

## Course Information

This handout describes basic course information and policies. Most of the sections will be useful throughout the course. The main items to pay attention to **NOW** are:

1. Please make sure you are signed up through Stellar, and talk to the TAs if there is a problem.
2. Please note the dates of the quizzes and make sure to keep these dates free.
3. Please note the collaboration policy for homeworks.
4. Please note the grading policy, and in particular, the penalty for *missed* problems.

## 1 Staff

The lecturers for this course are Prof. Srinivas Devadas and Prof. Ronitt Rubinfeld. Please see the stellar website for names and contact information for lecturers and teaching assistants.

The course website is at:

<https://stellar.mit.edu/S/course/6/fa12/6.046J/>

The staff e-mail is: 6046-tas@mit.edu

## 2 Registration for recitations

If you would like to switch the recitation assigned to you by the registrar, please contact Yotam Aron (yyaron@mit.edu). In the first week of class, requests for changes will generally be approved if space permits.

## 3 Prerequisites

This course is the header course for the MIT/EECS Engineering Concentration of Theory of Computation. You are expected, and strongly encouraged, to have taken:

- Either 6.006 *Introduction to Algorithms* or 6.001 *Structure and Interpretation of Computer Programs*, and
- Either 6.042J/18.062J *Mathematics for Computer Science* or 18.310 *Principles of Applied Mathematics*

and received grades of C or better.

Petitions for waivers will be considered by the course staff. Students will be responsible for material covered in prerequisites.

## 4 Lectures & Recitations

Lectures will be held in room 26-100 from 11:00 A.M. to 12:30 P.M. on Tuesdays and Thursdays. You are responsible for material presented in lectures, including oral comments made by the lecturer.

Students must also attend a one-hour recitation session each week. You are responsible for material presented in recitation. Attendance in recitation has been well correlated in the past with exam performance. Recitations also give you a more personalized opportunity to ask questions and interact with the course staff. Your recitation instructor will assign your final grade.

Recitations will be taught by the teaching assistants on Fridays.



## 5 Problem sets

Six problem sets will be assigned during the semester. The course calendar, available from the course webpage, shows the tentative schedule of assignments and due dates. The actual due date will always be on the problem set itself. Homework must be turned in by 11:59 pm on the due date.

- Late homework will generally not be accepted. If there are extenuating circumstances, you should make *prior* arrangements with your recitation instructor. *An excuse from the Dean's Office will be required if prior arrangements have not been made.* In all cases, late homework must be submitted online on the course website.
- Each problem must be written up separately, since problems may be graded by separate graders. Mark the top of each sheet with the following: (1) your name, (2) the name of your recitation instructor, and the time your recitation section meets, (3) the question number, (4) the names of any people you worked with on the problem (see Section 8), or "Collaborators: none" if you solved the problem completely alone.
- Answers should be submitted online to the Stellar website in PDF format. Formatting your problem set in  $\text{\LaTeX}$  will make it easier for us to read; however, any method of generating the PDF is acceptable (including scanning handwritten documents) as long as it is clearly legible.
- The problem sets includes exercises that should be solved but not handed in. These questions are intended to help you master the course material and will be useful in solving the assigned problems. Material covered in exercises will be tested on exams.

## 6 Guide to writing up homework

You should be as clear and precise as possible in your write-up of solutions. Understandability of your answer is as desirable as correctness, because communication of technical material is an important skill.

A simple, direct analysis is worth more points than a convoluted one, both because it is simpler and less prone to error and because it is easier to read and understand. Sloppy answers will receive fewer points, even if they are correct, so make sure that your handwriting and your thoughts are legible. If writing your problem set by hand, it is a good idea to copy over your solutions to hand in, which will make your work neater and give you a chance to do sanity checks and correct bugs. If typesetting, reviewing the problem set while typing it in often has this effect. In either case, going over your solution at least once before submitting it is strongly recommended.

You will often be called upon to "give an algorithm" to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of your essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. At least one worked example or diagram to show more precisely how your algorithm works.
3. A proof (or indication) of the correctness of the algorithm.
4. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions.

## 7 Grading policy

The final grade will be based on six problem sets, one in-class quiz, one take-home quiz, a final during final exam week, and participation during the weekly recitation sections. Quiz 1 will be in class on Thursday, October 11, 11:00 A.M. to 12:30 P.M. in room 26-100. Quiz 2 will be given out Thursday, November 8, at the end of lecture and will be due on Wednesday, November 14, at 5:00 P.M.

The grading breakdown is as follows:

Problem sets	25%
In-class quiz	20%
Take-home quiz	25%
Final exam	30%

Although the problem sets account for only 25% of your final grade, you are required to at least attempt them. The following table shows the impact of failing to attempt problems:

Questions skipped	Impact
0	None
1	One-hundredth of a letter grade
2	One-tenth of a letter grade
3	One-fifth of a letter grade
4	One-fourth of a letter grade
5	One-third of a letter grade
6	One-half of a letter grade
7	One letter grade
8	Two letter grades
9 or more	Fail

Please observe that this table is for *questions* skipped, not *problem sets*.



## 8 Collaboration policy

The goal of homework is to give you practice in mastering the course material. Consequently, you are encouraged to collaborate on problem sets. In fact, students who form study groups generally do better on exams than do students who work alone. If you do work in a study group, however, you owe it to yourself and your group to be prepared for your study group meeting. Specifically, you should spend at least 30–45 minutes trying to solve each problem beforehand. If your group is unable to solve a problem, talk to other groups or ask your recitation instructor.

**You must write up each problem solution by yourself without assistance**, however, even if you collaborate with others to solve the problem. You are asked on problem sets to identify your collaborators. If you did not work with anyone, you should write “Collaborators: none.” If you obtain a solution through research (e.g., on the web), acknowledge your source, but write up the solution in your own words. **It is a violation of this policy to submit a problem solution that you cannot orally explain to a member of the course staff.**

**No collaboration whatsoever is permitted on quizzes or exams.** The course has a take-home exam for the second quiz which you must do entirely on your own, even though you will be permitted several days in which to do the exam. More details about the collaboration policy for the take-home exam will be forthcoming in the lecture on Tuesday, November 15. Please note that this lecture constitutes part of the exam, and attendance is mandatory.

Plagiarism and other dishonest behavior cannot be tolerated in any academic environment that prides itself on individual accomplishment. If you have any questions about the collaboration policy, or if you feel that you may have violated the policy, please talk to one of the course staff. Although the course staff is obligated to deal with cheating appropriately, we are more understanding and lenient if we find out from the transgressor himself or herself rather than from a third party.

## 9 Textbook

The primary written reference for the course is the third edition of the textbook *Introduction to Algorithms* by Cormen, Leiserson, Rivest, and Stein. In previous semesters the course has used the first or second edition of this text. We will be using material and exercise numbering from the third edition, making earlier editions unsuitable as substitutes.

The textbook can be obtained from the MIT Coop, the MIT Press Bookstore, and at various other local and online bookstores.

## 10 Course website

The course website contains links to electronic copies of handouts, corrections made to the course materials, and special announcements. You should visit this site regularly to be aware of any changes in the course schedule, updates to your instructors’ office hours, etc. You will be informed

via the web page and/or email where and when the few handouts that are not available from the web page can be obtained.

In addition, you should use the Stellar website to submit problem sets and check on your grades.

## **11 Extra help**

Based on the desires of the students, the teaching staff will offer regular office hours. Details will be discussed in recitation during the first week of class. You may attend the office hours of any TA (not just your own).

Further help may be obtained through tutoring services. The MIT Department of Electrical Engineering and Computer Science provides one-on-one peer assistance in many basic undergraduate Course VI classes. During the first nine weeks of the term, you may request a tutor who will meet with you for a few hours a week to aid in your understanding of course material. You and your tutor arrange the hours that you meet, for your mutual convenience. This is a free service. More information is available on the HKN web page:

<https://hkn.mit.edu/tutoring/index.php>

Tutoring is also available from the Tutorial Services Room (TSR) sponsored by the Office of Minority Education. The tutors are undergraduate and graduate students, and all tutoring sessions take place in the TSR (Room 12-124) or the nearby classrooms. For further information, go to

<http://web.mit.edu/tsr/www>

**This course has great material, so HAVE FUN!**



**6.046/18.410 Design and Analysis of Algorithms****Calendar / Fall 2012**[Show academic calendar](#)[Subscribe to Calendar \(Beta\)](#)[Show documents](#)[Schedule view](#)

	Mon	Tue	Wed	Thu
<b>Sep</b>	3	4	5	6 11:00 a.m. – 12:30 p.m. Lecture 1: Overview, Interval Scheduling (Reading: CLRS 16.1)
	10	11 HW1 Out 11:00 a.m. – 12:30 p.m. Lecture 2: Divide and Conquer: Median Finding (Reading: CLRS 9.3)	12	13 11:00 a.m. – 12:30 p.m. Lecture 3: Divide and Conquer, FFT
	17	18 11:00 a.m. – 12:30 p.m. Lecture 4: Randomized Algorithms (Reading: CLRS Ch. 5, Ch. 7)	19	20 11:00 a.m. – 12:30 p.m. Lecture 5: More Randomized Algorithms (Reading: CLRS 31.8)
	24	25 HW1 Due HW2 Out 11:00 a.m. – 12:30 p.m. Lecture 6: Dynamic Programming (Reading: CLRS Ch. 15)	26	27 11:00 a.m. – 12:30 p.m. Lecture 7: All Pairs Shortest Path (Reading: CLRS Ch. 24–25)
<b>Oct</b>	1	2 11:00 a.m. – 12:30 p.m. Lecture 8: Greedy Algorithms (Reading: CLRS Ch. 16, Ch. 23)	3	4 Lecture 9: Max Flow, Min Cut (Reading: CLRS Ch. 26)
	8	9 HW2 Due	10	11 HW3 Out 11:00 a.m. – 12:30 p.m. Quiz 1
	15	16 11:00 a.m. – 12:30 p.m. Lecture 10: Matching (Reading: CLRS 26.3)	17	18 11:00 a.m. – 12:30 p.m. Lecture 11: P, NP and NP-Completeness (Reading: CLRS Ch. 34)
	22	23 11:00 a.m. – 12:30 p.m. Lecture 12: Problem Reduction (Reading: CLRS Ch. 34)	24	25 HW3 Due HW4 Out 11:00 a.m. – 12:30 p.m.

Mon	Tue	Wed	Thu
			Lecture 13: Linear Programming (Reading: CLRS Ch. 29)
29	30 11:00 a.m. – 12:30 p.m. Lecture 14: Linear Programming II (Reading: CLRS Ch. 29)	31	
<b>Nov</b>			1 11:00 a.m. – 12:30 p.m. Lecture 15: Randomized Algorithms II, Hashing (Reading: CLRS 11.3, 11.5)
5	6 HW4 Due HW5 Out 11:00 a.m. – 12:30 p.m. Lecture 16: Amortized Analysis (Reading: CLRS Ch. 17)	7	8 Quiz 2 Out 11:00 a.m. – 12:30 p.m. Lecture 17: Approximation Algorithms (Reading: CLRS Ch. 35)
12	13 No Lecture	14 Quiz 2 Due	15 11:00 a.m. – 12:30 p.m. Lecture 18: Approximation Algorithms II (Reading: CLRS Ch. 34)
19	20 11:00 a.m. – 12:30 p.m. Lecture 19: Parallel and Distributed Algorithms (Reading: CLRS Ch. 27)	21	22
26	27 HW5 Due HW6 Out 11:00 a.m. – 12:30 p.m. Lecture 20: Intro to Cryptography	28	29 11:00 a.m. – 12:30 p.m. Lecture 21: Clustering
<b>Dec</b>			
3	4 11:00 a.m. – 12:30 p.m. Lecture 22: Sublinear Algorithms	5	6 HW6 Due 11:00 a.m. – 12:30 p.m. Lecture 23: Compression
10	11 11:00 a.m. – 12:30 p.m. Lecture 23: Interactive Proofs	12	13
17	18	19	20
24	25	26	27
31			

## Course Objectives and Outcomes

### Course Objectives

This course assumes that students know how to analyze simple algorithms and data structures from having taken 6.006, and introduces students to design of computer algorithms, as well as analysis of sophisticated algorithms. Upon completion of this course, students will be able to do the following:

- Analyze the asymptotic performance of algorithms.
- Demonstrate a familiarity with major algorithms and data structures.
- Apply important algorithmic design paradigms and methods of analysis.
- Synthesize efficient algorithms in common engineering design situations.
- Understand the difference between tractable and intractable problems, and be familiar with strategies to deal with intractability.

### Course Outcomes

Students who complete the course will have demonstrated the ability to do the following:

- Argue the correctness of algorithms using inductive proofs and loop invariants.
- Analyze worst-case running times of algorithms using asymptotic analysis. Compare the asymptotic behaviors of functions obtained by elementary composition of polynomials, exponentials, and logarithmic functions. Describe the relative merits of worst-, average-, and best-case analysis.
- Analyze average-case running times of algorithms whose running time is probabilistic. Employ indicator random variables and linearity of expectation to perform the analyses. Recite analyses of algorithms that employ this method of analysis.
- Explain the basic properties of randomized algorithms and methods for analyzing them. Recite algorithms that employ randomization. Explain the difference between a randomized algorithm and an algorithm with probabilistic inputs.
- Describe the divide-and-conquer paradigm and explain when an algorithmic design situation calls for it. Recite algorithms that employ this paradigm. Synthesize divide-and-conquer algorithms. Derive and solve recurrences describing the performance of divide-and-conquer algorithms.



- Describe the dynamic-programming paradigm and explain when an algorithmic design situation calls for it. Recite algorithms that employ this paradigm. Synthesize dynamic-programming algorithms, and analyze them.
- Describe the greedy paradigm and explain when an algorithmic design situation calls for it. Recite algorithms that employ this paradigm. Synthesize greedy algorithms, and analyze them.
- Explain the major graph algorithms and their analyses. Employ graphs to model engineering problems, when appropriate. Synthesize new graph algorithms and algorithms that employ graph computations as key components, and analyze them.
- Describe a linear program and cite problems that can be solved using linear programming. Reduce problems to linear programming formulations. Understand the complexity of various linear programming approaches.
- Explain basic complexity classes such as P, NP, and NP-complete, and be able to use analysis and reduction techniques to show membership or non-membership of a problem in these classes.
- Understand and explain approaches to dealing with problems that are NP-complete such as the design of heuristic, approximation, or fixed-parameter algorithms.



## References

The principal text for this course is

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms* (Third Edition), MIT Press, 2009.

The following books are additional references.

### Texts on Algorithms

1. Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms*, McGraw-Hill, 2006.
2. Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2005.

### Algorithms + Programming

1. Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986. Applications of algorithm design techniques to software engineering.
2. Jon Bentley. *More Programming Pearls*. Addison-Wesley, 1988. More applications of algorithm design techniques to software engineering.
3. Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982. Performance hacking extraordinaire.
4. Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1997.
5. G. H. Gonnet. *Handbook of Algorithms and Data Structures*. Addison-Wesley, 1984. Pascal and C code, comparisons of actual running times, and pointers to analysis in research papers.

### Pointers to materials on advanced topics

1. Shimon Even. *Graph Algorithms*. Computer Science Press, 1979. Broad treatment of graph algorithms, including network flow and planarity.
2. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., San Francisco, 1979. Reference book devoted to NP-completeness. The second half contains an extensive list of NP-complete problems and references to algorithms in the literature for polynomial-time special cases.
3. Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997. General treatment of algorithms that operate on character strings and sequences.

4. Eugene L. Lawler. *Combinatorial Optimization*. Holt, Rinehart, and Winston, 1976. (Dense) graph algorithms, network flows, and linear programming. First few chapters are excellent.
5. Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982. Linear programming and its variants.
6. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Co., 1997. A good text on computability and complexity theory, with proof ideas to kick off each proof.
7. Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983. An advanced book with tons of good stuff.

### Background Mathematics

1. Kai Lai Chung. *Elementary Probability Theory with Stochastic Processes*. Springer-Verlag, 1974. Intuitive introduction to probability.
2. William Feller. *An Introduction to Probability Theory and Its Applications*. John Wiley & Sons, 1968 (Volume 1), 1971 (Volume 2). Excellent reference for probability theory.
3. C. L. Liu. *Introduction to Combinatorial Mathematics*. McGraw-Hill, 1968. Combinatorial mathematics relevant to computer science. Excellent problems.
4. Ivan Niven and Herbert S. Zuckerman. *An Introduction to the Theory of Numbers*. John Wiley & Sons, 1980. Readable introduction to number theory.

(Rahul is in this class)

(Sitting in the dark)

↳ Circuit breaker went out

Fixit has been called

Serini Davidas

- 25 years

)phoretic

Romni Drabentfield

6 TAs

Big sister of 6.006

Basic alg, data structure, programming

→ No implementing here  
But more design of algorithms

More rigor

Correctness + complexity

↑  
inc best sol as well

Runtime complexity  
Space

- asymptotically

②

Course policies on Stellar

Lecture notes - after lecture

(Rights fixed)

Prof could not find button - bad

10 sections

Soft limit 25 students

Read Course Collab Policy

Content

Some modules new + old  
"depending on Prof"

Divide + Conquer

- apply to new problems

- FFT

Optimization

- greedy

- dynamic programming



③

Network flow

intractability

- efficient = polynomial time

Linear programming

Reduction

break alg for A

if reduce B to A  $\Rightarrow$  solved!

Sublinear algorithm

Big Data

Sample the data - don't have time to look at all

Approximation

- it can't solve exactly

④

Ver, similar problems can have quite diff complexity

P class of problems solvable in poly time

$O(n^k)$  for constant  $k$   
generally 2 or 3

~~Shortest~~

- shortest path problems

NP non-deterministic polynomial time

= Class of probs verifiable in poly time

ie Hamiltonian cycle problem

Given a directed graph - can find cycle  
that visits each vertex 1x

Can verify, but can't find in P

NP-complete { problem is in NP and is

as hard as any problem in NP

↳ if any NPC prob can be solved in poly time

↳ all problems in NP can be solved in poly time  
ie. Hamiltonian cycle

5

Does  $P \stackrel{?}{=} NP$ ?

We do know  $P \subseteq NP$

## Interval Scheduling

resources + requests

start w/  
single  
one

$R = \{1, 2, \dots, n\}$

$s(i)$  = start time

$f(i)$  = finish time

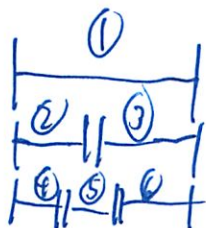
$s(i) < f(i)$   
can't be 0

two requests compatible if don't overlap

$[i, j]$

$f(j) \leq s(i)$

Basically



6

Goal: Select compatible subset of  $R$  w/ maximum size  
ie maximize # of units  
no credit limit

Can't have any overlap - at all

$n$ -requests means  $2^n$  subsets  
 $O(2^n)$

---

Compatibility check  
- brute force  
- terrible

~~(compat)~~  
is  $O(n^2)$  each  $\rightarrow$  so  $O(2^n n^2)$

---

Claim: A greedy alg solves this in poly time

Use a simple rule to select request  $i$   
Reject all requests ~~on~~ incompatible w/  $i$   
Repeat until all requests are processed

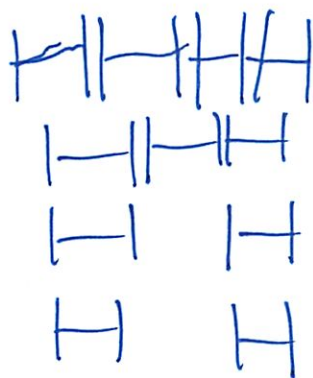
↓ "growing" subset



7

Some ideas (was not even paying attention!)

1. For each request  $\rightarrow$  find # incompatible  
Select the one w/ the least



$\uparrow$   
2 incompat

but this isn't optimal

2. Select earliest end time  
~~pick~~ Right, will come back

3. Select smallest



No

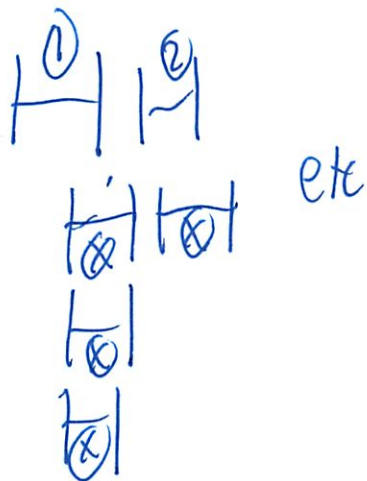
⑧

4. Starts earliest  
No

5. ~~Req.~~ Ends earliest

apply over + over

elim all incompatible req



But how prove that it works?  
Don't forget!

Will do in recitation two

Show contradiction if beat what Greedy does

Sketch: Assume optimal solution  $O$   
Greedy produces  $A$

⑨

Show that  $|A| = |O|$   
↑ cardinality of

Because greedy alg "stays ahead"

at the time  $O$  finishes or earlier  
show w/ induction

---

Complexity of greedy alg

$O(n \lg n)$

↳ Since sorting for earliest finish time  
Only need to sort once

---

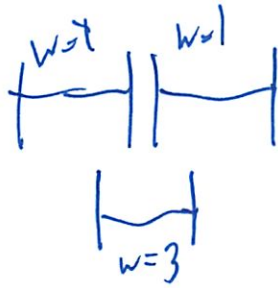
But what about weighted interval schedule

- have classes in priority - ish order  
ie you like some classes more than others

1. Each request  $i$  has weight  $w(i)$   
Schedule subset of requests w/ max weight



But



So obviously we need to look at weights

Greedy no longer solves this problem!

Are there other paradigms we could use?

1. Transform into graph

- might work

- but shortest path is greedy

<sup>or</sup> - could blow up (exponentially) size of graph

2. Dynamic Programming

"if you don't know what to do - guess"

and memoize

- only solve once!

- or grows exponential

⑩ ⑪

$R$  = set of requests




Subproblems

are  $R^x \{ \text{request } j \in R \mid s(j) \geq x \}$   
 (trying) to shrink size of problem solving (1)  
 defn' of subproblem for DP  
 a new problem-based on  $x$

Set  $x = f(i)$  for each  $i \rightarrow n$  subproblems

Can't pick any requests that overlap  $i$

Must start after  $j$  finishes

So if   
 pick 2   
 Can only   
 Pick from 3, 6

We are building sol in ordered fashion

(12)

So what does DP look like?

DP Guessing

Try each request  $i$  as a possible FIRST,  
remaining reqs  $R^{f(i)}$

$$\text{opt}(R) = \max(w_i + \text{opt}(R^{f(i)}))$$

~~at start~~

$1 \leq i \leq n$

at start

Try each request - just  $w_i$   
Others  $R^{f(i)}$  is drop all req before  
and incomputable

Then can write it up w/ memoization  
and try it  
and see if it works

Complexity?  $O(n^2)$

(need to think abt more!)



(13)

We can actually do better,  
In a few weeks

Hint Use sorting initially to reduce ~~DP~~ DP complexity  
to  $O(n) \rightarrow$   
 $\downarrow$  overall complexity  $O(n \log n)$

Is more complex  $\rightarrow$  15-20 min to explain

Are many related problems

- That can solve w/ greedy or DP

Can make problem more homogeneous

Multiple non-identical machines

Reqs  $1, \dots, n$   $s(i), f(i)$  as before

$m$  machine types  $T = \{T_1, \dots, T_m\}$

(14)  
 $Q(i) \subseteq T$  is set of machines that request  $i$   
can be performed on

ie Machine  $T_1$  can handle jobs A, B  
"  $T_2$  A only

Goal: Maximize # of jobs scheduled on  $n$   
machines

---

legality of solution

$\in NP$

↳ decision problems (yes or no)

---

Can  $k \leq n$  requests be scheduled?

NP-complete

---

So adding  $Q(i) \subseteq T$  made problem go  
from poly time to intractable

---

Optimization are NP-hard

(15)

## Deal w/ Intractability

1. Approx algorithms: Guarantee within some factors of optimal

$$1 - \frac{1}{e} \text{ approx alg}$$

↑  
2.718...

Guaranteed result is no worse than  
1.7 x current subset

does not always work

2. Pruning heuristics to reduce <sup>possibly</sup> exponential

- bounding - pruning heuristic
- can work well

3. Greedy or other suboptimal heuristics that work well

↳ but are not guaranteed

4. Reduction to "engines"  
Integer linear programming



10

For small enough find optimal

Or use pruning

6.046

4/6 ①  
L1

6.006 pre-requisite:

Data structures such as heaps, trees, graphs  
Algorithms for sorting, shortest paths,  
graph search, dynamic programming

Several modules:

Divide & Conquer - FFT, randomized algs

Optimization - greedy, dynamic prog

Network Flow

Intactability (and dealing with it)

Linear programming

Sublinear algorithms, approximation algs

Advanced topics

Read course information & objectives on Stellar  
Register on stellar for 6.046 (if you haven't  
and for a section already)

Pay particular attention to course collaboration  
policy!

②

## Theme of today's lecture

Very similar problems can have very different complexity.

Recall: P: class of problems solvable in polynomial time.  $O(n^k)$  for some constant  $k$ .  
Shortest paths in a graph  $O(V^2)$  e.g.

NP: class of problems verifiable in polynomial time.

Hamiltonian cycle a directed graph  $G(V, E)$  is a simple cycle that contains each vertex in  $V$ .

Determining whether a graph has a hamiltonian cycle is NP-complete but verifying that a cycle is hamiltonian is easy.

$P \subseteq NP$   
but is  
 $P = NP$ ?

NP-complete: problem is in NP and is as hard as any problem in NP.

If any NPC problem can be solved in poly time, then every problem in NP has a poly time solution.



# Interval Scheduling

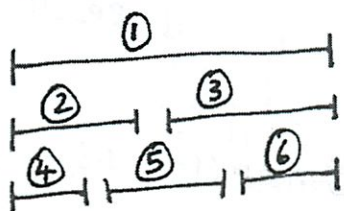
③

Resources & requests

Requests  $1, \dots, n$ , single resource

$s(i)$  start time,  $f(i)$  finish time  $s(i) < f(i)$

Two requests  $i$  &  $j$  are compatible if they don't overlap, i.e.,  $f(i) \leq s(j)$  or  $f(j) \leq s(i)$



3 compatible requests

Goal: select a compatible subset of requests of maximum size.

(Claim: We can solve this using a greedy algorithm.

A greedy algorithm is a myopic algorithm that processes the input one piece at a time with no apparent look ahead

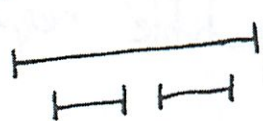
# Greedy Interval Scheduling

(4)

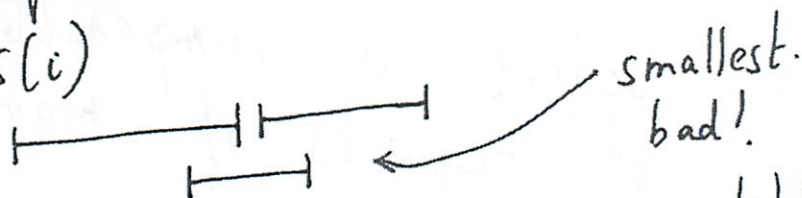
Use a simple rule to select a request  $i$ .  
Reject all requests incompatible with  $i$ .  
Repeat until all requests are processed.

## Possible rules?

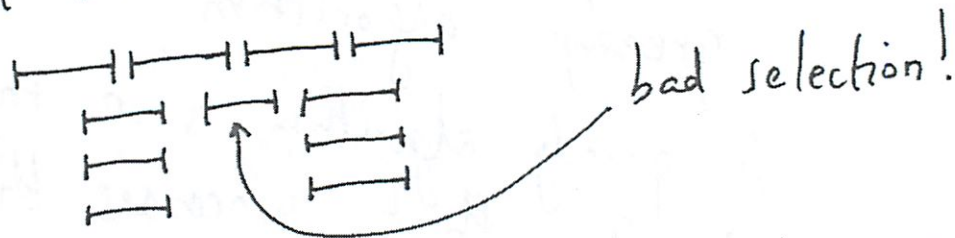
1. Select request that starts earliest, i.e., minimum  $s(i)$



2. Select request that is smallest, i.e., minimum  $f(i) - s(i)$



3. For each request find # incompatibles.  
Select the one with minimum # incompatibles.



4. Select request with earliest finish time, i.e., minimum  $f(i)$

Exercise: Prove greedy algorithm based on 5  
earliest finish time is optimal.

Sketch: Assume optimal solution  $O$ .

Greedy produces  $A$ . Show  $|A| = |O|$ ,  
by showing greedy algo "stays ahead".

Lemma: For all indices  $r \leq k$ ,  $f(i_r) \leq f(j_r)$   

$\nearrow$   
rth request  
Selected by A

$\nearrow$   
rth request  
Selected by O

Prove using induction.

Complexity? Sort in terms of earliest finishing time  
 $O(n \log n)$

Related problem: Interval Partitioning

Schedule all requests using a minimum  
number of identical resources.

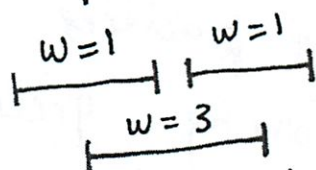
Similar algo works. Sort in terms of  
earliest finish times. Use existing resources  
if possible, else use a new/available one.



## Weighted Interval Scheduling

(6)

Each request  $i$  has weight  $w(i)$   
Schedule subset of requests with  
maximum weight.



greedy algo no longer works

## Dynamic Programming

Subproblems are

$$R^x = \{ \text{request } j \in R \mid s(j) \geq x \}$$

If we set  $x = f(i)$  then  $R^x$  is  
the set of requests later than request  $i$   
 $n$  different subproblems, one for each request  
Only need to solve each subproblem once &  
memoize

(7)

DP Guessing

Try each request  $i$  as a possible FIRST

If we pick request as the first request  
then remaining requests are  $Rf(i)$

Note: There may be requests compatible with  $i$  that are not in  $Rf(i)$  but we are picking  $i$  as the first request (i.e., we are going in order)

$$\text{opt}(R) = \max_{1 \leq i \leq n} (w_i + \text{opt}(Rf(i)))$$

Running time?  $O(n^2)$

Exercise: Use sorting initially & reduce  
DP complexity to  $O(n)$ . Overall  
complexity will be  $O(n \log n)$

## NON-IDENTICAL MACHINES.

⑧

requests  $1, \dots, n$ ,  $s(i)$ ,  $f(i)$  as before  
m machine types  $\mathcal{T} = \{T_1, \dots, T_m\}$

weight of 1 for each request.

$Q(i) \subseteq \mathcal{T}$  is set of machines that  
request  $i$  can be serviced on.

Maximize the number of jobs that can  
be scheduled on the m machines.

NP Can clearly check that any given subset  
of jobs with machine assignments is legal.

(Can  $k \leq n$  requests be scheduled? NP-complete  
Maximum requests should be scheduled. NP-hard.)

## Dealing with Intractability

- 1) Approximation algorithms: Guarantee within some factor of optimal in poly time.
- 2) Pruning heuristics to reduce (possibly exp) runtime on "real-world" examples
- 3) Greedy or other suboptimal heuristics that work well in practice  $\leftarrow$  no guarantees



# Reductions

For each request  $i$  and each machine  $j$   
 Boolean variable  $x_{ij}$  indicates whether  
 request  $i$  is scheduled on machine  $j$ .  
 If request  $i$  cannot be scheduled on machine  $j$ ,  
 i.e.  $j \notin Q(i)$ , set  $x_{ij} = 0$ .

$S$  = set of start times =  $\{s(i)\}$   
 For  $t \in S$ ,  $R(t)$  is set of requests containing  $t$ .  
 $\swarrow$  = 1 in unweighted case

$$\text{Max} \sum_{i=1}^n \sum_{j \in Q(i)} w(i) x_{ij} \quad \text{s.t.}$$

$$\forall i \sum_{j=1}^n x_{ij} \leq 1$$

Each job scheduled at  
most once

$$\forall j, \forall t \sum_{i \in R(t)} x_{ij} \leq 1$$

Each machine executes  
at most one job at  
each time point.

$$\forall i, \forall j \ x_{ij} \in \{0, 1\}$$

$$0 \leq x_{ij} \leq 1$$

Integer linear programming  
HARD!

Linear programming  
poly time!



## Strategies

2) Run ILP solver (e.g. CPLEX) that incorporates pruning heuristics to reduce runtime & provides optimal solution.  
— might not finish.

1) LP can be solved in poly time.  
But  $x_{ij}$  may be fractional in solution

Round up or down LP solution and show that solution is not too far off from optimal  $\Leftarrow$  approximation algo.

Bhatia et al  $(1 - 1/e)$  approximation algo.

Ai-zana

OH ~~all~~ M T W R 7-9 pm

half P-set due T<sup>half</sup> R

Email 6046-tas@mit.edu

no Piazza

2 quizzes - take home quiz

---

Interval Scheduling (cont.)

$$R = \{ R_1, \dots, R_n \}$$

$$\begin{array}{c} s(i) \\ f(i) \end{array} \quad s(i) \leq f(j)$$

Can't overlap at all  $f(i) \leq s(j)$

Greedy sol: Choose earliest finishing time 1st

Why?  $S = \{ R_{i_1}, \dots, R_{i_n} \}$  is in order

alternativ  $O = \{R_{j_1}, \dots, R_{j_m}\} \quad m \geq k$

Can prove w/ induction

~~AP~~ - recommended so sure it works

- must make sure alg correct

- much stricter than 6.06

## Induction claim

$$f(i_k) \leq f(j_k)$$

if tried to match request in order  
then finishing time sol is more than optimal  
then use optimal instead  
contradiction

③

Base case  $k=1$   $f(i_1) \leq f(j_1)$   
Since ar policy

If true for all previous  $k$ , consider  $k+1$

$$f(i_k) \leq f(j_k)$$

so  $S(j_{k+1})$  must be after  $f(i_k)$

So request is non-conflicting

$$\text{so } f(i_{k+1}) \leq f(j_{k+1})$$

$i_{k+1}$  must be smaller than optimal

$j_{k+1}$  is non conflicting w/  $i_k$

So greedy would have chosen earlier one

So this thing must hold

intervals not  
same length

but our sols we know are non-conflicting

h



(4)

Induction claim

$$m > k$$

$r_{\text{optimal}}$  is more than greedy

Then  $R_{j_{k+1}}$  is non conflicting

$$f(i_k) \leq f(j_k)$$

So  $j_{k+1}$  is non conflicting

So greedy would have added it

---

$R_{j_{k+1}}$  is non-conflicting w/  $S$

↓

Greedy solution will have chosen more requests

---

Must always specify an time

explain why - but not formal proof

$O(\ln n)$

5

## Weighted Interval Scheduling

$$R = \{R_1, \dots, R_n\}$$

$w(i)$  = weight

Goal is to choose max total weight

Yesterday: DP - solution

↳ not optimal though

Greedy algorithm: doesn't work!

↳ could find counter example

DP algorithm: Break into subproblems

Memorize

Combine

Don't redo work

↳ would be exponential

ie. fib sequence

6

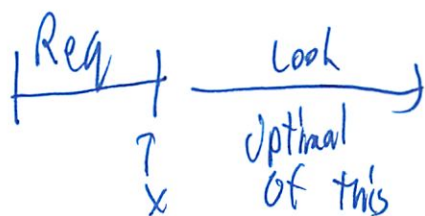
Subproblem:  $R^x = \{R_i \in R \mid s(i) \geq x\}$

= requests which start time at least  $x$

$x = f(i)$   
if finishing the other

Don't know what is optimal

So try all



How pre-sort requests?

Start-time

Sorting  $O(n \lg n)$

Weighted - interval - scheduling  $(\underbrace{\text{start}}_{R_i}, \underbrace{\text{num. req}}_n) \in$

indices of corresponding requests

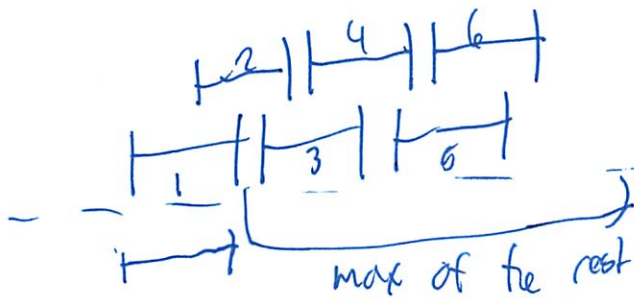
①

$R = \{R_1, \dots, R_n\}$  sorted by starting time  
↑ indices

# start is first request

So sol is weight + best of rest

$w(\text{start}) + \text{weighted\_interval\_scheduling}(\text{max of the rest}, n)$



best = 0

for ( $i = \text{start}, i \leq n, i++$ ) {

    Choose non-conflicting  $i$

$w(\text{start}) + \text{weighted\_interval\_scheduling}(i, n)$

    if best < total {

        best = total

    }

}

3



⑧

$m[\text{start}] = \text{best}$

Call the memoize stuff

## Class of problems

P: Class of problems solvable in polynomial time

NP: Class of problems verifiable in polynomial time

NP-Complete: Problems in NP that are as hard as any problem in NP.

(If any NP-C problem can be solved in polynomial time, ALL problems in NP can be solved in polynomial time)

Intractable: Not solvable in polynomial time

## Interval Scheduling

Scheduling use of Resource (classroom, time) to grant Requests(classes, appointments)

$R = \{R_1, R_2, \dots, R_n\}$  : n requests

$s(i)$  : start time

$f(i)$  : finish time       $s(i) < f(i)$

Find a schedule  $S \subseteq R$ , a set of compatible requests, that optimize some given criterion.

### 1. Interval Scheduling: As many requests as possible

Goal: select a compatible subset of R with maximum size

By brute force:

$O(2^n)$  subsets of requests \*  $O(n^2)$  compatibility checks for each subset

Exponential time. Not good

Greedy algorithm:

Use simple rule to select a request i

Reject all requests incompatible with i

Repeat until all requests are processed

Greedy algorithm with first-to-finish rule gives an optimal schedule in polynomial time

-Add a request that has earliest finishing time into the schedule

-Reject all request that overlaps with the request

-Repeat

Proof

- suppose greedy algorithm returns  $S = \{R_{i1}, R_{i2}, \dots, R_{ip}\}$
- $|S| = p$ , results are "in order" ( $R_{i1}$  finishes first)
- Let  $S^* = \{R_{j1}, R_{j2}, \dots, R_{jq}\}$  be some optimal schedule (in order).  $|S^*| = q = \text{opt}(R)$
- There exists an optimal schedule  $S^{**}$  starting with  $R_{i1}$ .  
(Relace  $R_{j1}$  with in  $R_{i1}$ ; OK since  $f(i1) < f(j1)$  by the greedy algorithm rule.)
- Let  $R^* = \{R_i \in R \mid s(i) \geq x\}$
- Then  $\{R_{j2}, \dots, R_{jq}\}$  must be optimal for  $R^{(i1)}$ ,  
otherwise we could replace it with better solution and  $S^{**}$  wouldn't be optimal.
- Repeat the same steps, then there exists an optimal schedule that starts with  $R_{i1}, R_{i2}, \dots, R_{ip}$ .
- S should be an optimal schedule, since S was returned by a greedy algorithm. There were no more compatible requests after  $f(R_{ip})$ .

Running time:  $O(n \lg n)$

-sort requests in order of increasing  $f(i)$

-consider each in turn, adding it to S iff it's compatible with last interval added to S.

## 6.046 Rec 1 9/7/12

### 2. Weighted Interval Scheduling: As much weight in the scheduled requests

Same as before, but now each request also has a "weight"  $w(i)$ .

Goal: select a compatible subset of  $R$  with maximum combined weight

Greedy algorithm doesn't work. Counterexamples?

Dynamic programming:

- Define subproblems

- Find the solution for entire problem using subproblems, memoization

Subproblems:  $R^x = \{R_i \in R \mid s(i) \geq x\}$

There are  $n$  different subproblems, one for each  $x = f(i)$ .

Save each subproblem once and save the solution for later use.

We don't have a rule telling us which interval to schedule first, so try each  $R_i$  in turn as possible "first".

Let's assume  $R = \{1, 2, \dots, n\}$  sorted in increasing order of start times.

Now if I want to compute  $R^x$ , where  $x = f(i)$ , then all I have to do is find the smallest index  $j$  such that  $s(j) \geq f(i)$ . And all subsequent indices/requests will also have  $s(k) \geq f(i)$ .

We have a dictionary/memo-table  $M$  with a request-index as the key, and the value is the optimal solution. The request-index corresponds to a subproblem with the request-index being the first request.

```
weighted_interval_scheduling(start = 1, num_req = n) {
```

```
    if (start > n) return 0
```

```
    Lookup  $M[start] = sol$ , if  $sol$  is not null, return  $sol$ .
```

```
    best = 0
```

```
    for ( $i = start$ ;  $i \leq n$ ;  $i++$ ) {
```

```
        /* Compute  $R^{f(i)}$  */
```

```
        Find smallest index  $t$  such that  $s(t) \geq f(i)$ 
```

```
         $tmp = w_i + \text{weighted\_interval\_scheduling}(t, n)$ 
```

```
        if ( $best < tmp$ )  $best = tmp$ 
```

```
    }
```

```
     $M[start] = best$ 
```

```
}
```

You can trace back to get the solution given the  $M$  array in  $O(n)$  time

### 3. "Multiple non-identical machines": Resources are multiple different things, more constraints on how to grant requests

Same as interval scheduling or weighted interval scheduling, but now with machine types  $T = \{T_1, T_2, T_3, \dots, T_m\}$

$Q(i) \subseteq T$ : set of machines that request  $i$  can be served on.

Goal: Maximize number of jobs that are scheduled on  $m$  machines

Legality of solution: NP

Can  $k \leq n$  requests be scheduled?: NP-Complete

Maximize the number of requests: NP-Hard

6.046 L2

9/11

## Divide + Conquer

Today: Master theorem  
Convex Hull  
Median Finding

~~Gen~~

Before Merge Sort

Straight forward Divide + Conquer

d

Next the FFT

---

Paradigm Given a problem of size  $n$

- Divide into subproblems of size  $\frac{n}{b}$   
 $b > 1$

Geometric decrease does count

- Solve each subproblem + solve
- Combine/merge ~~step~~ solutions
  - trickiest pt



②

We can write a recurrence

and discuss ways to solve recurrences

ie Master Theorem

How many ways do you break a problem down?

Want  $a$  to be small

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ a T\left(\frac{n}{b}\right) + \Theta(n^p \log^q n) & \text{if } n > 1 \end{cases}$$

$\underbrace{\hspace{10em}}_{\text{recursion}}$

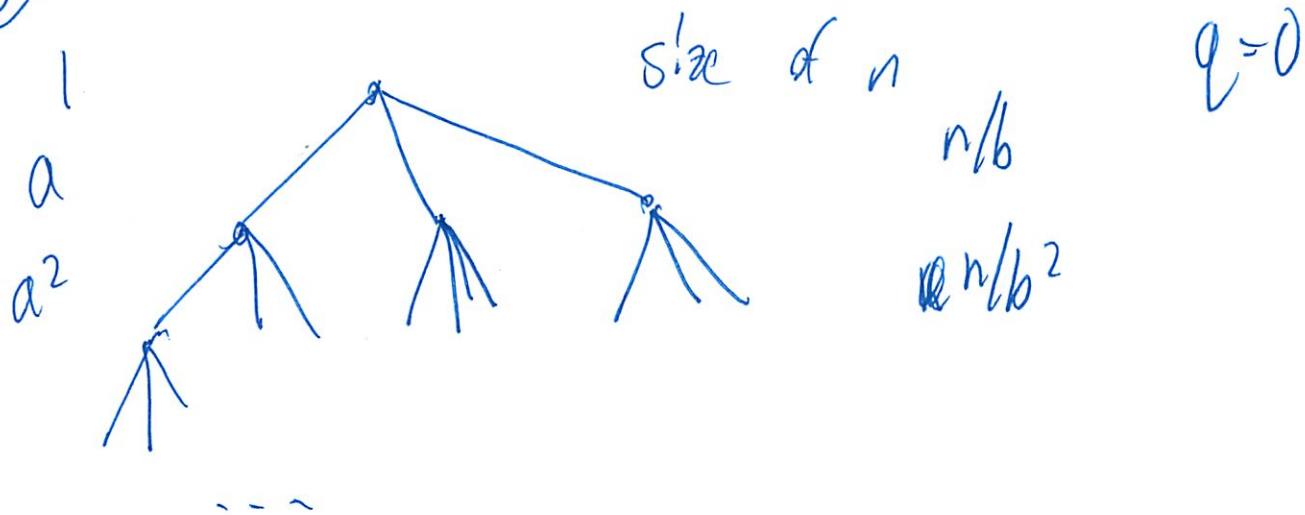
$\underbrace{\hspace{10em}}_{\text{divide} \quad \text{combine}}$

Then

$$T(n) = \begin{cases} \Theta(n^p \log^q n) & \text{if } p > \log_b a \\ \Theta(n^p \log^{q+1} n) & \text{if } p = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } p < \log_b a \end{cases}$$

So comparing the growth of the tree  
(What does each line mean?)

③



$a^h$

size-1

~~Proof for  $q=0$~~

~~Work for~~

Explainer for  $q=0$

Work

top  $n^p$

2nd  $a \left(\frac{n}{b}\right)^p$

3rd  $a^2 \left(\frac{n}{b^2}\right)^p$

At the bottom  $a^n (1)^p$

④

If sum cols - get total work from algorithm

$h = \text{height} = \# \text{ of levels}$

$$n = b^h$$

$$\begin{aligned} a^h &= (b^{\log_b a})^h \\ &= (b^h)^{\log_b a} \\ &= n^{\log_b a} \end{aligned}$$

Now can compare exponents  $a^h (1)^p$  vs.  $h^{\log_b a}$

3 cases  $\left( \begin{array}{l} \text{most work at root} \\ \text{middle} \\ \text{leaves} \end{array} \right.$

$$p > \log_b a$$

$$p = \log_b a$$

$$p < \log_b a$$

middle  $\Rightarrow$  equal amt of work at each level

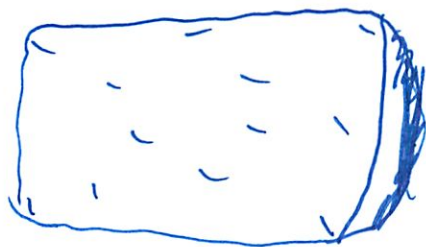
$$\Theta(n^p, h) = \Theta(n^p \log_b n)$$

5

## Convex Hull

$N$ -pts on a plane

What is the smallest polygon that includes  
all pts in plane



think abt all pts distinct  
no line w 3

Need the right order  
doubly linked list

So use divide + conquer

~~more~~ have 2 diff polygons we must merge



6

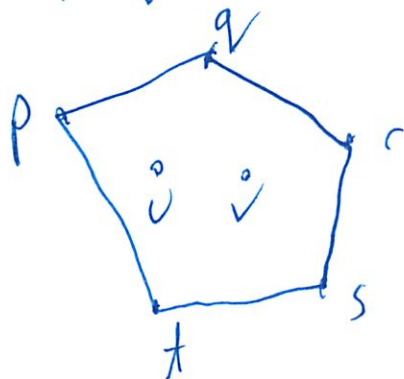
n-pts on plane

$$S = \{(x_i, y_i)\}$$

no two have same x  
no 3 on a line

Convex hull: smallest polygon containing all pts in S

5-sided polygons



$$CH(S) = [p \leftrightarrow q \leftrightarrow r \leftrightarrow s \leftrightarrow t \leftrightarrow p]$$

doubly linked list

note does not contain all pts

⑦

D+C

Sort pts by  $x$  coord

( $\text{fahh}$ )

Once + for all

$O(n \lg n)$  time

For input  $S$  of pts

- Divide into left half  $A$ , right half  $B$

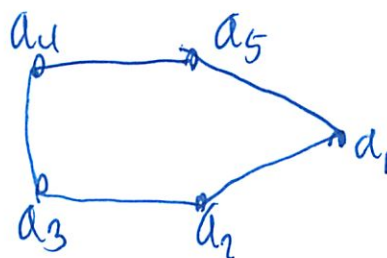
- Compute  $CH(A)$ ,  $CH(B)$

- Combine the two halves

$\hookrightarrow$  the fun part

merge sort: The two fingers - compare the two heads of the array

this is a bit more sophisticated

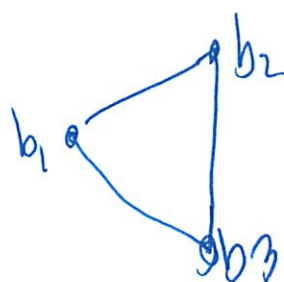


Clockwise Hing

8

$q_1$  = right most point

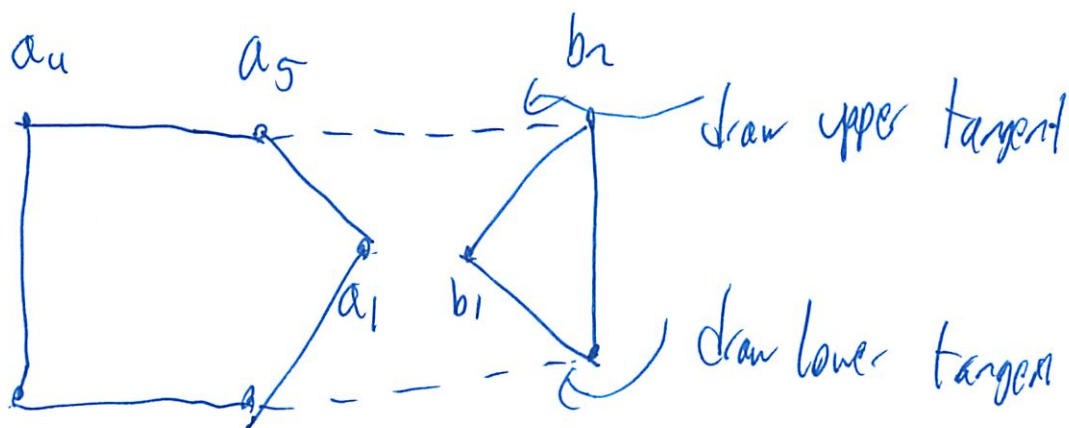
all  $b$  points will be to right of  $a$  points  
I guaranteed since how we broke  $y$



↪ naming

$b_1$  is the left-most pt

So to combine



(ah - so simple - and so intuitive!  
Think it through!)

9

Then some points fall out as well

$a_1, b_j$  = upper tangent  $a_5, b_2$

$a_k, b_m$  = lower tangent  $b_3, a_2$

$(a_1, a_2, a_3, a_4, a_5)$   $(b_1, b_2, b_3)$

So 1. link  $a_i$  to  $b_j$

2. Go down  $b$  list till see  $b_m$

3. Link  $b_m$  to  $a_k$

4. Continue along the  $a$  list until you return to  $a_i$ .

But how to find upper tangent?

We could try every  $a$  to every  $b$

Each will intersect  $L$  at some point

Take the one w/ highest intersection  $L$





(10)

Proof of convexity

But that's  $n^2$

↳ Proof: We are never satisfied w/  $n^2$

(∴ Prob use some heuristics)

Base cases - combine lines  
(∴ start at 3)

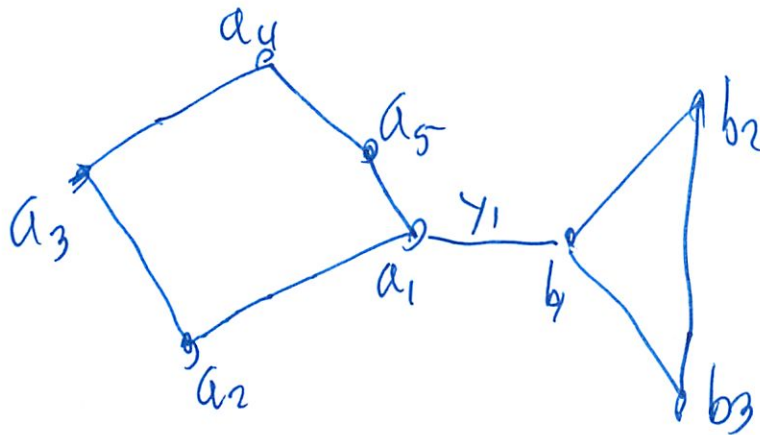
If we kept  $O(n^2)$ ,  
 $O(n^2 \lg n)$   $\hookrightarrow$  then overall

Instead, find the tangents

U.T. maximizes  $y(i, j)$  where  $y(i, j)$  is  
the  $y$ -intercept on line  $L$

②

Want to find the upper tangent  
L2 Finger algorithm



1. Start  $a_1, b_1 \rightarrow y_1$  the y intercept

2. Move CW  $b_1 \rightarrow b_2 \rightarrow y_2$

3. Check  $y_1$  vs  $y_2$   
if  $y_2$  (new) greater

B keeps moving CW

if ~~area~~  $area(y_2)$  smaller

A moves CCW

Retest

(12)

All of this linear time

So pseudocode

$i=1, j=1$   
↓

while ( $y(i, j+1) > y(i, j)$  or  $y(i-1, j) > y(i, j)$ );

terminate i when either conditions true  
two failures in a row

B Failed

Swap whose going

A Failed

When decide to stop?

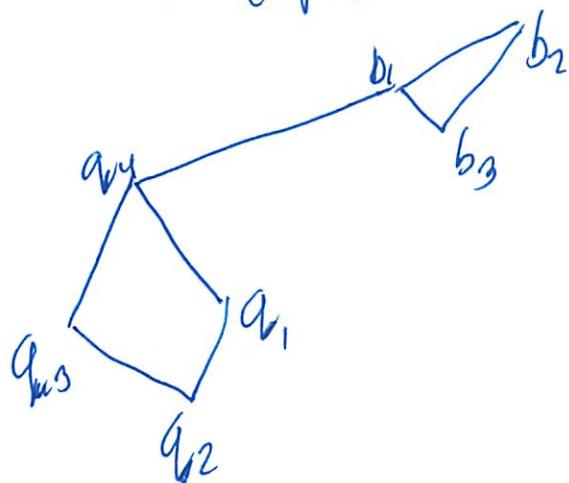
if ( $y(i, j+1) > y(i, j)$ ):

$j = j+1 \pmod{q}$

else:  
 $i = i-1 \pmod{p}$

⑬

~~Q~~ Q: Why not just pick the highest on each  
- his daughter is a MIT freshman



$b_2$  is highest pt

but V.T is  $a_4 - b_1$

not  $a_4 - b_2$  !

Complexity

↳ same as merge sort

~~lots of~~  $O(n)$  each step

dividing up  $a=2$   
 $b=2$

So  $O(n \lg n)$



(14)

## Median finding

finding the rank of a # within a list of #s

Given a set of  $n$  numbers

define  $\text{rank}(x)$  as number of #s in the  
set  $\leq x$

3, 1, 4, 2, 25

$$\text{rank}(2) = 1$$

$$\text{rank}(25) = 5$$

Median is special case of this

$\text{rank}\left[\frac{n+1}{2}\right]$  is lower median

Nice: Sort  $O(n \log n)$

Optimize around median case

15

Divide + Conquer

Select ( $S, i$ )

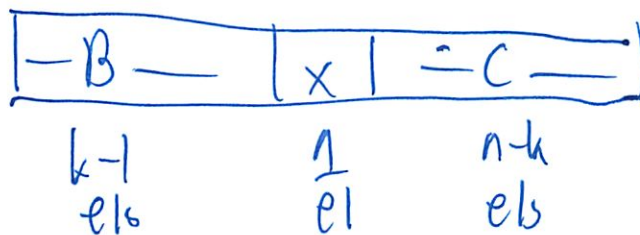
$S$  = set of  $n$  elements, rank of  $i$

- Pick  $x \in S$  (cleverly)

- Compute  $k = \text{rank}(x)$

$$B = \{y \in S \mid y < x\}$$

$$C = \{y \in S \mid y > x\}$$



Say randomly pick  $x$

Through linear traversal ( $n$ ) we can  
make  $B, C$  list

But often this is not in middle  
Need to pick  $x$  cleverly

(16)

Need  $x$  s.t. it is in the middle

Need lower bound on  $b$

ok

---

If  $k = i$ , ~~return x~~

→ else if  $k > i$

↳ return  $\text{Sel}(B, i)$

↳ not look at left side

→ else if  $k < i$

↳ return  $\text{Sel}(C, i - k)$

Get  $k$  close to  $\frac{n}{2}$

---

How do that?

Clever way from 1973

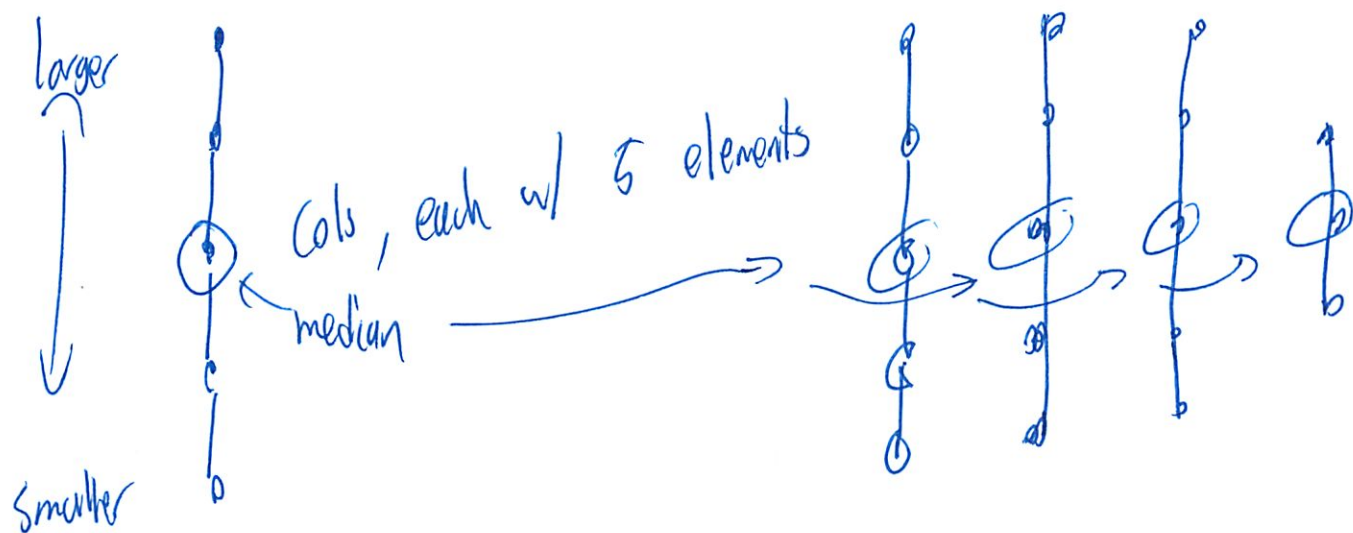
Need to pick  $x$  s.t.  $\text{rank}(x)$  is not extreme

(17)

Arrange  $S$  into columns of size 5  
 $\lceil n/5 \rceil$  cols

Sort each col  
big els on top  
linear time

Find "median of median" as  $x$



Want medians of medians as  $x$

Have  $\lceil \frac{n}{5} \rceil$  medians



18

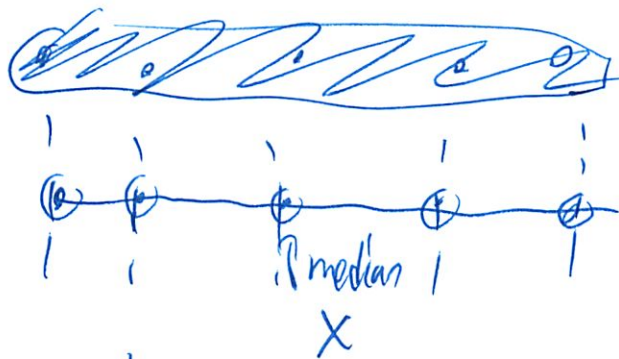
Can run on median of medians

$$X \rightarrow T\left(\left\lceil \frac{n}{3} \right\rceil\right)$$

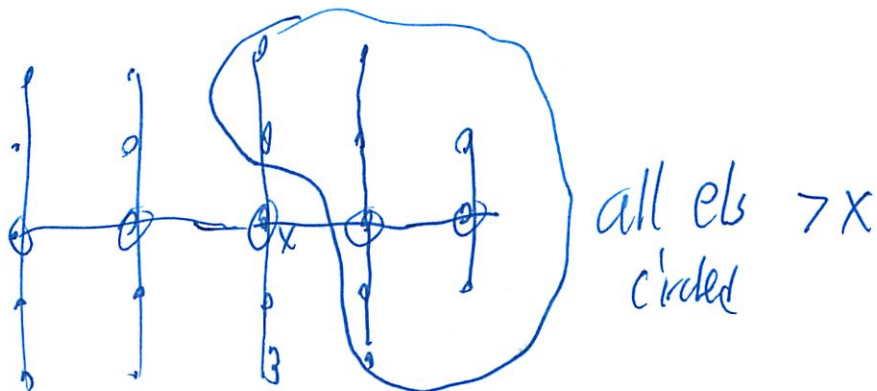
T size of problem

Computation of  $x$  requires recursive call

So take out 5 median of medians + sort them  
Well find the median of them



Take all the els  $> x$   
to the right



19

Half of the  $\lceil n/5 \rceil$  groups contribute at least  
3 elements  $> x$  except for 1 group  
w/ less than 5 elements + 1 group that contains  $x$

So have at least  $3(\lceil \frac{n}{5} \rceil - 2)$  els  $> x$

$$T(n) = \begin{cases} O(1) & \text{for small } n \\ T(\lceil \frac{n}{5} \rceil) + T(\frac{7n}{10} + 6) & \text{Median of medians} \end{cases}$$

discard  $\frac{3n}{10} + 6$  elements  
at least

How many elements the select will have

Still in world of guarantees + worst cases

2 Around 1/7 of the problem

$$b = \frac{10}{7}$$

where  $a = 1$

but has extra term - so no Master Theorem  
coefficient  $< 1$

## Divide & Conquer

- Master Theorem
- Convex hull
- Medians

## Paradigm

Given a problem of size  $n$

Divide it to subproblems of size  $\frac{n}{b}$

Solve each subproblem recursively ("conquer")

Combine solutions of subproblems to get overall solution

# Analysis of D&C ("Master" Theorem)

(2)

Ref: Ch 4

Suppose  $a \geq 1$  &  $b > 1$  are fixed integers

Let  $T(n)$  denote worst-case running time on input of size  $n$

Suppose  $n = b^n$  (same answer for general case)

$$\text{Suppose } T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ a T\left(\frac{n}{b}\right) + \Theta(n^p \log^q n) & \text{if } n > 1 \end{cases}$$

recursion

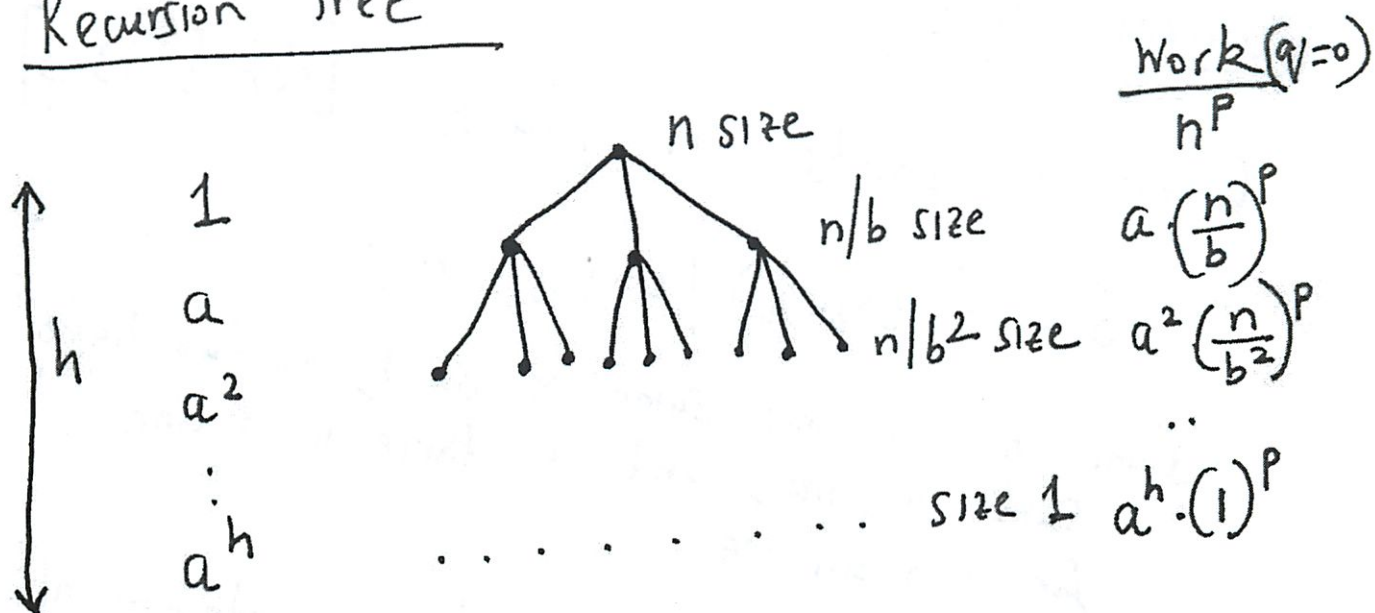
work for divide & combine

Then

$$T(n) = \begin{cases} \Theta(n^p \log^q n) & \text{if } p > \log_b a \\ \Theta(n^p \log^{q+1} n) & \text{if } p = \log_b a \\ \Theta(n^{\log_b a}) & \text{if } p < \log_b a \end{cases}$$



(3)

Recursion TreeSketch of argument

$$n = b^h, \quad a = b^{\log_b a}, \quad a^h = (b^{\log_b a})^h = (b^h)^{\log_b a} = n^{\log_b a}$$

ROOT  $n^P$ 

$a \left(\frac{n}{b}\right)^P$

$a^2 \left(\frac{n}{b^2}\right)^P$

 $\vdots$ LEAVES  $n^{\log_b a}$ 

if  $p > \log_b a$  ROOT dominates  
 $\Theta(n^P)$

if  $p = \log_b a$  all levels equal work  
 $\Theta(n^P \cdot \log n)$   $\log_b n = h$

if  $p < \log_b a$  LEAVES dominate  
 $\Theta(n^{\log_b a})$

(4)

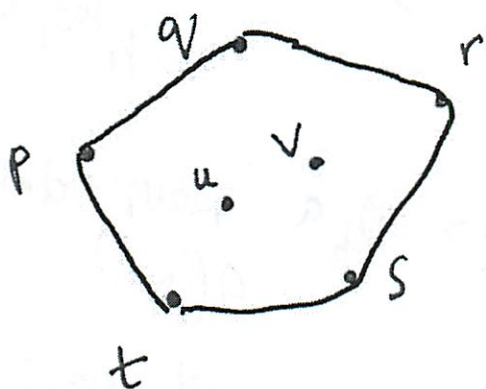
## Convex Hull

Given  $n$  points in plane [Ref § 33.3]

$$S = \{ (x_i, y_i) \mid i = 1, 2, \dots, n \}$$

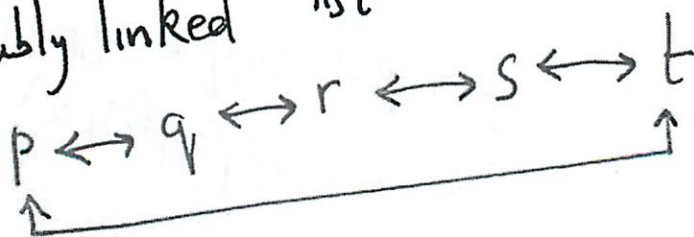
assume no two have same  $x$  coord, no two have same  $y$  coord, and no three in a line for convenience

Convex Hull: smallest polygon containing all points in  $S$   
 $CH(S)$



If points are nails, then  $CH(S)$  is shape of rubber band around all the nails

$CH(S)$  represented by the sequence of points on the boundary in order clockwise as doubly linked list



# D&C for Convex Hull

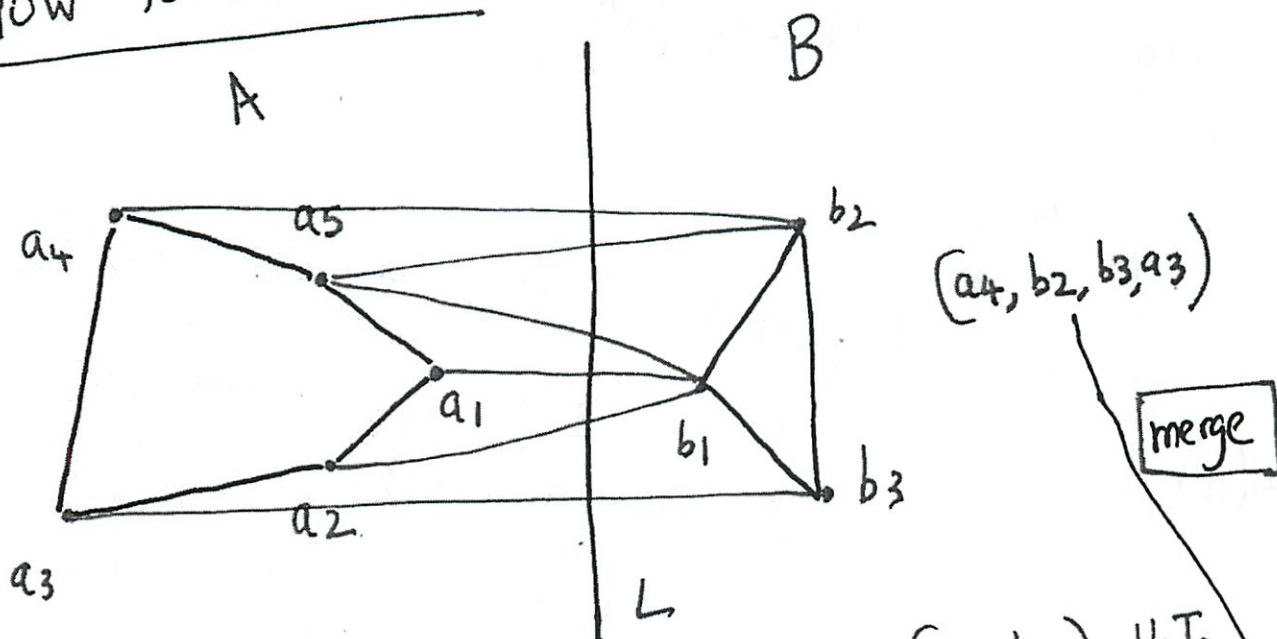
(6)

Sort points by x coord (once & for all,  $O(n \log n)$ )

For input set  $S$  of points:

- Divide into left-half  $A$  & right half  $B$
- by x coords
- Compute  $CH(A)$  &  $CH(B)$
- Combine  $CH$ 's of two halves (merge step)

## HOW TO MERGE?



Find upper tangent

Find lower tangent

Cut & paste in time  $\theta(n)$

First Link  $a_i$  to  $b_j$ , go down  $b$  list till you see  $b_m$  and link  $b_m$  to  $a_k$   
Continue along the  $a$  list until you return to  $a_i$

$(a_i, b_j)$  U.T.  
 $(a_k, b_m)$  L.T.  
 $(a_4, b_2)$  U.T.  
 $(a_3, b_3)$  L.T.  
 $(a_1, a_2, a_3, a_4, a_5)$   $(b_1, b_2, b_3)$



# FINDING TANGENTS

7

Assume  $a_1$  maximizes  $x$  within  $CH(A)$  ( $a_1, a_2, \dots, a_p$ )  
 $b_1$  minimizes  $x$  within  $CH(B)$  ( $b_1, b_2, \dots, b_q$ )

$L$  is the vertical line separating  $A$  &  $B$

Define  $y(i, j)$  as  $y$ -coordinate of pt of intersection between  $L$  & segment  $(a_i, b_j)$

CLAIM:  $(a_i, b_j)$  is upper tangent iff it maximizes  $y(i, j)$

Proof: Exercise.

Algorithm: Obvious  $O(n^2)$  algorithm looks at all  $a_i, b_j$  pairs

$\theta(n)$  [  $i = 1$   
 $j = 1$   
 while ( $y(i, j+1) > y(i, j)$  or  $y(i-1, j) > y(i, j)$ ):  
   if  $y(i, j+1) > y(i, j)$ : move right finger  $\uparrow$   
      $j = j+1 \pmod{q}$   
   else:  $i = i-1 \pmod{p}$  move left finger  $\uparrow$   
 return  $(a_i, b_j)$  as upper tangent

Similarly for lower tangent

$T(n) = 2T(\frac{n}{2}) + \theta(n)$  Master Theorem gives  $\theta(n \log n)$



# Median Finding

[Ref: § 9.3]

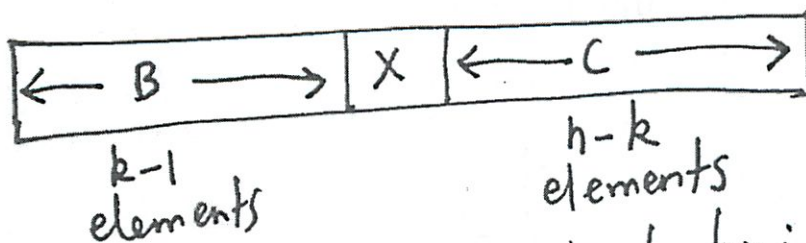
Given set of  $n$  numbers, define  $\text{rank}(x)$  as number of numbers in the set that are  $\leq x$

Find element of rank  $\lfloor \frac{n+1}{2} \rfloor$  : lower median  
(or element of rank  $i$ )  $\lceil \frac{n+1}{2} \rceil$  : upper median

Clearly sorting works in time  $\Theta(n \log n)$   
Can we do better?

Select( $S, i$ )

- Pick  $x \in S$  (cleverly) ← see next pg
- Compute  $k = \text{rank}(x)$
- $B = \{y \in S \mid y < x\}$
- $C = \{y \in S \mid y > x\}$



- if  $k = i$ : return  $x$  else if  $k > i$ : return Select( $B, i$ )  
else if  $k < i$ : return Select( $C, i - k$ )



## Solving the Recurrence

Master theorem does not apply

Prove  $T(n) \leq c \cdot n$  by induction, for  
some large enough  $c$

INTUITION:  
 $\frac{n}{5} + \frac{7n}{10} < n$

• True for  $n \leq 140$  by choosing large  $c$

•  $T(n) \leq c \cdot \lceil n/5 \rceil + c \left( \frac{7n}{10} + 6 \right) + a \cdot n$   
 (a needs to be large enough to cover  $\theta(n)$  term)

$$\leq \frac{cn}{5} + c + \frac{7nc}{10} + 6c + an$$

$$= cn + \underbrace{\left( -\frac{cn}{10} + 7c + an \right)}_{\text{if this is } \leq 0, \text{ we are done}}$$

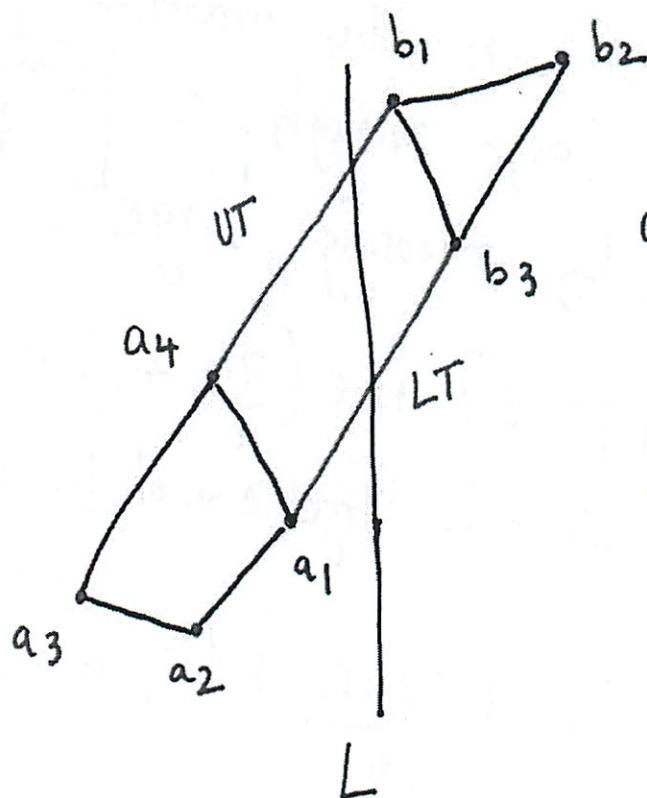
$$c \geq \frac{70c}{n} + 10a$$

ok for  $n \geq 140$  &  $c \geq 20a$  ☒

(A)

## COUNTER EXAMPLE

Why can't we pick the highest  $a_i$  and highest  $b_j$  to find upper tangent



$a_4, b_1$  is upper tangent

$a_1, b_3$  is lower tangent

$b_2$  higher than  $b_1$

$a_2$  lower than  $a_1$

Question: If  $a_i, b_j$  is an upper tangent then is one of these points the one with the highest  $y$  value?



You thought you could avoid FFT — nope!

Seti @ Home looking for constant low freq

## Today Fast Fourier Transform (FFT)

- why?
- polynomial multiplication
- representing <sup>— signal to impulse</sup> polynomials
- poly mult via change of representation
- how to change representation (FFT)

## FFT

- recursive idea
- complex roots of unity
- the recursive algorithm
- another view of the inverse algorithm

②

Adding polynomials of degree  $\leq n$

$$\begin{array}{r} 7x^2 - 10x + 9 \\ + \quad \quad \quad 4x - 5 \\ \hline 7x^2 - 6x + 4 \end{array}$$

takes  $O(n)$  time

Multiplying polynomials

$$\begin{array}{r} 7x^2 - 10x + 9 \\ \quad \quad \quad 4x - 5 \\ \hline -35x^2 + 50x - 45 \\ 28x^3 - 40x^2 + 36 \\ \hline 28x^3 - 75x^2 + 86x - 45 \end{array}$$

$O(n^2)$  time

Can we improve?

note:  $\deg(C) = \deg(A) + \deg(B)$

3

Hint: This looks awful like convolution!

## Representing Polynomials

1. Coefficient representation

add  $O(n)$ , multiply  $O(n^2)$

2. Represent w/ values at  $n$  distinct inputs

"Value Representation"

Crucial fact:  $\{(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})\}$

Linear algebra point

input, output pairs  
 $n$  of them

bijective mapping to  
set of deg  $n$   
polynomials

bi

degree  $n$  poly.

such that  $y_i = f(x_i) \quad \forall i = 0, \dots, n-1$

4

basically 2 pts determine a line

This is generalization to higher degree

Can <sup>add</sup> multiply this representation in  $O(n)$  time

add  $O(n)$       mul  $O(n)$

~~Given~~

Fix  $x_0 \dots x_n$  (distinct)

A represented by  $y_0 \dots y_{n-1}$  (ie  $y_i = A(x_i)$ )

B represented by  $y'_0 \dots y'_{n-1}$

A+B

$\forall x_i$

$$\begin{array}{c} A(x) + B(x) \\ y_i \quad y'_i \end{array} = (A+B)(x)$$

So

$$(y_0 + y'_0, y_1 + y'_1, \dots, y_{n-1} + y'_{n-1})$$



5

Same thing for multiplication

$$\forall x \quad A(x) \cdot B(x) = (A \cdot B)(x)$$

$$A \cdot B \quad (y_0 + y_0' \dots y_{n-1} + y_{n-1}')$$

problem:

degrees <sup>could be</sup> larger than  $n$

So need  $2n$  values

Must use a redundant representation

Need  $A, B$  evaluated at  $2n$  distinct points

We can still do this in  $O(n)$  time

~~3.2.2~~

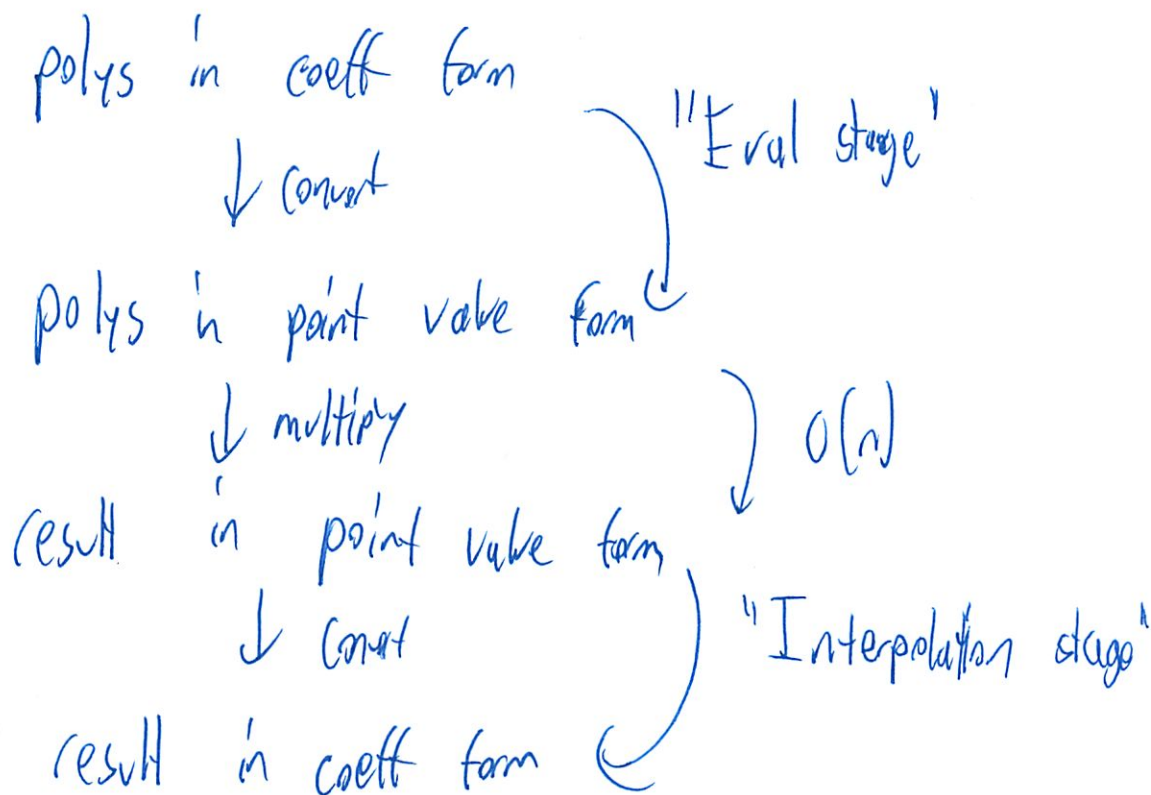
How to convert forms?

type of relation



(6)

Our plan



Eval stage

Nieve

Need to eval at  $n$  distinct locations

But eval at each pt is  $n$   
w/  $n$  pts  $) O(n^2)$  time

But this gets us nothing!

(7)

Main insight: Can pick any distinct points

So simultaneous computation will be overall cheaper

We reuse a lot of computation

↑ the real value of FFT!

So how can this insight help us

Choose  $n$  points  $\rightarrow \oplus$  and  $\ominus$  pairs

$$\pm x_0, \pm x_1, \dots, \pm x_{\frac{n}{2}-1}$$

$$A(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \dots$$

$$A(-x_i) = a_0 - a_1 x_i + a_2 x_i^2 + \dots$$

↑ almost the same coefficients

even same

odd  $\ominus$  of ~~even~~ other (mul  $-1$ )

(This is such a better way to explain)

8

Define  $A_{\text{even}}(z) = a_0 + a_2 z + a_4 z^2 + a_6 z^3$

$A_{\text{odd}}(z) = a_1 + a_3 z + a_5 z^2 + a_7 z^3$

Rewrite:

2 problems  
of degree  
 $< n$

$A(x_i) = A_{\text{even}}(x_i^2) + x_i A_{\text{odd}}(x_i^2)$

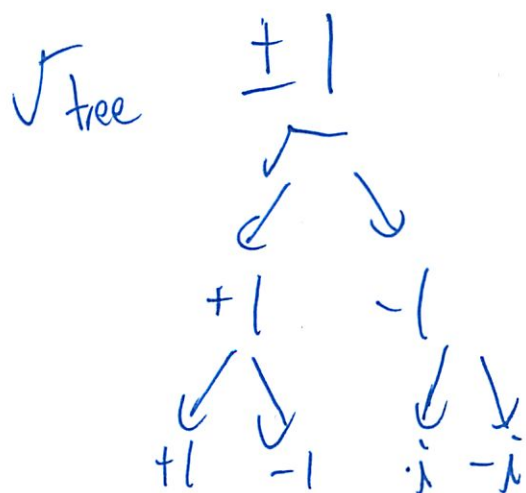
$A(-x_i) = A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2)$

↑ 2 problems of degree  $< \frac{n}{2}$

(Missed some discussion of complex #)

So can reuse results at 1 part of calculation

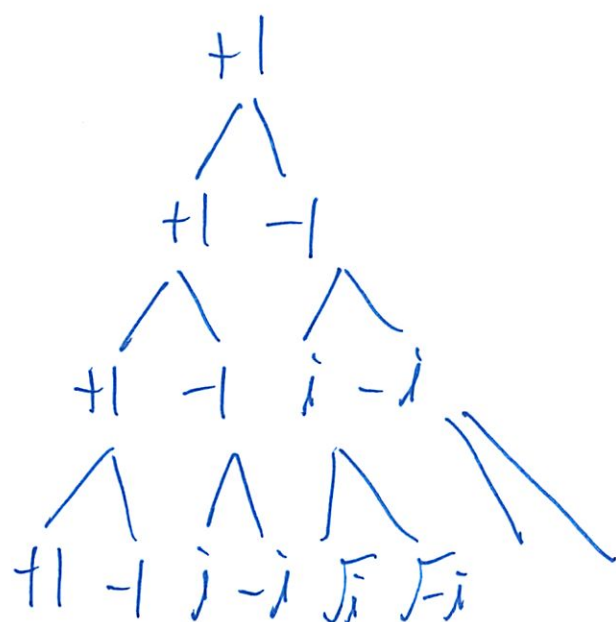
Complex Roots of unity





④

Recall



Better notation using  $e^{iu} = \cos(u) + i\sin(u)$

Solutions to  $w^n = 1$  are  $e^{2\pi i k/n}$

for  $k = 0, 1, \dots, n-1$   
 $i = \sqrt{-1}$

just call it  $\Theta(n)$

Good things to know

$$\underbrace{(e^{2\pi i/2})(e^{2\pi i/4})}_{-} = -w_n^{n/2+j}$$

1.  $n^{\text{th}}$  roots are  $(\pm)$ -paired

$$\text{ie. } w_n^j = -w_n^{n/2+j} \quad w_n^j = e^{\frac{2\pi i j (1/2+j)}{n}}$$

2. Squaring  $n^{\text{th}}$  roots gives exactly the  $n/2^{\text{th}}$  roots

$$(w_n^j)^2 = (e^{2\pi i j/n})^2 = e^{2\pi i j/(n/2)} = w_{n/2}^j$$

(10)

Recursive algorithm for FFT ( $n, n$ )  $\in \deg A$  is  $\leq \frac{n}{2}$   
 $w^{\log_2 n}$  power of 2

Runtime

- $O(1)$  - if  $n=1$  return  $A(1)$
- $O(1)$  - write  $A(x)$  as  $A(x) = A_{\text{even}}(x^2) + x A_{\text{odd}}(x^2)$
- $T(n/2)$  - FFT ( $A_{\text{even}}, n/2$ ) to eval  $A_{\text{even}}$  on powers  $w_{n/2}$
- $T(n/2)$  - FFT ( $A_{\text{odd}}, n/2$ ) " "  $A_{\text{odd}}$  " " "
- $O(1) \cdot n$  - Compute  $A$  at  $n$  powers of  $w_n$   
Via  $A(w_n^j) = A_{\text{even}}(w_n^{2j}) + w_n^j A_{\text{odd}}(w_n^{2j})$
- return  $A(w_n^0) A(w_n^1) \dots A(w_n^{n-1})$



$$T(n) = 2 T\left(\frac{n}{2}\right) + O(n)$$

$$= O(n \lg n)$$

much better than  $n^2$

II

## Interpolation step

Just computing FFT but w/ ~~multiplicative~~  
multiplicative inverse of roots of unity

This FFT is a special case of matrix multiply  
↳ a nice one

Can also look at inverse of matrix and see  
that it is related

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_n) \end{bmatrix} = \begin{matrix} M \\ \begin{bmatrix} 1 & x_0 & x_0^2 & x_0^3 & \dots \\ 1 & x_1 & x_1^2 & x_1^3 & \dots \\ \vdots & \vdots & \ddots & \ddots & \ddots \\ 1 & x_n & x_n^2 & x_n^3 & \dots \end{bmatrix} \end{matrix} \begin{matrix} \text{(Coeffs of } A) \\ \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \end{matrix}$$

evaluation  $\times M$

interpolation  $\times M^{-1}$

↑ Van der Monde  
matrix

When  $x_i$ s distinct, VdM matrices invertible

(17)

When  $n^{\text{th}}$  roots of unity, VDM satisfies

$$M_n(\omega_n) = \frac{1}{n} M_n(\omega_n^{-1})$$

$$M = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \dots \\ 1 & \omega_n^2 & (\omega_n^2)^2 & (\omega_n^2)^3 & \dots \\ \vdots & \vdots & (\omega_n^3)^2 & (\omega_n^3)^3 & \dots \\ 1 & \omega_n^{n-1} & (\omega_n^{n-1})^2 & (\omega_n^{n-1})^3 & \dots \end{pmatrix}$$

Powers of  
?  $n^{\text{th}}$   
root of unity

Sol to  $\omega^n = 1$

Are distinct  $x$ 's

← multiplicative inverse

$$M^{-1} = \begin{pmatrix} 1 & 1 & 1 & 1 & \dots \\ 1 & (\omega_n)^{-1} & (\omega_n^{-1})^2 & (\omega_n^{-1})^3 & \dots \\ 1 & (\omega_n^{-1})^2 & ((\omega_n^{-1})^2)^2 & ((\omega_n^{-1})^2)^3 & \dots \\ \vdots & \vdots & (\omega_n^{-1})^3 & (\omega_n^{-1})^4 & \dots \\ 1 & (\omega_n^{-1})^{n-1} & (\omega_n^{-1})^{n-2} & (\omega_n^{-1})^{n-3} & \dots \end{pmatrix}$$

beauty of using roots of unity  
don't need to write them all out



(13)

But standard Matrix multiple  $n^2$ . something

So use alg we have here

These specific matrix multiplies can be done much faster

---

1963 Tukey + Darwin thought this was easy

But used in 1930s

Also 1800s Gauss

## Lecture 3

### Fast Fourier Transform (FFT)

- why?
- polynomial multiplication
- representing polynomials
- polynomial mult via change of representation
- how to change representation  
i.e. FFT
  - A recursion idea
  - Complex roots of unity
  - A recursive algorithm
  - Another view via linear algebra
  - Inverse FFT

### Why?

- poly mult
- integer mult
- signal processing
  - speech, images ... e.g. search for aliens: Seti@home  
constant loud tones
- large data sets
- coding, compression ...
- Quantum computation

## Multiply polynomials

- Adding <sup>subtracting</sup> polys is easy -  $O(n)$  for degree  $\leq n$  polys

$$\begin{array}{r} \text{e.g. } 7x^2 - 10x + 9 \\ + \quad \quad 4x - 5 \\ \hline 7x^2 - 6x + 4 \end{array}$$

- What about multiplying?

Usual method:

$$\begin{array}{r} 7x^2 - 10x + 9 = A(x) \\ \times \quad 4x - 5 = B(x) \quad O(n^2) \text{ for degree } \leq n \text{ polys} \\ \hline -35x^2 + 50x - 45 \\ 28x^3 - 40x^2 + 36x \\ \hline 28x^3 - 75x^2 + 86x - 45 = C(x) \end{array}$$

↑  
degree of  $C$  = degree of  $A$  + degree of  $B$

Coefficient of  $x^k \rightarrow C_k = a_0 b_k + a_1 b_{k-1} + \dots + a_k b_0$   
 $= \sum_{i=0}^k a_i b_{k-i}$

Can we do better?

} similar computations  
in signal processing -  
response of signal  
to an "impulse"

Representing Polynomials

(1) via coefficients

$$A(x) = 9 - 10x + 7x^2 \longrightarrow (9, -10, 7)$$

$\uparrow \quad \quad \uparrow \quad \quad \uparrow$   
 $a_0 \quad a_1 \quad a_2$

add in  $O(n)$  time  
 multiply in  $O(n^2)$  time

(2) via values at  $n$  points  
 $\uparrow$   
 when degree of poly  $< n$

Why can you do this?

Crucial Fact: a degree  $d$  poly characterized by values  
 at any  $d+1$  distinct pts

i.e. given pts  $x_0, \dots, x_d$  (distinct)

$\{(x_0, y_0), (x_1, y_1), \dots, (x_d, y_d)\}$

for each possible  $(y_0, \dots, y_d)$

there is exactly one degree  $d$   
 poly  $f$  st.

$$f(x_0) = y_0$$

$$f(x_1) = y_1$$

$$f(x_d) = y_d$$

$\uparrow$  Bijection  
 $\downarrow$   
 degree  $d$  poly  
 $f$

(generalizes "2 pts determine a line")

Nice property of this representation:  
 add + multiply in  $O(n)$  time!



How to add: A represented by  $(y_0 \dots y_n)$ , B represented by  $(y'_0 \dots y'_n)$   
(Evaluated at same  $x_i$ 's)

$$\begin{array}{r} (y_0 \dots y_n) \\ + (y'_0 \dots y'_n) \\ \hline \end{array}$$

A+B  $(y_0 + y'_0, y_1 + y'_1, \dots, y_n + y'_n)$

since  $(A+B)(x) = A(x) + B(x)$

$O(n)$  time!

How to multiply:

well,  $(A \cdot B)(x) = A(x) \cdot B(x)$   
but  $\begin{array}{ccc} \uparrow & \uparrow & \uparrow \\ \deg \leq 2n & \deg \leq n & \deg \leq n \end{array}$

need  $A, B$  evaluated at  $2n+1$  distinct points

so A represented by  $(y_0 \dots y_n, y_{n+1} \dots y_{2n+1})$   
B " "  $(y'_0 \dots y'_n, y'_{n+1} \dots y'_{2n+1})$

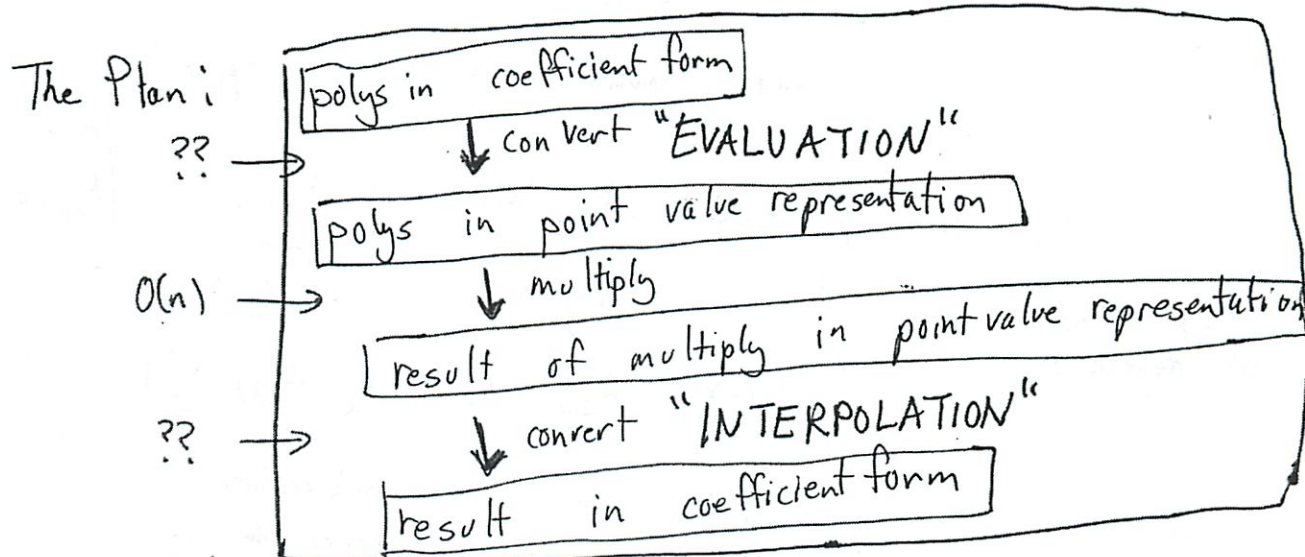
A · B " "  $(y_0 y'_0, y_1 y'_1, \dots, y_{2n+1} y'_{2n+1})$

$O(n)$  time!

But my polys are in coefficient form!

The main idea of today's lecture:

Can convert representations & do mult in easiest representation



Main issue:  
how long does conversion take?

naive method for evaluation:

evaluate poly at  $n$  points

- $\deg < n$  poly evaluation uses  $O(n)$  time
- $O(n^2)$  total time

↑  
how can we improve this?

MAIN INSIGHT • Choose  $n$  points carefully  
• st. simultaneous evaluation is cheaper!

An idea:

Choose the  $n$  points as  $\pm x_0, \pm x_1, \dots, \pm x_{\frac{n-1}{2}}$

$$A(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + \dots$$

$$A(-x_i) = a_0 - a_1 x_i + a_2 x_i^2 + \dots$$

- even powers are the same!
- odd powers are exact opposites

$$\begin{aligned} A_{\text{even}} &= a_0 + a_2 x + a_4 x^2 \\ A_{\text{odd}} &= a_1 + a_3 x + a_5 x^2 \end{aligned}$$

in fact

$$\begin{aligned} A(x_i) &= \underbrace{A_{\text{even}}(x_i^2)}_{\text{deg } n} + x_i \underbrace{A_{\text{odd}}(x_i^2)}_{\text{deg } n} \\ A(-x_i) &= A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2) \end{aligned}$$

2 problems of degree  $\leq n/2$

- reduce to 2 sub problems each of degree  $\leq n/2$

Problem: how to continue recursion?

need to choose  $x_0^2, \dots, x_{\frac{n-1}{2}}^2$   
so that they are plus-minus pairs,  
but they are all positive!  
unless we use complex #'s!!!

# Complex Roots of Unity

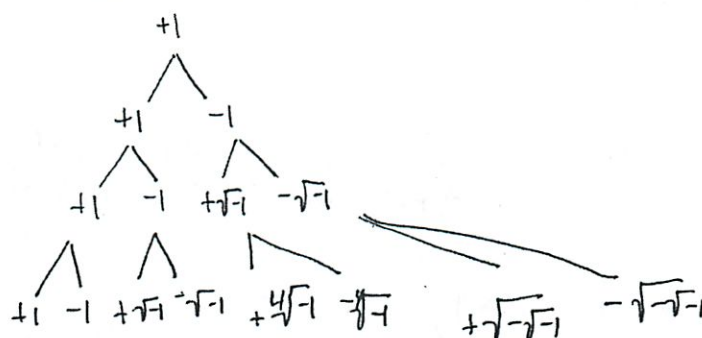
Solutions to

$$w = 1$$

$$w^2 = 1$$

$$w^4 = 1$$

$$w^8 = 1$$



Better notation:

use  $e^{iu} = \cos(u) + i \sin(u)$

solutions to  $w^n = 1$  are  $e^{2\pi i k/n}$  for  $k=0, 1, \dots, n-1$   
 $i = \sqrt{-1}$

Note:

1)  $n^{\text{th}}$  roots are  $(\pm)$ -paired

ie.  $w_n^{n/2+j} = -w_n^j$

Since  $e^{\frac{2\pi i (\frac{n}{2}+j)}{n}} = \underbrace{(e^{2\pi i /2})}_{-1} e^{2\pi i j/n}$

2) squaring  $n^{\text{th}}$  roots gives exactly the  $n/2^{\text{th}}$  roots

since  $(w_n^j)^2 = e^{2 \cdot 2\pi i j/n} = e^{2\pi i j/(n/2)} = w_{n/2}^j$

For the rest of this lecture,  
let  $w_n = e^{2\pi i /n}$



# Recursive Algorithm :

time  $\text{FFT}(A, n)$   $\left\{ \begin{array}{l} \text{degree} \leq n \text{ (without loss of generality, } n \text{ is power of 2)} \\ \text{poly in coeff representation} \end{array} \right.$

- if  $n=1$ , return  $A(1)$
- write  $A(x)$  as  $A(x) = A_{\text{even}}(x^2) + x \cdot A_{\text{odd}}(x^2)$

$T(n/2)$  • call  $\text{FFT}(A_{\text{even}}, n/2)$  to evaluate  $A_{\text{even}}$  at all powers of  $w_{n/2}$

$T(n/2)$  • call  $\text{FFT}(A_{\text{odd}}, n/2)$  to evaluate  $A_{\text{odd}}$  at all powers of  $w_{n/2}$

- compute  $A$  at  $n$  powers of  $w_n$  via

$n \cdot O(1)$

$$A(w_n^j) = A_{\text{even}}(w_n^{2j}) + w_n^j A_{\text{odd}}(w_n^{2j})$$

- return  $A(w_n^0) A(w_n^1) \dots A(w_n^{n-1})$

$$T(n) = 2T(n/2) + c \cdot n \Rightarrow T(n) = O(n \log n)$$

How do we convert back to coefficient representation?

[Polynomial Interpolation]

Amazing "coincidence": use FFT but with  $w_n^{-1}$  instead of  $w_n$

To see this:

Another view of Evaluation + Interpolation:

$$\begin{array}{c}
 \begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \dots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} \\
 \uparrow \qquad \qquad \qquad \uparrow \qquad \qquad \qquad \uparrow \\
 \text{value representation} \quad \text{Vandermonde Matrix } M \quad \text{coefficient representation}
 \end{array}$$

Nice property of Vandermonde matrix:

if  $x_i$ 's distinct,  $M$  invertible

Evaluation: multiply by  $M$  (FFT with  $w_n$ )

Interpolation: multiply by  $M^{-1}$  (FFT with  $w_n^{-1}$ )

Another nice property of  $M$  when  $(x_0 \dots x_{n-1}) = (1 w_n \dots w_n^{(n-1)})$ :

$$M = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & w_n & w_n^2 & \dots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & \dots & w_n^{2(n-1)} \\ 1 & w_n^3 & w_n^6 & \dots & w_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \dots & w_n^{(n-1)^2} \end{pmatrix}$$

$$M^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & 1 & \dots \\ & \omega_n^{-1} & (\omega_n^{-1})^2 & (\omega_n^{-1})^3 & \dots \\ & (\omega_n^{-1})^2 & (\omega_n^{-1})^4 & (\omega_n^{-1})^6 & \dots \\ & (\omega_n^{-1})^3 & (\omega_n^{-1})^6 & (\omega_n^{-1})^9 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \\ 1 & (\omega_n^{-1})^{n-1} & (\omega_n^{-1})^{2(n-1)} & (\omega_n^{-1})^{3(n-1)} & \dots \end{pmatrix}$$

i.e.  $M_n(\omega_n) = \frac{1}{n} M_n(\omega_n^{-1})$

So interpolation is as easy as evaluation!

### Comment

- matrix algebra shows relationship between FFT & FFT<sup>-1</sup>

• also gives algorithm

- is matrix mult faster than recursive formulation?
- can do fast matrix mult for Vandermonde using recursive formulation!

## FFT : a history

- 1963 Cooley + Tukey
  - ↑ programmer from IBM
  - ↑ mathematician from Princeton
- John Tukey explains to IBM's Dick Garwin
- Cooley implements
- Cooley wants to publish to avoid patent
  - Tukey thinks it must be known & too easy  
(algorithms not thought of as exciting back then?)

But actually ...

- 1930 British engineers

But even before ...

- 1800s Gauss on interpolation  
(in Latin)

So Tukey was right!



## 6.046 Recitation 2

9/18/14

- OH TR 7-9 pm

13-4101

plus M<sup>or</sup> W before p-set are

- PSI out

are 9/25

- No recitation next fri 9/21

Student holiday

- Mailing List

TAs assigned each day

- No Piazza

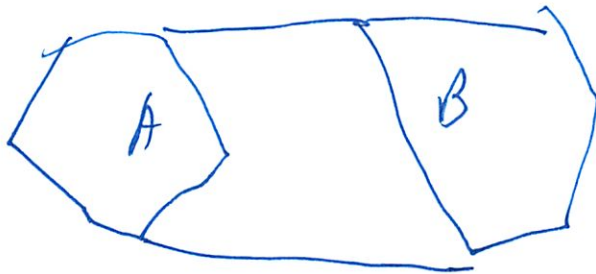
- Formal posts

②

Divide + Conquer

Convex hull

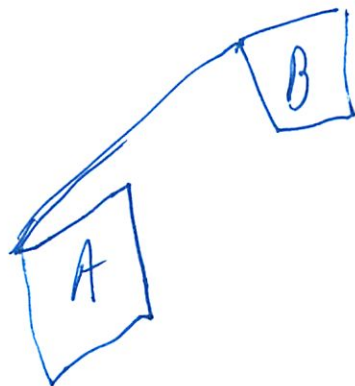
Want outside boundary



Do A, B individually

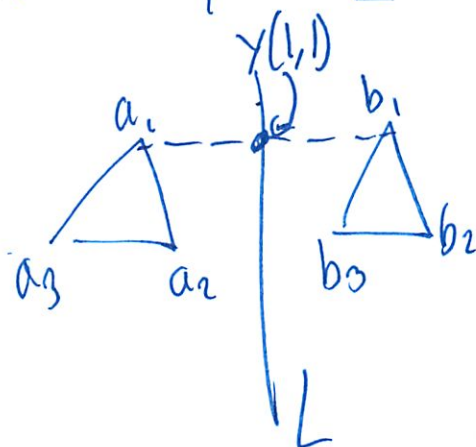
Then find upper, lower tangent

Note: not highest point



③

Can divide sets by  $L$



Claim:  $(a_i, b_j)$  is upper tangent iff  
it maximizes  $y(i, j)$  where  $y(i, j)$   
is the intersection w/ line  $L$

~~Proof~~

Proof

show

contradiction?

if upper tangent  $\rightarrow$  max

What can we say about line intersecting  
complex polygon?

9

→ Line  $L$  intersects complex polygon  $(S)$

Let  $L_0$  be a segment inside  $S$

$$L_0 \text{ is } L \cap S$$

Suppose  $\exists (a_i, b_j)$  s.t.  $y(i', j') > y(i, j)$

$$y(i', j') \notin L_0$$

↑ if there is a point that is higher, it must be higher than  $L_0$

Remember  $L_0$  is  $L \cap S$

Must also be intersection

$$y(i', j') \in L$$

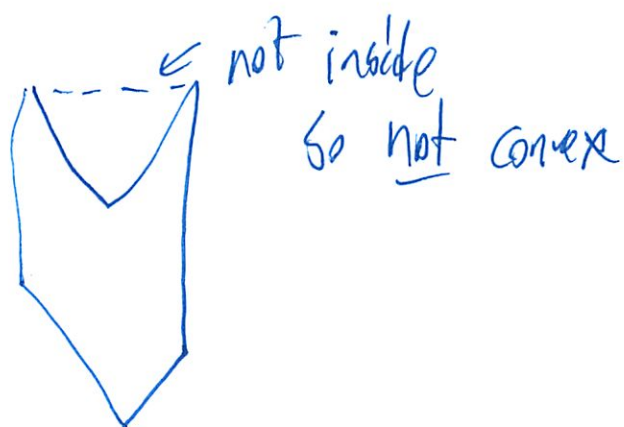
~~Since~~ Since when connect 2 pts, it must be inside  $S$

↳ Since 2 points on shapes otherwise we would not be the best complex hull



⑤

Def A planar polygon is convex if it contains all the line segments connecting any pair of its points



$$y(i', \text{~~some~~};') \in L \cap S = L_0$$

↳ Contradiction

Can't be any other pair  $a_i, b_j$  such that  $y$  is bigger because otherwise it won't be a convex hull anymore

Q

if max  $\rightarrow$  upper tangent

$(a_i, b_i)$  which max  $y(i, j)$

if not upper tangent  $\rightarrow$  must be interior of CH

So not boundary

~~False~~  $\rightarrow$  must be sides that cross above it  
contradiction

---

Median of medians

0 0 0 0

0 0 0

$O(n)$



0 0

0 0

But why 5?

7

What if we did 7?

Also  $O(n)$

$$\boxed{\text{For } 5} \quad 3\left(\left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{n}{5} \right\rceil - 2\right) = \# \text{ of groups}$$

$\uparrow$  half of 5, round up

$\uparrow$  what groups not include?  
~~the~~ the not the full amt group

$$4\left(\left\lceil \frac{n}{2} \right\rceil \left\lceil \frac{n}{7} \right\rceil - 2\right) = \# \text{ of groups}$$

$\uparrow$  half of 7, round up

$$\text{Then } T(n) \leq \begin{cases} O(1) & n < 70 \\ T\left(\left\lceil \frac{n}{7} \right\rceil\right) + T\left(\frac{5n}{7} + 8\right) + O(n) & n \geq 70 \end{cases}$$

$\uparrow$  median of medians

Same or similar proof in class

Induction claim for previous one

if know  $T(n) \leq c(n)$

$$T(n) \leq c\left(\left\lceil \frac{n}{7} \right\rceil\right) + c\left(\frac{5n}{7} + 8\right) + cn \leq \text{cont}$$

⑧

$$\leq cn - c \left( \frac{n}{7} - g \right) + an$$

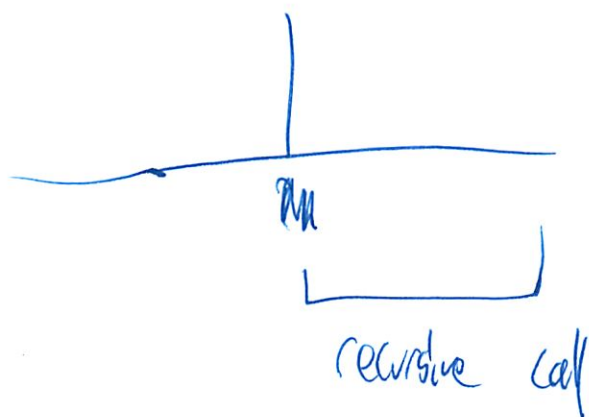
So

$$c \geq \frac{7a}{1 - \frac{63}{7}}$$

$$c \geq 70a$$

by induction  $\rightarrow$  also linear time

point is to choose ~~the~~ pivot



The claim guarantees median of median  
is somewhat in ~~the~~ middle

$\uparrow$  a claim about size of subproblem  
shrinking at known fraction



(9)

Choosing 3:  $2(\lceil \frac{1}{2} \lceil \frac{2}{3} \rceil \rceil - 2)$

$$T(n) \leq T(\lceil \frac{n}{3} \rceil) + T(\frac{2n}{3} + 4) + O(n)$$

$$T(n) = T(\alpha n) + T(\beta n) + O(n)$$

$\alpha + \beta < 1$  for linear

Sum of subproblems better be smaller

Don't want to recurse and not  
do anything!  
(shrink)

## Recitation 2: Divide-and-Conquer

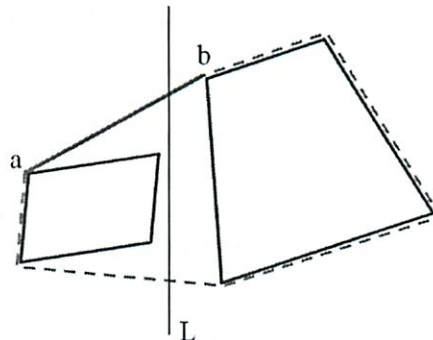
### 1 Convex hull and highest points

- Definitions:

- A planar polygon is *convex* if it contains all the line segments connecting any pair of its points. In other words,  $P \subset \mathbb{R}^2$  is a polygon if for any  $x, y \in P$ ,  $tx + (1 - t)y \in P$  for  $t \in [0, 1]$ .
- The *convex hull* of a set  $S$  of points, denoted  $\text{CH}(S)$ , is the smallest convex polygon for which each point in  $S$  is either on the boundary or in the interior.

See class notes for definitions on the upper/lower tangents, intercepts with the dividing line, and details on the divide-and-conquer method for finding  $\text{CH}(S)$ .

- The upper tangent is not always formed by connecting the two *highest points*, i.e. points with the largest  $y$ -coordinates (counterexample shown in class).
- Also, it is not true that one of the two points forming the upper tangent is the highest point in the subset. As a counterexample, consider the figure below, where the blue lines outline the convex hull. Points  $a$  and  $b$  form the upper tangent, marked with a solid blue line, but neither  $a$  or  $b$  is the highest point in their respective subsets.



- Claim: Consider all pairs of points  $a_i \in A, b_j \in B$  in the left- and right-subsets  $A$  and  $B$ , respectively. Let  $p(i, j) = (x(i, j), y(i, j))$  denote the intersection between the line segment  $(a_i, b_j)$  and the dividing line  $L$ . Then  $(a_i, b_j)$  is the upper tangent of  $\text{CH}(S)$  if and only if it maximizes  $y(i, j)$ .

*Proof.* First note that, by convexity, the intersection of  $L$  and any convex polygon is either empty, or a continuous line segment (including the case of a single point).

Let  $(a_i, b_j)$  be the upper tangent, and let  $L_0 = L \cap \text{CH}(S)$  be the line segment at which  $L$  intersects  $\text{CH}(S)$ . By definition of the upper tangent, all points in  $L_0$  have  $y$ -coordinates that are no greater than  $y(i, j)$ . Now suppose there exists a segment  $(a_{i'}, b_{j'})$  such that  $y(i', j') > y(i, j)$ , where  $i \neq i'$  or  $j \neq j'$ . By convexity, since  $p(i', j')$  is on the segment connecting  $a_{i'}$  and  $b_{j'}$ , we have  $p(i', j') \in \text{CH}(S)$ . Therefore,  $p(i', j') \in L \cap \text{CH}(S)$ , contradicting the fact that  $p(i', j') \notin L_0$  due to  $y(i', j') > y(i, j)$ .

Conversely, let  $(a_i, b_j)$  be the segment that maximizes  $y(i, j)$ . If it is not the upper tangent, then  $p(i, j)$  must be an interior point on  $L_0 = L \cap \text{CH}(S)$ . Therefore, the point at which the upper tangent intersects  $L$  has a greater  $y$ -coordinate than  $p(i, j)$ , a contradiction.  $\square$

## 2 Selection in worse-case linear time (Sec. 9.3)

- (See CLRS Section 9.3 for figures and details.)
- Dividing into groups of 7 elements:

The number of elements that are greater (or smaller) than the median-of-medians is at least

$$4 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{7} \right\rceil \right\rceil - 2 \right),$$

which is bounded below by  $\frac{2n}{7} - 8$ . Therefore, the complexity is

$$T(n) \leq \begin{cases} O(1), & n < 70, \\ T(\lceil \frac{n}{7} \rceil) + T(\frac{5n}{7} + 8) + O(n), & n \geq 70. \end{cases} \quad (1)$$

We show that  $T(n) = O(n)$  by substitution. First, choose a positive scalar  $c$  large enough such that  $T(n) \leq cn$  for all  $n < 70$ . Also, choose a positive scalar  $a$  such that the  $O(n)$  term above is upper-bounded by  $an$ . Now given  $n \geq 70$ , suppose  $T(k) \leq ck$  for all  $k < n$ . Then, by induction,

$$\begin{aligned} T(n) &\leq c \left\lceil \frac{n}{7} \right\rceil + c \left( \frac{5n}{7} + 8 \right) + an \\ &\leq c \left( \frac{n}{7} + 1 + \frac{5n}{7} + 8 \right) + an \\ &= cn - c \left( \frac{n}{7} - 9 \right) + an, \end{aligned}$$

which is no greater than  $cn$  if

$$c \geq \frac{an}{\frac{n}{7} - 9} = \frac{7a}{1 - 63/n}.$$

Note that the term on the right-hand side decreases with increasing  $n$ . For  $n = 70$ , we have  $\frac{7a}{1 - 63/n} = 70a$ . Therefore, by choosing  $c \geq 70a$ , we can ensure that the induction hypothesis holds for  $n$ , i.e.,  $T(n) \leq cn$ .

- Dividing into groups of 3 elements: The number of elements that are greater (or smaller) than the median-of-medians, is

$$2 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{3} \right\rceil \right\rceil - 2 \right),$$

which is bounded below by  $\frac{n}{3} - 4$ . Therefore, the complexity is

$$T(n) \leq \begin{cases} O(1), & n \text{ small,} \\ T(\lceil \frac{n}{3} \rceil) + T(\frac{2n}{3} + 4) + O(n), & \text{otherwise.} \end{cases} \quad (2)$$

The failure of substitution makes it clear that  $T(n)$  is not linear time. (Indeed, 5 is the smallest odd number for which the method works.)



## Supplementary Material: Fast Fourier Transform (Ch. 30)

- Representing polynomials of degree-bound  $n$  (i.e. degree strictly smaller than  $n$ ):

– Coefficient representation:  $A(x) = \sum_{j=0}^{n-1} a_j x^j$

– Point-value representation:  $\{(x_0, y_0), \dots, (x_{n-1}, y_{n-1})\}$

*Example.*  $f(x) = x^2 + 2x + 3$

(Evaluation) Choose points  $0, 1, -1$ .

$\Rightarrow f(x)$  can be represented by  $\{(0, f(0)), (1, f(1)), (-1, f(-1))\} = \{(0, 3), (1, 6), (-1, 2)\}$

$\Rightarrow O(n^2)$  time

(Interpolation)  $f(x) = ax(x-1) + bx(x+1) + c(x-1)(x+1)$

$\Rightarrow f(0) = -c = 3, f(1) = 2b = 6, f(-1) = 2a = 2$

$\Rightarrow O(n^2)$  time

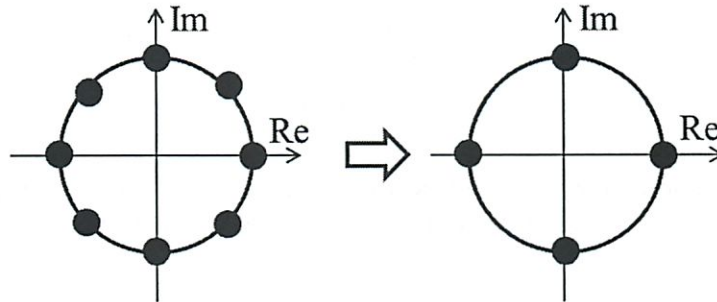
The *fast Fourier transform (FFT)* improves the time complexity of evaluation/interpolation to  $O(n \log n)$ , using a divide-and-conquer approach.

- Complex roots of unity:

$$\{\omega_n^k \mid k = 0, \dots, n-1\} := \{\omega \mid \omega^n = 1\} = \{e^{i\frac{2\pi k}{n}} \mid k = 0, 1, \dots, n-1\}$$

*Example.*  $n = 8$

$$\{\omega_8^k \mid k = 0, \dots, 7\} \quad \{(\omega_8^k)^2 \mid k = 0, \dots, 7\} = \{\omega_4^k \mid k = 0, 1, 2, 3\}$$



- Representing polynomials at complex roots of unity: (Assuming  $n = 2^m$ )

Representing  $A(x) = \sum_{j=0}^{n-1} a_j x^j$  with  $\{(\omega_n^k, y_k) \mid k = 0, \dots, n-1\}$  requires the following calculations for  $k = 1, \dots, n-1$ :

$$\begin{aligned} y_k &= A(\omega_n^k) = \sum_{j=0}^{n-1} a_j (\omega_n^k)^j = a_0 + a_1 \omega_n^k + a_2 (\omega_n^k)^2 + \dots + a_{n-1} (\omega_n^k)^{n-1} \\ &= a_0 + a_2 (\omega_n^k)^2 + a_4 (\omega_n^k)^4 + \dots + a_{n-2} (\omega_n^k)^{n-2} \\ &\quad + \omega_n^k (a_1 + a_3 (\omega_n^k)^2 + a_5 (\omega_n^k)^4 + \dots + a_{n-1} (\omega_n^k)^{n-2}) \\ &=: A_{\text{even}}((\omega_n^k)^2) + \omega_n^k A_{\text{odd}}((\omega_n^k)^2) \end{aligned}$$

where  $A_{\text{even}}$  and  $A_{\text{odd}}$  are polynomials of degree-bound  $\frac{n}{2}$ . Therefore, what we need is to compute values of  $A_{\text{even}}$  and  $A_{\text{odd}}$  at the set of points

$$\{(\omega_n^k)^2 \mid k = 0, \dots, n-1\} = \{\omega_{\frac{n}{2}}^k \mid k = 0, \dots, \frac{n}{2} - 1\}.$$



Moreover,  $y_{k+\frac{n}{2}} = A_{\text{even}}((\omega_n^{k+\frac{n}{2}})^2) + \omega_n^{k+\frac{n}{2}} A_{\text{odd}}((\omega_n^{k+\frac{n}{2}})^2) = A_{\text{even}}((\omega_n^k)^2) - \omega_n^k A_{\text{odd}}((\omega_n^k)^2)$ . Therefore, computation results for point  $\omega_{\frac{n}{2}}^k = (\omega_n^k)^2 = (\omega_n^{k+\frac{n}{2}})^2$  can be reused for both  $\omega_n^k$  and  $\omega_n^{k+\frac{n}{2}}$ .

- Runtime analysis:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

By the master theorem,  $T(n) = O(n \log n)$ .

- *Example.* Evaluation of a polynomial of degree-bound 4

Given:  $A(x) = a_0 + a_1x + a_2x^2 + a_3x^3$  (coefficient representation)

Goal: Obtain  $A(1), A(i), A(-1), A(-i)$  (point-value representation at points  $1, i, -1, -i$ )

Level-1 subproblems:

$$A(x) = A_e(x^2) + xA_o(x^2)$$

Level-2 subproblems:

$$\begin{aligned} A_e(y) &= a_0 + a_2y = A'_e(y^2) + yA'_o(y^2) \\ A_o(y) &= a_1 + a_3y = A''_e(y^2) + yA''_o(y^2) \end{aligned}$$

Level-3 subproblems: (base cases)

$$\begin{aligned} A'_e(z) &= a_0 \\ A'_o(z) &= a_2 \\ A''_e(z) &= a_1 \\ A''_o(z) &= a_3 \end{aligned}$$

The diagram below shows how the values are constructed. The lines indicate which subproblem values were queried in the computation of each value. The calculation of each value takes constant time.

Level-1 subproblems

Level-2 subproblems

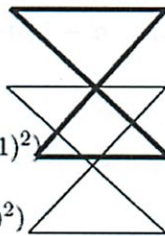
Level-3 subproblems

$$A(1) = A_e(1^2) + 1A_o(1^2)$$

$$A(i) = A_e(i^2) + iA_o(i^2)$$

$$A(-1) = A_e((-1)^2) - 1A_o((-1)^2)$$

$$A(-i) = A_e((-i)^2) - iA_o((-i)^2)$$



$$A_e(1) = A'_e(1^2) + yA'_o(1^2)$$

$$A_e(-1) = A'_e((-1)^2) + yA'_o((-1)^2)$$

$$A_o(1) = A''_e(1^2) + yA''_o(1^2)$$

$$A_o(-1) = A''_e((-1)^2) + yA''_o((-1)^2)$$

$$A'_e(1) = a_0$$

$$A'_o(1) = a_2$$

$$A''_e(1) = a_1$$

$$A''_o(1) = a_3$$

In each level, we are reducing the number of evaluation points by half, so there are  $O(\log n)$  levels of subproblems. At each level, two functions (even and odd) are created at each evaluation point, so there are still  $n$  values to calculate at each level. Therefore, we have a total of  $O(n \log n)$  values to calculate, which is why the runtime complexity of the FFT is  $O(n \log n)$ .

Co46 L

9/18

## Randomized Algorithms

Taking Probabilistic

Why Randomized?

Quick sort (randomized)

Skip list - data structure

---

Difficult or inefficient to find exact ans

But if you are ok w/ the occasional incorrect

---

Or always correct

But running time is variable

---

## Randomized algorithms

- Decisions by generating a random #  $r = \{1, \dots, R\}$
- On the same input, on diff executions, algorithm may run for a diff # of steps or even produce diff outputs possibly even wrong ~~ans~~<sup>ans</sup> with minuscule prob

②

Classified by gambling cities

~~Ala~~

Las Vegas

- always produce correct output
- run in expected poly time

Monte Carlo

- Always run in poly time
- Prob(output is correct) > high  
↳ ie close to 1

Examples

Test if  $n$  is prime

- ~~non~~ Deterministic  $n^6$
- But invent the prob. one on  $p$ -set
- High prob correct
- Matrix product  
( $\hat{=}$   $A \times B$ )

③

Quick sort

— LV algorithm - makes sense  
would be silly to have it be Monte Carlo

tries to improve  $\sqrt{n}$  time

interesting because of constant factors

best ya can do is  $O(n \log n)$

big  $O$  notation hides constant factors  
- what is practical

Small variations can affect things

—  
Divide + Conquer - work in divide step, rather  
than combine ~~sort~~

Sorts "in place" - only requires  $O(\log n)$   
auxiliary space

Merge sort  $O(n)$  aux space ~~instead~~  
trying to do in place is painful

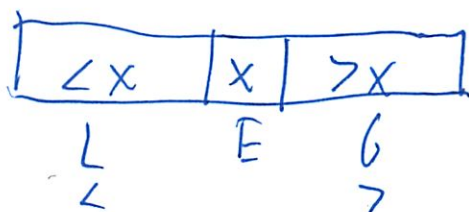


9

Diff variants: Basic, intellegent pivoting, randomized alg.

Quicksort  $n$ -element array  $A$

Divide: 1. Pick a pivot element  $x$  in  $A$   
~~Partition~~ Partition the array into sub-arrays



Conquer: 2. Recursively sort  $L, G$

Combine: 3. Trivial  $\rightarrow$  just append

Everything differs based on how you pick the pivot

Basic quicksort Pivot  $x = A[1]$   
1st el in array  $\uparrow$  1st element  $I \dots n$

- Remove in turn, each el  $y$  from  $A$  and  
insert  $y$  into  $L, E$ , or  $G$  depending on comparison  
to  $x$ .

⑤ Each insertion/removal takes  $O(1)$  time

Partition step takes  $O(n)$  time

To do this in place  $\rightarrow$  see code in CLRS

↳ doesn't matter for runtime  
but for aux space it does

So complexity

Worst case: reverse sorted

We want the halves to be roughly = sized

Don't want 0 vs  $n-1$  "halves"

↳ Not dividing into smaller parts

One side L or R has  $n-1$  elements  
then other empty

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

← divide step

6

Its an arithmetic series  $\Rightarrow$

$$T(n) = cn + c(n-1) + c(n-2) + \dots + 1$$

$$\hookrightarrow \theta(n^2)$$

~~People do this in practice~~

## Intelligence Not Selection

### Median finding

Worst-case  $O(n)$  for this part  
divide step must use  
recursive alg to find pivot

Can guarantee balanced L + R using median  
selection  $\theta(n)$  time

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) + \theta(n)$$

$\uparrow$  median select       $\uparrow$  divide step

$$= \theta(n \lg n)$$

①

So notice sometimes we care about constant factors and sometimes not

But this totally sucks in practice  
Selecting pivot through recursion

If randomize  $\Rightarrow$  it beats in practice

## Randomized Quick Sort

Just pick a random pivot  $x$

Lat each recursion, a random choice is made

Part of reading assignment to analyze

## "Paranoid" Quick Sort

Picks a random pivot

But does a quick check to make sure pretty even

~~Choose~~ Count size of subproblems

Until resulting partition is such that

$$|L| \leq \frac{3}{4}n \quad \text{AND} \quad |G| \leq \frac{3}{4}n$$



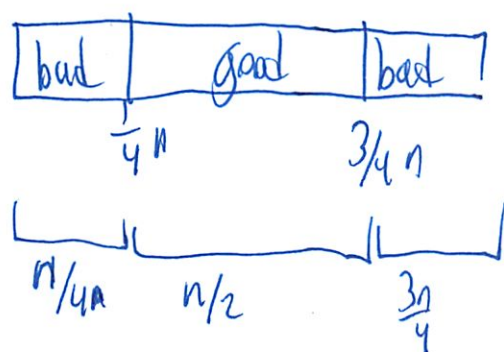
(8)

This guarantees not extremely unbalanced partitions

Complexity of loop 2

Good call: sizes of  $L$  +  $G \leq \frac{3}{4}n$  each

Bad call:  $L$  or  $G > \frac{3}{4}n$



A call is good call w/ prob  $> \frac{1}{2}$

So expected # of iterations  $\leq 2$

---

So we can write recurrence relation

$$T(n) \leq \max_{\frac{n}{4} \leq i \leq \frac{3n}{4}} (T(i) + T(n-i)) + E(\# \text{ of iterations}) \cdot cn + 2cn$$

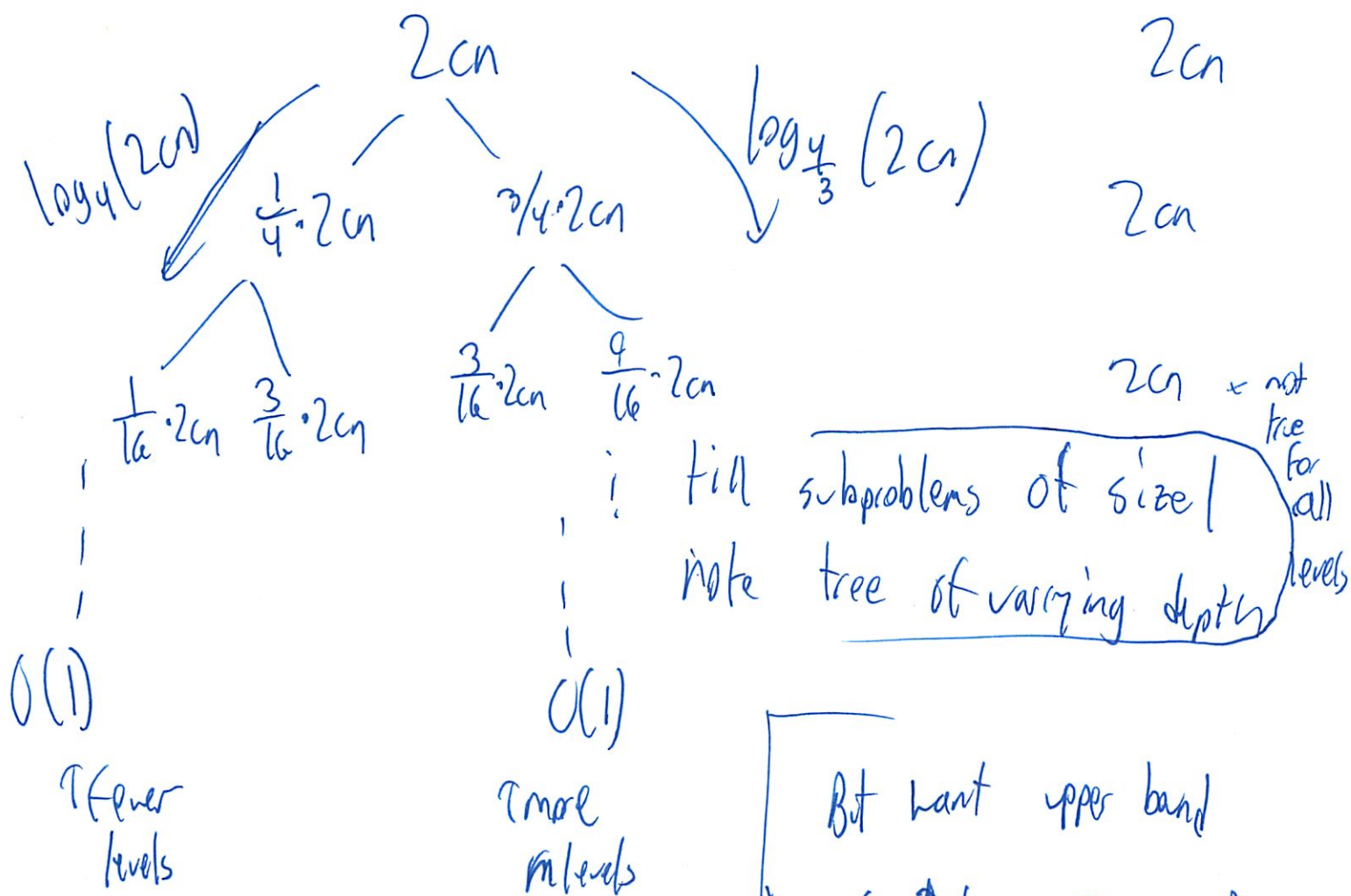
does not conform to master Thm

①

So must do recursive tree analysis

$$= T(n/4) + T(3n/4) + 2cn$$

So let's do it



But want upper bound

$$\leq \log_{4/3} 2cn \cdot 2cn$$

levels      size each level

So  $\Theta(n \lg n)$

10

## Thurs: Monte Carlo Alg

### Skip Lists

invented 1989

randomized data structure

a lot in common w/ randomized Bal BST

Balanced: Red Black, AVL, B tree

do rotations

so worst case  $O(\log n)$

Sorted linked list



↪

double linked list

↑ 7th Ave Subway  
in NYC

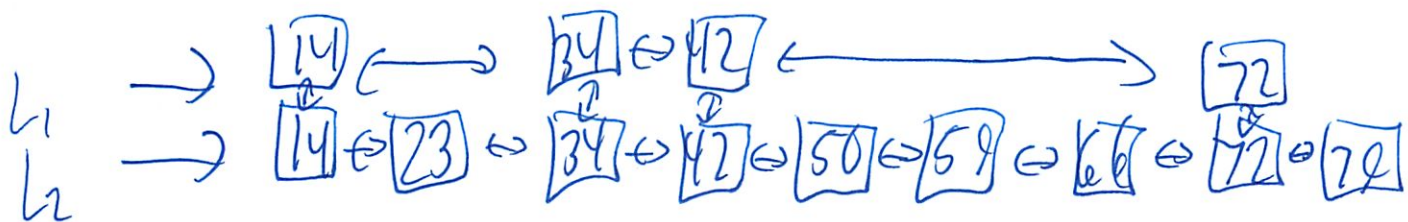
If sorted, can search in  $O(n)$  time  
must walk through list!

d

11

Better way?

Well let's add an express line

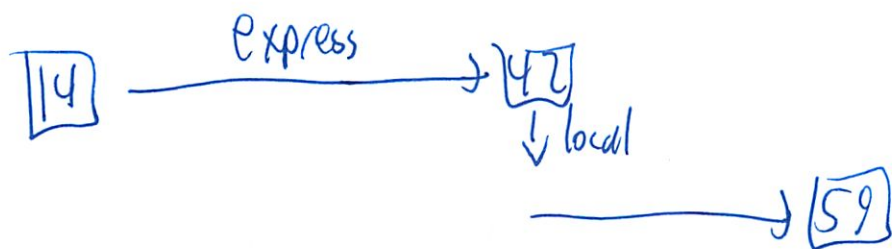


2 linked lists

bottom one always has all

top one has a subset of pts

So to go somewhere, like 59th st



ie, Ride in top linked list, until going right is faster  
(would not go to 72)  
then switch to local + go right till stop



(12)

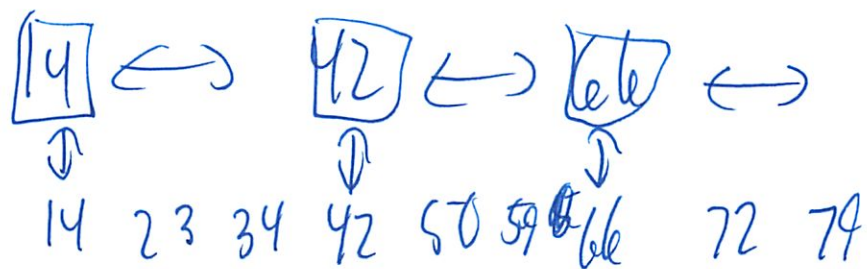
So a single linked list is  $O(n)$

But it's tedious select top list

2 we can reduce complexity of search in worst case

So it could rearrange top stations so lowest avg value...

1. Evenly space out stations



So Search cost  $\propto |L_1| + \frac{|L_2|}{|L_1|}$

$\nearrow$  express

$\nearrow$  local - only a limited # of stops

(13)

Stations at roughly = intervals

$$|L_1|^2 = |L_2| = n$$

$$|L_1| = \sqrt{n}$$

Not satisfied with

So express  $\hookrightarrow 2\sqrt{n}$  lines (3 lines)

$$\hookrightarrow 3 \sqrt[3]{n}$$

kth lines

$$\hookrightarrow k \sqrt[k]{n}$$

If

$$k = \lg n$$

$$\text{then } \lg n \cdot \sqrt[\lg n]{n}$$

$$= \lg n \cdot 2$$

$$= 2 \lg n$$

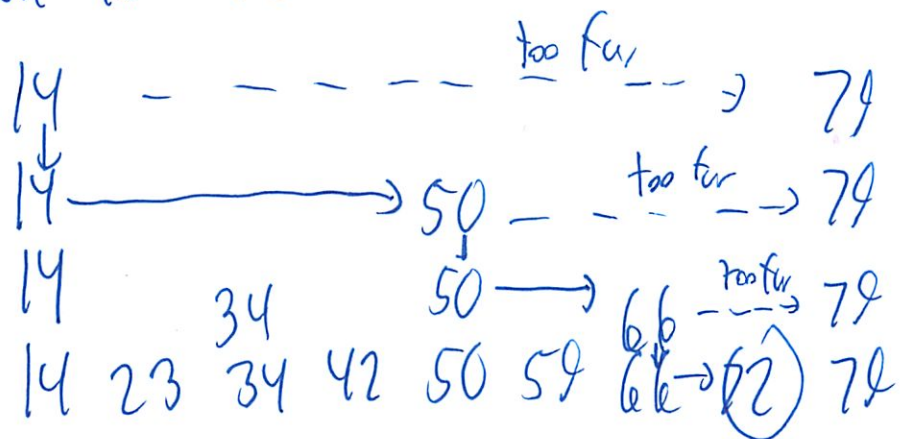
(14)

Whole point of Binary tree & didn't know what was coming

So here i not a good worst-case  
but a good avg-case

So can make a statement about the avg  
Case Complexity

So search for 72



But how do we maintain all this?

Insert() randomized structure

will for sure insert in bottom list

↳ search to see where x fits in bottom list

15

Insert selectively in list above

↳ by flipping a fair coin

if heads  $\rightarrow$  promote to next level

flip again

if tails or run out of levels  $\rightarrow$  done

At most  $\lg n$  linked lists

↳ Actually that is the optimal # of levels

So  $O(\lg n)$  is the search complexity we are looking for

On any  $\frac{1}{2}$  elements promotes 0 levels

$\frac{1}{2}$  promoted  $\geq 1$  level

$\frac{1}{4}$   $\geq 2$  levels

~~So expected steps in each linked list = 2~~



(16)

Expected time for search

Expected steps in each linked list = 2

Trace the search backwards from target until reach an element in next higher list.

Prob element is in next higher list =  $\frac{1}{2}$

$L_i \rightarrow L_{i-1}$

Basically follow bottom list

Prob that exists above =  $\frac{1}{2}$

So  $EV = \frac{1}{\text{prob}} = \frac{1}{\frac{1}{2}} = 2$  els to check  
before can pop up

Since # of linked list is  $\lg n$

Expected time is  $2 \lg n$

$= \Theta(\lg n)$

## Randomized Algorithms

Why randomized?

Quicksort

Skip lists

## Randomized or Probabilistic Algorithms

- Algorithm generates a random number  $r \in \{1, \dots, R\}$  and makes decisions based on  $r$ 's value
- On the same input on different executions randomized algorithm may
  - run for a different number of steps
  - produce different outputs

### Monte Carlo

- runs in poly time always
- $\text{prob}(\text{output is correct}) > \frac{2}{3}$ 
  - high variation due to  $r$

### Las Vegas

- always produces correct output
- runs in expected polynomial time

## Some Interesting Randomized Algorithms

(2)

Matrix product checker:  $C \stackrel{?}{=} A \times B$

Testing whether a number is a prime

Today: practical randomized sorting algo  
practical randomized data structure

### Quicksort

C.A.R. Hoare (1962)

Divide & conquer algorithm but work in divide step rather than combine

Sorts "in place" (like insertion sort, but not mergesort)  
requires  $O(n)$  auxiliary space

Different variants:

Basic: good in average case (for a random input)

Median-based pivoting: uses median-finding

Randomized: good for all inputs in expectation

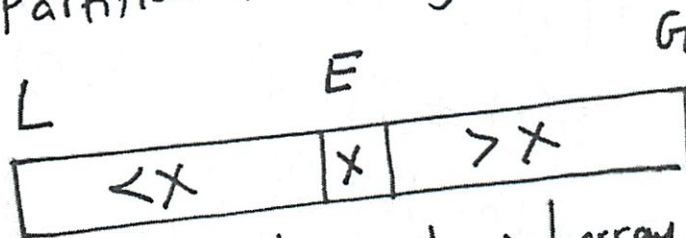
Las Vegas algorithm.

Quicksort

$n$ -element array  $A$

Divide:

1. Pick a pivot element  $x$  in  $A$   
Partition the array into sub-arrays



Conquer: Recursively sort subarrays  $L$  and  $G$

Combine: Trivial

Basic Quicksort

pivot  $x = A[1]$  or  $A[n]$ , first or last element

- Remove, in turn, each element  $y$  from  $A$  and
- Insert  $y$  into  $L$ ,  $E$  or  $G$  depending on the comparison with pivot  $x$
- Each insertion and removal takes  $O(1)$  time
- Partition step takes  $O(n)$  time
- To do this in place: see code in CLRS p 171



## Basic Quicksort analysis

- Input sorted or reverse sorted
- Partition around min or max elements
- One side L or R has  $n-1$  elements, other 0

$$\begin{aligned}
 T(n) &= T(0) + T(n-1) + \theta(n) \quad \leftarrow \text{divide step} \\
 &= \theta(1) + T(n-1) + \theta(n) \\
 &= T(n-1) + \theta(n) \\
 &= \theta(n^2) \quad (\text{arithmetic series})
 \end{aligned}$$

Does well on random inputs in practice

## Pivot Selection Using Median Finding

Can guarantee balanced L and R using rank/median selection algorithm that runs in  $\theta(n)$  time

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n) + \theta(n)$$

↙ recursive median selection
↘ divide

$$T(n) = \theta(n \log n)$$

This algorithm is slow in practice and loses to mergesort.

⑤

## Randomized Quicksort

$x$  is chosen at random from array  $A$   
(at each recursion, a random choice is made)

Expected time is  $O(n \log n)$  for all  
input arrays  $A$

See CLRS p181-4 for analysis; we will analyze  
here a variant quicksort

## "Paranoid" Quicksort

Repeat

choose pivot to be random element of  $A$

Perform Partition

Until resulting partition is such that

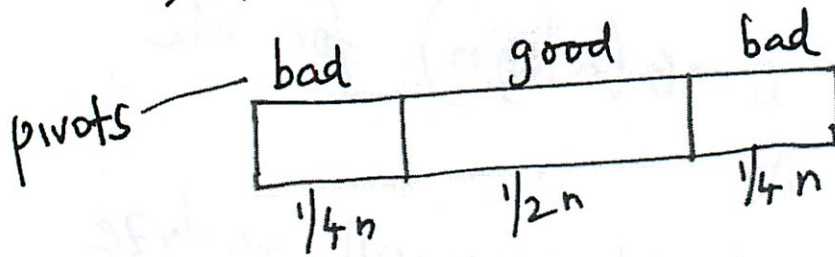
$$|L| \leq \frac{3}{4} |A| \quad \text{and} \quad |G| \leq \frac{3}{4} |A|$$

Recurse on  $L$  and  $G$

## "Paranoid" Quicksort Analysis

(6)

Good call : Sizes of  $L$  &  $R \leq \frac{3}{4}n$  each  
Bad call : One of  $L$  or  $R$  is  $> \frac{3}{4}n$



A call is good with probability  $> \frac{1}{2}$

Let  $T(n)$  be an upper bound on the expected running time on any array of  $n$  size

$T(n)$  comprises:

- Time needed to sort left subarray
- Time needed to sort right subarray
- The number of iterations to get a good call \*  $\underbrace{c \cdot n}_{\text{cost of partition}}$

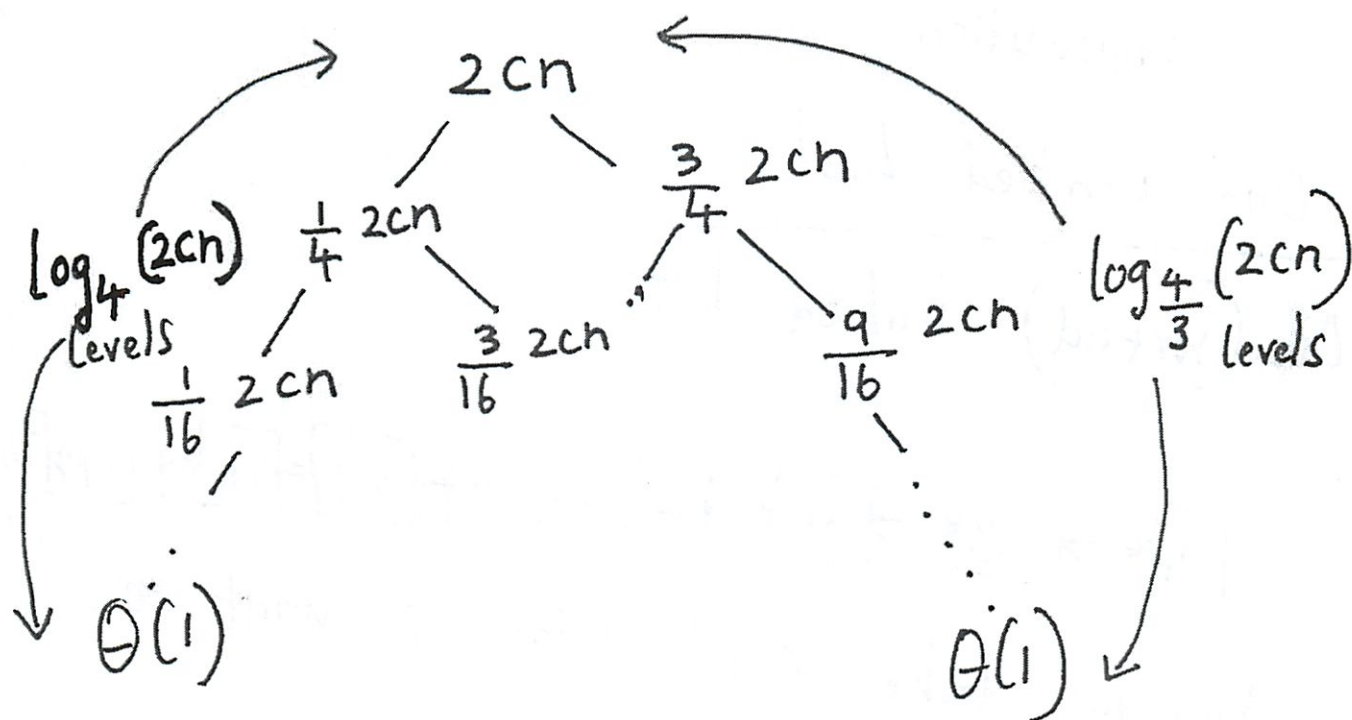
(7)

# Expectations

$$T(n) \leq \max_{n/4 \leq i \leq 3/4 n} (T(i) + T(n-i)) + E(\# \text{ iterations}) \cdot cn$$

$$E(\# \text{ iterations}) \leq 2 \quad \text{since prob of good call} > 1/2$$

$$= T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + 2cn$$



2cn work at each level  
max  $\log_{4/3}(2cn)$  levels

$\Theta(n \log n)$  expected runtime.



⑧

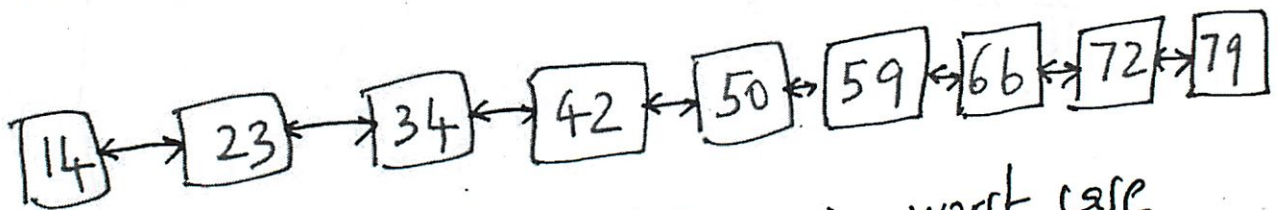
## Skip Lists

William Pugh (1989)

- Easy to implement (as compared to balanced trees)
- Maintains a dynamic set of  $n$  elements in  $O(\log n)$  time per operation in expectation

## One Linked List.

One (Sorted) linked list



Searches take  $O(n)$  time in worst case

Suppose we had two sorted linked lists

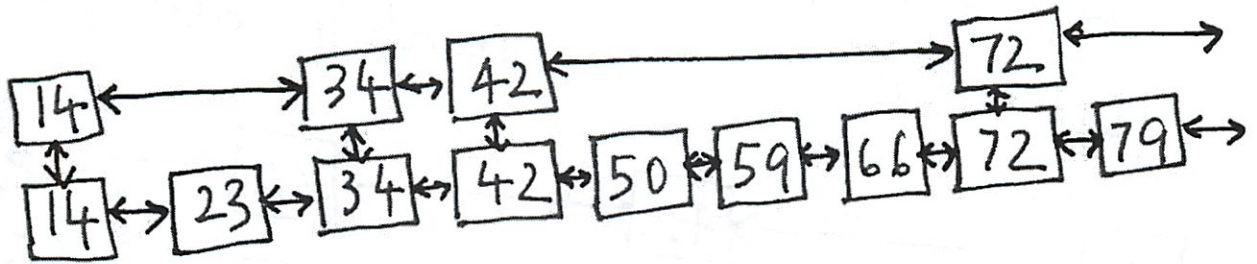
- each element can appear in one or both lists

(9)

## Two Linked Lists

Express and local subway lines  
(à la New York City 7th Avenue Line)

- Express line connects a few of the stations
- Local line connects all stations
- Links between lines at common stations



## Searching in Two Linked Lists

Search(x):

- Walk right in top linked list (L<sub>1</sub>) until going right would go too far
- Walk down to bottom linked list (L<sub>2</sub>)
- Walk right in L<sub>2</sub> until element found (or not)

Search(59)

## Analysis

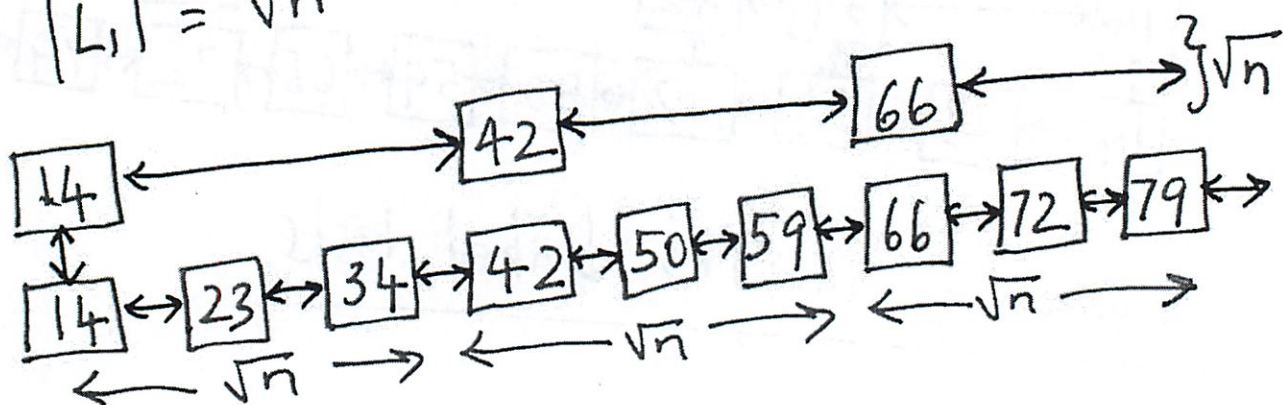
$$\text{Search cost} \approx |L_1| + \frac{|L_2|}{|L_1|}$$

Minimized when terms are equal

$$|L_1|^2 = |L_2| = n$$

$$|L_1| = \sqrt{n}$$

Search is  $\Theta(\sqrt{n})$



## More Linked Lists

$$2 \text{ sorted lists} \Rightarrow 2 \cdot \sqrt{n}$$

$$3 \text{ sorted lists} \Rightarrow 3 \cdot \sqrt[3]{n}$$

$$k \text{ sorted lists} \Rightarrow k \cdot \sqrt[k]{n}$$

$$\lg n \text{ sorted lists} \Rightarrow \lg n \cdot \sqrt[\lg n]{n}$$

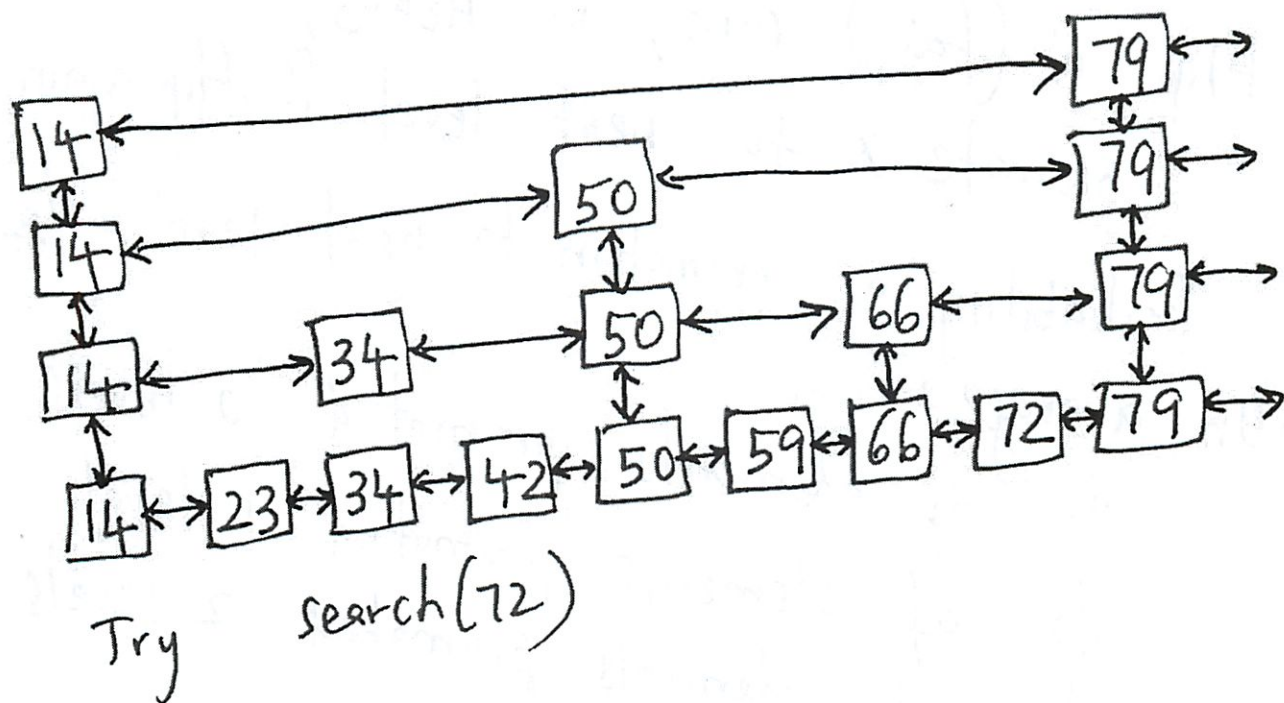
$$= 2 \lg n$$

like a binary tree!



# Searching in lgn Linked Lists

(11)



## Insert (x)

- To insert an element  $x$  into a skip list
- Search( $x$ ) to see where  $x$  fits into bottom list
  - Always insert into bottom list
  - Insert into some of the lists above which ones?



## Insert(x)

(12)

Flip a (fair) coin; if HEADS,  
promote  $x$  to next level & flip again

Probability of promotion to next level =  $\frac{1}{2}$

On average:

- $\frac{1}{2}$  of elements promoted 0 levels
- $\frac{1}{2}$  of elements promoted 1 level
- $\frac{1}{4}$  of elements promoted 2 levels

## Expected time for Search

Intuition: Expected number of steps in each  
linked list is 2

Trace a search path backwards from  
the target until reaching an element  
that appears in the next higher list

Probability of element being in the next  
higher list =  $\frac{1}{2}$

Since # linked lists is  $\lg n$ , expected time  
=  $2 \lg n = \Theta(\lg n)$

8/20

6.046  
LS Check Math Randomized  
Alg

- Checking matrix products
- Checking polynomial identities

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 3 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 4 \\ 3 & 3 & 6 \\ 4 & 12 & 8 \end{pmatrix}$$

"                      "                      "

A                      B                      C

← flipped  
↑ can we find

Is  $A \cdot B = C$ ?

Did ya make a calc error?

Given  $A, B, C$   $n \times n$  matrices  
entries from field (ie  $+$ ,  $\times$ ,  $\div$ )  
 $\uparrow \mathbb{R}$

Qst "Pass" if  $A \cdot B = C$   
"Fail" if  $A \cdot B \neq C$

②

Deterministic  $\rightarrow$  Compute  $A \cdot B$ , Compare w/  $C$   
 $O(n^3) \rightarrow O(n^2)$

Is a faster way to multiply matrices  
 $O(n^{2.81})$

Even faster  
 $O(n^{2.745})$

Faster still  
 $O(n^{2.376})$

Least year  
 $O(n^{2.272})$

Horrible constants so very impractical

Do we need to compute  $A \cdot B$

If  $A \cdot B \cdot r \neq C \cdot r$  return "not equal"

3

Now allow a prob of error

↳ w/ error prob  $\leq \frac{1}{2}$   
for not pass

So if pass  $\rightarrow$  always pass

not pass  $\rightarrow \geq \frac{1}{2}$  time  $\rightarrow$  not pass

↳  $\leq \frac{1}{2} \rightarrow$  pass

(False neg or False pos?)

Freivald's idea: multiply both sides by a  
random vector

$$A \cdot B \cdot r = C \cdot r$$

↑ associative

So can compute  $B \cdot r$  (at  $O(n^2)$ )  
↑ left w/ vector  $O(n^2)$



4

Algorithm:

1. - pick  $n$ -vector  $r$  randomly

-  $n$  entries

- each entry chosen from  $\{0, 1\}$

- choose ind ~~at~~ each choice

- uniform  $P(r_i = 0) = P(r_i = 1) = \frac{1}{2}$

$O(n)$

2. a. Compute  $B \cdot r$

b. Compute  $A \cdot (B \cdot r)$

c. Compute  $c \cdot r$

$O(n^2)$

$O(n^2)$

$O(n^2)$

3. Pass  $A \cdot B \cdot r = c \cdot r$

$O(n^2)$

Output  $\neq$  if not

---

$O(n^2)$

↑  
note  $n$  cols, cons  
so problem is  $n^2$

But does it work?

5  
if  $A \cdot B = C$  then  
 $A \cdot B \cdot r = C \cdot r$   
then alg will pass w/ prob 1

If  $A \cdot B \neq C$

Problem: some vectors don't "work"  
Lse  $A \cdot B \neq C$

eg if  $r = \begin{pmatrix} 0 \\ 0 \\ \vdots \end{pmatrix}$  then  $A \cdot B \cdot r = C \cdot r$   
 $\forall A, B, C$

in our example: other bad  $R$ s

- That has 2 0s  
will 0 out the problem  $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$

-  $\begin{pmatrix} 0 \\ 1 \\ \vdots \end{pmatrix}$  doesn't work either  
(too big / too small) cancel at

- might be other bad ones

- try them at home!

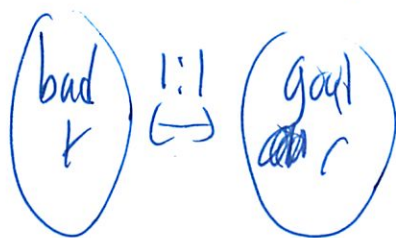
(6)

But a lot of  $f$ s do work  
↳ at least half

Still  $\geq \frac{1}{2}$   $r$ 's are "good"

↳ we won't know which

idea For each bad  $r \rightarrow$  give a mapping to a good  $r$   
Make sure mapping is 1:1



Then at least half  
have to be good

$$\# \text{ good } r \geq \# \text{ bad } r$$

$$\hookrightarrow \geq \frac{1}{2} \text{ } r \text{ s are good}$$

Claim if  $A \circ B \neq C$

Then  $\geq \frac{1}{2}$  choices of  $r$  from  $\{0, 1\}$   
Satisfied  $A \circ B \circ r \neq C \circ r$

⑦

Freivald's  
algorithm

Now let's prove this claim

let  $D = A \circ B - C$   
 $L \times n$  matrix

$$D \cdot r = A \cdot B_r - C_r$$

$$\neq \begin{pmatrix} 0 & 0 & 2 & \dots & 0 \\ 0 & 0 & & & \\ 0 & 0 & & & \\ & & & & 0 \end{pmatrix} \text{ by assumption (prob almost 0)}$$

$\exists D_{ij} \neq 0 \leftarrow$  maybe many, pick one

Want proof  $P[D_r \neq 0] \geq \frac{1}{2}$

$$D = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -4 & 4 \end{pmatrix} \text{ pick } (i,j) = (3,3)$$

Map  $r$  to  $r + \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix} \leftarrow e^{(i)}$   
 $\leftarrow j\text{th location}$

For bad  $r$ ,  $D_r = \bar{0}$

$$D_{r'} = D_r + D e^{(i)}$$



8

~~Addition~~

XOR

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Now must figure out what these are

$$D_{r'} = \underbrace{D_r}_{=0 \text{ by assumption}} + D_{\begin{pmatrix} 0 \\ 0 \\ \pm 1 \\ 0 \end{pmatrix}}$$

+1 if  $r_j = 0$   
-1 if  $r_j = 1$

has non 0  
el in  $j$ th  
location

$$\underbrace{D_{\begin{pmatrix} 0 \\ 0 \\ \pm 1 \\ 0 \end{pmatrix}}}_{j\text{th entry}} \leftarrow e^{(j)} = \sum_k D_{ik} \cdot e_k^{(j)} = \underbrace{D_{ij}}_{\neq 0} \cdot \underbrace{e_j^{(j)}}_{\pm 1}$$

$\downarrow$  0 if  $k \neq j$

⑨

This is a mapping from bad  $r$  to good  $r$

Now need to show its  $1:1$

$$\begin{array}{ccc} \text{So bad } r & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \\ \downarrow & & \\ r' & \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{array}$$

Why  $1-1$ ?

$$\begin{array}{l} \text{if } r_1 \oplus e^{(i)} = r' \\ r_2 \oplus e^{(i)} = r' \end{array} \quad \left. \vphantom{\begin{array}{l} r_1 \oplus e^{(i)} = r' \\ r_2 \oplus e^{(i)} = r' \end{array}} \right\} \text{ then } r_1 = r_2$$

$\uparrow$   $\rightarrow$

$r_1, r_2, r'$  are all 0-1 vectors

So we make the mathematical version of XOR

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

due to a property of XOR, these must be =

(10)

So if XOR 2 distinct vectors w/  $p(k)$

You get 2 distinct ans

So must be 1!

Sanity check: - Did we use that entries are over a field?

- What if  $r$  is chosen from a larger set?
  - ie vectors w/ entries from  $0 \dots q$
- How can we improve Error prob?

If  $A \cdot B \neq C$  prob of error is  $\frac{1}{2^k}$   
exponentially  $\downarrow$  error w/ # of loops

---

### Polynomial identities

Check  $(x^2+1)^2(x^2-1)^2 + (x-1)^2 \stackrel{?}{=}$   
 $x^8 - 2x^4 + 2x^2 - 2x + 2$

Could have up to  $n$  terms

(11)

We want to ~~know~~ know for all  $x$ !

$$\text{Is } \det \begin{pmatrix} 2x^2 & 4x^2+3 \\ 4x & 8x \end{pmatrix} + 12x = 0$$

Do you need to try all  $x$ ?

↳ but  $\infty$  many  $x$

(Some notes on slide)

Remember Given degree  $\leq n$  poly  $f$  then

Or  $\exists ! f(x) = 0$  for all  $0$   
Or  $\forall x, f$  has at most  $n$  roots

So we use this simple fact for polynomial testing

---

Given  $f$  a poly of deg  $\leq n$

may query  $f(x)$  in one step

$x_i \rightarrow$  (Oracle)  $f(x)$  = 1 step



(12)

Have no idea what poly. is.

---

If  $f \equiv 0 \rightarrow \text{output } "\equiv 0"$

if  $\exists x \text{ s.t. } f(x) \neq 0 \rightarrow \text{output } "\neq 0"$   
 $\hookrightarrow \text{w/ prob } \geq \frac{1}{2}$

---

Deterministic: Eval  $f$  at  $n+1$  distinct locations

if  $f(x_i) = 0 \forall i = 0, \dots, n$  then

$f$  has  $n+1$  roots  $\rightarrow f(x) = 0$

So output  $"f \equiv 0"$

if  $\exists x_i \text{ s.t. } f(x_i) \neq 0$  then output  
 $"f \neq 0"$

Randomized

eval  $f(x)$

$$L \neq 0$$
$$P[\text{output} = 0] = \frac{\# \text{ roots in } [1, 2]}{2n}$$

$$\leq \frac{n}{2n} = \frac{1}{2}$$

Then  $wald$  be  $O(1)$

Even a little randomness helps a lot

Is random is even needed

$L_{\text{prob}}$  - but no definite ans

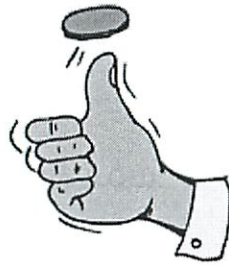
Crazy things would happen otherwise

Today:

## Check your math with randomized algorithms

Prof. Ronitt Rubinfeld

6.046 Lecture 5



Warning: see lecture notes for more details

### Checking matrix products

$$\text{Is } \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 3 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 3 & 4 \\ 3 & 3 & 6 \\ 4 & 12 & 8 \end{pmatrix} ???$$

- Checking matrix products
- Checking polynomial identities

### The question and an attempt

- Given 3  $n \times n$  matrices  $A, B, C$ :
  - Is  $A \cdot B = C$ ??
- A first try:
  - Compute  $A \cdot B$  and compare result to  $C$

Runtime:  
 $O(n^3)$  multiplications  
+  $O(n^2)$  more work

5

9/20

# Fast Matrix Multiplication

- Multiply 2x2 matrices with 7 multiplications gives  $O(n^{2.81})$  time [Strassen]
- Multiply 70x70 matrices with 143640 multiplications gives  $O(n^{2.795...})$  time [Pan '76]
- ...
- $O(n^{2.376...})$  time [Coppersmith-Winograd'87]
- $O(n^{2.372...})$  time [Virginia Williams '12]

Do we really need to compute  $A \cdot B$ ?

Is  $O(n^2)$  possible?

## Freivald's algorithm

- Pick random  $n$ -vector  $r$ 
  - Each entry independently and uniformly from  $\{0,1\}$
- If  $A \cdot B \cdot r \neq C \cdot r$  return “not equal”

## Sanity Checks:

- Did we use that entries are over a field?
- What if  $r$  is chosen from a larger set?
  - i.e., vectors with entries from  $0..q$
- How can we improve error probability?



## Freivald's algorithm (improved error)

- Do  $k$  times:
  - Pick random  $n$ -vector  $r$ 
    - Each entry independently and uniformly from  $\{0,1\}$
  - If  $A \cdot b \cdot r \neq C \cdot r$  return “Not equal”
- Return “Pass”

## Checking Polynomial Identities

- Is it the case that

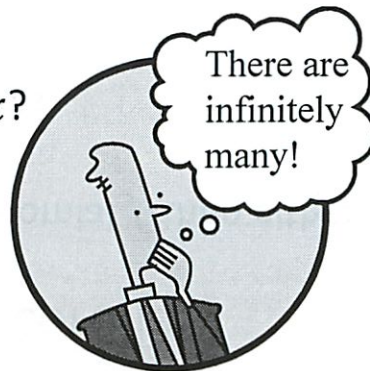
$$(x^2 + 1)^2(x^2 - 1)^2 + (x - 1)^2 \equiv x^8 - 2x^4 + 2x^2 - 2x + 2$$

Notation: “ $\equiv$ ” is “FOR ALL  $x$ ”

## Another example:

- Is  $\text{Det} \begin{pmatrix} 2x^2 & 4x^2 + 3 \\ 4x & 8x \end{pmatrix} + 12x \equiv 0$ ?

- Do you need to try all  $x$ ?



## Recall from last time:

- 2 points determine a line
- 3 points determine a quadratic polynomial
- ...
- Given  $x_0, x_1, \dots, x_n$  (distinct), and  $y_0, \dots, y_n$  (not necessarily distinct), there is EXACTLY one degree  $\leq n$  polynomial  $p$  such that
$$f(x_0) = y_0, f(x_1) = y_1, \dots, f(x_n) = y_n$$

## Important corollary

- Given degree  $\leq n$  polynomial  $f$  then either
  - $f(x) \equiv 0$
  - Or  $f$  has at most  $n$  roots

9/20

©  
Fall 2012

## Lecture 5

Randomized Algorithms continued ...

- Checking Matrix products
- Checking polynomial identities

## Checking Matrix Products

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 3 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 & 1 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \end{pmatrix} \stackrel{?}{=} \begin{pmatrix} 1 & 3 & 4 \\ 3 & 3 & 6 \\ 4 & 12 & 8 \end{pmatrix}$$

$\begin{matrix} \text{"} \\ A \end{matrix}$ 
 $\begin{matrix} \text{"} \\ B \end{matrix}$ 
 $\begin{matrix} C \end{matrix}$

should be 12  
 should be 8

Is  $\underbrace{A \cdot B = C}_{\substack{\text{nxn matrices} \\ \text{from field}}} ?$

First try:

Simple matrix mult algorithm

- compute  $A \cdot B$  + check if result =  $C$

-  $O(n^3)$  mults

↑

assume mult is  $O(1)$  time



Improvements:

faster mat mult algs to compute  $A \cdot B$

- multiply  $2 \times 2$  matrices with 7 mults

$\Rightarrow$  mat mult in  $O(n^{2.81})$  time

[Strassen '69]

- mult  $70 \times 70$  matrices with 143640 mults

$\Rightarrow O(n^{2.795...})$  [Pan '78]

⋮

-  $O(n^{2.376...})$

[Coppersmith-Winograd 1990]

⋮  
-  $O(n^{2.372...})$

[Virginia Williams 2012]

Can we do better?

do we need to compute  $A \cdot B$ ?

Freivalds idea: (for Matrices over a field)

Multiply both sides by a random vector

more details:

- pick  $n$ -vector  $r$  randomly
  - $n$  entries
  - each entry chosen from  $\{0,1\}$ 
    - independently
    - $\Pr[r_i=0] = \Pr[r_i=1] = 1/2$
- Compute  $A(B \cdot r)$
- Compute  $C \cdot r$
- Pass if equal, "Not equal" if not

Why faster?

$$\begin{array}{ll}
 B \cdot r & O(n^2) \\
 A \cdot (B \cdot r) & O(n^2) \\
 \quad \uparrow & \text{n-vector} \\
 C \cdot r & O(n^2)
 \end{array}$$

Why correct?

$$\text{if } A \cdot B = C \text{ then } A \cdot B \cdot r = C \cdot r$$

for every choice of  $r$

$\Rightarrow$  Algorithm always correct

What if  $A \cdot B \neq C$ ?

Problem:

Some vectors don't "work"

e.g. if  $r = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$  then  $A \cdot B \cdot r = C \cdot r \quad \forall A, B, C$

in our example,  $r \in \left\{ \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \right\}$  don't work.

Still, will show that many (at least  $\frac{1}{2}$ ) of  $r$ 's

show that  $A \cdot B \neq C$ .

idea for each "bad"  $r$  ( $r$  s.t.  $A \cdot B \cdot r = C \cdot r$ )

give a mapping to a "good"  $r'$  ( $r'$  s.t.  $A \cdot B \cdot r' \neq C \cdot r'$ )

→ ensure mapping is 1-1  $\Rightarrow \geq \frac{1}{2}$  of  $r$ 's are good

Claim if  $A \cdot B \neq C$

then  $\geq \frac{1}{2}$  choices of  $r$  from  $\{0, 1\}^n$

satisfy  $A \cdot B \cdot r \neq C \cdot r$

Pf.

Let  $D \leftarrow A \cdot B - C$  ( $n \times n$  matrix) [so  $D \cdot r = A \cdot B \cdot r - C \cdot r$ ]

by assumption,  $D \neq (0)$

so  $\exists i, j$  s.t.  $D_{ij} \neq 0$

(maybe many such  $i, j$  - just pick one!)

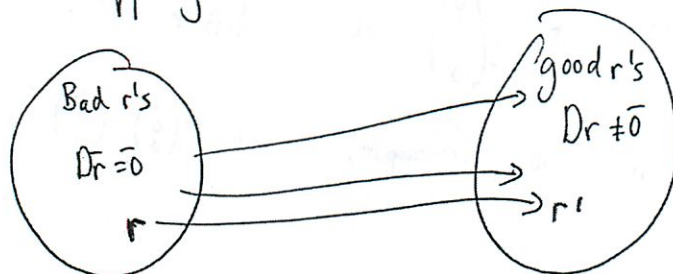
In our example:

$$D = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -4 & 4 \end{pmatrix}$$

pick  $(i,j) = (3,3)$   $((3,2)$  works as well

so  $j=3$

The mapping:



For any bad  $r$ :

map  $r$  to  $r' = r \oplus \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \leftarrow j\text{th position}$

Call this  $e^{(j)}$  - all 0 entries except for 1 in  $j$ th position

So  $r'$  is just  $r$  with  $j$ th bit flipped

(i.e. if  $r_j = 0$  then  $r'_j = 1$   
else (if  $r_j = 1$ ) then  $r'_j = 0$ )

So how does  $Dr = 0$  compare to  $Dr'$ ?

$i$ th entry of  $Dr'$  is  $\sum_k D_{ik} r'_k = \left( \sum_{k \neq j} D_{ik} r_k \right) + D_{ij} r'_j$

so  $Dr - Dr' = D_{ij} (r_j - r'_j)$   
by construction  $r_j \neq r'_j$

so,  $Dr' \neq Dr$  by assumption  
+ therefore  $Dr' \neq 0$

Same as  $r_k$  when  $k \neq j$

Why is mapping 1-1?

if  $r \oplus e^{(j)} = r' \oplus e^{(j)}$  then  $r = r \oplus e^{(j)} \oplus e^{(j)} = r' \oplus e^{(j)} \oplus e^{(j)} = r'$



Conclusion:

$$\# \text{ bad } r\text{'s} \leq \# \text{ good } r\text{'s}$$

$$\text{so fraction of } r\text{'s st. } A \cdot B \cdot r = C \cdot r \leq \frac{1}{2}$$

Thm  $O(n^2)$  algorithm st.

$$\text{if } AB = C$$

$$\Pr[\text{PASS}] = 1$$

$$\text{if } AB \neq C$$

$$\Pr[\text{Fail}] \geq \frac{1}{2}$$

In our example, picking  $j=3$

bad  $r$

$$r_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 6 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} = r'_1$$

$$A \cdot B \cdot r_1 = \begin{pmatrix} 1 & 3 & 4 \\ 3 & 3 & 6 \\ 4 & 8 & 12 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 7 \\ 9 \\ 20 \end{pmatrix} \text{ bad}$$

$$C \cdot r_1 = \begin{pmatrix} 1 & 3 & 4 \\ 3 & 3 & 6 \\ 4 & 12 & 8 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 7 \\ 9 \\ 20 \end{pmatrix}$$

$$r_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \oplus \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = r'_2$$

$$A \cdot B \cdot r'_1 = \begin{pmatrix} 3 \\ 3 \\ 8 \end{pmatrix} \text{ good}$$

$$C \cdot r'_1 = \begin{pmatrix} 3 \\ 3 \\ 12 \end{pmatrix}$$

$$\left. \begin{array}{l} A \cdot B \cdot r_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \\ C \cdot r_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \end{array} \right\} \text{ bad}$$

$$\left. \begin{array}{l} A \cdot B \cdot r'_2 = \begin{pmatrix} 4 \\ 6 \\ 12 \end{pmatrix} \\ C \cdot r'_2 = \begin{pmatrix} 4 \\ 6 \\ 8 \end{pmatrix} \end{array} \right\} \text{ good}$$

## Sanity Checks

• Where did we use that our entries are over a field?  
(showing 1-1)

• what if  $r$  chosen from a larger set?  
ie. vectors with entries from  $0..q$   
how should mapping differ? what happens to error probability?  
• how can we improve error probability?  
(repeat  $K$  times  $\Rightarrow 2^{-K}$  error)

• Monte Carlo or Las Vegas?

small prob of error  
fast on all matrices

Freivald's algorithm with error  $\leq 2^{-k}$ :

Do  $k$  times

pick random  $n$ -vector  $r$  from  $\{0,1\}^n$

if  $A \cdot B \cdot r \neq C \cdot r$  return "not equal"

return "pass"

Why correct?

if  $A \cdot B = C$

always passes

if  $A \cdot B \neq C$

$P_r[\text{survive a single loop}] \leq \frac{1}{2}$

$P_r[\text{survive } k \text{ loops}] \leq \left(\frac{1}{2}\right)^k$

## Checking Polynomial Identities

$$\text{Is } \forall x \underbrace{(x^2+1)^2 (x^2-1)^2 + (x-1)^2}_{p(x)} \equiv \underbrace{x^8 - 2x^4 + 2x^2 - 2x + 2}_{q(x)} ?$$

identity  
↓

$$\text{Is } \forall x \det \begin{pmatrix} 2x^2 & 4x^2+3 \\ 4x & 8x \end{pmatrix} + 12x \equiv 0 ?$$

notation:  $p(x) \equiv q(x)$  if  $\forall x, p(x) = q(x)$

How do you test?

do you need to try all  $x$ ?

Problem:

Given  $p(x), q(x)$  polys of degree  $\leq n$

Output if  $p(x) \equiv q(x)$  output " $\equiv$ "  
if  $\exists x$  st.  $p(x) \neq q(x)$  output " $\neq$ " (with prob  $\geq \frac{1}{2}$ )

Equivalent problem:

Given  $r(x)$  poly of degree  $\leq n$

Output if  $r(x) \equiv 0$  output " $0$ "  
if  $\exists x$  st.  $r(x) \neq 0$ , output "non 0"

(Take  $r(x) = p(x) - q(x)$   
then  $r(x) \equiv 0$  iff  $p(x) \equiv q(x)$ )



# Checking Polynomial Identities

Recall from last time ...

## Crucial Fact

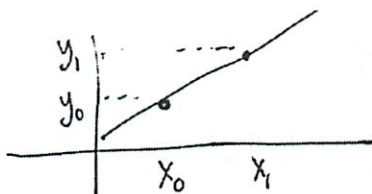
Given inputs  $x_0, \dots, x_n$  distinct, for each possible  $y_0, \dots, y_n$  <sup>do not have to be distinct</sup>  
there is exactly one degree  $\leq n$  poly  $f$  st.

$$\begin{aligned} f(x_0) &= y_0 \\ f(x_1) &= y_1 \\ &\vdots \\ f(x_n) &= y_n \end{aligned}$$

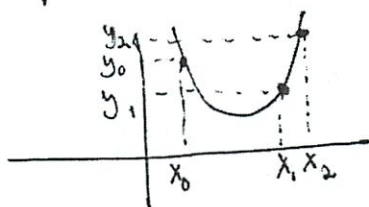
e.g. "2 points determine a line"

given  $x_0 \neq x_1$  &  $y_0, y_1$

there is exactly one line  $l$  st  $l(x_0) = y_0$   
 $l(x_1) = y_1$



3 points determine a quadratic polynomial



### Corollary

if  $f(x_i) = 0 \quad \forall i = 0 \dots n$

then  $f \equiv 0$

is. if  $f \not\equiv 0$  then  $f$  has at most  $n$  roots

↑  
poly of degree  $\leq n$

### Algorithm idea

#### deterministic algorithm

evaluate  $f$  at  $x_0 \dots x_n$  distinct inputs

if  $f(x_i) = 0 \quad \forall i = 0 \dots n$

then  $f$  has

$n+1$  roots  $\Rightarrow f \equiv 0$

else,  $\exists x_i$  st.  $f(x_i) \neq 0$  so  $f \not\equiv 0$

runtime  $O(n)$

#### randomized algorithm

pick  $x \in [1 \dots n]$  randomly

evaluate  $f(x)$

if  $f(x) = 0$  output

" $f \equiv 0$ "

else,

output

" $f \not\equiv 0$ "

runtime  $O(1)$

but, does it work?

Thm a) if  $f(x) \equiv 0$  then always outputs " $f \equiv 0$ "  
b) if  $f(x) \not\equiv 0$  then  $\Pr[\text{outputs } "f \equiv 0"] < 1/4$

pt. a)  $\checkmark$

b) since  $f$  has  $\leq n$  roots,

$$\Pr[\text{output } "f \equiv 0"] = \Pr[X \text{ is a root}] \leq \frac{n}{4n} = \frac{1}{4}$$

## Problem Set 1

This problem set is due **at 11:59pm on Tuesday, September 25, 2012.**

Both exercises and problems should be solved, but *only the problems* should be turned in. Exercises are intended to help you master the course material. Even though you should not turn in the exercise solutions, you are responsible for material covered by the exercises.

Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date and the names of any students with whom you collaborated.

Each problem must be turned in separately to stellar.

You will often be called upon to “give an algorithm” to solve a certain problem. Your write-up should take the form of a short essay. A topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:

1. A description of the algorithm in English and, if helpful, pseudo-code.
2. A proof (or indication) of the correctness of the algorithm.
3. An analysis of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions *which are described clearly*. Convoluting and obtuse descriptions will receive low marks.

---

**Exercise 1-1.** Do Exercise 2.3-3 in CLRS on page 39.

**Exercise 1-2.** Do Exercise 2.3-4 in CLRS on page 39.

**Exercise 1-3.** Do Exercise 3.1-2 in CLRS on page 52.

**Exercise 1-4.** Do Exercise 3.1-3 in CLRS on page 53.

**Exercise 1-5.** Do Exercise 3.1-4 in CLRS on page 53.

**Exercise 1-6.** Do Exercise 4.3-6 in CLRS on page 87.

**Exercise 1-7.** Do Exercise 4.4-8 in CLRS on page 93.

---

### Problem 1-1. Asymptotic Growth

Decide whether these statements are **always true**, **never true**, or **sometimes true** for asymptotically nonnegative functions  $f$  and  $g$ . You must justify all your answers to receive full credit by either giving a short proof (1-2 sentences) or exhibiting a counter-example.



- (a)  $f(n) = \Omega(g(n))$  and  $f(n) = o(g(n))$
- (b)  $f(n) = O(g(n))$  and  $g(n) = o(h(n))$  implies  $h(n) = \omega(f(n))$
- (c)  $f(n) + g(n) = \omega(\max(f(n), g(n)))$
- (d) Rank the following functions by order of growth. In other words, find an arrangement  $g_1, g_2, \dots, g_{16}$  of the functions satisfying  $g_1 = \Omega(g_2)$ ,  $g_2 = \Omega(g_3)$ ,  $\dots$ ,  $g_{15} = \Omega(g_{16})$ . Partition your list into equivalence classes such that  $f(n)$  and  $g(n)$  are in the same class if and only if  $f(n) = \Theta(g(n))$ .

$$\begin{array}{cccc}
 \setminus 2^{n^2} & \setminus \lg n & \setminus 60^{46^{6046}} & \setminus n! \\
 \setminus n & \setminus \sum_{k=1}^n k & \setminus 2^{(2^n)} & \setminus \sqrt{n} \lg n \\
 \setminus \log^2 n & \setminus n2^n & \setminus \log_4 n & \setminus n^{\log n} \\
 \setminus \log \log n & \setminus n^{2 \log n} & \setminus 3^n & \setminus n^2
 \end{array}$$

### Problem 1-2. Recurrences

Give asymptotic upper and lower bounds for  $T(n)$  in each of the following recurrences. For parts (a)–(e), assume that  $T(n)$  is constant for  $n \leq 2$ . Make your bounds as tight as possible, and justify your answers.

- (a)  $T(n) = 3T(n/4) + 5n$
- (b)  $T(n) = 9T(n/3) + n^2$
- (c)  $T(n) = 5T(n/2) + \log n$
- (d)  $T(n) = 4T(\sqrt{n}) + \lg^2 n$
- (e)  $T(n) = T(n-1) + n$

**Problem 1-3. Adding Many Little Numbers**

You have  $n$  numbers that are each a single bit (either 1 or 0). You wish to determine their sum. However, you only have a one-bit adder (with carry). This means, in order to add an  $a$ -bit number and a  $b$ -bit number, you will need to spend  $\Theta(\max(a, b))$  time to add each pair of bits sequentially starting from the least significant bit.

- (a) Assume you simply go through the list adding each bit in turn to a running total. Analyze the running time of this algorithm. What is the upper bound? What can we say about the lower bound? What would you expect the typical running time to be if the 1s and 0s are approximately evenly distributed?
- (b) Give a better algorithm to add the numbers together efficiently, and analyze your algorithm's running time.
- (c) Do you think a different algorithm can perform asymptotically better than the one you presented in part (b)? Why or why not?

**Problem 1-4. Donut Building**

The donut building is shaped like a circle of  $n$  connected rooms such that each classroom is next to two other classrooms. The rooms are numbered 1 through  $n$  clockwise around the building such that the room  $i$  is next to  $i - 1$  and  $i + 1$ , and room  $n$  and room 1 are next to each other. Each classroom  $1 \leq i \leq n$  has a capacity  $v_i$ , which is the maximum number of students who can be seated in the room. Unfortunately, the walls are not soundproof, and it is impossible to have classes in two neighboring classrooms at the same time. We want to maximize the number of students who can attend class at once. To do this, we wish to select a set of rooms of maximum total capacity. Assume that  $v_i \neq v_j$  for  $i \neq j$ .

- (a) Show by example that the "greedy" approach of selecting the highest capacity  $v_i$  and then continuing with what remains does not necessarily select the set of rooms of maximum total capacity.
- (b) Give an efficient algorithm to find the set of rooms with maximal total capacity. What is the run time?

# First Look: P-Set 1

9/17

## Exercises

out of book

not to turn in

and sols are never revealed

L total JIP

I can't just do solved exercises to practice  
of my own

## Problem

Each problem must be turned in separately

Write up like a short essay

1. Summary / problem statement

2. Description of algorithm

Line, pseudo

3. Proof or indication of correctness

??

4. Running time

\* Must describe clearly \*

②

So what skills are on this?

Asy growth

L I still suck at these

But review from last time

Recurrences

I mostly get this

Adding many HS

(divide + conquer

Don't building

that other algorithm we did

like



So we learned all the skills already  
that's a nice lead time...



(3)

Look at some exercises

Recurrences problem

Review notes again

$$T(n) = \begin{cases} \Theta(1) & n=1 \\ 2T(n/2) + \Theta(n) & n>1 \end{cases} \leftarrow \text{bottom}$$

$\log_2 n$  levels

each costing  $\Theta(n \log n)$



asy upper

asy lower

O upper

$\Omega$  lower

Master Theorem

$$T(n) = aT(n/b) + f(n)$$

(never reviewed the easy way...)

(4)

$f(n)$  is the time not doing recurrence

$\frac{a}{b} = \#$  of recursive calls  
like half is  $\frac{n}{2}$

So the first 2 means do for each

Look at  $n^{\lg_b a}$  vs  $f(n)$   
if  $a=b \Rightarrow 1$

$n^c \cdot (\lg n)^d$  for each

In order

1. Compare  $c$ s

if " $c$ s" different  $\rightarrow$  bigger  $c$  wins  
"polynomial difference"

2. Next compare  $d$ s

Slightly different

Answer to recurrence  $\underbrace{T(n)}_{\text{do this}} \cdot \lg n$

do this mean  $f(n)$ ;  
or the  $n^c \cdot (\lg n)^d$   
I think

\* It's  $n^{\lg_b a}$

(5)

The official 3 cases

1. If  $f(n) = O(n^{\lg_b a - \epsilon})$  for some  
easy upper  
bound

constant  $\epsilon > 0$  then  $T(n) = \Theta(n^{\lg_b a})$

So this if they are = ?  
polynomially diff

2. If  $f(n) = \Theta(n^{\lg_b a})$  then ~~then~~

$$T(n) = \Theta(n^{\lg_b a} \lg n)$$

if ns are the same  
slightly diff

(gr my notes from last year are missing details!)

but can see that whenever slightly diff  
its the  $n^{\lg_b a}$

④

3. If  $f(n) = \Omega(n^{\log_b a + \epsilon})$  for  $\epsilon > 0$   
and if  $a f(n/b) \leq c f(n)$   
for some constant  $c < 1$  and sufficiently  
large  $n$ , then  $T(n) = \Theta(f(n))$

So this is comparing  $d$ :

Or this is  $c$ s diff, but larger is  $f(n)$

Since that's what we are taking

## Review Class Notes

Interval scheduling  
Greedy

Ends earliest

How are we supposed to know?

Draw it out and try to figure it out



⑦

Weighted

$w(i)$  for each req

greedy no longer solves

So dynamic programming

I think I generally get how this works

I still don't have a reliable way for runtime  
Just internal feeling...

Can use sorting to  $\downarrow$  initial complexity

Non identical machines

how intractable

8

Proof

induction

- first correct

- subsequent correct  $\rightarrow$  adding 1

Basic Proof Techniques

induction

proof by contradiction

pigeon hole principle

~~Q~~

6042 stuff

Proof by contradiction

if a prop were false  $\rightarrow$  then some false fact  
would be true

Since <sup>the fact</sup> ~~its~~ false, the prop better be true

I don't think I've ever seen it written like that

9)

ex  $\sqrt{2}$  is irrational

Can be written  $\frac{p}{q}$   
(I need to remember defn!)

Suppose  $\sqrt{2}$  is rational

Then we can write  $\sqrt{2} = \frac{n}{d}$  ← pos int lowest terms

If we can't finish yet - have not shown anything  
If we square both sides

$$2 = \frac{n^2}{d^2}$$

$$2d^2 = n^2$$

~~n~~ must be multiple of 2

$n^2$  is

4

But since  $2d^2 = n^2$  we know

$2d^2$  is a multiple of 4

$d^2$

2

So  $n, d$  have 2 as common factor

So not in lowest terms

Contradiction

$\sqrt{2}$  must be irrational

(10)

I don't get this stuff

What if actually rational

eg  $5/9$  is 'irrational'  
↳ false

$$\text{So } \frac{5^2}{9^2} = \frac{5}{9} = \frac{n}{d}$$
$$\frac{25}{81} = \frac{n^2}{d^2}$$

$$25d^2 = 81n^2$$

but  $\sqrt{\quad}$

$$5d = 9n$$

No common factor

So in lowest terms

No contradiction



⑪

## Pigeon hole principle

if  $n$  items  
 $m$  holes

$n > m$  then must double up

(counting argument)

---

Back to problem

---

$$f(i_x) \leq f(j_x)$$

$$x=1 \quad f(i_1) \leq f(j_1)$$

↑ finish time  
of one and another?

$$x+1 \quad f(i_{x+1}) \leq f(j_{x+1})$$

(I don't get what  $f$  is not conflicting w/  $i_x$   
the symbols mean...)

(12)

Induction Claim

$$m > k$$

$\cdot A_{j_{k+1}}$  is not conflicting w/  $S$

$$f(i_k) \leq f(j_k)$$

~~$\phi$~~

Master Theorem

A 3rd way to write in the notes!

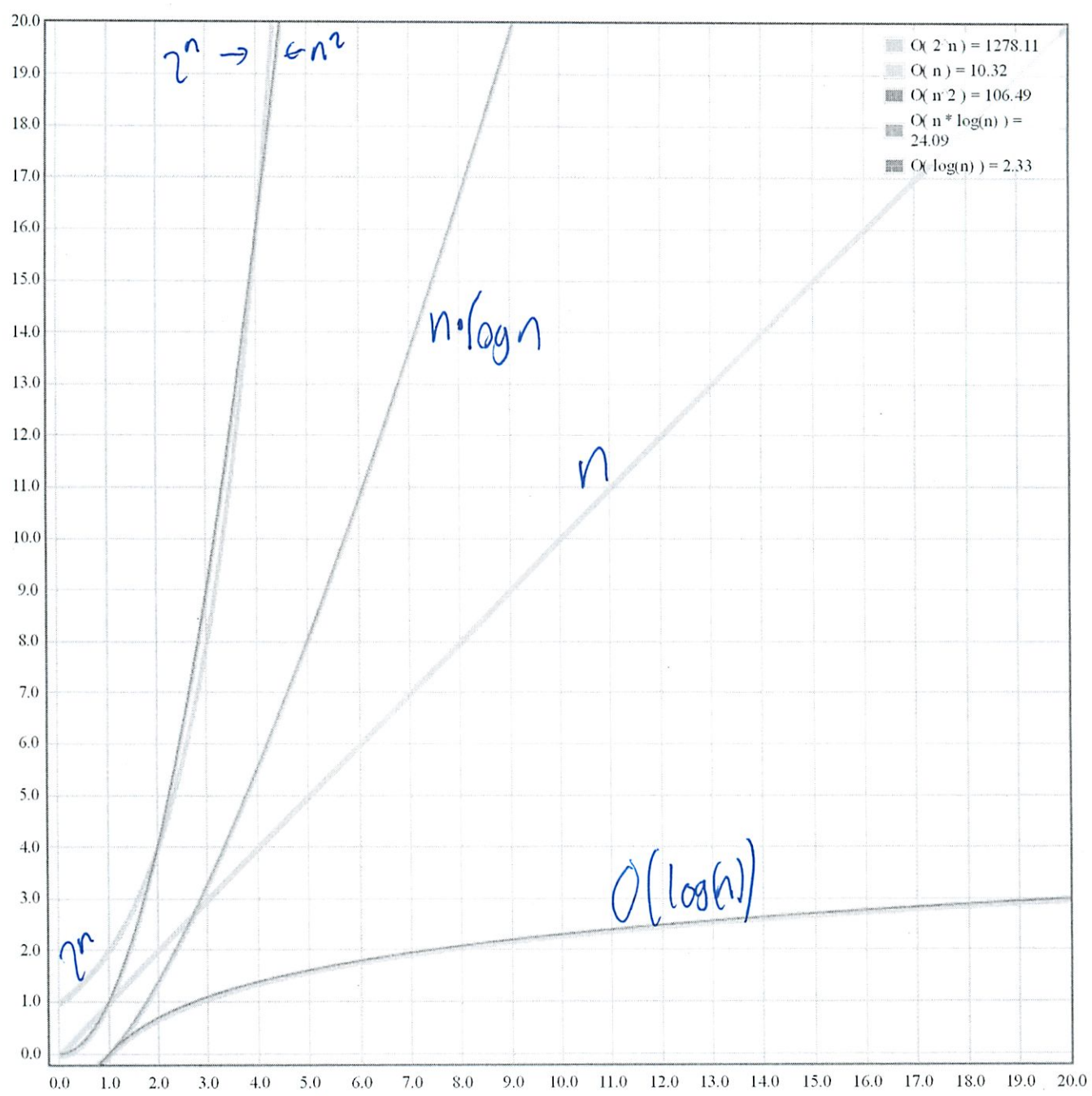
Next time: Review FFT

fun!

O-Notation Visualizer

About

out



↑  
at 2.0

↑  
at 5.0

↑  
at 10

$$2^n = 2$$
$$n^2 = 1$$

$$2^n = 31$$
$$n^2 = 25$$

$$2^n = 1000$$
$$n^2 = 100$$

Read 9/23 opt

# Big O notation

From Wikipedia, the free encyclopedia

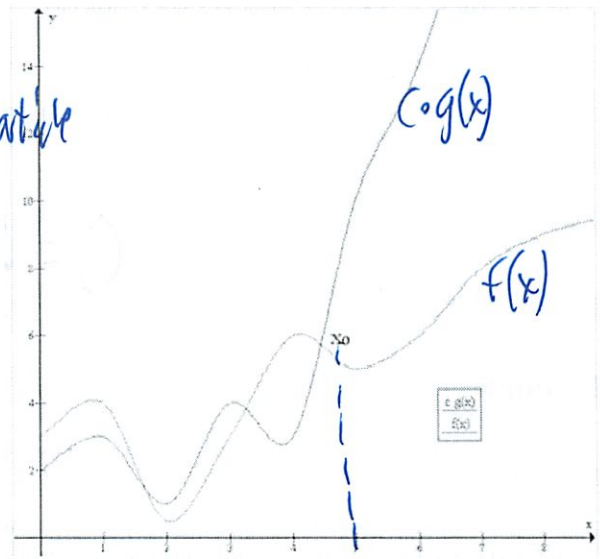
In mathematics, **big O notation** is used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity, usually in terms of simpler functions. It is a member of a larger family of notations that is called **Landau notation**, **Bachmann–Landau notation** (after Edmund Landau and Paul Bachmann), or **asymptotic notation**. In computer science, big O notation is used to classify algorithms by how they respond (e.g., in their processing time or working space requirements) to changes in input size.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols  $o$ ,  $\Omega$ ,  $\omega$ , and  $\Theta$ , to describe other kinds of bounds on asymptotic growth rates.

Big O notation is also used in many other fields to provide similar estimates.

## Contents

- 1 Formal definition
- 2 Example
- 3 Usage
  - 3.1 Infinite asymptotics
  - 3.2 Infinitesimal asymptotics
- 4 Properties
  - 4.1 Product
  - 4.2 Sum
  - 4.3 Multiplication by a constant
- 5 Multiple variables
- 6 Matters of notation
  - 6.1 Equals sign
  - 6.2 Other arithmetic operators
    - 6.2.1 Example
  - 6.3 Declaration of variables
  - 6.4 Multiple usages
- 7 Orders of common functions
- 8 Related asymptotic notations
  - 8.1 Little-o notation
  - 8.2 Family of Bachmann–Landau notations
  - 8.3 Use in computer science
  - 8.4 Extensions to the Bachmann–Landau notations
- 9 Generalizations and related usages
  - 9.1 Graph theory
- 10 History (Bachmann–Landau, Hardy, and Vinogradov notations)
- 11 See also
- 12 References
- 13 Further reading
- 14 External links



Example of Big O notation:  $f(x) \in O(g(x))$  as there exists  $c > 0$  (e.g.  $c = 1$ ) and  $x_0$  (e.g.  $x_0 = 5$ ) such that  $f(x) < c g(x)$  whenever  $x > x_0$ .

Constant Factor?  
after  $x_0$

## Formal definition

Let  $f(x)$  and  $g(x)$  be two functions defined on some subset of the real numbers. One writes

$$f(x) = O(g(x)) \text{ as } x \rightarrow \infty$$

if and only if there is a positive constant  $M$  such that for all sufficiently large values of  $x$ ,  $f(x)$  is at most  $M$  multiplied by  $g(x)$  in absolute value. That is,  $f(x) = O(g(x))$  if and only if there exists a positive real number  $M$  and a real number  $x_0$  such that

$$|f(x)| \leq M|g(x)| \text{ for all } x > x_0.$$

In many contexts, the assumption that we are interested in the growth rate as the variable  $x$  goes to infinity is left unstated, and one writes more



simply that  $f(x) = O(g(x))$ . The notation can also be used to describe the behavior of  $f$  near some real number  $a$  (often,  $a = 0$ ): we say

$$f(x) = O(g(x)) \text{ as } x \rightarrow a$$

if and only if there exist positive numbers  $\delta$  and  $M$  such that

$$|f(x)| \leq M|g(x)| \text{ for } |x - a| < \delta.$$

If  $g(x)$  is non-zero for values of  $x$  sufficiently close to  $a$ , both of these definitions can be unified using the limit superior:

$$f(x) = O(g(x)) \text{ as } x \rightarrow a$$

if and only if

$$\limsup_{x \rightarrow a} \left| \frac{f(x)}{g(x)} \right| < \infty.$$

$O =$  asy upper bound  
Can be tight or not  
 $\Theta =$  not asy tight  
 $\Theta =$  asy tight

## Example

In typical usage, the formal definition of  $O$  notation is not used directly; rather, the  $O$  notation for a function  $f(x)$  is derived by the following simplification rules:

- If  $f(x)$  is a sum of several terms, the one with the largest growth rate is kept, and all others omitted.
- If  $f(x)$  is a product of several factors, any constants (terms in the product that do not depend on  $x$ ) are omitted.

For example, let  $f(x) = 6x^4 - 2x^3 + 5$ , and suppose we wish to simplify this function, using  $O$  notation, to describe its growth rate as  $x$  approaches infinity. This function is the sum of three terms:  $6x^4$ ,  $-2x^3$ , and  $5$ . Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of  $x$ , namely  $6x^4$ . Now one may apply the second rule:  $6x^4$  is a product of  $6$  and  $x^4$  in which the first factor does not depend on  $x$ . Omitting this factor results in the simplified form  $x^4$ . Thus, we say that  $f(x)$  is a big-oh of ( $x^4$ ) or mathematically we can write  $f(x) = O(x^4)$ . One may confirm this calculation using the formal definition: let  $f(x) = 6x^4 - 2x^3 + 5$  and  $g(x) = x^4$ . Applying the formal definition from above, the statement that  $f(x) = O(x^4)$  is equivalent to its expansion,

$$|f(x)| \leq M|g(x)|$$

for some suitable choice of  $x_0$  and  $M$  and for all  $x > x_0$ . To prove this, let  $x_0 = 1$  and  $M = 13$ . Then, for all  $x > x_0$ :

$$\begin{aligned} |6x^4 - 2x^3 + 5| &\leq 6x^4 + |2x^3| + 5 \\ &\leq 6x^4 + 2x^4 + 5x^4 \\ &\leq 13x^4 \\ &\leq 13|x^4| \end{aligned}$$

so

$$|6x^4 - 2x^3 + 5| \leq 13|x^4|.$$

asy tight means only  
differs by constant factor?  
✓ CLRS pg 45

## Usage

Big O notation has two main areas of application. In mathematics, it is commonly used to describe how closely a finite series approximates a given function, especially in the case of a truncated Taylor series or asymptotic expansion. In computer science, it is useful in the analysis of algorithms. In both applications, the function  $g(x)$  appearing within the  $O(\dots)$  is typically chosen to be as simple as possible, omitting constant factors and lower order terms. There are two formally close, but noticeably different, usages of this notation: infinite asymptotics and infinitesimal asymptotics. This distinction is only in application and not in principle, however—the formal definition for the "big O" is the same for both cases, only with different limits for the function argument.

## Infinite asymptotics

Big O notation is useful when analyzing algorithms for efficiency. For example, the time (or the number of steps) it takes to complete a problem of size  $n$  might be found to be  $T(n) = 4n^2 - 2n + 2$ . As  $n$  grows large, the  $n^2$  term will come to dominate, so that all other terms can be neglected — for instance when  $n = 500$ , the term  $4n^2$  is 1000 times as large as the  $2n$  term. Ignoring the latter would have negligible effect on the expression's value for most purposes. Further, the coefficients become irrelevant if we compare to any other order of expression, such as an expression containing a term  $n^3$  or  $n^4$ . Even if  $T(n) = 1,000,000n^2$ , if  $U(n) = n^3$ , the latter will always exceed the former once  $n$  grows larger than  $1,000,000$  ( $T(1,000,000) = 1,000,000^3 = U(1,000,000)$ ). Additionally, the number of steps depends on the details of the machine model on which the algorithm runs, but different types of machines typically vary by only a constant factor in the number of steps needed to execute an algorithm. So the big O notation captures what



remains: we write either

$$T(n) = O(n^2)$$

*at most*

or

$$T(n) \in O(n^2)$$

and say that the algorithm has *order of*  $n^2$  time complexity. Note that "=" is not meant to express "is equal to" in its normal mathematical sense, but rather a more colloquial "is", so the second expression is technically accurate (see the "Equals sign" discussion below) while the first is a common abuse of notation.<sup>[1]</sup>

## Infinitesimal asymptotics

Big O can also be used to describe the error term in an approximation to a mathematical function. The most significant terms are written explicitly, and then the least-significant terms are summarized in a single big O term. For example,

$$e^x = 1 + x + \frac{x^2}{2} + O(x^3) \quad \text{as } x \rightarrow 0$$

expresses the fact that the error, the difference  $e^x - (1 + x + x^2/2)$ , is smaller in absolute value than some constant times  $|x^3|$  when  $x$  is close enough to 0.

## Properties

If a function  $f(n)$  can be written as a finite sum of other functions, then the fastest growing one determines the order of  $f(n)$ . For example

$$f(n) = 9 \log n + 5(\log n)^3 + 3n^2 + 2n^3 = O(n^3).$$

In particular, if a function may be bounded by a polynomial in  $n$ , then as  $n$  tends to *infinity*, one may disregard *lower-order* terms of the polynomial.  $O(n^c)$  and  $O(n^d)$  are very different. If  $c$  is greater than one, then the latter grows much faster. A function that grows faster than  $n^c$  for any  $c$  is called *superpolynomial*. One that grows more slowly than any exponential function of the form  $e^n$  is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential; examples of this include the fastest known algorithms for integer factorization.  $O(\log n)$  is exactly the same as  $O(\log(n^c))$ . The logarithms differ only by a constant factor (since  $\log(n^c) = c \log n$ ) and thus the big O notation ignores that. Similarly, logs with different constant bases are equivalent. Exponentials with different bases, on the other hand, are not of the same order. For example,  $2^n$  and  $3^n$  are not of the same order. Changing units may or may not affect the order of the resulting algorithm. Changing units is equivalent to multiplying the appropriate variable by a constant wherever it appears. For example, if an algorithm runs in the order of  $n^2$ , replacing  $n$  by  $cn$  means the algorithm runs in the order of  $c^2 n^2$ , and the big O notation ignores the constant  $c^2$ . This can be written as  $c^2 n^2 \in O(n^2)$ . If, however, an algorithm runs in the order of  $2^n$ , replacing  $n$  with  $cn$  gives  $2^{cn} = (2^c)^n$ . This is not equivalent to  $2^n$  in general. Changing of variable may affect the order of the resulting algorithm. For example, if an algorithm's running time is  $O(n)$  when measured in terms of the number  $n$  of *digits* of an input number  $x$ , then its running time is  $O(\log x)$  when measured as a function of the input number  $x$  itself, because  $n = \Theta(\log x)$ .

## Product

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 f_2 \in O(g_1 g_2)$$

$$f \cdot O(g) \subset O(fg)$$

## Sum

$$f_1 \in O(g_1) \text{ and } f_2 \in O(g_2) \Rightarrow f_1 + f_2 \in O(|g_1| + |g_2|)$$

This implies  $f_1 \in O(g)$  and  $f_2 \in O(g) \Rightarrow f_1 + f_2 \in O(g)$ , which means that  $O(g)$  is a convex cone.

If  $f$  and  $g$  are positive functions,  $f + O(g) \in O(f + g)$

## Multiplication by a constant

Let  $k$  be a constant. Then:

$$O(kg) = O(g) \text{ if } k \text{ is nonzero.}$$

$$f \in O(g) \Rightarrow kf \in O(g).$$

## Multiple variables

Big O (and little o, and  $\Omega$ ...) can also be used with multiple variables. To define Big O formally for multiple variables, suppose  $f(\vec{x})$  and  $g(\vec{x})$  are two functions defined on some subset of  $\mathbb{R}^n$ . We say

*I think I get this better now*

$$f(\vec{x}) \text{ is } O(g(\vec{x})) \text{ as } \vec{x} \rightarrow \infty$$

if and only if

$$\exists C \exists M > 0 \text{ such that } |f(\vec{x})| \leq C|g(\vec{x})| \text{ for all } \vec{x} \text{ with } x_i > M \text{ for all } i.$$

For example, the statement

$$f(n, m) = n^2 + m^3 + O(n + m) \text{ as } n, m \rightarrow \infty$$

asserts that there exist constants  $C$  and  $M$  such that

$$\forall n, m > M: |g(n, m)| \leq C(n + m),$$

where  $g(n, m)$  is defined by

$$f(n, m) = n^2 + m^3 + g(n, m).$$

Note that this definition allows all of the coordinates of  $\vec{x}$  to increase to infinity. In particular, the statement

$$f(n, m) = O(n^m) \text{ as } n, m \rightarrow \infty$$

(i.e.,  $\exists C \exists M \forall n \forall m \dots$ ) is quite different from

$$\forall m: f(n, m) = O(n^m) \text{ as } n \rightarrow \infty$$

(i.e.,  $\forall m \exists C \exists M \forall n \dots$ ).

## Matters of notation

### Equals sign

The statement " $f(x)$  is  $O(g(x))$ " as defined above is usually written as  $f(x) = O(g(x))$ . Some consider this to be an abuse of notation, since the use of the equals sign could be misleading as it suggests a symmetry that this statement does not have. As de Bruijn says,  $O(x) = O(x^2)$  is true but  $O(x^2) = O(x)$  is not.<sup>[2]</sup> Knuth describes such statements as "one-way equalities", since if the sides could be reversed, "we could deduce ridiculous things like  $n = n^2$  from the identities  $n = O(n^2)$  and  $n^2 = O(n^2)$ ."<sup>[3]</sup> For these reasons, it would be more precise to use set notation and write  $f(x) \in O(g(x))$ , thinking of  $O(g(x))$  as the class of all functions  $h(x)$  such that  $|h(x)| \leq C|g(x)|$  for some constant  $C$ .<sup>[3]</sup> However, the use of the equals sign is customary. Knuth pointed out that "mathematicians customarily use the = sign as they use the word 'is' in English: Aristotle is a man, but a man isn't necessarily Aristotle."<sup>[4]</sup>

### Other arithmetic operators

Big O notation can also be used in conjunction with other arithmetic operators in more complicated equations. For example,  $h(x) + O(f(x))$  denotes the collection of functions having the growth of  $h(x)$  plus a part whose growth is limited to that of  $f(x)$ . Thus,

$$g(x) = h(x) + O(f(x))$$

expresses the same as

$$g(x) - h(x) \in O(f(x)).$$

### Example

Suppose an algorithm is being developed to operate on a set of  $n$  elements. Its developers are interested in finding a function  $T(n)$  that will express how long the algorithm will take to run (in some arbitrary measurement of time) in terms of the number of elements in the input set. The algorithm works by first calling a subroutine to sort the elements in the set and then perform its own operations. The sort has a known time complexity of  $O(n^2)$ , and after the subroutine runs the algorithm must take an additional  $55n^3 + 2n + 10$  time before it terminates. Thus the overall time complexity of the algorithm can be expressed as

$$T(n) = O(n^2) + 55n^3 + 2n + 10.$$

This can perhaps be most easily read by replacing  $O(n^2)$  with "some function that grows asymptotically no faster than  $n^2$ ". Again, this usage disregards some of the formal meaning of the "=" and "+" symbols, but it does allow one to use the big O notation as a kind of convenient placeholder.

### Declaration of variables



Another feature of the notation, although less exceptional, is that function arguments may need to be inferred from the context when several variables are involved. The following two right-hand side big O notations have dramatically different meanings:

$$\begin{aligned} f(m) &= O(m^n), \\ g(n) &= O(m^n). \end{aligned}$$

The first case states that  $f(m)$  exhibits polynomial growth, while the second, assuming  $m > 1$ , states that  $g(n)$  exhibits exponential growth. To avoid confusion, some authors use the notation

$$g(x) \in O(f(x)).$$

rather than the less explicit

$$g \in O(f),$$

Multiple usages

In more complicated usage,  $O(\dots)$  can appear in different places in an equation, even several times on each side. For example, the following are true for  $n \rightarrow \infty$

$$\begin{aligned} (n+1)^2 &= n^2 + O(n) \\ (n + O(n^{1/2}))(n + O(\log n))^2 &= n^3 + O(n^{5/2}) \\ n^{O(1)} &= O(e^n). \end{aligned}$$

The meaning of such statements is as follows: for *any* functions which satisfy each  $O(\dots)$  on the left side, there are *some* functions satisfying each  $O(\dots)$  on the right side, such that substituting all these functions into the equation makes the two sides equal. For example, the third equation above means: "For any function  $f(n) = O(1)$ , there is some function  $g(n) = O(e^n)$  such that  $n^{f(n)} = g(n)$ ." In terms of the "set notation" above, the meaning is that the class of functions represented by the left side is a subset of the class of functions represented by the right side. In this use the "=" is a formal symbol that unlike the usual use of "=" is not a symmetric relation. Thus for example  $n^{O(1)} = O(e^n)$  does not imply the false statement  $O(e^n) = n^{O(1)}$ .

Orders of common functions

Further information: Time complexity#Table of common time complexities

Here is a list of classes of functions that are commonly encountered when analyzing the running time of an algorithm. In each case,  $c$  is a constant and  $n$  increases without bound. The slower-growing functions are generally listed first.

Notation	Name	Example
$O(1)$	constant	Determining if a number is even or odd; using a constant-size lookup table
$O(\log \log n)$	double logarithmic	Finding an item using interpolation search in a sorted array of uniformly distributed values.
$O(\log n)$	logarithmic	Finding an item in a sorted array with a binary search or a balanced search tree as well as all operations in a Binomial heap.
$O(n^c), 0 < c < 1$	fractional power	Searching in a kd-tree
$O(n)$	linear	Finding an item in an unsorted list or a malformed tree (worst case) or in an unsorted array; Adding two $n$ -bit integers by ripple carry.
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear	Performing a Fast Fourier transform; heapsort, quicksort (best and average case), or merge sort
$O(n^2)$	quadratic	Multiplying two $n$ -digit numbers by a simple algorithm; bubble sort (worst case or naive implementation), Shell sort, quicksort (worst case), selection sort or insertion sort
$O(n^c), c > 1$	polynomial or algebraic	Tree-adjointing grammar parsing; maximum matching for bipartite graphs
$L_n[\alpha, c], 0 < \alpha < 1 = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	L-notation or sub-exponential	Factoring a number using the quadratic sieve or number field sieve
$O(c^n), c > 1$	exponential	Finding the (exact) solution to the travelling salesman problem using dynamic programming; determining if two logical statements are equivalent using brute-force search
$O(n!)$	factorial	Solving the traveling salesman problem via brute-force search; generating all unrestricted permutations of a poset; finding the determinant with expansion by minors.

The statement  $f(n) = O(n!)$  is sometimes weakened to  $f(n) = O(n^k)$  to derive simpler formulas for asymptotic complexity. For any  $k > 0$

and  $c > 0$ ,  $O(n^c(\log n)^k)$  is a subset of  $O(n^{c+\epsilon})$  for any  $\epsilon > 0$ , so may be considered as a polynomial with some bigger order.

## Related asymptotic notations

Big  $O$  is the most commonly used asymptotic notation for comparing functions, although in many cases Big  $O$  may be replaced with Big Theta  $\Theta$  for asymptotically tighter bounds. Here, we define some related notations in terms of Big  $O$ , progressing up to the family of Bachmann–Landau notations to which Big  $O$  notation belongs.

### Little-o notation

The relation  $f(x) \in o(g(x))$  is read as " $f(x)$  is little-o of  $g(x)$ ". Intuitively, it means that  $g(x)$  grows much faster than  $f(x)$ , or similarly, the growth of  $f(x)$  is nothing compared to that of  $g(x)$ . It assumes that  $f$  and  $g$  are both functions of one variable. Formally,  $f(n) = o(g(n))$  as  $n \rightarrow \infty$  means that for every positive constant  $\epsilon$  there exists a constant  $N$  such that

$$|f(n)| \leq \epsilon |g(n)| \quad \text{for all } n \geq N. \quad [3]$$

Note the difference between the earlier formal definition for the big- $O$  notation, and the present definition of little- $o$ : while the former has to be true for *at least one* constant  $M$  the latter must hold for *every* positive constant  $\epsilon$ , however small.<sup>[1]</sup> In this way little- $o$  notation makes a stronger statement than the corresponding big- $O$  notation: every function that is little- $o$  of  $g$  is also big- $O$  of  $g$ , but not every function that is big- $O$  of  $g$  is also little- $o$  of  $g$  (for instance  $g$  itself is not, unless it is identically zero near  $\infty$ ).

If  $g(x)$  is nonzero, or at least becomes nonzero beyond a certain point, the relation  $f(x) = o(g(x))$  is equivalent to

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = 0.$$

For example,


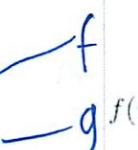
- $2x \in o(x^2)$
- $2x^2 \notin o(x^2)$
- $1/x \in o(1)$

Little- $o$  notation is common in mathematics but rarer in computer science. In computer science the variable (and function value) is most often a natural number. In mathematics, the variable and function values are often real numbers. The following properties can be useful:

- $o(f) + o(f) \subseteq o(f)$
- $o(f)o(g) \subseteq o(fg)$
- $o(o(f)) \subseteq o(f)$
- $o(f) \subset O(f)$  (and thus the above properties apply with most combinations of  $o$  and  $O$ ).

As with big  $O$  notation, the statement " $f(x)$  is  $o(g(x))$ " is usually written as  $f(x) = o(g(x))$ , which is a slight abuse of notation.

### Family of Bachmann–Landau notations

Notation	Name	Intuition	Informal definition: for sufficiently large $n$ ...	Formal Definition	Notes
 $f(n) \in O(g(n))$ <i>f is below g</i>	Big Omicron; Big O; Big Oh	$f$ is bounded above by $g$ (up to constant factor) asymptotically	$ f(n)  \leq g(n) \cdot k$ for some $k$	$\exists k > 0 \exists n_0 \forall n > n_0  f(n)  \leq  g(n)  \cdot k$ or $\exists k > 0 \exists n_0 \forall n > n_0 f(n) \leq g(n) \cdot k$	
 $f(n) \in \Omega(g(n))$	Big Omega	$f$ is bounded below by $g$ (up to constant factor) asymptotically	$f(n) \geq g(n) \cdot k$ for some positive $k$	$\exists k > 0 \exists n_0 \forall n > n_0 g(n) \cdot k \leq f(n)$	Since the beginning of the 20th century, papers in number theory have been increasingly and widely using this notation in



					the weaker sense that $f = o(g)$ is false.
$f(n) \in \Theta(g(n))$	Big Theta	$f$ is bounded both above and below by $g$ asymptotically	$g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$ for some positive $k_1, k_2$	$\exists k_1 > 0 \exists k_2 > 0 \exists n_0 \forall n > n_0$ $g(n) \cdot k_1 \leq f(n) \leq g(n) \cdot k_2$	
$f(n) \in o(g(n))$	Small Omicron; Small O; Small Oh	$f$ is dominated by $g$ asymptotically	$ f(n)  \leq  g(n)  \cdot \varepsilon$ for every $\varepsilon$	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0  f(n)  \leq  g(n)  \cdot \varepsilon$	
$f(n) \in \omega(g(n))$	Small Omega	$f$ dominates $g$ asymptotically	$f(n) \geq g(n) \cdot k$ for every $k$	$\forall k > 0 \exists n_0 \forall n > n_0 g(n) \cdot k < f(n)$	
$f(n) \sim g(n)$	On the order of	$f$ is equal to $g$ asymptotically	$f(n)/g(n) \rightarrow 1$	$\forall \varepsilon > 0 \exists n_0 \forall n > n_0 \left  \frac{f(n)}{g(n)} - 1 \right  < \varepsilon$	

Bachmann–Landau notation was designed around several mnemonics, as shown in the *As  $n \rightarrow \infty$ , eventually...* column above and in the bullets below. To conceptually access these mnemonics, "omicron" can be read "o-micron" and "omega" can be read "o-mega". Also, the lower-case versus capitalization of the Greek letters in Bachmann–Landau notation is mnemonic.

- The *o-micron mnemonic*: The o-micron reading of  $f(n) \in O(g(n))$  and of  $f(n) \in o(g(n))$  can be thought of as "O-smaller than" and "o-smaller than", respectively. This *micro*/smaller mnemonic refers to: for sufficiently large input parameter(s),  $f$  grows at a rate that may henceforth be **less** than  $cg$  regarding  $g \in O(f)$  or  $g \in o(f)$ .
- The *o-mega mnemonic*: The o-mega reading of  $f(n) \in \Omega(g(n))$  and of  $f(n) \in \omega(g(n))$  can be thought of as "O-larger than". This *mega*/larger mnemonic refers to: for sufficiently large input parameter(s),  $f$  grows at a rate that may henceforth be **greater** than  $cg$  regarding  $g \in \Omega(f)$  or  $g \in \omega(f)$ .
- The *upper-case mnemonic*: This mnemonic reminds us when to use the upper-case Greek letters in  $f(n) \in O(g(n))$  and  $f(n) \in \Omega(g(n))$ : for sufficiently large input parameter(s),  $f$  grows at a rate that may henceforth be **equal** to  $cg$  regarding  $g \in O(f)$ .
- The *lower-case mnemonic*: This mnemonic reminds us when to use the lower-case Greek letters in  $f(n) \in o(g(n))$  and  $f(n) \in \omega(g(n))$ : for sufficiently large input parameter(s),  $f$  grows at a rate that is henceforth **inequal** to  $cg$  regarding  $g \in O(f)$ .

Aside from Big O notation, the Big Theta  $\Theta$  and Big Omega  $\Omega$  notations are the two most often used in computer science; the Small Omega  $\omega$  notation is rarely used in computer science.

## Use in computer science

For more details on this topic, see *Analysis of algorithms*.

Informally, especially in computer science, the Big O notation often is permitted to be somewhat abused to describe an asymptotic tight bound where using Big Theta  $\Theta$  notation might be more factually appropriate in a given context. For example, when considering a function  $T(n) = 73n^3 + 22n^2 + 58$ , all of the following are generally acceptable, but tightnesses of bound (i.e., numbers 2 and 3 below) are usually strongly preferred over laxness of bound (i.e., number 1 below).

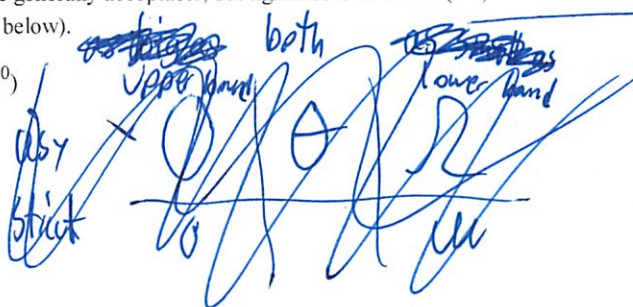
1.  $T(n) = O(n^{100})$ , which is identical to  $T(n) \in O(n^{100})$
2.  $T(n) = O(n^3)$ , which is identical to  $T(n) \in O(n^3)$
3.  $T(n) \in \Theta(n^3)$ , which is identical to  $T(n) \in \Theta(n^3)$ .

The equivalent English statements are respectively:

1.  $T(n)$  grows asymptotically no faster than  $n^{100}$
2.  $T(n)$  grows asymptotically no faster than  $n^3$
3.  $T(n)$  grows asymptotically as fast as  $n^3$ .

So while all three statements are true, progressively more information is contained in each. In some fields, however, the Big O notation (number 2 in the lists above) would be used more commonly than the Big Theta notation (bullets number 3 in the lists above) because functions that grow more slowly are more desirable. For example, if  $T(n)$  represents the running time of a newly developed algorithm for input size  $n$ , the inventors and users of the algorithm might be more inclined to put an upper asymptotic bound on how long it will take to run without making an explicit statement about the lower asymptotic bound.

## Extensions to the Bachmann–Landau notations



What would  $o$  be?

$g$  grows faster than  $f$



Another notation sometimes used in computer science is  $\tilde{O}$  (read *soft-O*):  $f(n) = \tilde{O}(g(n))$  is shorthand for  $f(n) = O(g(n) \log^k g(n))$  for some  $k$ . Essentially, it is Big O notation, ignoring logarithmic factors because the growth-rate effects of some other super-logarithmic function indicate a growth-rate explosion for large-sized input parameters that is more important to predicting bad run-time performance than the finer-point effects contributed by the logarithmic-growth factor(s). This notation is often used to obviate the "nitpicking" within growth-rates that are stated as too tightly bounded for the matters at hand (since  $\log^k n$  is always  $o(n^\varepsilon)$  for any constant  $k$  and any  $\varepsilon > 0$ ). The L notation, defined as

$$L_n[\alpha, c] = O\left(e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}\right),$$

is convenient for functions that are between polynomial and exponential.

## Generalizations and related usages

The generalization to functions taking values in any normed vector space is straightforward (replacing absolute values by norms), where  $f$  and  $g$  need not take their values in the same space. A generalization to functions  $g$  taking values in any topological group is also possible. The "limiting process"  $x \rightarrow x_0$  can also be generalized by introducing an arbitrary filter base, i.e. to directed nets  $f$  and  $g$ . The  $o$  notation can be used to define derivatives and differentiability in quite general spaces, and also (asymptotical) equivalence of functions,

$$f \sim g \iff (f - g) \in o(g)$$

which is an equivalence relation and a more restrictive notion than the relationship " $f$  is  $\Theta(g)$ " from above. (It reduces to  $\lim f/g = 1$  if  $f$  and  $g$  are positive real valued functions.) For example,  $2x$  is  $\Theta(x)$ , but  $2x - x$  is not  $o(x)$ .

## Graph theory

It is often useful to bound the running time of graph algorithms. Unlike most other computational problems, for a graph  $G = (V, E)$  there are two relevant parameters describing the size of the input: the number  $|V|$  of vertices in the graph and the number  $|E|$  of edges in the graph. Inside asymptotic notation (and only there), it is common to use the symbols  $V$  and  $E$ , when someone really means  $|V|$  and  $|E|$ . This convention simplifies asymptotic functions and make them easily readable. The symbols  $V$  and  $E$  are never used inside asymptotic notation with their literal meaning, since the number of vertices and edges must be non-negative, so this abuse of notation does not risk ambiguity. For example  $O(E + V \log V)$  means  $O((V, E) \mapsto |E| + |V| \cdot \log |V|)$  for a suitable metric of graphs. Another common convention—referring to the values  $|V|$  and  $|E|$  by the names  $n$  and  $m$ , respectively—sidesteps this ambiguity.

## History (Bachmann–Landau, Hardy, and Vinogradov notations)

The notation was first introduced by number theorist Paul Bachmann in 1894, in the second volume of his book *Analytische Zahlentheorie* ("analytic number theory"), the first volume of which (not yet containing big O notation) was published in 1892.<sup>[5]</sup> The notation was popularized in the work of number theorist Edmund Landau; hence it is sometimes called a Landau symbol. It was popularized in computer science by Donald Knuth, who re-introduced the related Omega and Theta notations.<sup>[6]</sup> Knuth also noted that the Omega notation had been introduced by Hardy and Littlewood<sup>[7]</sup> under a different meaning " $\neq o$ " (i.e. "is not an  $o$  of"), and proposed the above definition. Hardy and Littlewood's original definition (which was also used in one paper by Landau<sup>[8]</sup>) is still used in number theory.

Hardy's symbols were (in terms of the modern  $O$  notation)

$$f \preceq g \iff f \in O(g) \text{ and } f \prec g \iff f \in o(g);$$

(Hardy however never defined or used the notation  $\ll$ , nor  $\lll$ , as it has been sometimes reported). It should also be noted that Hardy introduces the symbols  $\preceq$  and  $\prec$  (as well as some other symbols) in his 1910 tract "Orders of Infinity", and makes use of it only in three papers (1910–1913). In the remaining papers (nearly 400!) and books he constantly uses the Landau symbols  $O$  and  $o$ .

Hardy's notation is not used anymore. On the other hand, in 1947, the Russian number theorist Ivan Matveyevich Vinogradov introduced his notation  $\ll$ , which has been increasingly used in number theory instead of the  $O$  notation. We have

$$f \ll g \iff f \in O(g),$$

and frequently both notations are used in the same paper.

The big-O, standing for "order of", was originally a capital omicron; today the identical-looking Latin capital letter O is used, but never the digit zero.

## See also

- Asymptotic expansion: Approximation of functions generalizing Taylor's formula
- Asymptotically optimal: A phrase frequently used to describe an algorithm that has an upper bound asymptotically within a constant of a lower bound for the problem
- Limit superior and limit inferior: An explanation of some of the limit notation used in this article
- Nachbin's theorem: A precise method of bounding complex analytic functions so that the domain of convergence of integral transforms can be

# Asy Notation

8/23

$$f(n) = O(g(n))$$

asy upper

$$0 \leq f(n) \leq c g(n)$$

$$f(n) = O(g(n))$$

upper

$$0 \leq f(n) < c g(n)$$

$$f(n) = \Omega(g(n))$$

asy lower

$$0 \leq c g(n) \leq f(n)$$

$$f(n) = \omega(g(n))$$

a lower

$$0 \leq c g(n) < f(n)$$

$$f(n) = \Theta(g(n))$$

asy tight upper + lower

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Must do Latek? — Can do but written acceptable


# 1. Asy Growth (Reading)

a)  $f(n) = \Omega(g(n))$        $f(n) = o(g(n))$

True always      Sometimes  
never

$\Omega$  is asy lower bound  $f$  —  $g$  —  $f$  must be above  $g$   
or not  
 $o$  is a non-asy upper bound

$g$  must be above  $f$   
non asy





②

Is it:

	$f < g$ upper bound	both	$g < f$ lower bound
=	asy tight	$\Theta$	
$\leq$	<del>asy</del> either asy	$O$	$\Omega$
$<$	non asy tight	$\Theta$	$\Omega$

but must be off by  
constant factor

Then what is  $O$ :

$$0 \leq f(n) < c g(n)$$

$$O) \quad 0 \leq f(n) \leq c g(n)$$

Then what is

$$0 \leq f(n) \leq c g(n)$$

asy tight upper  
not defined

posted on Piazza

③

Any way back to problem

$$2 \quad 0 \leq cg(n) \leq f(n)$$

$$0 \quad 0 \leq f(n) < cg(n)$$

↑ Forget this

Simplify

$$a \leq b$$

$$b < a$$

“Though things can  
change over time”  
So not really valid”

~~but~~

but from a factor pair

No → can't be true

↑  
I think it is  
valid...

4

b)  $f(n) = O(g(n))$

$$0 \leq f(n) \leq c g(n)$$

$$g(n) = o(h(n))$$

$$0 \leq \cancel{g(n)} \overset{g}{f}(n) < c \cancel{g}(n)$$

$$h(n) = w(f(n))$$

$$0 \leq c \cancel{g}(n) < \cancel{h}(n)$$

So

$$\cancel{f \leq g}$$

$$\cancel{g < h}$$

$$\cancel{h < f}$$

$$\cancel{f \leq g < h}$$
$$\cancel{h < f}$$

No

even if  $f=g$   $g \not\leq h$

5

Wait Redo Ordering  
- didn't pay attention

$$f \leq g$$

$$g < h$$

$$f < h$$

$$f \leq g < h$$

$$f < h?$$

always



⑥ c)  $f(n) + g(n) = O(\max(f(n), g(n)))$

$$0 \leq C \max(f, g) < f + g$$

non reg

$C$  can be 0

if  $f = 0$

$$g < g \quad \otimes \text{ No}$$

Sometimes true

d) Piazza: Forget true + false

Ad Rank by order of growth

$$g_1 = \Omega(g_2)$$

" $g_1$  is a <sup>asy</sup> lower bound to  $g_2$ "

$$0 \leq C g(n) \leq f(n)$$

$$g_2 \leq g_1$$

~~So largest to smallest~~

Smallest to largest

7

I don't think asy signifies any thing  
as in

asy tight upper bound =

vs  
asy upper bound  $\leq$

vs  
Upper bound  $<$

Asked Tanya

So my table  $\Rightarrow$  

1st + 2nd rows no distinction

asy tight just means both upper + lower  
~~asy~~

8

So then

	$f < g$ Upper	both tight	$g < f$ lower
$\leq$ asy bound	$O$	$\Theta$	$\Omega$
$<$ bound	$o$		$u$

But then p50 confuses it

~~lower~~  $\Omega(n^2) = O(n^2)$  is asy tight

So then that actually goes back to 1st chart

	upper	both	lower
$=$	$\rightarrow$	$\Theta$	$\leftarrow$
$\leq$	$O$	$\times$	$\Omega$
$<$	$o$	$\times$	$u$

But add arrows + xs to reflect  
impossibilities

-feel much better about this now

9

11

d) Rank

Wait  $g_1 = \lambda(g_2)$   $g_2 \leq g_1$   
= ?  
 $g_2 = \phi(g_1)$   $g_2 \leq g_1$   
Yes - is same  
So  $g_2$  is larger than  $g_1$

Would they want  $W, 0$  since  
want = differently  
 $\angle \theta$



10

Wikipedia Chart has it

Loh claims not in order

$$\lg(\lg(n)) \rightarrow \lg(2046) = \text{constant}$$

$$\ln \lg n \rightarrow \lg n = \lg n$$

$$\lg^2(n) \rightarrow n = \cancel{2046} \quad \cancel{2046} \quad n \lg n$$

$$n^2 \quad \sum_{k=1}^n k$$

$$2^{n^2}$$

$$3^n$$

$$n! \leftarrow n 2^n$$

$$2^{2n}$$

$$2^{n^2}$$

Question is  $\rightarrow$  do they differ by a constant?

(10)

~~log~~  $\lg n$  vs  $\lg_y n$

$$\frac{\log(n)}{\log(10)} \text{ vs } \frac{\log(n)}{\log(y)}$$

$\therefore$  same  $\frac{1}{\log 10}$  vs  $\frac{1}{\log y}$  = constant factor  
- (ignore)

---

$$n^{\lg n} \stackrel{?}{=} n$$

↳ No

Actually as  $y$  yes  
past  $x=1$ .

---

$$2^{(2^n)}$$

do paren lot  $2^n$

~~like~~ is expon

2 raised to that is double exp!

WA  $\rightarrow$  Not same

17

$$n! \text{ vs } 2^{2^n} \text{ vs } 2^n$$

b/w

---

$$\sqrt{n} \lg n$$

multiplies  $\lg n$

but still  $\leq n$

---

$$\lg_y n$$

is  $\frac{\lg(n)}{\lg(y)}$

A constant factor  $\frac{1}{\lg(y)}$

Oh did already

---

$$n 2^n$$

is  $2^{(n+1)}$  ;

(13)

say  $n=5$

$$5 \cdot 2^5$$

$$\frac{5}{32} \cdot 2^{10} = 160$$

$$\text{not } 2^6 = 62$$

So no

less than  $2^{2n}$

after  $n!$

---

$$n^{2 \log n}$$

asy diff from  $n^{\log n}$

Since ~~it~~ not a constant factor

---

$$\sum_{k=1}^n k$$

So this is  $1+2+3+4+5+6$

(I suck at moving these to def form)

$$\text{WA: } \frac{1}{2} n(n+1)$$

I remember from 6.042



(14)

This is  $n^2$

---

$$2^{n^2}$$

more than  $n^2$

~~but~~

Much more than  $3^n$

||

$$n^{2^n}$$

$$2^{2^n}$$

(How would you  
figure out w/o  
WA  
- try it -)

---

$$\log^2(n)$$

is this  $n$  - no  
less than  $n$

more than  $\lg(n)$

---

$$\lg(\lg(n))$$

is that how parenthesis - pretty sure

(15)

less than  $\log^2 n$

less than  $\log n$

less than  $\ln \log n$

more than 1

---

Done!

Don't show work - just explain

---

Wait ~~the~~  $n^{\log n}$

is not  $n$

$$10^{\log 10} = 200$$

---

$2^{n^2}$  is twice

## Michael E Plasmeier

---

**From:** Katherine Fang <katfang@MIT.EDU>  
**Sent:** Sunday, September 23, 2012 4:37 PM  
**To:** Michael E Plasmeier  
**Cc:** 6046-tas@mit.edu  
**Subject:** Re: [6046-tas] What does it mean exactly to be an asymptotically tight bound?

Hi Michael,

You're correct in that an asymptotically tight bound refers to  $\Theta$  and that  $O$  is a bound that may not be tight. It's common to use  $O$  as an asymptotically tight upper bound in algorithms, but  $\Theta$  is more appropriate.

There is no notion of as an asymptotically tight upper-only bound. One way to think about this is say you have the function  $f(n) = 3 \cdot n^3$ . Then the asymptotically tight upper bound  $O(f) = n^3$ . That is to say,  $c \cdot n^3 \geq 3 \cdot n^3$  for some  $c$ . If it's truly an asymptotic upper bound, then it would follow extremely close (but above) to the function for large  $n$ . Were you to decrease this by a factor of 2, you would suddenly get a lower bound.

-- Katherine

On Sun, Sep 23, 2012 at 3:59 PM, Michael E Plasmeier <[theplaz@mit.edu](mailto:theplaz@mit.edu)> wrote:

What does it mean exactly to be an asymptotically tight bound?

$\Theta$  is a asy. tight upper and lower bound.  $O$  is an asy upper bound that may or may not be tight. What is a asy tight upper-only bound?

Or am I confusing things?

[#pset1](#)

---

6046-tas mailing list  
[6046-tas@lists.csail.mit.edu](mailto:6046-tas@lists.csail.mit.edu)  
<https://lists.csail.mit.edu/mailman/listinfo/6046-tas>

Part 2  
Recurrences

9/23  
6:30P

2. Upper + lower bound for each

Master Theorem

Quick review

$$T(n) = a T\left(\frac{n}{b}\right) + \Theta(f(n))$$

~~Recurrence~~

$$n^{\lg_b a} \text{ vs } f(n)$$

$$n^c \cdot (\lg n)^d \text{ for each}$$

1. Compare  $C_s$

$\hookrightarrow$  bigger  $c \rightarrow \text{poly}$ ,  $\downarrow$  if

2. Compare  $d_s$

return that  $n^{\lg_b a}$  or  $f(n)$

$$\hookrightarrow T(n) \cdot \lg n$$

$$\uparrow$$
$$n^{\lg_b a}$$



⑦

(each of these formats seem to be diff

can't really see how eq'v)

Now I kinda do

CLAS 1. ~~n~~  $n^{\lg a}$  c bigger  
poly 2.  $c^s$  same  $\rightarrow d$  then  $n^{\lg b a}$   $\lg n$   
3.  $f(n)$  c. bigger

a)  $T(n) = 3T(n/4) + 5n$

So  $a=3$   
 $b=4$

~~n~~  $n^{\lg 4 3}$  vs  $5n$

Look at  $c_s$

$1.78$  vs  $1$

So  ~~$n^{\lg 4 3}$~~   $5n$  is bigger

$T(n) = f(n) = \underline{5n}$

(correct)

③

What is that extra step we should be doing?

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

for some  $c < 1$

$$3f\left(\frac{n}{4}\right) \leq c \cdot 5n$$

So at  $n=20$

$$3 \cdot f(5) \leq c \cdot 100$$

$$c = .99$$

$$3 \cdot 25 \leq .99 \cdot 100$$

✓

✓ It works

When does it not work

Must we always check?

- Seems to be in addition to before -  
So should check

11/1/14

$$b) T(n) = 9T(n/3) + n^2$$

$$n^{\lg_3 9} \quad \text{vs} \quad n^2$$

$$n^2 \quad \text{vs} \quad n^2$$

So compare ds  $\rightarrow$  same

$$\text{So } (n^2 \cdot \lg n)$$

$$c) T(n) = 5T(n/2) + \lg n$$

$$a=5$$

$$b=2$$

$$n^{\lg_2 5}$$

$$n^{2.32}$$

$\uparrow$  bigger C

$$n^0 \lg n$$

$$(n^{2.32})$$

Should I  
be able  
to solve  
in head??

5

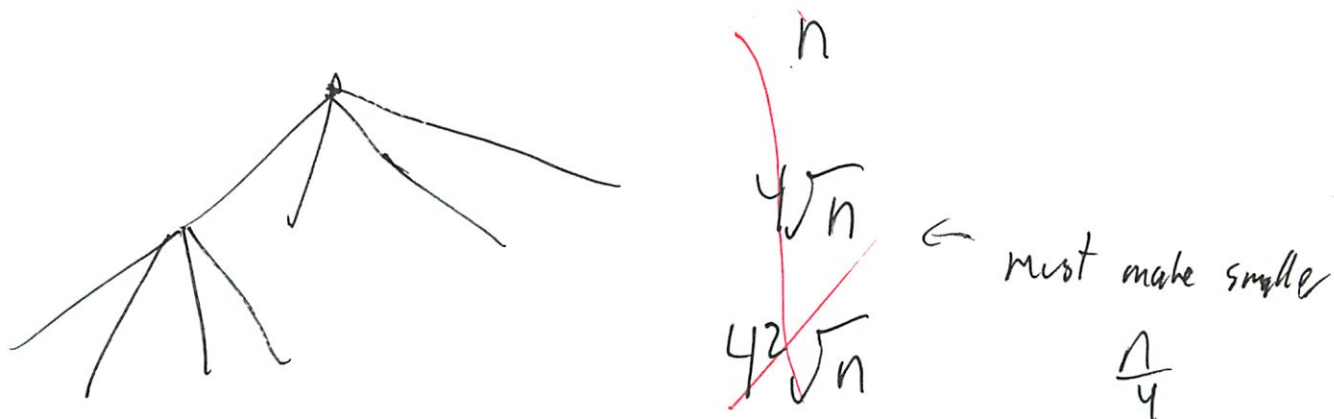
d)  $T(n) = 4 T(\sqrt{n}) + \lg^2 n$

~~Q/A~~ No master Theorem!

Old fashioned way:

So 4 branches

each  $\sqrt{n}$  amt of work



(What is  $p$  in notes)

Book calls eq'n  $f(n)$

In notes  $f(n) = n^p$

Then

$p > \lg_b a \rightarrow \Theta(n^p)$

$p = \lg_b a \rightarrow \Theta(n^p \lg n)$

$p < \lg_b a \rightarrow \Theta(n^{\lg_b a})$  (not  $f(n)$ )

w/  $n^p = f(n)$

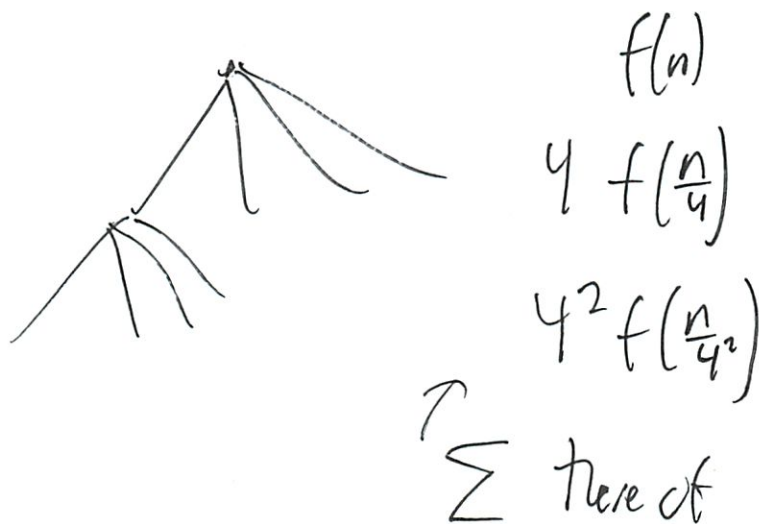
Flipped order 3.  
2.  
1.



6

I'm going to treat  $n^p$  as  $f(n)$

So



Matches notes

Then last level

$$4^h f(1)$$

$\uparrow$  however # of levels

①

Generally

$$a^h f(\frac{n}{b^{ah}})$$

How do we know how many levels?

~~Set~~ Set it = !

(Should have thought of)

$$\frac{n}{4^{ah}} = 1$$

⑦

Then solve

$$n = 4^h$$

$$h = \log_4(n) \quad \textcircled{1}$$

Note ~~must include root~~

$$\log_4(n) + 1$$

I don't know why

Some level we didn't include

Since we number from 0  
at root

Note book converts

$$a^h f\left(\frac{n}{b^h}\right)$$

- Oh we don't have constants

Add up

Gr another

convert ~~regions~~ sigma to definite

But could use

$\uparrow$   
as

geometric series instead  
upper bound

⑧

$$T(n) = \sum_{i=0}^{\lg_4(n)-1} a^i f\left(\frac{n}{b^i}\right) + \Theta(\lg^2 n)$$

How do they convert  $\Theta(n^2)$  to  $cn^2$

↳ Some constant  $c > 0$  since  $\Theta$

So here  $\Theta$  was  $n^2$  so  $f(n) = n^2$

So our  $f(n) = \lg^2 n$

Oh our recurrence was set up already

So not  $\frac{n}{4}$  since we branched by 4

So ours is  $f(n)$

$4 f(\frac{n}{4})$

$4^2 f(\frac{n}{4^2})$

↳ normally  $\frac{n}{b^2}$

is it  $\sqrt{n}$

Since  $\frac{n}{b^2}$  is  $(\frac{n}{b}) / b$

⑨

So  $4^2 f(4\sqrt{n})$

$$\sqrt[4]{n} = 4\sqrt{n}$$

Where  $f() = \lg^2 n$

~~So the time is~~

$$\sqrt{\sqrt{n}} = \sqrt[8]{n}$$

So  $n^{1/8}$

1  
4  
8  
16

So  $n^{1/8} = 1$

WA

$$h = \pm \frac{\sqrt{\log n}}{\sqrt{2\pi} \sqrt{i c_1}}$$

$$c_1 \neq 0$$

$$Or \quad h = \pm \sqrt{\frac{\lg(n)}{2\pi}} \sqrt{i n}$$

forget that



(10)

$$T(n) = \sum_{h=0}^{h_{\text{need}}-1} 4^h \lg^2(n^{1/h^2}) + \Theta(\lg^2 n)$$

What is this?  
time for last node

must convert to  
closed form

WA is not really working

They were able to convert to format

$$4^h \lg^2(n^{1/h^2})$$

We can't really break it at

When I try a # for h it says ind.

Skip



(2) So  $h$  should disappear

On the 'ineq

$$\text{that} < \sum_{i=0}^{\infty} -\frac{1}{2} (n+1) (2h-n)$$

So ind integral

---

Series diverges

$$n + n-1 + n-2 + \dots \text{infinite times if } n \rightarrow \infty$$

---

So basic

$$= -\frac{1}{2} (n+1) 2(n+1-n) + O(n)$$

$$= -\frac{1}{2} n - 2 + O(n)$$

$$= O(n)$$

8/23  
8P

Paul Woods

Substitution  
method ??

forgot about

2d)

$$T(n) = 4T(n/4) + \lg^2 n$$

$$\lg n \cdot \lg n$$

$$\frac{1}{2} \lg n \cdot \frac{1}{2} \lg n = \frac{1}{4} \lg n$$

$$= 4(4T(n^{1/4}) + \frac{1}{4} \lg^2 n) + \lg^2 n$$

I didn't write this in my  
method

$$= 16T(n^{1/4}) + 2 \lg^2 n$$

$$= 16(4T(n^{1/8}) + \frac{1}{16} \lg^2 n) + 2 \lg^2 n$$

$$= 64T(n^{1/8}) + 3 \lg^2 n$$

$$= \uparrow \quad h^? \lg^2 n$$

Need to take to limit

problem size ~~1/4~~  $n^{1/h^2}$

Set  $=$  to 1

6.042

open  $\rightarrow$   
closed  
form



②

$$h^{1/h^2} = 1$$

WA solve  $\rightarrow$  really weird

But it for ~~limit~~ (upper?)  $h = \infty$

$$\text{Then } \propto \lg^2 n$$

$$\sum_{h=0}^{\infty} \lg^2 n$$

Could we say  $h = n$

$$n \lg^2 n$$

Can't be that - since reduces by  $\sqrt{2}$  each  
time

So  $h = \lg n$  levels if

$$\text{Or } h = \sqrt{n}$$

(3)

Rest of parts

~~#26~~

It appears  $T(n) = 4^k T(n^{1/2^k}) + k \lg^2 n$

Verify  $T(n) = 4^k T(n^{1/2^k}) + k \lg^2 n$   
 $= 4^k \left( 4 T(n^{1/2^{k+1}}) + \lg^2 n^{1/2^k} \right) + k \lg^2 n$

$$= 4^{k+1} T(n^{1/2^{k+1}}) + (k+1) \lg^2 n$$

$$T(n) = \Omega(\sqrt{n} \lg^2 n)$$
$$= O(\sqrt{n} \lg^2 n)$$

Q

How many levels?

What is generic ans?

Example

$$T(n) = 3 T(n/4) + cn^2$$

Then sub problem size  $\frac{n}{4^h}$

Since

$$\begin{array}{c} n \\ \downarrow \\ \frac{n}{4} \\ \downarrow \\ \frac{n}{4^2} = \frac{n}{16} \end{array}$$

$$\frac{n}{4^h} = 1 \quad \text{solve for } h$$

I got that

5

But visually

each time  $\frac{1}{4}$  the problem

here each time  $\sqrt{n}$  of the problem

So  $n=25$  up front

25

5

$$\sqrt{5} = 2.23$$

$$\sqrt{\sqrt{5}} = 1.47 = 5^{1/4}$$

$n$

$$\sqrt{n} = n^{1/2}$$

$$\sqrt{\sqrt{n}} = n^{1/4}$$

$$\sqrt{\sqrt{\sqrt{n}}} = n^{1/8}$$

each level

What is  $(\text{inv})$  of  $\text{fact } i$   
What is even the right term?

$$\begin{array}{l} \log \\ x = b^y \\ y = \log_b(x) \end{array}$$



6

So  $n^{1/h^2}$

is

$$\frac{1}{h^2} = \log_n(n)$$

$\uparrow$  items at each level  
 $\uparrow$  items at each level  
 items at each level

Paul ~~John~~  $h = \lg \lg n$

Want  $h$

$$h = \frac{1}{\lg_n(\lg_n(\text{each level}))}$$

How get rid of  $\frac{1}{\text{over}}$  that is inverse

---

So  $\frac{1}{h^2}$  is  $h^2$

$$h^2 = \frac{1}{\lg_n(n)}$$

$\lg_h^2 =$  we want the base pulled out

9

that's  $h = \sqrt{\frac{1}{\lg n(n)}}$   
Each level is  $\log^2 n$

So  $\sqrt{\frac{1}{\lg(\log^2 n)}}$

Idk!

## Michael E Plasmeier

---

**From:** Nils Molina <nilsmolina@gmail.com>  
**Sent:** Sunday, September 23, 2012 8:32 PM  
**To:** Michael E Plasmeier  
**Subject:** Re: Are you good at recurrences?

Here's a similar problem: <http://math.stackexchange.com/questions/128503/recurrence-tn-t-sqrt-n-theta-log-logn>

So, substitute some things like the author did, getting as an intermediate step

$$S(m) = 4*S(m/2) + m^2$$

which can be solved using Master's theorem

On Sun, Sep 23, 2012 at 8:01 PM, Michael E Plasmeier <[theplaz@mit.edu](mailto:theplaz@mit.edu)> wrote:

I'm stuck on  $T(n) = 4T(\sqrt{n}) + \log^2(n)$  for 6.046

Are you around at any point today or tomorrow?

Thanks so much! -Plaz

**Recurrence**  $T(n) = T(\sqrt{n}) + \Theta(\log(\log(n)))$ 

I need to find the bounds of the above recurrence .

I've tried the following however got stuck :

$$T(n) = T(\sqrt{n}) + \Theta(\log(\log(n))) =$$

$$n = 2^m, \quad m = \log(n)$$

$$T(2^m) = T(\sqrt{2^m}) + \Theta(\log(\log(2^m))) = T(2^{m/2}) + \Theta(\log(m))$$

Now define:  $S(m) = T(2^m)$  then:

$$S(m) = S(m/2) + \log(m)$$

Now define :  $q = \log(m)$  ,  $m = 2^q$

$$\text{And we get : } S(2^q) = S(2^q/2) + \Theta(q)$$

$$\text{And finally , define : } R(q) = S(2^q) \implies R(q) = R(q-1) + \Theta(q)$$

But how can I continue from here ?

Regards

EDIT:

$$R(q-1) = R(q-2) + \Theta(q-1) \implies R(q) = R(q-2) + \Theta(q) + \Theta(q-1)$$

$$R(q-2) = R(q-3) + \Theta(q-2) \implies R(q) = R(q-3) + \Theta(q) + \Theta(q-1) + \Theta(q-2)$$

What am I suppose to do with all the :  $\Theta(q) + \Theta(q-1) + \Theta(q-2)$  ?

Thanks

(recurrence-relations)

edited Apr 5 at 21:54

asked Apr 5 at 20:51



ron

307 5

77% accept rate

You really shouldn't forget the constant in the big- $O$  notation. The recurrence  $R$  you derive is one of the simplest recurrences: I bet you actually have it memorized, but didn't think to think about it! If you really can't see it, try writing out the first few terms explicitly in terms of  $R(0)$  .... – Hurkyl Apr 5 at 21:21

@Hurkyl: I've made some changes , and would appreciate if you could check them out . – ron Apr 5 at 21:55

While your work is no longer *wrong*, you've taken a step backwards. You *can't* do anything with  $\Theta(q) + \Theta(q-1) + \dots$  , because any strange thing could be happening with the constants hidden in the big- $\Theta$  notation. You have to take advantage of the fact it all originated with the original  $\Theta(\log \log n)$ . I.E. that you know

there are  $M$  and  $N$  so that  $T(\sqrt{n}) + M \log \log n \leq T(n) \leq T(\sqrt{n}) + N \log \log n$  . – Hurkyl Apr 5 at 23:59

@Hurkyl : You mean that all the *define*  $S$  that I've made are not necessary ? – ron Apr 6 at 0:07

- 1 No, you have the right underlying idea. See Didier's answer where he finished up the proof, rather than suggesting how you might see it for yourself. (The main thing I was hoping you'd notice is that you'd recognize  $Cq + C(q-1) + C(q-2) + \dots$  as being a sum of consecutive integers) – Hurkyl Apr 6 at 0:49

feedback

## 2 Answers

You know that  $T(\sqrt{n}) + A \log \log n \leq T(n) \leq T(\sqrt{n}) + B \log \log n$  for some constants  $A$  and  $B$ . As in your post, let  $R(q) = T(2^{2^q})$ , then

$$R(q-1) + A(q \log 2 + \log \log 2) \leq R(q) \leq R(q-1) + A(q \log 2 + \log \log 2),$$

hence there exists  $A'$  and  $B'$  such that  $A'q \leq R(q) - R(q-1) \leq B'q$ . Summing this from 1 to  $q$ , one gets

$$\frac{A'}{2} q^2 \leq A' \sum_{k=1}^q k \leq R(q) - R(0) \leq B' \sum_{k=1}^q k \leq B' q^2.$$

Finally,

$$T(n) = \Theta((\log \log n)^2).$$

answered Apr 6 at 0:36



did

66.2k

6

53

130

feedback

After you get  $S(m) = S(m/2) + \log(m)$ , you can use the Master Theorem:

$$f(m) = \log m = \Theta(m^{\log_2 1} \times (\log m)^1) = \Theta(\log m). \text{ Therefore}$$

$$S(m) = \Theta(m^{\log_2 1} \times (\log m)^2) = \Theta((\log m)^2)$$

Or:

$$T(n) = \Theta((\log \log n)^2)$$

answered Apr 6 at 7:45



MNos

121

3



Nils  
Notes: Mdp

4/23

Sweet  $\rightarrow$  almost exactly  
(I should have Googled!)

Why convert  $\sqrt{n}$  to  $2^m$  ?  
how

well  $m = \frac{1}{2}$

Then  $S(m) = S(m/2) + \lg(m)$

$\uparrow$  So what we  
are doing is  $\lg$ :

$\frac{1}{n^2}$  Totally not

$q = \lg(m) \quad m = 2^q$

⑦

$$T(\sqrt{n}) + A \log \log n \leq T(n) \leq T(\sqrt{n}) + B \log \log n$$

for constants  $A, B$

$$R(q) = T(2^{2^q})$$

$$R(q-1) + A(q \log 2 + \log \log 2)$$

$$\leq R(q) \leq R(q-1) + A(q \log 2 + \log \log 2)$$

So exists  $A', B'$  s.t.

$$A'q \leq R(q) - R(q-1) \leq B'q$$

Sum from 1 to  $q$

$$\frac{A'}{2} q^2 \leq A' \sum_{k=1}^q k \leq R(q) - R(0)$$

$$\leq B' \sum_{k=1}^q k \leq B' q^2$$

(3)

$$\text{So } T(n) = \Theta(\lg \lg n^2)$$

Our ans likely sure?

No the 4 matters

Nils

$$S(m) = 4 \cdot S\left(\frac{m}{2}\right) + m^2$$

So that is  $m = \lg n$ ?

In original  $S(m) = S(m/2) + \lg(m)$   
(Ours is the same)

I really think Nils meant  $+ \lg(m)$   
not  $m^2$

Where  $m$  is  $\lg m$

~~So it can be either~~

④

$$\Theta(m^{\log_2 1} \cdot \log m) = \Theta(\log m)$$

$$\begin{aligned} S(m) &= \Theta(m^{\log_2 1} \times (\log m)^2) \\ &= \Theta((\log m)^2) \end{aligned}$$

I kinda see it

So just solve - ah easy

$$a = 4$$

$$b = 2$$

$$m^{\log_2 4} \text{ vs } \log m$$

$$m^{1/2}$$

$$m^0$$

& bigger

So

$$\Theta(\sqrt{\log n})$$

If  $a = 1$

5

$m^2$  vs  $\lg m$   
 $\uparrow$  bigger

$$\Theta((\log n)^2)$$

$$\Theta(\lg^2(n))$$

Verify Original  $\rightarrow$  Same if  $a=1$

$m \lg 2$

$m^0$  vs  $m^0 \lg m$

$\rightarrow \lg m \lg m$

$\lg^4 m$  (?)

Am I collapsing correctly?

but not  $\log(\log(n))$   $\left\{ \begin{array}{l} (\log(n))^2 \text{ vs } \log^2(n) \text{ are} \\ \text{same? } \checkmark \text{ yes} \\ \text{also } \log \log n \checkmark \end{array} \right.$



⑥

So  $\log m \log m$

$$\log(\log(m)) \log(\log(m))$$

Which matches given  $(\log \log m)^2$

But I think I get the practical aspects  
of this

---

Rewrite ~~on~~  
— on blank sheet

$$n = 2^m$$

also  $\hookrightarrow m = \log_2(n)$

$$\log^2(2^m)$$

$$\log \underbrace{\log_2(2^m)}_{\log(m)} \quad \text{ah clever}$$

---

I still don't feel good about  
 $S(m)$  step

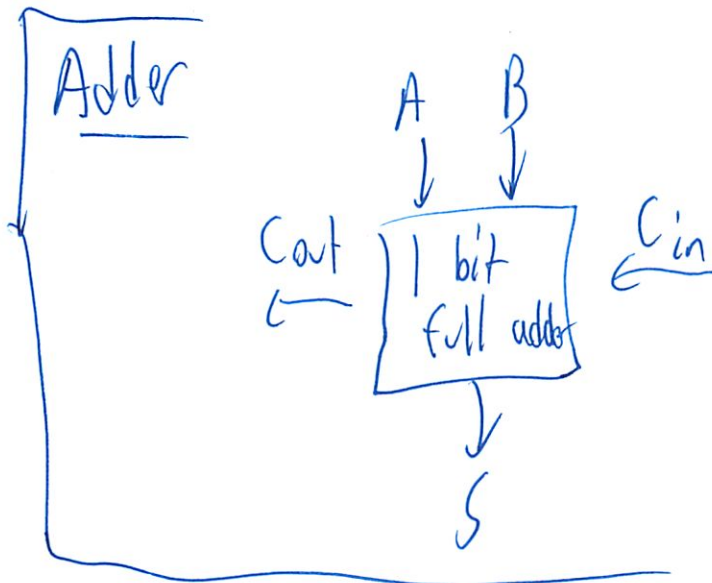
# P-set 1 #3

9/24  
8:15P

n #s 1 or 0

want sum

but only 1 bit adder & carry



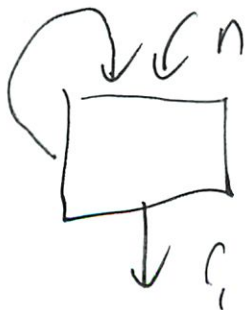
So  $\Theta(\max(a, b))$

Why

~~So the numbers are 1 or 0~~

So have n #s to add

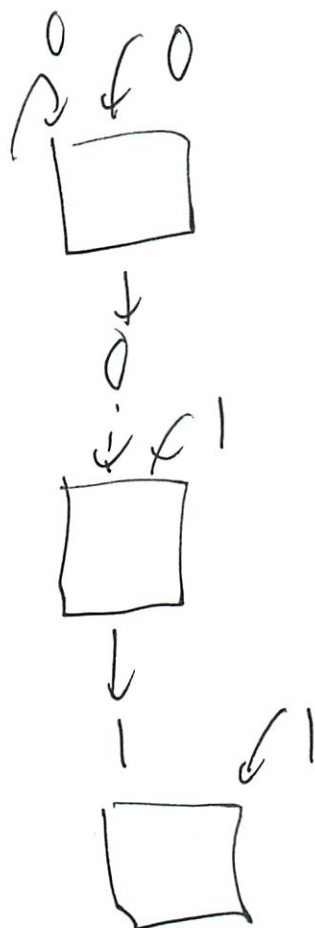
So



②

So let's say have 1, 1, 1, 0  
 $\uparrow$   $n=4$

So



no thats not going to work  
it can only output 1 or 0

~~Add each pair~~

Add each pair of digits sequentially

but then can't just be 1 or 0

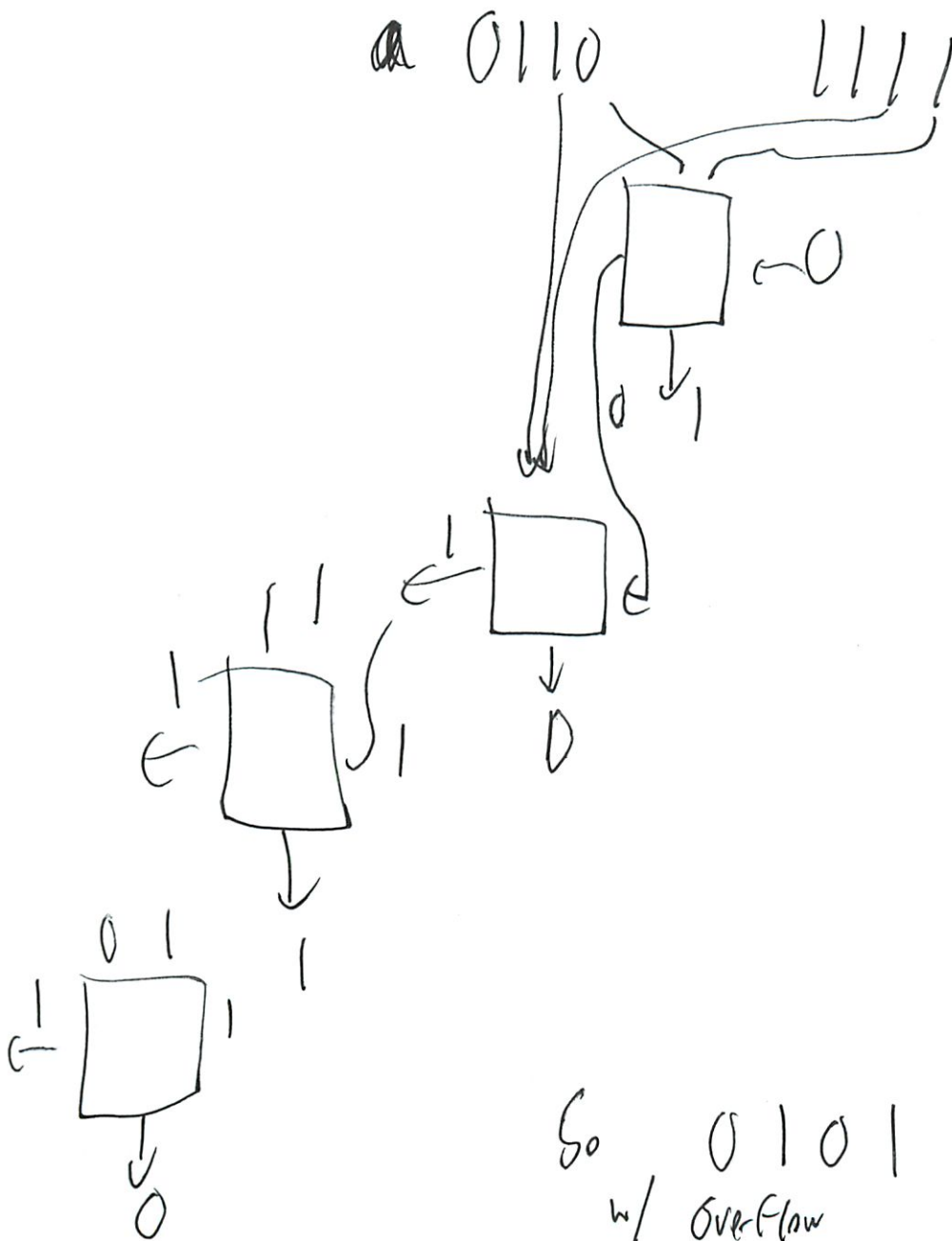
Confused

(3)

(I can't go further if don't get the adder)

a-bit #      b-bit #

So this makes more sense



④

So # of steps is  $O(n)$  for the longest string

① Makes sense

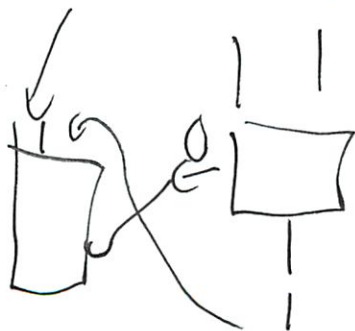
a) Assume you go through the list adding each bit in turn to a running total  
Ok now back to the  $n$  #'s

1 1 0 1

then do that:



1 1 0 1



but this can be multiple bits



5

is this the problem?

Ripple - how represent?

So



is  $1+1=2$

Oh so output 0 now  
and save up the 2  
carry ↓ output  
10

$1+1=2$  binary digits  
# digits

$2+1=3$  binary digits

$3+1=4$

$$\text{So } \sum_{i=1}^n (i+1)$$

$$= \frac{1}{2} n(n+3)$$

(6)

So that 'is  $O(n^2)$   
↑ worst case

but if all #s are 0

$$\begin{array}{c} 0 + 0 \\ \text{digit} \quad \# \end{array} = 0$$

$$0 + 0 = 0$$

So takes  $n$  time  $\Omega(n)$   
↑ at least

What if evenly distributed?

So 25% the  $0 + 0 \rightarrow$  same # digits

50%  $1 + 0 \rightarrow$  same # digits

25%  $\rightarrow$  digits level up for rest

(We have the prob. before  $\rightarrow$  really guessing here)

So 75%  $n^{\#}$  25%  $n^{\#} + 1$

$\therefore$  How do you prove this?

⑦

$$\therefore \frac{1}{4} n^2 + \frac{3}{4} n \rightarrow \text{still } n^2$$

Probabilistic Analysis section in textbooks

Random values

$$X_{ij}$$

$$E[X_{ij}]$$

$$X = \sum_i \sum_j X_{ij}$$

$$E[X] = E \left[ \sum_i \sum_j X_{ij} \right]$$

~~linearity of expectations~~

How do they jump to  $\Theta(n)$

When  $E > 1$

I don't think this is what we need

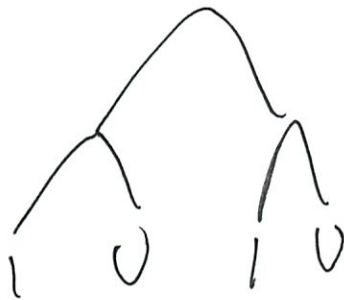
⑧

Just say ~~that~~  $\frac{1}{4}n^2 + \frac{3}{4}n$

Not asy

b) Better algorithm to add ~~the~~ together efficiently

Think what we did in class: divide + conquer  
Do we still just have our 1-bit adder



Or just elim all 0s

Then add the 1s

Do we have the logic to strip 0s

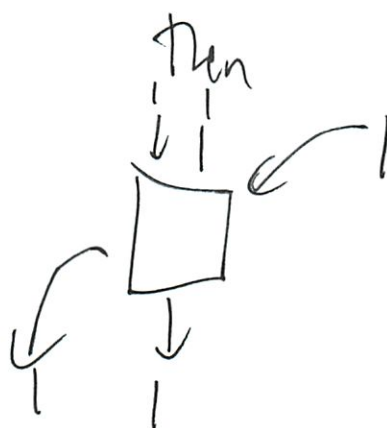
(hint: just have 1-bit adder)

9

But we need some intermediate ram...  
I don't know what we actually have  
Prob something that



So for every group of 6s



~~But~~ So for 5 1s  
we want 101



 now have to add that



(10)

But only the 1 bit adder

What other tricks did we learn

- Interval Scheduling
- Divide + Conquer
- Shape thing
- FFT
- Randomized alg

It seems lectures have () to do  $\forall$  p-set

We did picking pivot in randomized quick sort

This is not really like that

look at expected # of iterations  $\rightarrow 2$  in given case

---

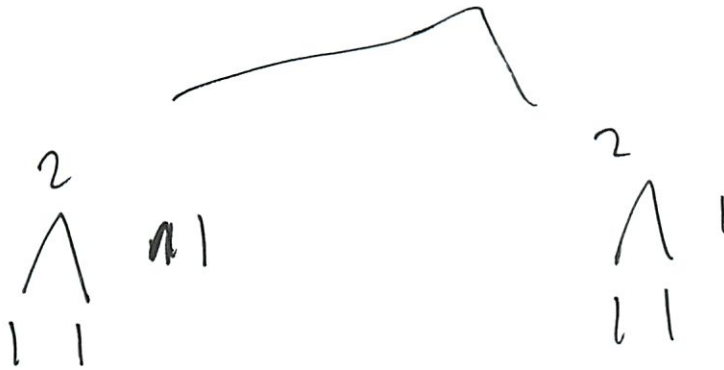
Paul: Add in pairs



linear time worst case

(11)

Let me think more about that



2 + 2 is 4 operations?  
no - 2 operations

10 10

then we have

$$\begin{array}{c} n \\ n \\ n \\ \sum_{i=1}^n n \end{array}$$

$$= \frac{1}{2} n(n+1)$$

$$= O(n^2)$$

(12)

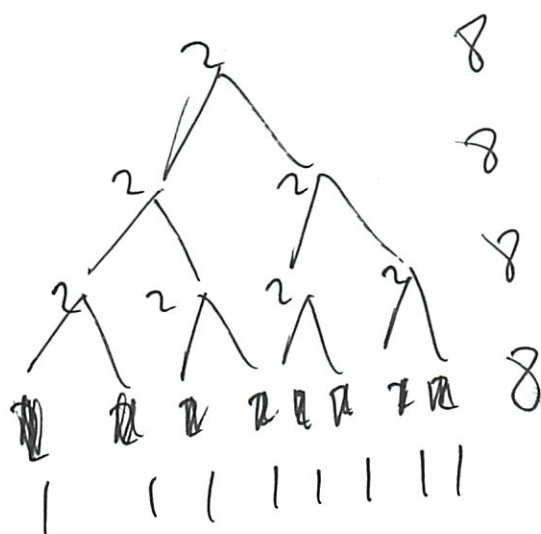
Which is  $2 T(\frac{n}{2}) + \Theta(1)$  right?

$$\lg_2 2$$

$$N \text{ is } n^0 \text{ or } 1$$

$$T_{No} N$$

This is



$$n=8$$

$$\text{or } 8.4$$

What is  $h$

$$\frac{n}{h^2} = 1$$

$$n = h^2$$

$$h = \sqrt{n}$$

(13)

Book

$$\frac{n}{4^i} = 1$$

$$i = \log_4 n$$

$$\text{so } n = 4^i$$

Oh I see

$$\frac{n}{2^h} = 1$$

$$n = 2^h$$

$$h = \log_2(n)$$

$$\text{so } \sum_{i=1}^{\log_2(n)} n$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

but that does not apply here

(14)

So that is  $\lg^2(n)$

Let me review this

we have  $\frac{n}{2^h}$  levels

we do that much work on each

$$2^h \frac{n}{2^h} = n$$

So # levels

$$\frac{n}{2^h} = 1$$

$$2^h = n$$

$$\lg_2(n) = h$$

So we have

$\lg(n)$  of  $n$

$$\underbrace{n + n + n}_{\lg(n) \text{ times}}$$



(15)

What does that mean?

if we had

$$\underbrace{1 + 1 + \dots}_{n \text{ times}}$$

we'd have  $n$   
no 1's

if we had

$$\underbrace{n + n + \dots}_{n \text{ times}}$$

That would be  $n^2$  not  $n$

So  $n \lg(n)$

That sounds good

↳ but Paul said  $n$

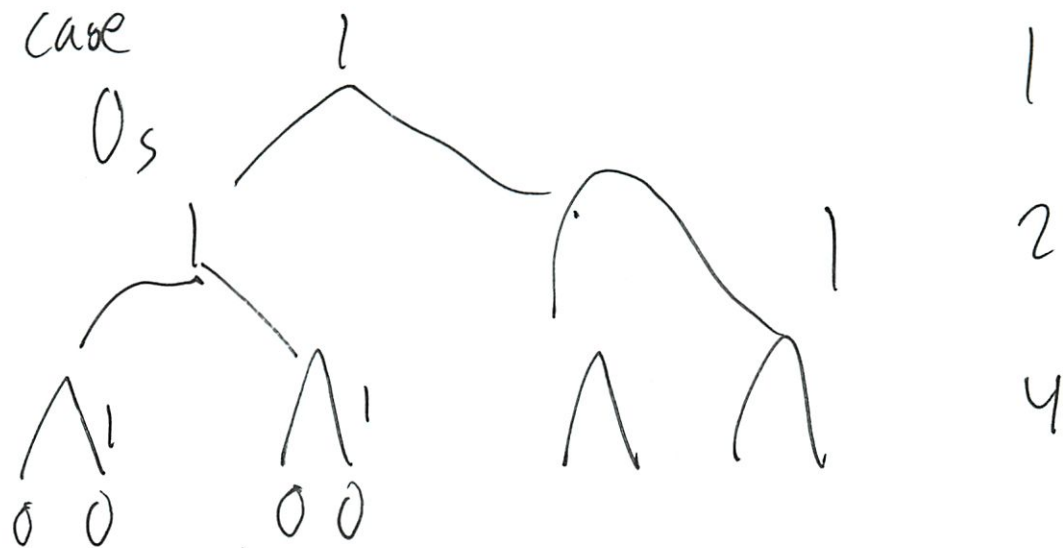
or ~~there~~ was phase two?

(16)

Best

case

0s



This is  $\sum_{n=0}^{\log(n)} 2^n$

is this right

$$1 + 2 + 4 + 8 + 16$$

if  $n = 16$

so this is  $2n$

(my WA adding method does not seem to be valid)

or  $n$  best case

$O(n \log n)$

$\Omega(n)$

(17)

~~$n \log n$~~   
 ~~$n$~~   $\leftarrow$  correct

c) Can we do better?

well with linear - certainly not

but since we only have a 1 bit  
adder

make this vague reference to that

So must add each character that many times

$\Theta(\max(a, b))$  for each group

(but that would be  $n^2$  - ...?)

Vague proof

Ok for now - see if other people better

---

keep thinking

a  $n$ -bit # requires at least

$\frac{n}{2} + \frac{n}{2}$  ie  $500 + 500 = 1000$

no!

(18)

Confusing # and length

5 is 101 101  
is

1010

So always are longer  
than both

So a-bit # is

a-1 b-1

That means at <sup>least</sup> ~~most~~ n?

but how to get from ~~na~~ to n

a	1
a-1 a-1	2
a-2 a-2 a-2 a-2	4

Operations  
 $= 2n$

So claim could do it in n  
possibly

(19)

(Write up)

ok wait 3 is 2 bits as well

11

So

2	2 bits	}	2
3	2		
4	3	}	4
5	3		
6	3		
7	3		
8	4	}	8
15	4		
16	5	}	16

So # of binary bits per # is  
(I suck at this type of math)  
- try it



20  
~~8~~ 8 is 4 bits

$$\frac{8^2}{4^2} =$$

How many times  $\gg$  till 0

$$\text{bits} = \lfloor \log_2(\#) \rfloor \leftarrow \text{what was looking for}$$

$$\# = \frac{\text{bits}}{2} \text{ bits and up}$$

but how amortize?

$$2 + 4 + 8 + 16$$

$$2 \cdot 2 + 4 \cdot 3 + 8 \cdot 4 + 16 \cdot 5$$

$$S_n = \sum_{i=1}^n 2^i \cdot (i+1)$$

(21)

w/ h to be

$$\lfloor \log_2(\#) \rfloor$$

$$\sum_{i=1}^{\log_2 \#} 2^i \cdot (i+1)$$

$$= 2 \cdot \lg(\#)$$

Then my method

---

But growth of  $1 + 0$

Oh its not pairs here

---

Is height half?

—no same  
but for our purposes of growth  
hmm



27

Draw



Grows half as often  
but height same?  
though summation less



(23)

b) : same since  
but  $n \lg n$

$\begin{matrix} & 2 \\ & \wedge \\ 1 & & 1 \end{matrix}$

c)  $2T(n-1) + O(1)$

~~Ques~~

$$l = n - 1$$

so  $n$  steps

~~T(n)~~

$$\sum_{i=1}^n \cancel{2^i} (n-i)$$

$$= O(2^a)$$

but how does that relate to  $n$ ?

if  $n$  was all 1s it would be

$$q = \lfloor \log_2(n) \rfloor$$

$$\cancel{O(n)} \in O(n)$$

(29)

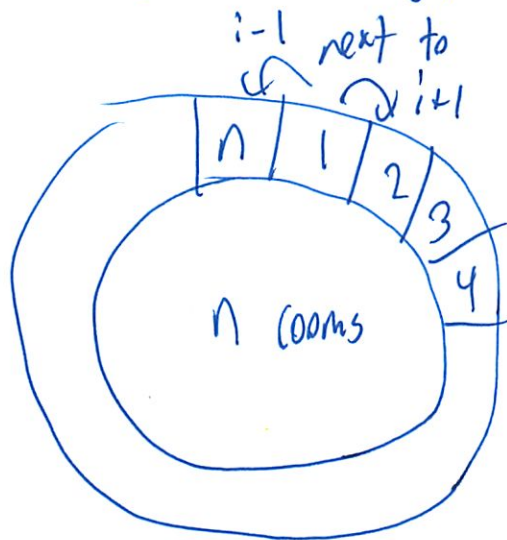
So that is better

— b just asks for better — not the best

— actually that proof might not be all that conclusive  
just showed that we can do no better than Alg  
but not ~~the~~ what we can do w/ 1-bit add

w/e



#4 Donut BuildingDonut in circle of  $n$  rooms

$V_i$  = capacity of  
each room  
can vary

But rooms are not sound proof

So if class in  $i$ , can't be  $i \pm 1$ 

Paul's hint: Like interval scheduling

weighted

Set to beginning + end time

Sounds pretty obvious  $\rightarrow$  would I have  
thought on my own?

②

## Interval Scheduling Review L1

Resources + requests

$s(i)$

$f(i)$

$s(i) < f(i)$

$f(j) \leq s(i)$

Greedy

↳ locally optimal at the time

Pick: finishes earliest

$O(n \log n)$  for sorting

DP as well

for non-greedy

was this the same problem? for non-weighted

For non-even machines

↳ which is over window

③

~~Go DP~~

So DP seems more robust

Reductions

boolean variable  $x_{ij}$  if req  $i$  can be  
sch on  $j$

then try to maximize

Can run ILP solvers (Cplex)

↳ no exp w/

this whole optimization market

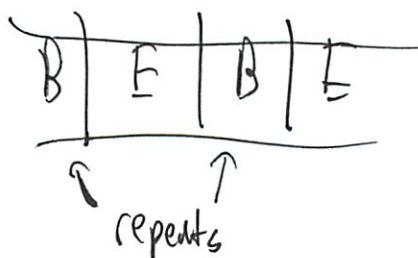
---

Ok start to think

No direct link of course

---

Can't just book 3 blocks since



(4)

They want greedy list

So we can pick max then block off  $i-1$   
 $i+1$

a) basically asks for a counter example

So if we had

20 25 20

1 1 1

Then greedy picks 25, 1, 1

But optimal is 20, 20, 1

5

b) Give an efficient algorithm

DP?

Only other option besides Greedy

Would make this problem rather straight forward though

Try each ~~element~~<sup>room</sup> as possible 1st  
Then best of the remaining

From  
example  
in lecture

$$\text{Opt}(A) = \max_{1 \leq i \leq n} (w_i + \text{opt}(A \setminus \{i\}))$$

$O(n^2)$

Then make it better w/ sorting

want to do big rooms 1st

but also try the ults. - just in case



⑥

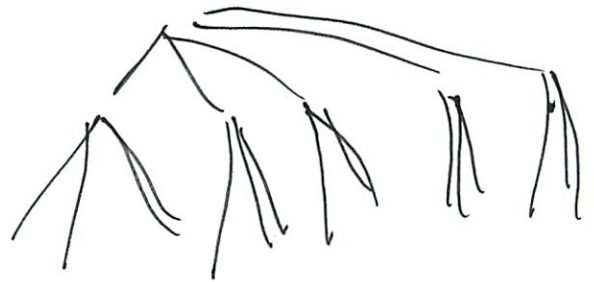
Though I don't get how w/ sorting  
we get  $O(n)$

(didn't get in lecture either)

So basic try  $n$

$n$   $n-1$

$n^2$   $n-2$



but memoization

look at

bottom

↳ forget great rules



↑ do less packed

So Pick

(I should understand)

⑦

Book: This is  $2^n$  - exponential!

w/ memoization  $O(n^2)$  bottom up  
doubly nested structure

for  $j = 1$  to  $n$

for  $i = 1$  to  $j$

→  
not cutting

$q = \max$

top down is same

So basically time for each sub problem & # of

So each an  $O(1)$  comparison

but  $n + n-1 + n-2 + \dots$

well memoization

8  
But how did that relate to intervals?  
and how did sorting help?

Skip for now

---

(write up)

Nothing special for the neighbor coms  
cross off

Perhaps consider  $V_{i-1} + V_i + V_{i+1}$

But DP should be faster + more general

---

could sort by that?

Do we want to?

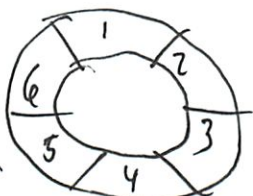
Look at what taking out of commission  
Or actually using?  
It's just a heuristic anyway

9

They only consider later in earlier example  
Must figure out antine!

So must actually do antine

So if  $n=6$  rooms



So this is # + capacity

Sorted 5:15  
6:12

~~5:12~~

4:12

3:9

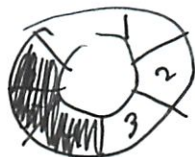
~~2:6~~

1:8

2:6

(4)

5 + optimal of rest



next on list

3'



1



gives 9

Is that optimal?

No!

Don't include extra rooms  
↳ want to minimize

(that wasn't what I was trying  
to test there anyway...)



①

Redo



2

So  $n$  possible choices  $+ n - \cancel{2}^3 + n - \cancel{4}^5$

Normally DP is  $n^2$

$$n + \sum_{i=1}^{n-3} n - 3 - 2i$$

$$\text{fill } n - 3 - 2i = 1$$

$$n = 3 + 2i$$

$$i = \frac{n-3}{2}$$

is  $O(n^2)$  that sounds good

(12)

Ordering has no ~~practical~~ advantage here  
easy

but a practical advantage

---

I should do a proof-why

They don't specifically require one

---

Does it work on counter example



+ optimal rest



2 more pulls



but must try optimal for each upfront

So  $n \cdot n$

13

---

In general I didn't really follow the format  
that closely.

No topic para

No real proof why correct

---

Do I need something as formal as  
recitation?

↳ Don't think so

Is what I have enough?

Can say it will try every combo

---

Is this heuristic repeatable?

---

Is  $n^2$  efficient?

Is there a robust cutoff?

If not  $\Rightarrow$  do heuristic

# Stuff to Review

1c rethink 1C when both ~~non neg~~ pos

1d  $\log^2 n$  vs  $\ln \lg n$

2d CLRS - substitution method  
p86

Recitation notes

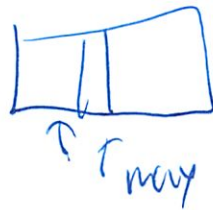
3b each step  $\log n$  space

$2+2=3$  bit ~~over~~ ans

4  $O(n)$

divide + conquer kindy

kinda like peak finding



then more right + left

$$V_i + V(i+2)$$

$$O(n)$$

$$O + V(i+1)$$





# Review

9/25  
2:45p

$\log^2 n$  vs  $\sqrt{n} \lg n$

Ops - fix

$\lg n$

$\log^2(n)$

$\sqrt{n} \lg(n)$

---

lc both ~~are~~ neg pos

So

$f = x$

$g = x$

$f + g = x + x$

at  $x = 10$

20

~~It~~ <sup>max</sup> Should be  $< 20$

① it is

$f = 5$

$g = 5$

$5 < 10$

②

$$2x \quad 14$$

at  $x=10$

$$\max_{14}^{20} < 34$$

No - I think this <sup>has</sup> ~~is~~ correct

---

CLRS p86 (p83 in my book)

Substitution method

I think I have a diff version of the book

- don't see the example

They say guess  $1st$   
and substitute it in  
to find that bound is valid

③

(this seems like a diff sub method)

---

Now Notes Paul sent

Basically same guess I always do

Along w/ a verifier

---

Wo on another page

$$T(n) \leq 2T(\sqrt{n}) + \lg(n)$$

$$m = \lg(n)$$

$$n = 2^m$$

$$\begin{aligned} T(2^m) &= 2T(\sqrt{2^m}) + \lg(2^m) \\ &= 2T(2^{m/2}) + m \end{aligned}$$

Wait  $\lg^2(m)$  is  
 $\lg(m) \lg(m)$   
not  $\lg(\lg(m))$

④

That might have been my issue

$$\log(2^m) \log(2^m)$$

$$m \cdot m$$

That's consistent w/ what nls said

Makes more sense now!

$$\Theta = (\lg^2(n) \lg(\lg(n)))$$

Long - but what  
(little had I think

③

$$\downarrow T(n) = 4T(\sqrt{n}) + \lg^2 n$$

define  $m = \lg_2(h)$

↳ so  $n = 2^m$

Plug in

~~$T(2^m) = 4T(\sqrt{2^m}) + \log^2(2^m)$~~

~~$T(2^m) = 4T(\sqrt{2^m}) + \log^2(2^m)$~~

~~$= 4T(2^{m/2}) + \log(2^m) \log(2^m)$~~

define  $S(m) = T(2^m)$

$$S(m) = 4S(m/2) + \log(m)$$

Master Theorem

$$a=4 \quad b=2$$

$$n^{\lg_2 4} \quad \text{vs} \quad n^0 \lg n$$

$n^2$  is bigger

So

$$\Theta = \lg^2 \left( \frac{n}{m} \right)$$

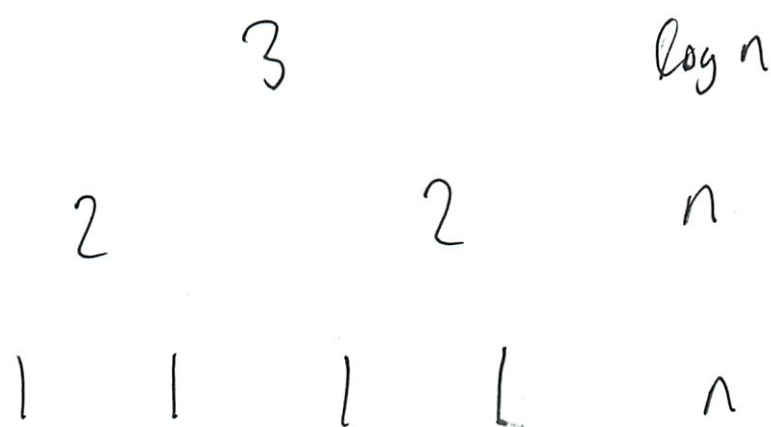


5

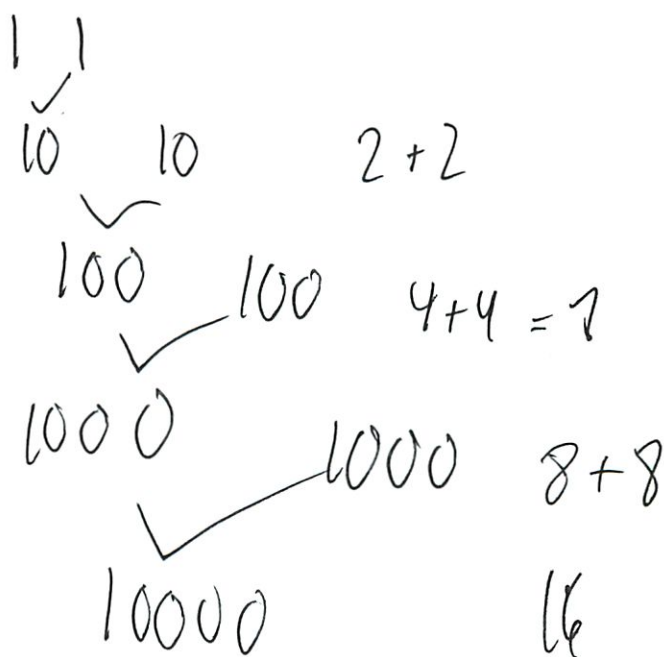
3b

So 2 bits + 2 bits = 3 bits

Never fixed on b



So actual results



How do you do this w/o writing it out?

6

So the bits is

Oh need freq

$$n \cdot 1$$

$$\frac{n}{2} \cdot 2$$

$$\frac{n}{4} \cdot 3$$

$$\frac{n}{8} \cdot 4$$

$$\frac{n}{16} \cdot 5$$

$$\sum_{i=1}^{\lg_2(n)} \frac{n}{2^i} \cdot i$$

$$= O(n)$$

still  $\lg n$  levels I think

as expected from others

no wrong problem

but  $3c$  was  $O(n)$

⑦

Why am I unsure how to translate

$n \lg n$  vs  $n$   
(smaller!)

It's  $n$  I'll say

Proof

$n$

$n^2$

at  $x=5$

5

25

$25 < 30$

0

$n$

$n \nless n$

Sometimes

8

$$\lim \frac{n}{n^2} \quad \frac{n+n^2}{n^2} \quad \lim \frac{\infty}{\infty}$$

diff meaning

One that relies more on asy

I think I will leave it...

Rahul

$n^2$  vs  $n^2 \lg n$   
bigger

$n 2^n$  below  $3^n$   
bigger

So I was right

Rahul

3a  $O(n \lg n)$  not  $O(\sqrt{n} \lg n)$  avg case  
I agree

2c  $\Theta(n^2)$  not  $\Theta(n)$

makes sense - fix

9

4, So should I change it?

Their solution was  $V_i + V(i+2)$   
~~0~~  $+ V(i+1)$

Do a very light proof

I don't even really get how this works

25	20	1	1	1	20
----	----	---	---	---	----

Kinda like the cut + rest

25 + best ~~1/1/1/1/1/20~~

Oh it will wrap around  
So if no

20 + best ~~1/1/1/1/20/25~~

So handle it



Michael Plasmier  
6.046 A07

P-Set 1  
#1

pwoals, chisty, (ah)caj

~~#~~ a)  $f(n) = \Omega(g(n))$        $f(n) = o(g(n))$

$$0 \leq c g(n) \leq f(n) \quad 0 \leq f(n) < c g(n)$$

$$g \leq f$$

$$f < g$$

f can't be both  $\geq g$  and  $< g$   
at any point in time

Never true

b)  $f(n) = O(g(n))$   
 $f(n) \leq c g(n)$   
 $f \leq g$

$$g(n) = o(h(n))$$
$$g(n) < h(n)$$
$$g < h$$

$$h(n) = \omega(f(n))$$
$$f(n) < h(n)$$
$$f < h$$

$$f \leq g < h$$
$$f < h$$

Always true

②

$$c) f(n) + g(n) = \omega(\max(f(n), g(n)))$$

$$\text{c. } \max(f, g) < f + g$$

$f, g$  Nonnegative  $\rightarrow$  so can be 0

If  ~~$g=0$~~  then

$$\max(f, 0) = f$$

$$f + \text{0} = f$$

$f < f$  can't be true

But if both  $f, g$  positive ( $> 0$ )

Then the sum of both will be  
greater than any one of them  
individually

Sometimes true

(3)

d)

$$e^{O(1)} = \text{constant}$$

$$\lg(\lg(n))$$

$$\lg n, \lg_4(n)$$

$$\lg^2(n)$$

$$\sqrt{n} \lg(n)$$

$$n \lg(n)$$

$$n^2 \lg(n)$$

$$\cancel{n^2} n^2$$

$$3^n$$

$$n!$$

$$n 2^n$$

$$2^{n^2}$$

$$2^{2n}$$

Michael Plasmeier  
6.046 R07

P-Set 1  
#2

pwoods, chridy, rehrhaj

a)  $T(n) = 3 T(n/4) + 5n$

$$a=3$$
$$b=4$$

$$n^{\lg_4 3}$$

$$vs \quad n^1$$

$$n^{.78}$$

$$vs \quad n^1$$

? bigger so

also  $3f(n/4) \leq c \cdot 5n$   
(✓)

$$\Theta(n) = f(n) = 5n$$

b)  $T(n) = 9 T(n/3) + n^2$

$$n^{\lg_3 9} \quad vs \quad n^2$$

$$n^2 \quad vs \quad n^2$$

So  $\Theta(n) = n^2 \lg n$

②

$$c) T(n) = 5T\left(\frac{n}{2}\right) + \lg n$$

$$a=5$$

$$b=2$$

$$n^{\lg_2 5}$$

$$vs \quad n^0 \lg n$$

$$n^{2.32}$$

$$vs \quad n^0 \lg n$$

? bigger

$$\boxed{\Theta(n) = n^{2.32}}$$

$$d) T(n) = 4T(\sqrt{n}) + \lg^2 n$$





2)

$$\downarrow) \quad T(n) = 4T(\sqrt{n}) + \lg^2 n$$

$$\text{define } m = \lg_2(n)$$

$$\text{so } n = 2^m$$

Plug in

$$T(2^m) = 4T(\sqrt{2^m}) + \log_2^2(2^m)$$

$$T(m) = 4T(\sqrt{m/2}) + m^2$$

Master theorem

$$a=4 \quad b=2$$

$$m^{\lg_2 4} \quad \text{vs} \quad m^2$$

$$\Theta(m^2 \lg m)$$

Convert back

$$\Theta = (\log^2(n) \lg(\lg(n)))$$

④

$$e) T(n) = T(n-1) + n$$

$$\left| \begin{array}{l} T(n) \\ T(n-1) \\ T(n-2) \end{array} \right.$$

$$n + (n-1) + (n-2) + (n-3) - \dots + 1$$

$$T(n) = \sum_{h=0}^n n-h + O(n)$$

$$= -\frac{1}{2} (n+1) (2n-n) + O(1)$$

$$= -\frac{1}{2} n - 2 + O(n)$$

$$\boxed{= \Theta(n^2)}$$

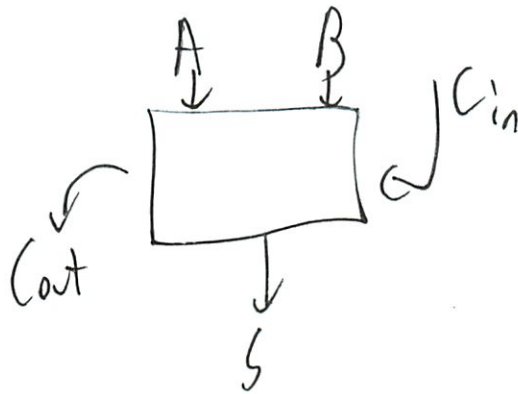
Michael Plasmeier  
6.046 R07

P-Set 7  
#3

Priya, Christy, Rahulraj

a) Adding to a running total

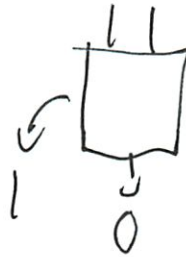
1-bit Adder w/ carry



For worst-case  
→ all  
1s

if we were to add in series

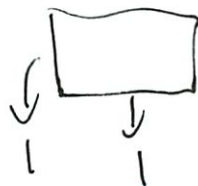
1+1



2 bits out

1+10

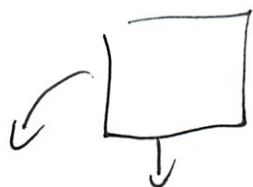
↓ ↓



2 bits

2

1 + 11



1 0 0 3 bits

So  
2: 2 bits  
3: 2 bits  
4: 3 bits  
5: 3 bits  
6: 3 bits  
7: 3 bits  
8: 4 bits

$$\text{bits} = \lfloor \log_2(\#) \rfloor$$

So then

~~2 + 4 + 8 + 16~~

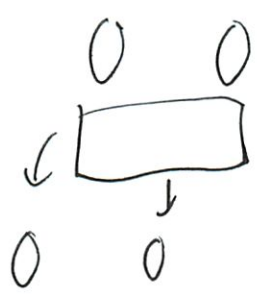
$$2 \cdot 2 + 4 \cdot 3 + 8 \cdot 4 + 16 \cdot 5$$

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} 2^i (i+1)$$

$$= O(n \lg n) \text{ worst case}$$

(3)

For best case  
all 0s



Will always be 0

So just  $n$  additions

$$2(n)$$

For avg case

assume = dist of 0, 1

~~0~~  $\rightarrow$  ~~addition~~  
no growth

~~1~~  $\rightarrow$  grows

So only half of the operations lead to growth

So EV is that each ~~operation~~ segment goes twice as far  
? ill define



4

$$2 \cdot 2 \cdot 2 + 2 \cdot 4 \cdot 3 + 2 \cdot 8 \cdot 4 + 2 \cdot 16 \cdot 5$$

and height will be roughly half

$$\sum_{i=1}^{\frac{\log_2(n)}{2}} 2 \cdot 2^i \cdot (i+1)$$

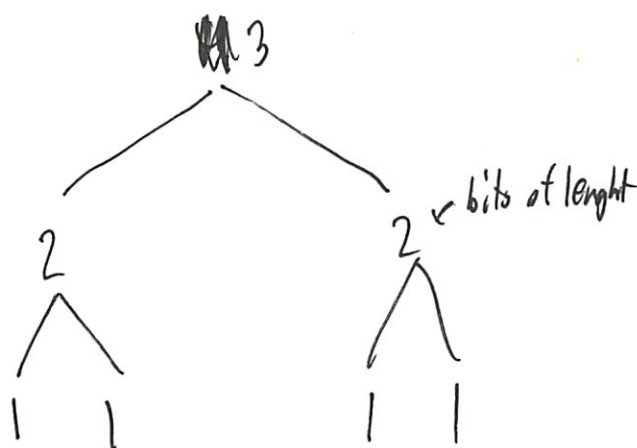
~~Still  $O(n \lg n)$  Avg case~~

Avg case  $O(n \lg n)$

b) A better way still w/ 1 bit adder

We could use a divide and conquer strategy

Worst  
case all  
1s



$$\frac{n}{4} = 3$$

$$\frac{n}{2} = 2$$

$$n = 1$$

5

# of levels

$$\frac{n}{2^h} = 1$$

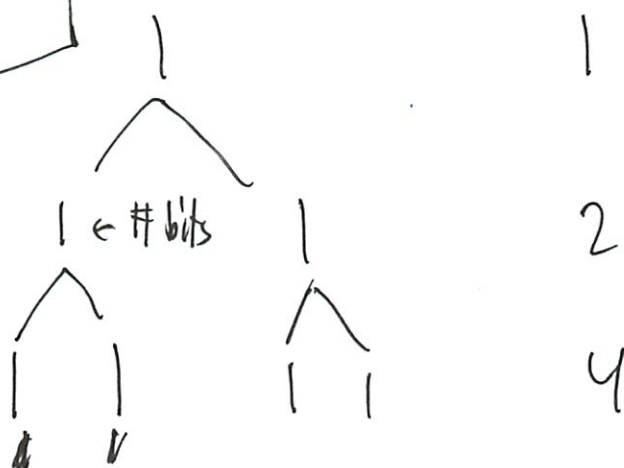
$$n = 2^h$$

$$h = \lg_2(n)$$

$$\sum_{i=1}^{\lg_2(n)} i \cdot \frac{n}{2^i}$$

$$= \cancel{O(n^2 \lg n)} O(n)$$

Best case  
all 0s

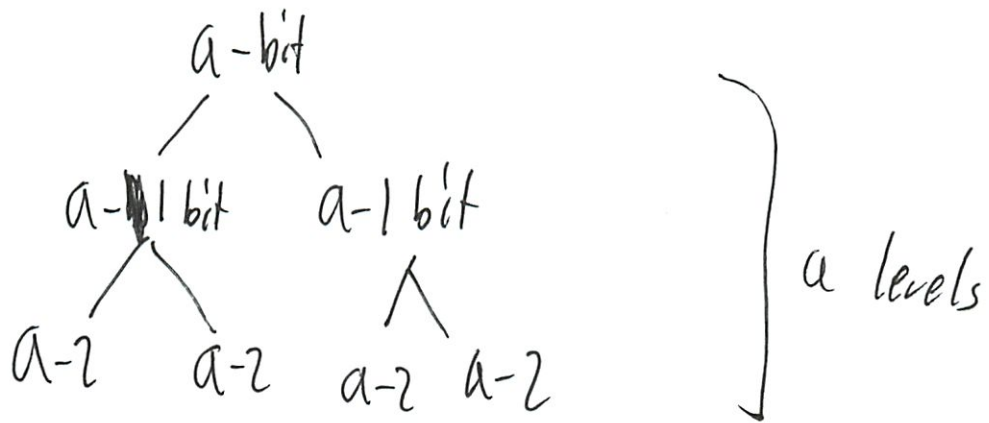


$$\sum_{i=0}^{\lg_2(n)} 2^i = 2n \quad \mathcal{O}(n)$$

6)

c) Can we do better?

An  $a$ -bit number is created from adding two  $a-1$  bit numbers



at most we will have

$$\sum_{i=1}^n 2^i (n-i)$$

$$= O(2^n)$$

and if  $a$  is all 1s it would be

$$a = \lfloor \log_2(n) \rfloor$$

$$\text{So } 2^{\log_2(n)} = n \rightarrow \boxed{O(n)}$$

Michael Plasmeier  
6.046 ~~###~~ A07

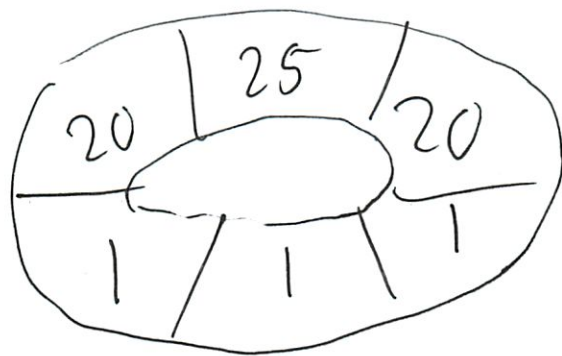
P-Set #1

#4

Pwoods, chris, rahul raj

a) Show greedy approach is not optimal

A counter example:



The greedy solution would return  $25, 1, 1 = 27$

The optimal solution is  $20, 20, 1 = 41$

b) Give an efficient algorithm

Use Dynamic Programming

To speed up, let's sort by room size first  
↳  $O(n \log n)$  as typical

②

Then try each room and the best possible  
~~max~~ of the remainder (taking into account  
the 2 rooms that will be out of  
service)

$$\text{Opt}(V) = \max_{1 \leq i \leq n} (V_i + \text{Opt}(V_{\text{rest}}))$$

This will recurse until no rooms are  
available

Make sure to memoize or else an time  
will be bad.

So for example



pick 6



(3)



pick 4



pick 2

That was 1 particular example (which happened to be optimal)

But for



that would not be

instead we must try each item 1st

then test the rest of the remaining rooms each

This is

$$n + \sum_{i=1}^{n-3} (n-3-2i)$$

Solve for  $i$

$$\text{or } O(n^2)$$

which makes sense

9

The sorting was ~~mostly~~ helpful in practice  
we could build a heuristic where it is  
not worth exploring after a certain point

This will produce a correct answer because  
it will try all possible combinations

---

An  $O(n)$  solution

Same DP idea.

Look at each one individually

take the best of  $V_i + V(i+2)$   $\leftarrow$  include  
 $0 + V(i+1)$   $\leftarrow$  skip

When we get to the end, wrap around

This is  $O(n)$  since we scan the list once

5

So for example

25	20	1	1	1	20
----	----	---	---	---	----

25 + best ~~20~~ 1 1 1 ~~20~~

or ~~this doesn't work it adds~~

20 + best ~~25~~ 1 1 1 20 25

Then it recurses

With memoization this will be  $O(1)$

Since it will go down the list once

---

## Problem Set 1 Solutions

This problem set is due at **11:59pm** on **Tuesday, September 25, 2012**.

---

**Exercise 1-1.** Do Exercise 2.3-3 in CLRS on page 39.

**Exercise 1-2.** Do Exercise 2.3-4 in CLRS on page 39.

**Exercise 1-3.** Do Exercise 3.1-2 in CLRS on page 52.

**Exercise 1-4.** Do Exercise 3.1-3 in CLRS on page 53.

**Exercise 1-5.** Do Exercise 3.1-4 in CLRS on page 53.

**Exercise 1-6.** Do Exercise 4.3-6 in CLRS on page 87.

**Exercise 1-7.** Do Exercise 4.4-8 in CLRS on page 93.

---

### Problem 1-1. Asymptotic Growth

Decide whether these statements are **always true**, **never true**, or **sometimes true** for asymptotically nonnegative functions  $f$  and  $g$ . You must justify all your answers to receive full credit by either giving a short proof (1-2 sentences) or exhibiting a counter-example.

(a)  $f(n) = \Omega(g(n))$  and  $f(n) = o(g(n))$

**Solution:** Never true. By definition,  $f(n) = \Omega(g(n))$  means that for some constant  $c_1 > 0$ , it holds that  $f(n) \geq c_1 g(n) \geq 0$  for large enough  $n$ . On the other hand,  $f(n) = o(g(n))$  would imply that for any constant  $c_2 > 0$ , it holds that  $c_2 g(n) > f(n) \geq 0$  for large enough  $n$ . These statements are directly contradictory because if we choose  $c_2$  to take the value of  $c_1$ , we will get  $c_1 g(n) > f(n)$ , which contradicts  $f(n) \geq c_1 g(n)$ .

(b)  $f(n) = O(g(n))$  and  $g(n) = o(h(n))$  implies  $h(n) = \omega(f(n))$

**Solution:** Always true. By definition,  $f(n) = O(g(n))$  means that for some constant  $c_1 > 0$ ,  $c_1 g(n) \geq f(n) \geq 0$  for large  $n$  and  $g(n) = o(h(n))$  means that for any constant  $c_2 > 0$ ,  $c_2 h(n) > g(n) \geq 0$  for large  $n$ . This implies that for a fixed constant  $c_1 > 0$  and any constant  $c_2 > 0$ , we have  $c_1 c_2 h(n) > f(n) \geq 0$ . Setting  $c_1 c_2 = c$ , we get  $ch(n) > f(n) \geq 0$  for any  $c > 0$ . This can be seen by setting  $c_2 = c/c_1$ , which is allowed because the relation between  $g(n)$  and  $h(n)$  holds for all  $c_2 > 0$ .

(c)  $f(n) + g(n) = \omega(\max(f(n), g(n)))$

**Solution:** Never true.  $f(n) + g(n) = O(\max(f(n), g(n)))$ , so it cannot be  $\omega(\max(f(n), g(n)))$ .

- (d) Rank the following functions by order of growth. In other words, find an arrangement  $g_1, g_2, \dots, g_{16}$  of the functions satisfying  $g_1 = \Omega(g_2)$ ,  $g_2 = \Omega(g_3)$ ,  $\dots$ ,  $g_{15} = \Omega(g_{16})$ . Partition your list into equivalence classes such that  $f(n)$  and  $g(n)$  are in the same class if and only if  $f(n) = \Theta(g(n))$ .

$2^{n^2}$	$\lg n$	$60^{46^{6046}}$	$n!$
$n$	$\sum_{k=1}^n k$	$2^{(2^n)}$	$\sqrt{n} \lg n$
$\log^2 n$	$n2^n$	$\log_4 n$	$n^{\log n}$
$\log \log n$	$n^{2 \log n}$	$3^n$	$n^2$

**Solution:** The functions, grouped asymptotically from largest to smallest, are as follows (functions  $f, g$  on the same line are such that  $f(n) = \Theta(g(n))$ ):

$$\begin{array}{c}
 2^{2^n} \\
 2^{n^2} \\
 n! \\
 3^n \\
 n2^n \\
 n^{2 \log n} \\
 n^{\log n} \\
 n^2 \quad \sum_{k=1}^n k \\
 n \\
 \sqrt{n} \lg n \\
 \log^2 n \\
 \log_4 n \quad \lg n \\
 \log \log n
 \end{array}$$



$$60^{46^{6046}}$$

A few notes about how these results are obtained:

$2^{n^2}$  is asymptotically greater than  $n!$ , which can be seen since its logarithm is asymptotically greater ( $n^2 = \omega(n \log n)$ ), where the latter logarithm comes from Stirling's approximation, which is that  $n! \approx \sqrt{2\pi n}(n/e)^n$ .

$3^n = \Omega(n2^n)$  because their ratio is  $\frac{3^n}{n2^n} = \frac{1}{n}(\frac{3}{2})^n$ . Since  $\frac{3}{2} > 1$ ,  $(\frac{3}{2})^n$  grows faster than  $n$ , which means that the original fraction grows without bound.

$60^{46^{6046}}$ , despite being an unfathomably large number, is still a constant.

### Problem 1-2. Recurrences

Give asymptotic upper and lower bounds for  $T(n)$  in each of the following recurrences. For parts (a)–(e), assume that  $T(n)$  is constant for  $n \leq 2$ . Make your bounds as tight as possible, and justify your answers.

(a)  $T(n) = 3T(n/4) + 5n$

**Solution:**  $T(n) = \Theta(n)$ . Case 3 of the master method.

(b)  $T(n) = 9T(n/3) + n^2$

**Solution:**  $T(n) = \Theta(n^2 \log(n))$ . Case 2 of the master method.

(c)  $T(n) = 5T(n/2) + \log n$

**Solution:**  $T(n) = \Theta(n^{\log_2 5})$ . Case 1 of the master method.

(d)  $T(n) = 4T(\sqrt{n}) + \lg^2 n$

**Solution:** Change variables. Assume that  $n$  is a power of 2, let  $n = 2^m$ . We get  $T(2^m) = 4T(2^{m/2}) + m^2$ . If we define  $S(m) = T(2^m)$ , then we get  $S(m) = 4S(m/2) + m^2$ . By case 2 of the master method,  $S(m) = \Theta(m^2 \lg m)$ . Changing the variable back ( $m = \lg n$ ), we have  $T(n) = \Theta(\lg^2 n \lg \lg n)$ .

(e)  $T(n) = T(n-1) + n$

**Solution:**  $\Theta(n^2)$ . This actually just becomes  $\sum_{k=1}^n k$ , which is just  $\Theta(n^2)$ .

**Problem 1-3. Adding Many Little Numbers**

You have  $n$  numbers that are each a single bit (either 1 or 0). You wish to determine their sum. However, you only have a one-bit adder (with carry). This means, in order to add an  $a$ -bit number and a  $b$ -bit number, you will need to spend  $\Theta(\max(a, b))$  time to add each pair of bits sequentially starting from the least significant bit.

- (a) Assume you simply go through the list adding each bit in turn to a running total. Analyze the running time of this algorithm. What is the upper bound? What can we say about the lower bound? What would you expect the typical running time to be if the 1s and 0s are approximately evenly distributed?

**Solution:** At each step you add the next number to the total of the previous  $n - 1$ . The previous  $n - 1$  have a sum that is  $O(\log n)$  bits. Thus,  $T(n) = T(n - 1) + O(\log n)$ . This has the solution  $T(n) = O(n \log n)$ . We can't say that the algorithm always takes  $\Theta(n \log n)$  time, though—if the numbers are all zeroes, then we only incur a cost of  $\Theta(1)$  at each step (since the running total is 0). However, since we iterate through a list of size  $n$ , we know the algorithm takes  $\Omega(n)$  time. Furthermore, if the 1s and 0s are about even, then the running total after  $n$  steps will be approximately  $n/2$ , which takes  $\Theta(\log n)$  bits to represent; thus, in this case the total running time is  $\Theta(n \log n)$ .

- (b) Give a better algorithm to add the numbers together efficiently, and analyze your algorithm's running time.

**Solution:** Use a standard divide-and-conquer strategy to add smaller numbers together rather than always adding a single bit to an ever-growing number. Thus,

```

ADD(L)
1  if size(L) = 1
2    then
3      return L[0]
4  else
5    return ADD(L[0 : size(L)/2 - 1]) + ADD(L[size(L)/2 : size(L) - 1])
6

```

**Running Time:** At each level, we perform two subcalls on equal-sized subproblems, and we add the two results. Each result, however, is a value that is at most one-half the size of this subproblem, and therefore can be expressed in the logarithm of that many bits. This results in the recurrence  $T(n) = 2T(n/2) + O(\log n)$ . (Note that the number of bits in the subproblem is at most  $\log(n/2) = \log n - 1$ , which is  $O(\log n)$ .) Thus,  $T(n) = \Theta(n)$  by Case 1 of the master method.



- (c) Do you think a different algorithm can perform asymptotically better than the one you presented in part (b)? Why or why not?

**Solution:** It takes  $\Theta(n)$  time to simply read all of the bits, let alone add them to anything. Therefore any algorithm that adds them together must take  $\Omega(n)$  time. Thus, the  $\Theta(n)$  algorithm is asymptotically optimal.

#### Problem 1-4. Donut Building

The donut building is shaped like a circle of  $n$  connected rooms such that each classroom is next to two other classrooms. The rooms are numbered 1 through  $n$  clockwise around the building such that the room  $i$  is next to  $i - 1$  and  $i + 1$ , and room  $n$  and room 1 are next to each other. Each classroom  $1 \leq i \leq n$  has a capacity  $v_i$ , which is the maximum number of students who can be seated in the room. Unfortunately, the walls are not soundproof, and it is impossible to have classes in two neighboring classrooms at the same time. We want to maximize the number of students who can attend class at once. To do this, we wish to select a set of rooms of maximum total capacity. Assume that  $v_i \neq v_j$  for  $i \neq j$ .

- (a) Show by example that the “greedy” approach of selecting the highest capacity  $v_i$  and then continuing with what remains does not necessarily select the set of rooms of maximum total capacity.

**Solution:** Example  $v = \{1, 9, 10, 8\}$ . The greedy approach would select the set  $\{10, 1\}$  which is not optimal. The optimal set is  $\{9, 8\}$ .

- (b) Give an efficient algorithm to find the set of rooms with maximal total capacity. What is the run time?

**Solution:** For each  $k = 1 \dots n$ , let  $C(k)$  be the maximum capacity which can be chosen from the rooms  $\{1, 2, \dots, k\}$  under the assumption that the first room is not chosen

$$C(1) = 0.$$

For  $1 < k \leq n$ ,  $C(k)$  can be calculated by the following recursive equation

$$C(k) = \max\{C(k-1), C(k-2) + v_k\}.$$

Similarly, for each  $k = 1 \dots n$ , let  $D(k)$  be the maximum capacity that can be chosen from  $\{1, 2, \dots, k\}$  under the assumption that the first room is chosen

$$D(1) = v_1 \text{ and } D(n) = D(n-1)$$

since once the first room is chosen, the  $n$ th room cannot be chosen. For  $1 < k < n$ ,  $D(k)$  can be calculated by the same recursive equation

$$D(k) = \max\{D(k-1), D(k-2) + v_k\}.$$

The maximum number of students that can take classes at the same time can be calculated as

$$\max\{D(n), C(n)\}$$

**Running Time:**  $O(n)$  because it takes  $O(1)$  to solve one subproblem for  $C$  or  $D$  and there are  $n$  subproblems for each  $C$  and  $D$ .

6.046  
6.046 Dynamic Programming

9/26

Prof.  
DP is on the -pset

Today: Longest Palindromic Sequence  
Optimal Binary Search trees  
Alt. Coin game

DP works for a lot of problems where greedy doesn't work  
But must memoize  $\rightarrow$  or exponential  
(technically must do this to be DP)

DP Notions

1. Characterize optimal solution  
 $\rightarrow$  defining subproblems
2. Recursively define the value of an optimal soln based on optimal sols of subproblems
3. Compute the value of an opt. sol in bottom-up fashion
  - recurse + memoize (top-down)
  - iterative (bottom-up)

$\hookrightarrow$  more mechanical



Q2

4. Compute the actual opt sol from the computed into

## Longest palindromic subseq

Subseq  $\neq$  substring  
contiguous set of chars

↑  
not contiguous

↓

foobw!    foar

bar

palindrome: string that is unchanged when reversed

radar

+

.bb

redder

input character  
|    ||||

c a c a c    len = 5

3

Need an algorithm to do this

A string  $x[1 \dots n]$   $n \geq 1$   
starting at 1

Then answer must be  $\geq 1$   
<sup>remember</sup>  
L Single character counts

What is the subproblem?

Strategy  $L(i, j)$  = length of longest  
palindromic subseq of  
 $x[i \dots j]$  for  $i \leq j$

def  $L(i, j)$ :

if  $i == j$ : return 1

if  $x[i] == x[j]$ :

if  $j + 1 == j$ : return 2

else: return 2 +  $L(i+1, j-1)$

else: return  $\max(L(i+1, j), L(i, j-1))$

⑨

DP  $\rightarrow$  any time you are stuck  $\rightarrow$  guess  
try everything

So try 1 and then try other to see which is best

Each subproblem is  $O(1)$

$n^2$  subproblems

$$\underline{O(n^2)}$$

But one other issue

- not the max
- it's that it's not memoized!
- So will take exponential time!

Trivial to memoize



$$T(n) = \begin{cases} 1 \\ 2T(n-1) \end{cases}$$
$$= 2^{n-1}$$

$$n=1$$
$$n>1$$

$\epsilon$  should  
have the  
intuition to  
see the patterns

5

Subproblems :  $\binom{n}{2} = \Theta(n^2)$  subproblems  
 $L(i, j) \quad i < j$

$i = j \Rightarrow O(n)$  subproblems

Memoize use a dictionary  
keys are the parameters

but don't know about dictionaries here

could use 2D array  
 $L[i][j]$

$O(1)$  time to look up under either case

So complexity

$$\underbrace{\Theta(n^2)}_{\text{\# subproblems}} \cdot \underbrace{\Theta(1)}_{\substack{\text{time to} \\ \text{solve each} \\ \text{given smaller sized subproblems} \\ \text{have been solved}}} = \Theta(n^2)$$

6

# subproblems  $i < j$

$i < n$

$j < n$

Choose 2 values from  $n$

Or  $n$  values for  $i$

$n$  values for  $j$   
~~And~~

---

2 ← since  $i < j$

So  $\frac{n^2}{2} \rightarrow \Theta(n^2)$

Note we didn't return  
Ans

instead return a tuple

Or drop  $[k]$  /  $k \in \text{take}$   
maintain through recursion

↓

Either  $i$  or  $j$  character

drop  $[i]$  ← true  
 $i+1$

drop  $[j]$  ← true  
 $j+1$



7

# Optimal Binary Search Trees

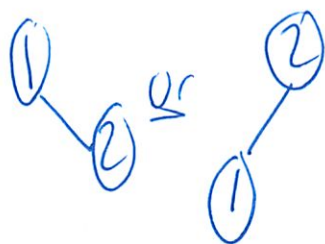
$k_1, k_2, \dots, k_n \in \text{static set of keys}$   
 $= 1, 2, \dots, n \leftarrow \text{does not have to be}$   
 $w_1, w_2, \dots, w_n \leftarrow \text{weights}$

Find BST  $T$  minimizes

$$\sum_{i=1}^n w_i \cdot (\text{depth}_T(k_i) + 1)$$

$\uparrow$  cost function

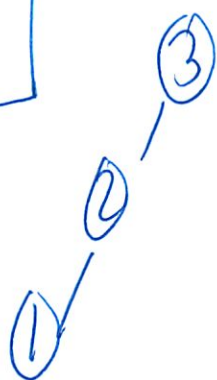
$n=2$   
 $\text{depth} = 0$   
 $= 1$



$$w_1 + 2w_2 \quad 2w_1 + w_2$$

8

$n=3$

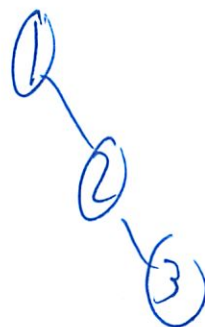


$$3w_1 + 2w_2 + w_3$$



$$2w_1 + 3w_2 + w_3$$

---



$$w_1 + 2w_2 + 3w_3$$

# of options explode fast

perhaps  $\Theta(4^n / n^{3/2})$

verify in  
URS 15.5

want minimal BST that best satisfies

How is this interesting?

- app has static set of keys
- it want to make a dictionary  
(a book in trad. sense)

9

So # steps translation takes is minimized

Could be linked list (missed)

Do look ups ~~where~~  
leaves is further than costs

Add up to get cost function

Strategy  $w(i, j) = w_i + w_{i+1} + \dots + w_j$

$e(i, j) = \text{cost of optimal BST on}$   
 $w_i, w_{i+1}, \dots, w_j$

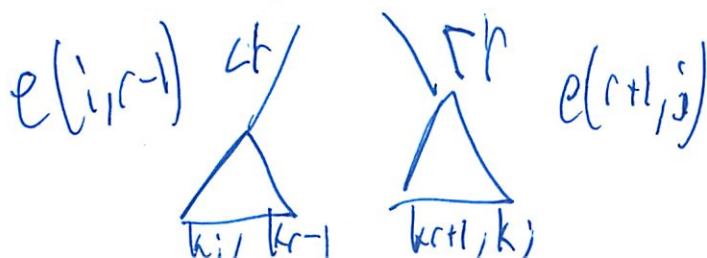
We want  $e(i, n)$

if choose

$(k_r)$  as root

What subproblems do you have

So



10

(I think I understood  
this stuff much better  
now)

Would a greedy strategy work?

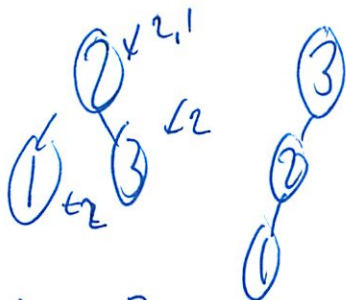
No



$$\begin{array}{ccc} 2w_1 & +w_2 & +2w_3 \\ \downarrow & \downarrow & \downarrow \\ 1 & 1.9 & 2 \\ = 2 + 1.9 + 4 \\ = 7.9 \end{array}$$

Must compare with each other tree  
Might or might not be optimal

Show that highest weight doesn't necessarily  
have to be at the root



$$\begin{array}{l} 2w_1 + w_2 + 2w_3 \\ 2 \cdot 2 + 2.1 + 2 \cdot 2 \\ 10.1 \end{array}$$

$$\begin{array}{l} 3 \cdot 2 + 2 \cdot 2.1 + 2 \cdot 3 \\ 6 + 4.2 + 6 \\ 16.2 \end{array}$$

⑩

So running a DP on this

When don't know which keys to check  
↳ check from all

$$e(i, j) = \begin{cases} w_i & i=j \\ \min_{i \leq r \leq j} (e(i, r-1) + e(r, j) + w(i, j)) \end{cases}$$

Guessing all possible keys for the root

This is  $\Theta(n)$  since the min

$$\text{So } \underbrace{\Theta(n^2)}_{\# \text{ subp.}} \cdot \underbrace{\Theta(n)}_{\text{time each subp.}} = \Theta(n^3)$$



(12)

Game

Remi + Christing

Value  
of Coins

→ 4 42 39 17 25 6

Can only pick outside edge

ie 4, 6

You keep the coins you pick

Want max value

(This is like AT)  
Chess

④ ④② ③⑨ ①⑦ ②⑤ ⑥

Why can the 1st player always win?

$L_n$  even

$V_1$   $V_2$   $V_3$  ...  $V_{n-1}$   $V_n$

(13)

We can look ahead

Can pick  $V_1$  or  $V_n$

$V_1, V_2, V_3 \dots V_{n-1}, V_n$

↑

↑

↑

4 + 39 + 25 = 68    or  
42 + 17 + 6 = 65

$(V_1 + V_3 + \dots + V_{n-1})$  or  
 $(V_2 + V_4 + \dots + V_n)$

Remi + Christina also made a mistake

Remi pick 41

Then Christina has 42 or 6  
either is in her set

Then you always pick what is  
in your subset

(14)

So linear time alg to guarantee victory

But are we satisfied  $\rightarrow$  no!

$V(i, i)$  : max value we can definitely win if it  
is our turn and only  $V_i \dots V_j$  remain

Start w/  $V(1, n)$

$V(i, i)$

$\downarrow$   
just pick  $i$  - no choice

$V(i, i+1)$

$\downarrow$   
pick max

$V(i, i+2)$

$V(i, i+3)$

$\downarrow$  increasing size

(15)

$$V(i, j) = \max \left\{ \begin{array}{l} \text{range is } i+1, j \\ \text{range } i, j-1 \end{array} \right. + V_i + V_j \left. \vphantom{\begin{array}{l} \text{range is } i+1, j \\ \text{range } i, j-1 \end{array}} \right\}$$

pick  $V_i$       pick  $V_j$

$$V(i+1, j)$$

Opponent can pick  $i+1$   
↓  
 $j$

then you are left w/

$$\min \left\{ \begin{array}{l} V(i+1, j-1) \\ V(i+2, j) \end{array} \right\}$$

So



$$V(i, j) = \max \left\{ \min \left\{ \begin{array}{l} V(i+1, j-1) \\ V(i+2, j) \end{array} \right\} + V_i, \min \left\{ \begin{array}{l} V(i, j-2) + V_j \\ V(i+1, j-1) \end{array} \right\} \right\}$$

(16)

Complexity

$$\theta(n^2) \cdot \theta(1) = \theta(n^2)$$

# subproblems    each  
                         subproblem



9/25

6.046

①

L6

# Dynamic Programming

Longest palindromic sequence

Optimal binary search trees

Alternating coin game

## DP notions

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems
3. Compute the value of an optimal solution in bottom-up fashion (recursion & memorization).
4. Construct an optimal solution from the computed information

## Longest Palindromic Sequence

(2)

Def: A palindrome is a string that is unchanged when reversed

Examples: radar, civic, t, bb, redder

Given: A string  $x[1..n]$   $n \geq 1$

To find: Longest palindrome that is a subsequence

Example: Given "character"

output "carac"

Answer will be  $\geq 1$  in length

### Strategy

$L(i, j)$ : length of longest palindromic subsequence of  $x[i..j]$  for  $i \leq j$

def  $L(i, j)$ :

if  $i == j$ : return 1

if  $x[i] == x[j]$ :

if  $i+1 == j$ : return 2

else: return  $2 + L(i+1, j-1)$

else:  
return  $\max(L(i+1, j), L(i, j-1))$

Exercise: compute the actual solution

(3)

## Analysis

As written, program can run in exponential time: Suppose all symbols  $X[i]$  are distinct

$T(n)$  = running time on input of length  $n$

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(n-1) & n>1 \end{cases}$$
$$= 2^{n-1}$$

## Subproblems

But there are only  $\binom{n}{2} = \theta(n^2)$  distinct subproblems: each is an  $(i,j)$  pair with  $i < j$

By solving each subproblem only once, running time reduces to

$$\underbrace{\theta(n^2)}_{\# \text{ subproblems}} \cdot \underbrace{\theta(1)}_{\substack{\text{time to solve} \\ \text{subproblem, GIVEN} \\ \text{that smaller ones} \\ \text{are solved}}} = \theta(n^2)$$

↓  
memoize  $L(i,j)$  value, and look up hash table to see if the subproblem is already solved, else recurse. hash inputs to get output



(4)

## Memoizing Vs. Iterating

- ① Memoizing uses a dictionary for  $L(i, j)$  where value of  $L$  is looked up by using  $i, j$  as a key. Could just use a 2-D array here where null entries signify that the problem has not yet been solved.
- ② Can solve subproblems in order of increasing  $j-i$  so smaller ones are solved first.

## Optimal Binary Search Trees: CLRS 15.5

Given: keys  $K_1, K_2, \dots, K_n$   $K_1 < K_2 < \dots < K_n$   
 weights  $w_1, w_2, \dots, w_n$  WLOG  $K_i = i$   
 (search probabilities)

Find: BST  $T$  that minimizes:

$$\sum_{i=1}^n w_i \cdot (\text{depth}_T(K_i) + 1)$$

Example:  $w_i = p_i =$  probability of searching for  $K_i$

Then, we are minimizing expected search cost.

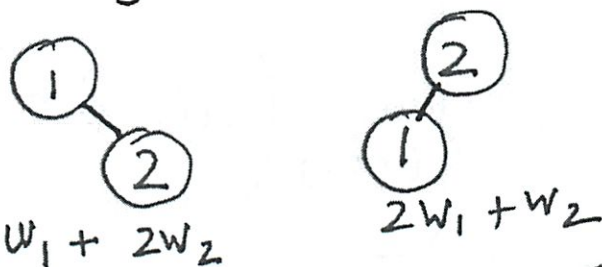
(say we are representing an English  $\rightarrow$  French dictionary and common words should have greater weight.)

5

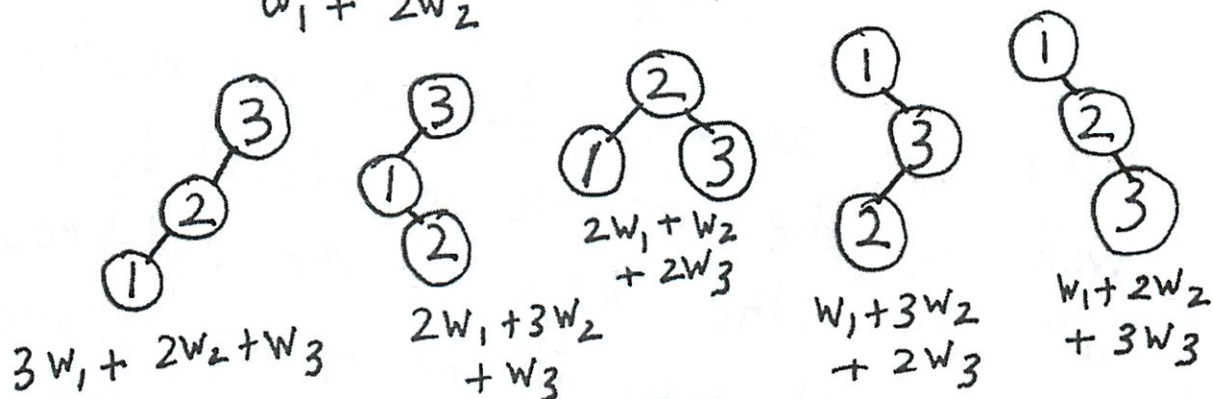
# Enumeration

Exponentially many trees

$n=2$



$n=3$



## Strategy

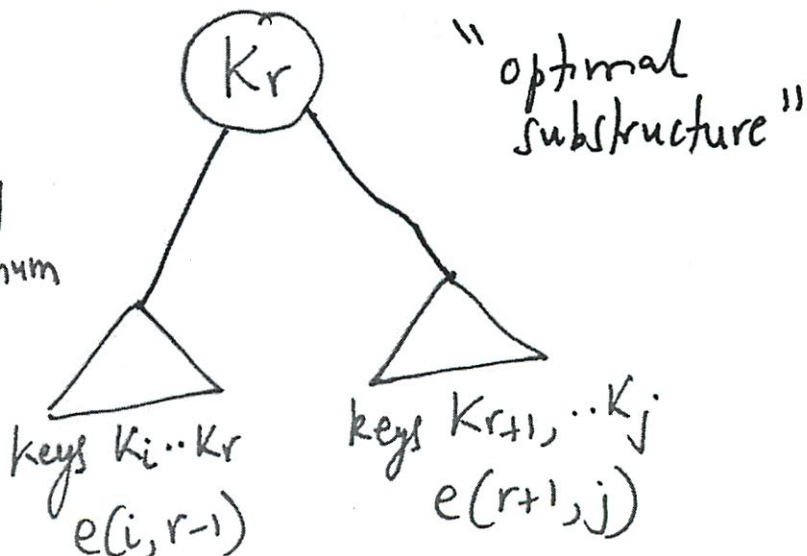
$$w(i, j) = w_i + w_{i+1} + \dots + w_j$$

$e(i, j)$  = cost of optimal BST on  $k_i, k_{i+1}, \dots, k_j$ .  
 Want  $e(1, n)$

Greedy solution?

Pick  $k_r$  in some greedy fashion, e.g,  $w_r$  is maximum

greedy doesn't work





DP Strategy: Guess all roots

(6)

$$e(i, j) = \begin{cases} w_i & \text{if } i = j \\ \min_{i \leq r \leq j} (e(i, r-1) + e(r+1, j) + w(i, j)) & \text{else} \end{cases}$$

+  $w(i, j)$  accounts for  $w_r$  of root  $K_r$  as well as the increase in depth by 1 of all the other keys in the subtrees of  $K_r$ .  
(DP tries all ways of making local choice & takes advantage of overlapping subproblems.)

Complexity:  $\underbrace{\Theta(n^2)}_{\text{\# subproblems}} \cdot \underbrace{\Theta(n)}_{\text{time per subproblem}} = \Theta(n^3)$

$1 \leq i \leq j \leq n$   
 $\binom{n}{2}$  subproblems

Taking the min  
from  $i$  to  $j$

## Alternating Coin Game

⑦

Row of  $n$  coins of values  $V_1, \dots, V_n$   $n$  even

In each turn, a player selects either the first or last coin from the row, removes it permanently, and receives the value of the coin.

### Question

Can the first player always win?

Try: 4 42 39 17 25 6

### Strategy:

1) Compare  $V_1 \quad V_2 \quad V_3 \quad V_4 \quad \dots \quad V_{n-2} \quad V_{n-1} \quad V_n$   
 $V_1 + V_3 + \dots + V_{n-1}$  against  
 $V_2 + V_4 + \dots + V_n$

And pick whichever is greater.

2) During the game only pick from the chosen subset (you will always be able to!)

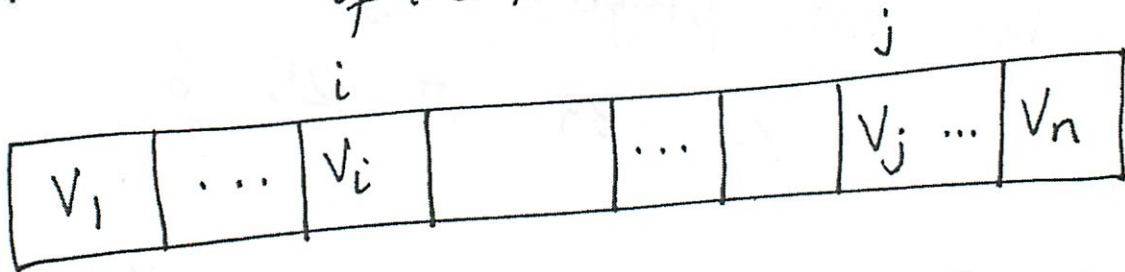
How to maximize the amount of money won assuming you move first?

⑧

## Optimal Strategy

$V(i, j)$  : max value we can definitely win if it is our turn and only coins  $V_i \dots V_j$  remain

$V(i, i)$   $V(i, i+1)$   $V(i, i+2)$   $V(i, i+3) \dots$   
just pick  $i$  for all values of  $i$   
pick the maximum of the two



$$V(i, j) = \max \left\{ \underbrace{\langle \text{range becomes } (i+1, j) \rangle + V_i}_{\text{pick } V_i} \right\}$$

$$\underbrace{\langle \text{range becomes } (i, j-1) \rangle + V_j}_{\text{pick } V_j} \}$$

## Solution

(9)

$V(i+1, j)$  subproblem with opponent picking  
 $\Rightarrow$  we are guaranteed  $\min\{V(i+1, j-1), V(i+2, j)\}$   
Opponent picks  $V_j$       Opponent picks  $V_{i+1}$

We have:

$$V(i, j) = \max \left\{ \min \left\{ \begin{matrix} V(i+1, j-1) \\ V(i+2, j) \end{matrix} \right\} + V_i, \min \left\{ \begin{matrix} V(i, j-2) \\ V(i+1, j-1) \end{matrix} \right\} + V_j \right\}$$

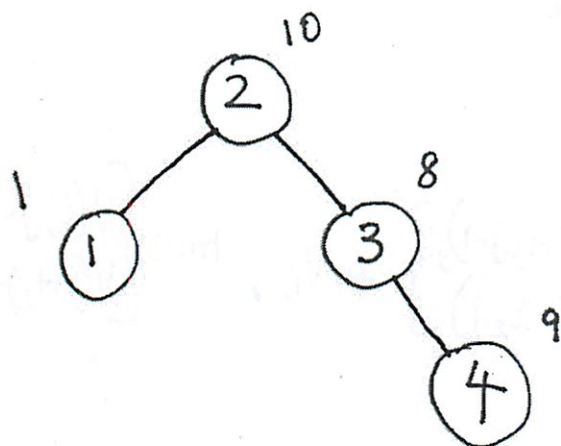
Complexity?

$$\underbrace{\Theta(n^2)}_{\text{\# subproblems}} \cdot \underbrace{\Theta(1)}_{\text{time per subproblem}} = \Theta(n^2)$$

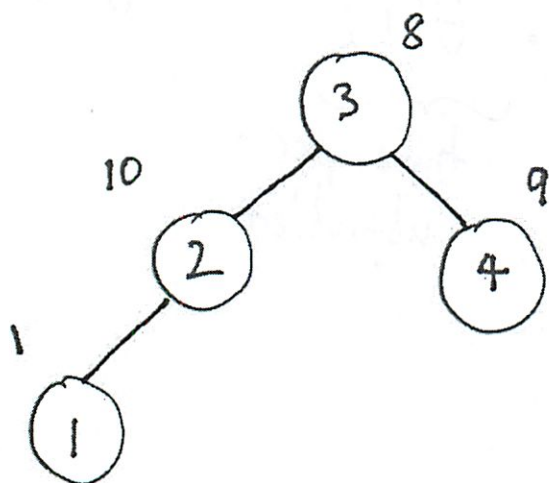


# Example of Greedy Failing for Optimal BST problem

Thanks to Nick Davis!



$$\begin{aligned}
 \text{Cost} &= \overset{w_1}{1} \times \overset{\text{depth}+1}{2} + \overset{w_2}{10} \times 1 \\
 &\quad + 8 \times 2 + 9 \times 3 \\
 &= 55
 \end{aligned}$$



$$\begin{aligned}
 \text{Cost} &= \overset{w_1}{1} \times \overset{w_2}{3} + \overset{w_2}{10} \times 2 \\
 &\quad + \overset{w_3}{8} \times 1 + 9 \times 2 \\
 &= 49
 \end{aligned}$$

Choosing highest weight key of 2  
as root doesn't work.



9/27

6c046  
L7 Shortest Path

Last time: DP

Toby: Shortest path  
but uses DP

---

→ DP #1

- "Matrix Multiplication" - the formulation

- Floyd - Warshall  
↳ better DP

- Johnson's Algorithm

---

Single Source Shortest Path

$$G = (V, E)$$

weights  $w: E \rightarrow \mathbb{R}$

$\forall v \in V$  find

$\delta(s, v)$  shortest weight path  $s \rightarrow v$

$n = |V|$   $m = |E|$  notation for 6c046

②

## Why Shortest Paths?

- Google maps, mapquest
- Internet routing
- How well connected are we
  - ↳ 6 degrees of separation
- Plus other optimization problems
  - Google → scheduling shortest paths

6.006

Unweighted	BFS	$O(n+m)$
non neg weights	Dijkstra	$O(m + n \lg n)$ special data structures
general	Belman Ed	$O(nm)$
acyclic graph	Topo sort + 1 pass Belman Ford	$O(n+m)$

(3)

All pairs shortest paths

$$G = (V, E)$$

$\hookrightarrow$  weights  $w: E \rightarrow \mathbb{R}$

Find  $\delta(u, v) \cdot \forall u, v \in V$

<u>Situation</u>	<u>Alg</u>	<u>Time</u>
Unweighted	n · BFS	$O(n \cdot m)$
Nonneg weights	n · Dijkstra	$O(n \cdot m + n^2 \log n)$
General case	n · Bellman Ford	$O(n^2 m)$

Dense case

$$m = \Theta(n^2)$$

Unweighted

$$O(n^3)$$

nonneg

$$O(n^3 + n^2 \lg n)$$

general

$$O(n^4)$$

4

Today

time

dense case

general

??

$$O(m + n^2 \lg n)$$

$$O(n^3 + n^2 \lg n)$$

Can assume any weight, as long as no  $\ominus$  cycles

\* If have  $\ominus$  weight cycle  $\rightarrow$  then some shortest path does not exist



We have the optimal substructure  $\rightarrow$  subpath of shortest path is a shortest path



5

## Method 1

DP |  $O(n^4)$

There are some choices how to structure multiple may work

Some might be better than the other

$d_{uv}^{(m)}$  = weight of shortest  $u-v$  path  
using  $\leq m$  edges

The guess:

What is the last edge  $(x, v)$  traversed  
on  $u-v$  path?

Recursive Soln

$$d_{uv}^{(m)} = \min_{x \in V} (d_{ux}^{(m-1)} + w(x, v))$$

Assume  $w(x, x) = 0$

base case  
 $d_{uv}^{(0)} = \begin{cases} 0 & \text{if } u=v \\ \infty & \text{if otherwise} \end{cases}$   
Can't go anywhere in 0 steps



6

The min you go around in a cycle  $\Rightarrow$  not shortest path

or

All shortest paths use  $\leq n-1$  edges

$$\text{So } d_{uv}^{(p)} = d_{uv}^{(n-1)} \quad \forall p \geq n-1$$

---

Relaxation algorithm

bottom-up

historical name  $\rightarrow$  don't think about it!

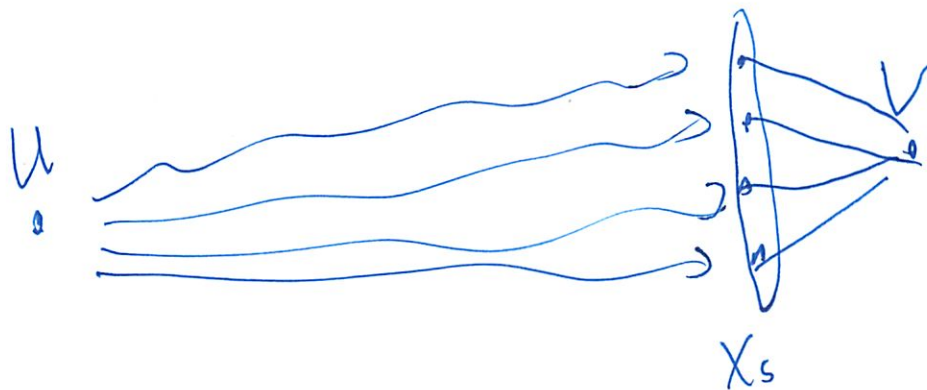
For  $j = 1$  to  $n$

For all  $u \in V$

For all  $v \in V$

For all  $x \in V$

[ So find shortest path  $u \rightarrow v$   
Look at all the  $x$ s (1 before)  
if  $d_{uv} > d_{ux} + w_{xv}$   
 $d_{uv} \leftarrow d_{ux} + w_{xv}$



So ~~all~~ compute all  
 $X_s \rightarrow v$

but this is  $O(n^4)$

Can we do better?

Method 2 Matrix Multiplication  $O(n^3 \log n)$

$A, B$   $n \times m$  matrices

$$C = A \circ B$$

$$C_{ij} = \sum_{k=1}^h A_{ik} \cdot B_{kj}$$

Connection to shortest paths

$$\oplus \equiv \min$$

$$\odot \equiv +$$

⑧  
Then

$$C = A \odot B$$

$$C_{ij} = \min_k (a_{ik} + b_{kj})$$

define  $D^{(k)} = (d_{ij}^{(k)})$

$$W = (w_{ij})$$

$$D^{(k)} = D^{(k-1)} \odot W = \dots = W^{(k)}$$

Since we'd multiplication  
its a power

Alg is to compute this

What is the best way to compute  $w^{(k)}$ ?

How?

a)  $n-2$  multiplications

$$= O(n \cdot n^3) \text{ fine}$$

$$O(n^4)$$

b) repeated squaring

$$(((w^2)^2)^2)$$

9

60  $w^2$  sign? time

$$= (d(i, j))$$

$O(n^3 \lg n)$  time

Can't do Virginia Williams

Transitive Closure - some special case (missed)

---

Method 3 Floyd - Warshall  $O(n^3)$

- important

- Another DP formulation

Problem substructure: larger + larger set of ind. vertices

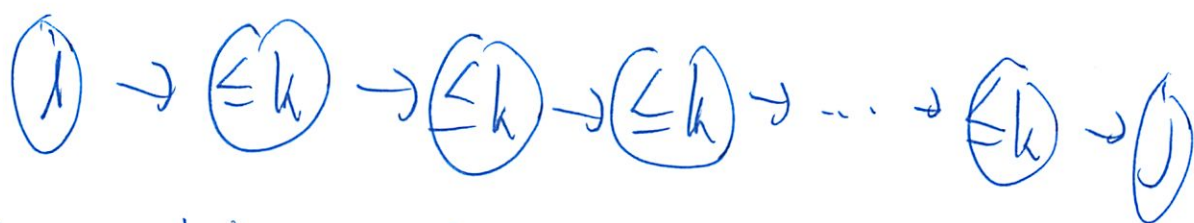
Can use node 1 as a shortcut

Then nodes 1, 2 "

Then " 1, 2, 3 "

10

Subproblem  $C_{uv}^{(x)}$  = wt of shortest  $u, v$   
path w/ intermediate vertices  
only in  $\{i, k\}$



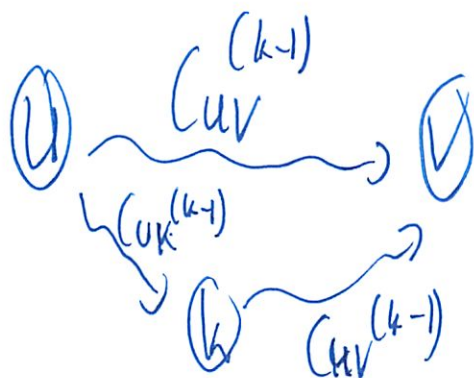
the guess: does shortest path use matrix  $k$ ?

Recursive Soln

$$C_{uv}^{(k)} \in \min \left( C_{uv}^{(k-1)}, C_{uk}^{(k-1)} + C_{kv}^{(k-1)} \right)$$

best sol w/  $k-1 \rightarrow k-1$

Can we use  
vertex  $k$ ?



if see same node twice

either - neg weight cycle  $\rightarrow$  stop

- Cycle  $\rightarrow$  not shortest path

So no node visited 71



(11)

$$C_{uv}^{(0)} = w(u, v)$$

$$\delta(u, v) = C_{uv}^{(1)}$$

So  $C_{uv}^{(0)}$  is all possible paths

relaxation: Our estimates are getting better + better

Bottom up via relaxation

matrix  $C$  ( $w(u, v)$ )

For  $k = 1, \dots, n$

For  $u \in V$

For  $v \in V$

if  $C_{uv} > C_{uk} + C_{kv}$

let  $C_{uv} \leftarrow C_{uk} + C_{kv}$

(12)

if going through  $u$  gives you shorter path  
update  $C_{uv}$

Can omit superscript

↳ it is ok

but need to think about why

---

## Method 4 Johnson Algorithm

- best alg
- not DP
- esp good on sparse graphs
  - each node only points to a few nodes
- want  $D_{ij}$ , but may have  $\ominus$  weights
  - ~~but~~  $\hookrightarrow$  transform so  $\oplus$ 
    - adding a constant to every edge weight
    - but doesn't work
    - (we saw this in 6.006)

(13)

Can we do this while maintaining graph  $\rightarrow$  Yes we can  
Shortest path same  
but lengths may change

$$O(nm + n^2 \lg n)$$

As good as  $Dijkstra$

but can have  $-$  weights

but not  $-$  weight cycles

Graph reweighting given function  $h: V \rightarrow \mathbb{R}$   
~~must~~ get  $h$  in special way  
Shortest path w/ Bellman Ford

Now must reweight edges  $(u, v) \in E$  by

$$w_h(u, v) = w(u, v) + h(u) - h(v)$$

Claim For every  $u, v$  all paths from  $u$  to  $v$   
are reweighted by the same amount  
 $\rightarrow$  So shortest path should be same

(14)

PF

let  $p = \overset{u}{\underset{||}{V_1}} \rightarrow V_2 \rightarrow \dots \rightarrow \overset{V}{\underset{||}{V_k}}$

be any  $u \rightarrow v$  path

any path of any length

$$\text{Then } w_h(p) = \sum_{i=1}^{k-1} w_h(V_i, V_{i+1})$$

$$= \sum_{i=1}^{k-1} (w(V_i, V_{i+1}) + h(V_i) - h(V_{i+1}))$$

So look at each part of sum separately

$$= \underbrace{\sum_{i=1}^{k-1} w(V_i, V_{i+1})}_{w(p)} + \sum_{i=1}^{k-1} h(V_i) - h(V_{i+1})$$

So this is  
 $h(V_1) - h(V_2) + h(V_2)$   
 $- h(V_3) + h(V_3)$

telescoping sum where  
 everything drops out!

$$= w(p) + h(V_1) - h(V_k)$$

$$= w(p) + h(u) - h(v)$$



⑤

Note it does not depend on path

↳ Shortest path will be preserved

Corr  $\bar{\sigma}_h(u, v) = \sigma(u, v) + h(u) - h(v)$

Goal find  $h: V \rightarrow \mathbb{R}$  s.t.

$$W_h(u, v) \geq 0 \quad \forall (u, v) \in E$$

We can use Dijkstra

$$W_h(u, v) \geq 0 \Leftrightarrow w(u, v) + h(u) - h(v) \geq 0$$
$$\Leftrightarrow h(v) - h(u) \leq w(u, v) \quad \forall u, v$$

must be true for all  $u, v$

Called system of difference constraints  
has a soln' that we can find

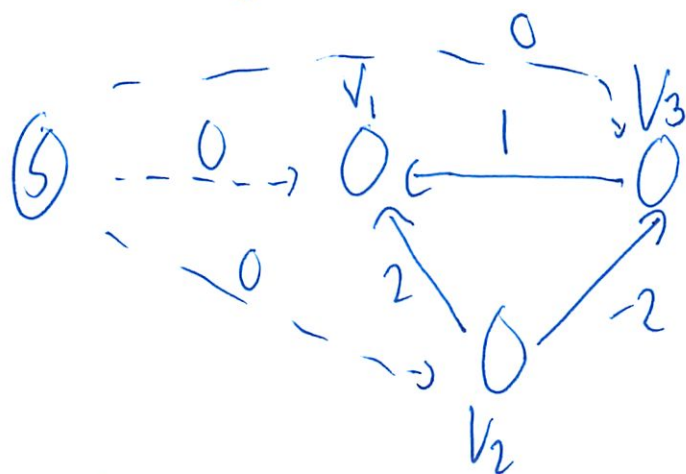


(16)

Theorem 1 if  $\exists$  neg weight cycle, then  
no sol'n to difference constraints

Theorem 2 if no neg weight cycle, there is a  
sol'n and we can find it

✓  
Proof add to G new node S



add  $(s, v) = 0$  for all  $v$

let  $h(v) \leftarrow \overbrace{d(s, v)}^{\text{be shortest path}}$  can compute w/ Bellman Ford

no edges back to S

So no cycles esp of the neg weight kind

So if original has no neg weight cycles, new graph doesn't

(17)

Now what happens to  $h(u, v) - h(u) \leq w(u, v)$

$$\Leftrightarrow \delta(s, v) - \delta(s, u) \leq w(u, v)$$

$$\Leftrightarrow \delta(s, v) \leq \delta(s, u) + w(u, v)$$

Triangle inequality  
So true

So proved is a solution  
and we can find it

### Algorithm

1. Find  $h: V \rightarrow \mathbb{R}$  s.t.  $w_h(u, v) \geq 0 \quad \forall (u, v) \in E$

Set  $h(v) = \delta(s, v)$  using Bellman Ford  
from  $s$

2. Reweight all edges via

$$w_h(u, v) = w(u, v) + h(u) - h(v)$$

3. Run Dij for all source nodes  $u \in V$   
using  $w_h$

4. Reweight all edges via  $w(u, v) = w_h(u, v) - h(u) + h(v)$

Runtime

$$\left. \begin{array}{l} 1. O(n \cdot m) \\ 2. O(m) \\ 3. n \cdot O(m + n \lg n) \\ 4. O(n) \end{array} \right) O(nm + n^2 \lg n)$$

Prove Theorem 1

if  $\exists$  neg wt cycle then no sol to diff constraints

Why? Say  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$   
neg weight

$$\text{if } \exists \text{ sol : } \left. \begin{array}{l} h(v_1) - h(v_0) \leq w(v_0, v_1) \\ h(v_2) - h(v_1) \leq w(v_1, v_2) \\ \vdots \\ h(v_k) - h(v_{k-1}) \leq w(v_{k-1}, v_k) \\ h(v_0) - h(v_k) \leq w(v_k, v_0) \end{array} \right\} \text{sum up}$$

(19)

$$\text{So } \text{sum} = 0$$

$$\text{sum } \cancel{10} < 0$$

Contradiction

# Single Source Shortest Path Problem

- Given digraph  $G=(V,E)$ 
  - with edge weights  $w: E \rightarrow \mathbb{R}$
- $\forall v \in V$  find  $\delta(s, v)$  = shortest path weight from  $s \rightarrow v$

$$n = |V|, \quad m = |E|$$

## Shortest Paths

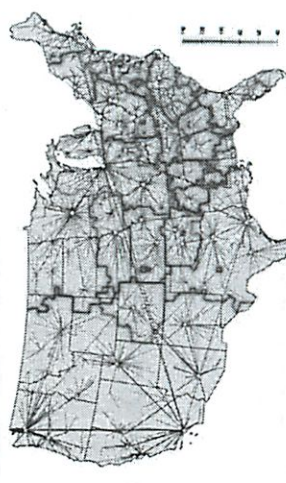
Ronitt Rubinfeld

6.046 Fall 2012

Lecture 7

## Why shortest paths?

- Directions:
  - Google maps, mapquest,...
  - Internet routing



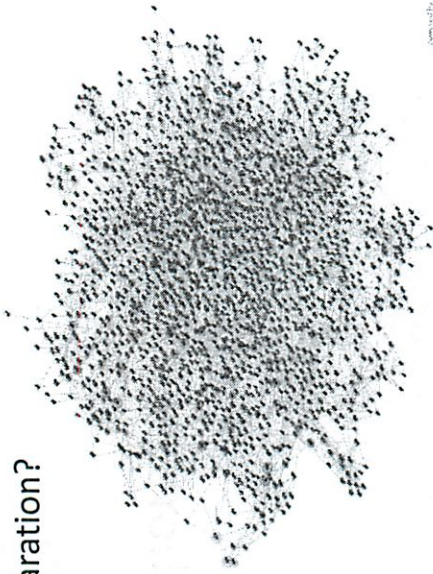
9/27/17  
L7

## Why shortest paths?



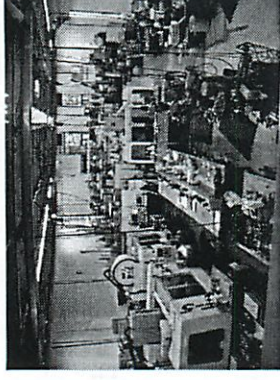
## Why shortest paths?

- How well connected are we?  
– 6 degrees of separation?



## Why shortest paths?

- Relationship with other optimization problems



## Single Source Shortest Path Algorithms

Situation	Algorithm	Time
Unweighted ( $w=1$ )	BFS	$O(n+m)$
Nonnegative weights	Dijkstra	$O(m + n \log n)$
general	Bellman Ford	$O(nm)$
Acyclic graph	Topological sort + 1 pass of Bellman Ford	$O(n+m)$

## Recall:

- If  $G$  has a negative weight cycle, then some shortest paths do not exist!
- Optimal substructure:
  - Subpath of a shortest path is a shortest path!

## Johnson's Algorithm

- Main idea:
  - Want to use Dijkstra's algorithm from each node,
    - but Dijkstra needs positive weights
  - So --- let's transform the weights to make them positive!
    - Can we do this while still keeping the same shortest paths?
    - YES WE CAN!
      - But the path lengths will change, so we'll need to figure out how to transform back to the right path length.

## Johnson's Algorithm

1. Find  $h: v \rightarrow \mathbb{R}$  such that  $w_{h(u,v)} \geq 0, \forall (u,v) \in E$ 
  - Set  $h(v) \leftarrow \delta(s, v)$  using Bellman-Ford from  $s$
2. Reweight all edges via
 
$$w_h(u, v) \leftarrow w(u, v) + h(u) - h(v)$$
3. Run Dijkstra for all source nodes  $u \in V$  using  $w_h$
4. Reweight all edges via
 
$$w(u, v) \leftarrow w_h(u, v) - h(u) + h(v)$$

## Solving difference constraints:

- Multimedia scheduling
- Temporal reasoning
- ...

9/27  
L7

## Lecture 7

### All-Pairs Shortest Paths

- Dynamic Programming #1
- "Matrix Multiplication"-like formulation
  - transitive closure
- Floyd - Warshall  
(Dynamic Programming #2)
- Johnson's Algorithm

# Recall

Single source shortest paths: (6.006, Ch24)  $|V|=n$

- given digraph  $G=(V,E)$

$s \in V$

edge weights  $w: E \rightarrow \mathbb{R}$

$|E|=m$

- find  $\delta(s,v)$  = shortest path weight  $s \rightarrow v \quad \forall v \in V$

<u>Situation</u>	<u>Algorithm</u>	<u>Time</u>
unweighted ( $w=1$ )	BFS	$O(n+m)$
non neg edge weights	Dijkstra	$O(m+n \lg n)$ Fib heaps
general	Bellman-Ford	$O(nm)$
acyclic graph	topological sort + 1 pass Bellman Ford	$O(n+m)$

all are best known

# All-pairs shortest paths

- given graph  $G=(V,E)$   
edge weights  $w:E \rightarrow \mathbb{R}$

$$|V|=n$$

$$|E|=m$$

- Find  $\delta(u,v) \quad \forall \underline{u,v \in V}$

<u>Situation</u>	<u>Algorithm</u>	<u>Time</u>	<u>dense <math>E=\Theta(V^2)</math></u>
unweighted	$n \times \text{BFS}$	$O(nm)$	$O(n^3)$
nonneg wts	$n \times \text{Dijkstra}$	$O(nm + n^2 \lg n)$	$O(n^3)$
general	$n \times \text{B-F}$	$O(n^2m)$	$O(n^4)$
general	TODAY	$O(nm + n^2 \lg n)$ $\uparrow$ good for sparse graphs	$O(n^3)$

all (except 3rd) are best known

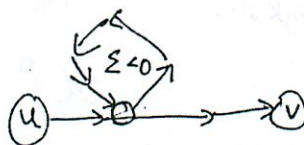
Today: Assume  $V = \{1, 2, \dots, n\}$   
so  $|V|=n$



Recall from 6.006:

Well-definedness:

if  $G$  has a negative wt cycle,  
then some shortest paths do not exist



i.e.  $\delta(u, v) = -\infty$

# Method 1      Dynamic Program      $O(n^4)$

Problem Substructure:

$d_{uv}^{(m)}$  = wt of shortest path  
from  $u \rightarrow v$  using  $\leq m$   
edges

Optimal substructure:

subpath of a  
shortest path is  
a shortest path

the guess:

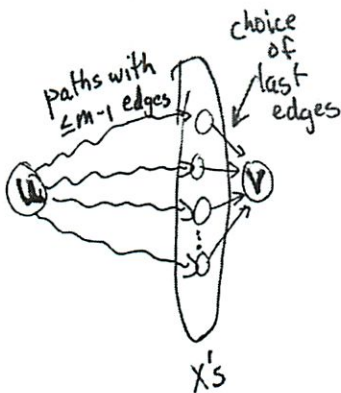
what is last edge  $(x, v)$  traversed on  $u \rightarrow v$ ?

$\Rightarrow$  the recursive solution

$$d_{uv}^{(m)} = \min_{x \in V} (d_{ux}^{(m-1)} + w(x, v)) \quad \text{assume } w(v, v) = 0$$

with "base case"

$$d_{uv}^{(0)} = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{else} \end{cases}$$



Why can we terminate?

assume no neg-wt cycles  $\Rightarrow$  shortest paths are all simple (B-F analysis)

$\Rightarrow$  all shortest paths use  $\leq n-1$  edges

$$\Rightarrow d_{uv}^{(m)} = d_{uv}^{(n-1)} \quad \forall m \geq n-1$$

Bottom up relaxation algorithm:

For  $m = 1$  to  $n$

For all  $u \in V$ :

For all  $v \in V$ :

For all  $x \in V$ :

if  $d_{uv} > d_{ux} + w_{xv}$

$d_{uv} \leftarrow d_{ux} + w_{xv}$  } relaxation step

sanity check: why is it ok to drop superscripts?

Method 2 Matrix Multiplication  $O(n^3 \log n)$

$A, B$   $n \times n$  matrices

$$C = A \cdot B \quad : \quad c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Connection to shortest paths:

$$\oplus \equiv \min$$

$$\odot \equiv +$$

then  $C = A \odot B$

$$c_{ij} = \min_k (a_{ik} + b_{kj})$$

define  $D^{(k)} = (d_{ij}^{(k)})$      $W = (w_{ij})$      $V = \{1..n\}$

$$\Rightarrow D^{(k)} = D^{(k-1)} \odot W$$

$$= W^{(k)} \quad \text{where} \quad W^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \\ \infty & \infty & 0 & \infty \\ & & & \ddots \\ & & & & 0 \end{pmatrix}$$

$W^{(k)}$  makes sense since

$\odot$  associative

algorithm: Compute  $W^{(n-1)}$

•  $n-2$  multiplications  $\Rightarrow O(n^3 \cdot n) = O(n^4)$  time

• repeated squaring:

$$\underbrace{((W^2)^2)^2 \dots}_{\log n} = W^{2^{\log n}} = W^{n-1}$$

$= (\delta_{ij})$  if no neg-wt-cycles

time:  $O(n^3 \lg n)$

• why can't you use Strassen or other fast matrix mult?

Special Case when can use fast matmult:

### Transitive Closure

$$t_{ij} = \begin{cases} 1 & \text{if } \exists \text{ path from } i \text{ to } j \\ 0 & \text{else} \end{cases}$$

$$= [ \text{is } \delta(i,j) \text{ finite?} ]$$

$$\oplus = \text{or}$$

$$\odot = \text{and}$$

$$C_{ij} = \bigvee_k (a_{ik} \wedge b_{kj})$$

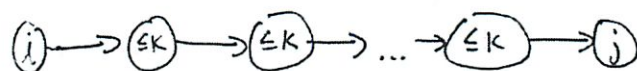
Can use Strassen + repeated squaring  $\Rightarrow O(n^{2.376} \lg n)$   
time

### Method 3 Floyd-Warshall Algorithm $O(n^3)$

Another dynamic program formulation - this one is faster!

### Problem Substructure

subproblem  $C_{uv}^{(k)}$  = weight of shortest  $u \rightarrow v$  path  
with intermediate vertices only in  $\{1, 2, \dots, k\}$



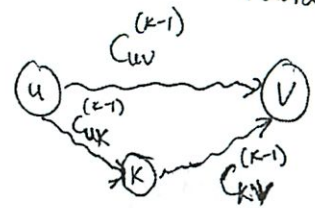
the guess:

does shortest path use vertex  $k$ ?



the recursive solution:

$$C_{uv}^{(k)} = \min (C_{uv}^{(k-1)}, C_{uk}^{(k-1)} + C_{kv}^{(k-1)})$$



note - no neg cycles  $\Rightarrow$  can use node k only once

$$C_{uv}^{(0)} = w(u, v)$$

$$\delta(u, v) = C_{uv}^{(n)}$$

time:  $O(n^3)$  subproblems  $\cdot$  2 choices  $\cdot O(1)$   
 $= O(n^3)$

Bottom up via relaxation

(simple & efficient in practice)

matrix  $C \leftarrow (w(u, v))$

for  $k = 1, 2, \dots, n$ :

for  $u \in V$ :

for  $v \in V$ :

if  $C_{uv} > C_{uk} + C_{kv}$

let  $C_{uv} \leftarrow C_{uk} + C_{kv}$

Check: ok to omit superscripts

# Method 4 Johnson's Algorithm $O(nm + n^2 \log n)$

Main idea: would like to use Dijkstra, but Dijkstra needs positive wts.

can transform wts to positive to get shortest paths!  
(but the new path lengths are not right, so you need to transform them back)

Graph reweighting given fctn  $h: V \rightarrow \mathbb{R}$  will choose this in a special way

reweight each edge  $(u, v) \in E$  by  
 $w_h(u, v) = w(u, v) + h(u) - h(v)$

then  $\forall u, v$ , all paths from  $u$  to  $v$  are reweighted by same amount.

Proof:

Let  $p = v_1 \xrightarrow{u} v_2 \rightarrow \dots \rightarrow v_k \xrightarrow{v}$  be any  $u \rightarrow v$  path (of any length)

$$\text{then } w_h(p) = \sum_{i=1}^{k-1} w_h(v_i, v_{i+1})$$

$$= \sum_{i=1}^{k-1} (w(v_i, v_{i+1}) + h(v_i) - h(v_{i+1}))$$

$$= h(v_1) - h(v_k) + \underbrace{\sum_{i=1}^{k-1} w(v_i, v_{i+1})}_{w(p)}$$

$$= w(p) + (h(u) - h(v))$$

telescoping "action"  
ie.  $h(v_1) - h(v_2) + h(v_2) - h(v_3) + \dots + h(v_{k-1}) - h(v_k)$   
 $= h(v_1) - h(v_k)$

$\Rightarrow$  shortest path preserved (but its weight is offset)  
same for all paths from  $u$  to  $v$

Corollary  $\delta_h(u, v) = \delta(u, v) + h(u) - h(v)$

Goal Find  $h: V \rightarrow \mathbb{R}$  st.

$$w_h(u, v) \geq 0 \quad \forall (u, v) \in E$$

so can run Dijkstra from each vertex.

Need  $w_h(u, v) \geq 0 \quad \forall u, v$

but  $w_h(u, v) = w(u, v) + h(u) - h(v) \geq 0$

iff

$$h(v) - h(u) \leq w(u, v) \quad \forall u, v$$

$\Rightarrow$  system of "difference constraints"

Thm 1 if  $\exists$  negative wt cycle  
 then no solution to difference constraints

Pf. say  $v_0 \rightarrow v_1 \rightarrow \dots \rightarrow v_k \rightarrow v_0$  neg wt.

if  $\exists$  soln then we have:

$$h(v_1) - h(v_0) \leq w(v_0, v_1)$$

$$h(v_2) - h(v_1) \leq w(v_1, v_2)$$

$\vdots$

$$h(v_k) - h(v_{k-1}) \leq w(v_{k-1}, v_k)$$

$$+ \quad h(v_0) - h(v_k) \leq w(v_k, v_0)$$

Sum

$$0 \leq w(\text{cycle}) < 0 \rightarrow \text{contradiction}$$

Thm 2 If no neg wt cycle  
then  $\exists$  solution!

Pf

add to  $G$  a new node  $s$

+ add wt 0 edges  $(s, v) \forall v \in V$

- this introduces no negative wt cycles since can't get back to  $s$
- $s \rightarrow v$  path exists

$\Rightarrow \delta(s, v)$  finite  $\forall v \in V$

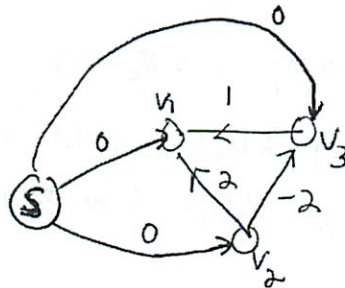
- assign  $h(v) \leftarrow \delta(s, v)$

can use Bellman Ford!

so  $h(v) - h(u) \leq w(u, v) \Leftrightarrow \delta(s, v) - \delta(s, u) \leq w(u, v)$

$\Leftrightarrow \delta(s, v) \leq \delta(s, u) + w(u, v)$

true since it's  $\Delta \neq$



$$h(v_1) = \delta(s, v_1) = 0$$

$$h(v_2) = \delta(s, v_2) = 0$$

$$h(v_3) = \delta(s, v_3) = -2$$

## Johnson's Algorithm

runtime

- Find  $h: V \rightarrow \mathbb{R}$  s.t.  $w_h(u,v) \geq 0 \quad \forall (u,v) \in E$

using Bellman Ford from  $s$   
(+ reweight all edges)

$O(n \cdot m)$

$O(m)$

or find existence of neg cycle + halt

- Run Dijkstra using  $w_h$  from every  $u \in V$  to compute  $\delta_h(u,v) \quad \forall v \in V$   $O(n \cdot m + n^2)$   
(can remove  $s$  now!)

- reweight all  $(u,v)$  pairs via

$$\delta(u,v) = \delta_h(u,v) - h(u) + h(v)$$

$O(n^2)$

$O(nm + n^2 \log n)$

Also:

- B-F can solve any system of difference constraints  $x - y \leq c$

in  $O(V \cdot E)$

where

$V = \# \text{vars}$

$E = \# \text{constraints}$



~~Check~~ another class stole our room

↳ only 2 students

---

## Randomization Algorithms

- Multivariate polynomial identity testing
  - String equality testing
- 

### Univariate Polynomial Testing

$P, Q$

is  $P$  same as  $Q$

Evaluate at  $n$  distinct points

$$D = P - Q = 0?$$

---

Class coverage: way of thinking

②

2 diff alg : deterministic or randomized

Then error is bounded  $\leq \frac{1}{2} \quad \frac{1}{4}$

So what is diff w/ multivariate

$P(x, y) = xy$  over  $\mathbb{R}$

is it 0?

$\infty$  # of points it can be 0

Univariate  $D = 0$  at  $n+1$  points is 0  
no more than  $n$  roots

$$P(x_1, \dots, x_n) = (x_1 + 2x_2 + \dots + x_n)^d$$

if try to expand + compare

$\hookrightarrow$  will be exponential in  $d$

③

[0 - polynomial  $\rightarrow 0$  at every point

What is degree of polynomial?

Defn Total degree of multi variet polynomial  $p$   
is the maximum degree of a particular term

Where the degree of a term is  
the sum of the variable exponents  
 $x_1^2 x_2^3 x_3 + x_1^6 x_2 + x_3^8$

No known deterministic alg.

Must use randomization

Algorithm Eval polynomial at diff pts

Compare to 0

What is the error?

4

Given poly  $P$  over a field  $F$  w/  $n$   
variables  $x_1, \dots, x_n$  want to know  
 $P(x_1, \dots, x_n) \equiv 0$

1.  $S \subseteq F$

choose an arbitrary subset of point

$$|S| \geq 2d$$

2. Choose  $r_1, \dots, r_n$  randomly from  $S$

3. If  $P(x_1, \dots, x_n) = 0$

Then Output  $\equiv 0$

Else output  $\neq 0$

~~will always return same~~

---

If  $P \neq 0$  we want to know prob of error  
↳ Theorem 1



⑤

## Schwartz Zippel Lemma

$$\text{Prob}[P(x_1, \dots, x_n) = 0 \mid P \neq 0] \leq \frac{d}{|S|}$$

When  $x_1, \dots, x_n \in S$

Note: Important That  $S$  is fixed  
( $n$ -dim cube  $S^n$ )

† Choose points from the same  $S$

Can't just choose randomly  
points along a dimension  $\sim n$  dimensions

Proved using induction  $\rightarrow$  see online if interested

So the error when  $P \neq 0$  is  $\leq \frac{d}{|S|} \leq \frac{1}{2}$



# ⑥ Man on the Moon

binary strings  $a, b \rightarrow$  same length  
↳ padded w/ 0s

Want to know  $a \stackrel{?}{=} b$

Want to minimize comm costs

↳ size of max string

General approach: finger ~~printing~~ printing  
↓  
hash!

$P(\text{finger-prints differ}) \approx \text{very high}$

2 solutions

1. Polynomial identity testing ~~and~~

Compare string to polynomial

Compare polynomials

⑦

Remember we want low comm cost  
So willing to do a lot of local com

$$a(x) = \sum_{i=0}^n a_i x^i \quad \text{where } a = a_0, a_1, \dots, a_n$$

Think of strings as polynomial

$$a(x) = b(x)^2$$

But it might be too big

So do things (mod  $\text{prime}_p$ )

So polynomial over  $\mathbb{Z}_p$

So ~~at most~~  $O(\log p)$

prime  $p$  should be small enough so

$$\log p \in O(\log n)$$

②

but big enough so  $P(\text{collision}) = \text{very low}$

## Solution 2

$A = a_0 \dots a_{n-1}$   $\leftarrow$  large binary #s, each  $a_i = 1 \text{ char}$

$$a = \sum_{i=0}^{n-1} 2^i a_i$$

Can do mod  $p$   $a_i \in \{0, 1\}$

Must eval prob of error

$\hookrightarrow$  Detailed notes online

Want to compare ~~diff~~

$$a > b \pmod{p}$$

(must choose randomly  
to guarantee something  
about the error)

$$c = |a - b| \equiv 0 \pmod{p} \text{ iff } p \text{ is a divisor of } c$$

⑨

Want to choose  $T$  (threshold) s.t.

- prime  $p$  is chosen randomly  
from primes  $< T$

- want  $T$  small enough, so  $p$  small

- but want bound on #s that can divide  $C$

So # of prime divisors of  $C$  is at most  $n$

Proof:  $C < 2^n$

Can it have more than  $n$  prime divisors?

No

$$C = p_1^{d_1} \dots p_k^{d_k}$$

$T$  can be bigger or equal to 2

Can only divide  $n$  times

Can't have many prime divisors

the  $p$ s that actually divide  $C$

(10)

If  $p$  actually divides  $c$  - get wrong ans!

So must ... (see sol

$$T = xn \log(xn)$$

$x = \text{large \#}$

Prime \# Theorem says that \# of primes

$< T$  is going to be about

$$\sim \frac{T}{\ln T}$$

$$\text{Will get } p(\text{error}) \leq \frac{n}{\left(\frac{T}{\ln T}\right)} \leq \frac{2}{x}$$

Can choose  $T$  large enough, so small prob  
of error



①

Choose  $\uparrow$  larger than necessary

9/28

## Recitation 3: Randomized Algorithms

### 1 Multivariate Polynomial Identity Testing

**Problem:** Given two polynomials  $P$  and  $Q$ , we want to know whether  $P \equiv Q$ , or equivalently  $P - Q \equiv 0$ .

**Univariate Polynomials.** Recall (from the lecture 5) how we did polynomial identity testing. By evaluating a  $d$ -degree polynomial  $P(x)$  at different points, we can decide if  $P(x) \equiv 0$  or not. We saw a deterministic algorithm running in  $O(d)$  time and a randomized (Monte Carlo) algorithm running in  $O(1)$  time. The crucial fact in those algorithms was that a  $d$ -degree non-zero polynomial can have at most  $d$  roots. What happens if instead of univariate polynomials, we consider multivariate polynomials? Is there an efficient deterministic algorithm polynomial in the total degree and in number of variables?

**Definition 1.** The total degree of a multivariate polynomial  $P$  is the maximum degree of any term in  $P$ , where the degree of a particular term is the sum of the variable exponents.

For example, the total degree of  $x_1^2 x_2^3 x_3 + x_1^6 x_2 x_4$  is 8.

**Multivariate Polynomials.** What is different in the case of multivariate polynomials?

- The number of roots can be infinite. For example, consider  $P(x, y) = xy$  over field of  $\mathbb{R}$  numbers. The number of points where  $P$  evaluates to 0 is infinite. Thus, checking at finite number of points is not enough.
- Expanding polynomials into monomials can result in exponential number of terms. For example, consider  $P(x_1, x_2, \dots, x_n) = (x_1 + x_2 + \dots + x_n)^d$ . Thus, comparing term by term is exponential in degree  $d$ .

In case of multivariate polynomials, there is no known polynomial time algorithm. Thus, randomness seems to be necessary.

## Algorithm

The following randomized algorithm was independently discovered in the late 1970's by DeMillo and Lipton, Schwarz, and Zippel. The general idea is to evaluate the polynomial at random points selected from a sufficiently large range. If any of these values are non-zero, then we report that the polynomial is not identically zero.

**Definition 2.** The notation  $x \in_R S$  means that  $x$  is sampled uniformly at random from the set  $S$ .

Given a polynomial  $P$  over a field  $F$  with  $n$  variables  $x_1, \dots, x_n$  and a total degree  $d$ , we wish to find if  $P \equiv 0$ .

1. Let  $S$  be an arbitrary subset of  $F$  of size at least  $2d$ , i.e.  $S \subseteq F$  and  $|S| \geq 2d$ .
2. Choose  $x_1, \dots, x_n \in_R S$ , i.e. randomly sample  $x_i$  from  $S$ .
3. If  $P(x_1, \dots, x_n) = 0$ , output " $\equiv 0$ ", else " $\not\equiv 0$ ".

If  $P \equiv 0$ , the algorithm always returns the correct answer. If  $P \not\equiv 0$ , then the probability of getting the wrong answer is bounded by the Schwartz-Zippel lemma which states the following.

**Theorem 1** (Schwartz-Zippel Lemma).  $Pr[P(x_1, \dots, x_n) = 0] \leq \frac{d}{|S|}$  when  $P \not\equiv 0$  and  $x_1, \dots, x_n \in_R S$ .

Proof of this fact is done by induction on the number of variables  $n$ . (If interested, the proof can be found on Wikipedia). It is important that the point  $(x_1, \dots, x_n)$  is chosen uniformly at random from  $n$ -dimensional cube  $S^n$ . Arbitrary distributions, even over large support, will not in general work because a multivariate polynomial can have lots of zeros.

Thus, probability of getting the wrong answer in the case of  $P \not\equiv 0$  is bounded by  $\frac{d}{|S|} \leq \frac{1}{2}$ . We can amplify the probability of correctness as needed. This randomized algorithm is very useful because no efficient deterministic algorithm is known.

## 2 Man on the Moon problem or Testing Equality of Strings

Testing equality of strings has many applications in real life. For example, multiple copies of the same document may exist in different places and we would like to know if the two copies are equal. However, the communication cost may be expensive and we would like to test the equality without sending the whole document over the communication channel. This problem is commonly called “man on the moon problem”.

Given two binary strings  $a$  and  $b$ , we would like to know if  $a = b$ . There are many solutions possible, but the general approach is to use fingerprinting: apply some function that is unlikely to be equal for non-equal strings and compare the values. We will consider two different solutions.

### 2.1 Solution 1

Let  $a = a_0a_1a_2 \dots a_n$  and  $b = b_0b_1b_2 \dots b_n$  (where  $a_i, b_i \in \{0, 1\}$ ). We can think of a binary string as a polynomial:

$$a(x) = \sum_{i=0}^n a_i x^i$$

and similar for  $b$ . If we think of string  $a$  and  $b$  as polynomials, then two strings are equal iff  $a(x) \equiv b(x)$ . We already know how to solve this problem. If we choose random  $r$  from set  $S$ , then probability that  $a(r) = b(r)$  is at most  $\frac{n}{|S|}$ . By choosing  $|S| > 2n$ , we can get more than  $\frac{1}{2}$  chance of detecting the difference between strings. We can make sure that  $r$  requires only  $\log n$  bits to represent, but  $a(r)$  and  $b(r)$  can be very large numbers, even larger than the original string because the degree of the polynomial is  $n$ . This is bad, because we’re trying to reduce the number of bits that we transmit.

The solution to this problem is to work over a finite field instead of over  $\mathbb{Z}$ . Let’s operate over  $\mathbb{Z}_p$ , i.e. do all calculations mod  $p$ , where  $p$  is some prime number. Now,  $a(r)$  and  $b(r)$  require at most  $\log p$  bits to represent. What are the constraints on the value of  $p$ ?



- prime  $p$  must be large enough to have enough distinct values for set  $S$ . Thus,  $p > |S|$  should hold.
- prime  $p$  should be small enough so that  $\log p$  is  $O(\log n)$ .

We can see that by operating over  $\mathbb{Z}_p$ , we can reduce the number of bits sent to be  $O(\log n)$ .

## 2.2 Solution 2

Another way of solving the same problem is to think of strings  $a$  and  $b$  as large integer numbers given in the binary representation  $a = a_0a_1 \dots a_{n-1}$  and  $b = b_0b_1 \dots b_{n-1}$ , i.e. the integer values are

$$a = \sum_{i=0}^{n-1} 2^i a_i$$

and similar for  $b$ . Define fingerprint function

$$F_p(x) = x \mod p$$

for a prime number  $p$ . Instead of comparing strings  $a$  and  $b$  directly, we will compare  $F_p(a)$  and  $F_p(b)$  for some random prime  $p$ . The number of transmitted bits are now  $O(\log p)$ . We want  $p$  to be small.

We would like to know what is the probability of the error when  $a \neq b$ . Let  $c = |a - b| \neq 0$ , then  $F_p(a) = F_p(b)$  iff  $c \equiv 0 \mod p$ . Hence,

$$Pr[F_p(a) = F_p(b) | a \neq b] = Pr[c \equiv 0 \mod p]$$

We know that  $c < 2^n$ , this means that the number of distinct prime divisors of  $c$  is at most  $n$ . Choose a threshold  $T > n$  and choose  $p$  to be random prime less than  $T$ . Use the prime number theorem to evaluate the number of primes less than  $T$ .

**Theorem 2** (Prime Number Theorem). *For any  $k \in \mathbb{N}$ , let  $\pi(k)$  be the number of distinct primes less than  $k$ . Then, asymptotically*

$$\pi(k) \approx \frac{k}{\ln k}$$



The number of primes smaller than  $T$  is  $\pi(T) \approx \frac{T}{\ln T}$ . Among those prime numbers, at most  $n$  can be a divisor of  $c$ . Choose  $T = tn \log tn$  for some large constant  $t$ . Then,

$$Pr[F_p(a) = F_p(b) | a \neq b] < \frac{n}{T/\ln T} < \frac{2}{t}$$

(for large enough  $n$ ). We have a good chance of finding the difference in  $a$  and  $b$ . The number of transmitted bits are  $O(\log T) = \log(n)$

Notice that in both solutions, the randomness is necessary to guarantee the bound on the probability of the error.

Missed lecture due  
to interview

## Lecture 8

### Greedy Algorithms & Minimum Spanning Tree (MST)

- MST problem definition

- greedy choice

- Prim's algorithm

- Kruskal's algorithm

- Union Find

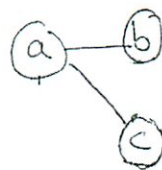
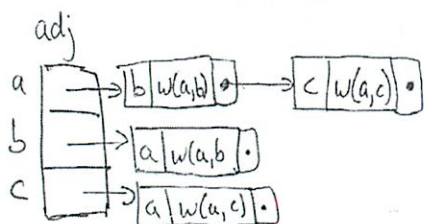
## Quick Review

undirected graph  $G=(V,E)$  with weight fctn  $w: E \rightarrow \mathbb{R}$

$|V|=n$   
 $|E|=m$

adjacency list representation of  $G$

- for each  $u \in V$ ,  $\text{Adj}[u]$  is a linked list of  $u$ 's nbrs  
ie.  $\{v \mid (u,v) \in E\}$



connects every (or most) vertices  
w/o loops

Tree - connected graph with no cycles

maximal set of edges

Spanning tree of  $G$  -

subset of edges that form a tree  
spanning all vertices  
(touching)

Fact: spanning tree has  $n-1$  edges

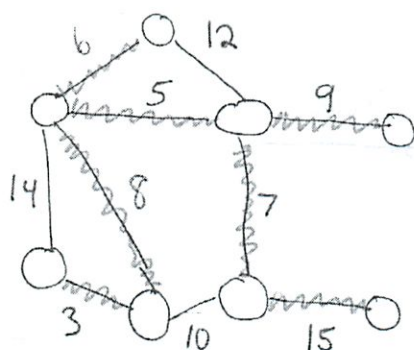
# Minimum Spanning Tree (MST) Problem:

given:  $G=(V,E)$  + edge wts  $w: E \rightarrow \mathbb{R}$

find: spanning tree  $T \subseteq E$  of min wt:

$$w(T) = \sum_{e \in T} w(e) \quad \text{weighted}$$

example



weights  
min = MST

Is it unique?

-if edge wts distinct, yes!

-else maybe could be tree w/ same weights

Applications?

Many!

e.g. connecting cities with min amt of fiber optic cable  
not the speediest

Try to find

best (min wt)  
subset (T) tree;  
of given set ( $\subseteq E$ ) edges  
that is legal (connected / acyclic i.e. spanning tree)

"Greedy" approach:

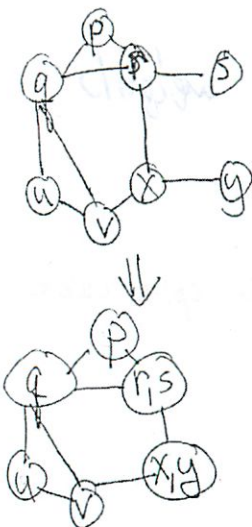
- locally optimal choices might lead to  
globally optimal solution

- works if (a) can identify some edge in MST  
 (b) after committing to that edge,  
 remaining problem has same form  
 = "optimal substructure"

So no backwards?  
 well its not DP

In our example:

we know  $s, y$  have to be in MST  
 why



Can "contract" edges to make new graph  
 $G'$  with supernodes  $(q, r, s)$  &  $(x, y)$  ah  $\rightarrow$  combine  
 what are weights of new edges?  
 how do we keep track of weight of contractions?  
 minimum sum up



The "choice":

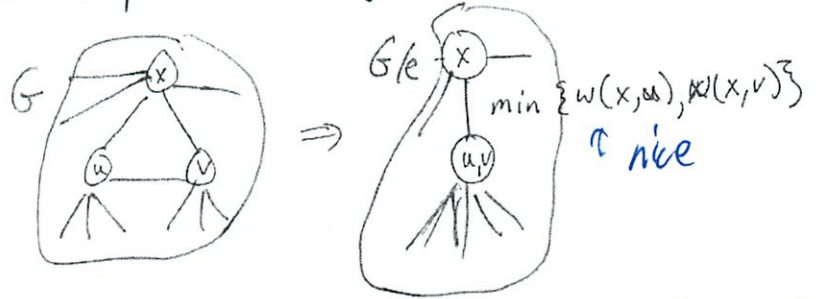
i.e. if edge  $e = (u, v)$  Known to be in MST

"Merge" operation:

Contract( $e$ ): merge  $(u, v)$  to  $(uv)$

- if multiple copies of edge, keep only lowest weight

denote by  $G/e$

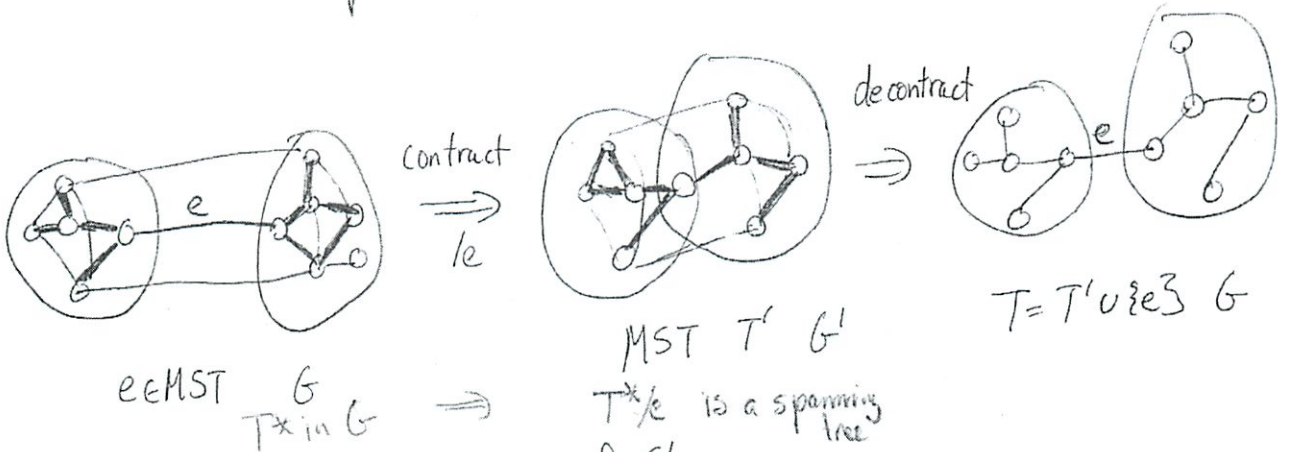


Claim if  $T'$  is an MST for  $G' = (V', E') = G/e + e$  and  $e$  is in some MST  $T^*$  of  $G$ , then  $T = T' \cup \{e\}$  is an MST for  $G$ .

remap edges to "pre-contracted form"

one edge?  $e = (u, v)$

what is the divided by symbol?



pf  $T^*/e$  is spanning tree of  $G'$

$\Rightarrow w(T') \leq w(T^*/e)$

$\Rightarrow w(T) = w(T') + w(e)$

$\leq w(T^*/e) + w(e)$

$= w(T^*)$

$\Rightarrow T$  is MST

since  $T'$  is MST

construction/definition of  $T$

by previous step!

def of  $T^*/e$

Moral:

if we have procedure to find any edge  $e$  in any MST  
we can commit + solve remaining smaller  
problem on  $G' = G/e$

at end, grow MST  $T$  edge by edge  
by uncontracting

"Supernodes" correspond to connected components  
found so far.

Don't get how showed greedy works

Dynamic program attempt:

guess edge  $e$  in MST  
Contract  $e$  to get subproblem  
recurse  
decontract + add  $e$

problem  
lots of subproblems!

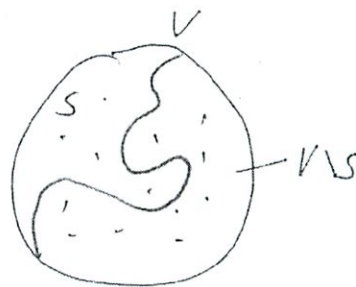
Def a cut of  $G=(V,E)$  is partition of  $V$

into 2 nonempty subsets:

$$S + V \setminus S$$

neither  $= \emptyset$

(denote by  $(S, V \setminus S)$ )



How do we pick an edge that is in some MST?

Idea For any cut  $(S, V \setminus S)$

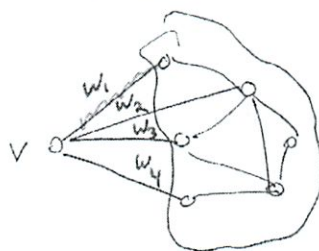
pick min wt edge  $e=(u,v)$  "crossing cut"  
i.e.  $u \in S, v \in V \setminus S$

Thm  $\exists$  MST  $T^*$  st.  $e \in T^*$

before we prove thm:

example

$$S = \{v\}$$

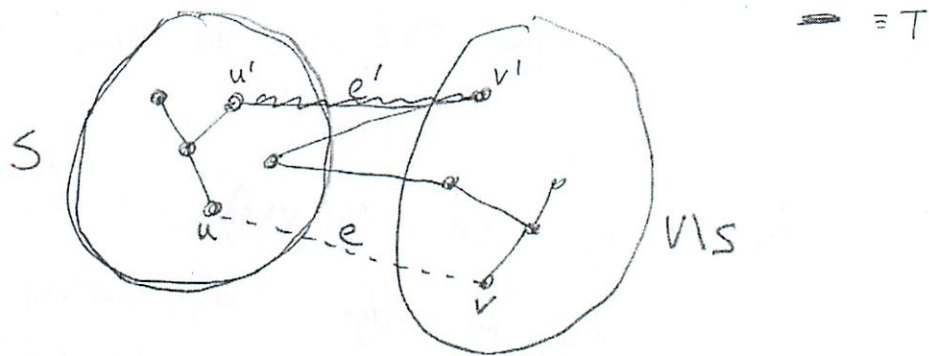


if  $w_1 < w_2 < w_3 < w_4$ :

Thm  $\Rightarrow w_1$  must be in MST

Pf

- let  $T$  be MST
- If  $e$  in  $T$ , we are done ( $T^* \leftarrow T$ )
- Else, assume  $e \notin T$   
 $(u, v)$  st.  $u \in S, v \in V \setminus S$



- there is path from  $u$  to  $v$  in  $T$
- let  $e' = (u', v')$  be 1st edge on path crossing cut  
 (path must cross cut at least once since  $u \in S, v \in V \setminus S$ )

• let  $T^* \leftarrow T \setminus \{e'\} \cup \{e\}$

•  $T^*$  is spanning tree:

any path that used  $e'$  can be restructured to use  $e$

•  $w(e) \leq w(e')$  since  $e$  is min wt across cut

• so  $w(T^*) = w(T) - w(e') + w(e) \leq w(T)$

$\Rightarrow T^*$  is also MST

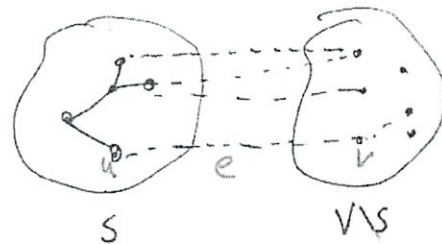
Prim's AlgorithmHow is this diff. than earlier?idea grow supernode until it includes all of  $V$ 

$$S \leftarrow \{v_0\}$$

arbitrary start pt

$$T \leftarrow \emptyset$$

empty tree

while  $S \neq V$ :let  $e \leftarrow \min \text{wt edge } (u,v) \text{ crossing from}$   
 $S \text{ to } V \setminus S$ add  $v$  to  $S$ add  $e$  to  $T$ Output  $T$ 

Why correct?

previous thm + induction,  $T$  is always subset of some MST  $T^*$



Efficient Implementation

- maintain priority  $Q$  on  $V \setminus S$ :

priority  $Q$ :

- Collection of pairs  $(v_1, k(v_1)), (v_2, k(v_2)), \dots$

elements  $v_1 \dots v_k$   
keys  $k(v_1) \dots k(v_k)$

here: • elements are nodes in  $V \setminus S$   
• keys are min wt edge to  $S$   
ie.  $k(v) \leftarrow \min_{u \in S} (w(u, v))$   
or  $\infty$  if no such  $(u, v)$

- Operations supported

extract\_min - output  $v \in Q$   
st.  $k(v)$  minimized

decrease-key - given  $v$  + new (lower)  $k(v)$   
updates value of  $k(v)$

- Algorithm:

- Initialize  $Q$ :  $\{S \leftarrow \emptyset\}$

$Q \leftarrow$  all  $v \in V$  with  $k(v) \leftarrow \infty$

- pick arbitrary start vertex  $v_0$

$k(v_0) \leftarrow 0$

- while  $Q \neq \emptyset$ :

$v \leftarrow \text{extract\_min}(Q)$

for  $x \in \text{adj}[v]$

if  $x \in Q$  and  $w(v, x) < k(x)$

$k(x) = w(v, x)$

decrease-key  $(Q, x)$

$x.\text{parent} = v$

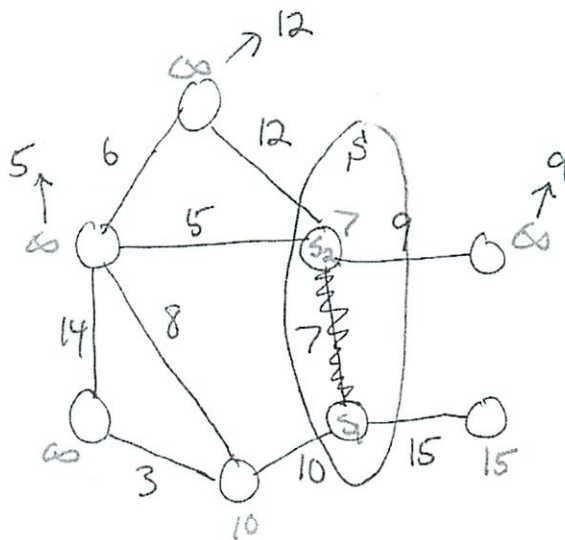
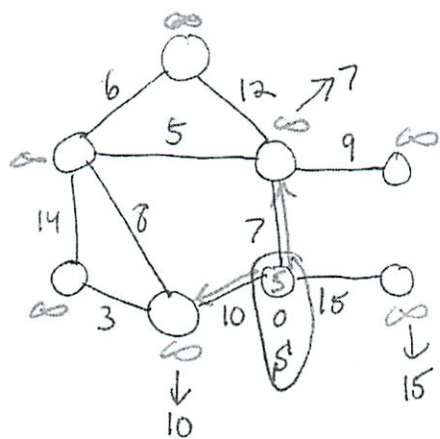
as MST

Finds new min wt edge to  $S$

ie. remember that  $(v, x)$  is MST edge

- return  $\{(v, v.\text{parent}) \mid v \in V \setminus \{v_0\}\}$

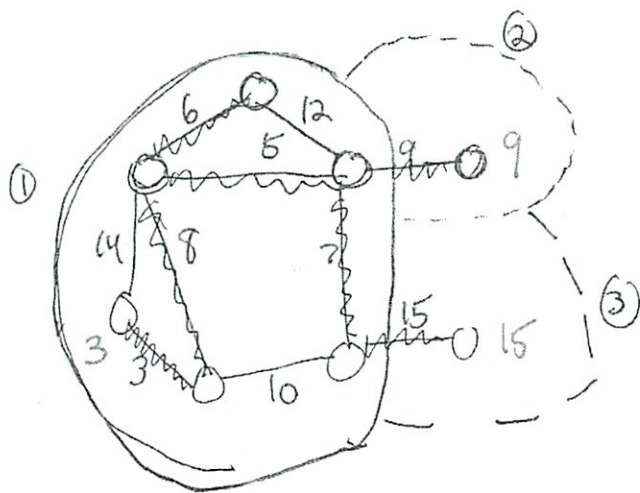
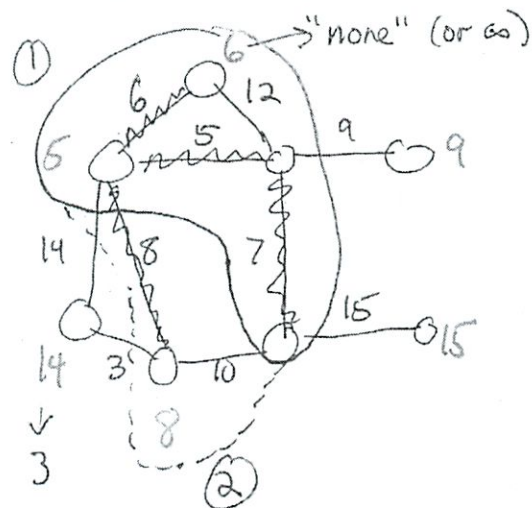
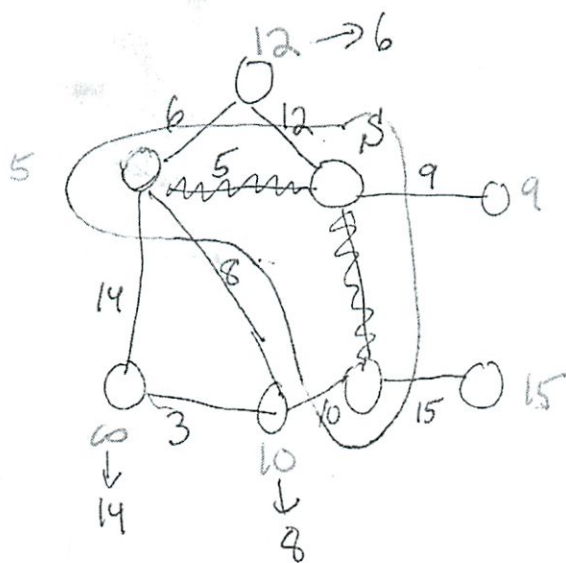
Example :



Initially  $Q \leftarrow S_1$   
Keys of nbrs of  $S_1$   
are decreased

$Q \leftarrow Q \cup \text{nbr with min at edge } S_2$   
Keys of nbrs of new node  
decreased

edge  $(S_1, S_2)$  put in MST  $T$



Time: (let  $n = \# \text{nodes}$ ,  $m = \# \text{edges}$ )

$$= O(n) \cdot T_{\text{extractmin}} + O(m) \cdot T_{\text{decrease-key}}$$

↑  
Since each edge can  
be decreased at most twice  
(once from each endpt)

<u>priority queue implementation</u>	<u><math>T_{\text{extractmin}}</math></u>	<u><math>T_{\text{decreasekey}}</math></u>	<u>Total</u>
array (nothing)	$O(n)$	$O(1)$	$O(n^2)$
binary heap	$O(\log n)$	$O(\log n)$	$O(m \log n)$
Fibonacci heap (CLRS ch19)	$O(\log n)$	$O(1)$	$O(m + n \log n)$
	<div style="text-align: center;"> <span style="border-top: 1px solid black; display: inline-block; width: 150px;"></span>  amortized </div>		

# Greedy Algorithms and Minimum Spanning Tree

Ronitt Rubinfeld

Lecture 8

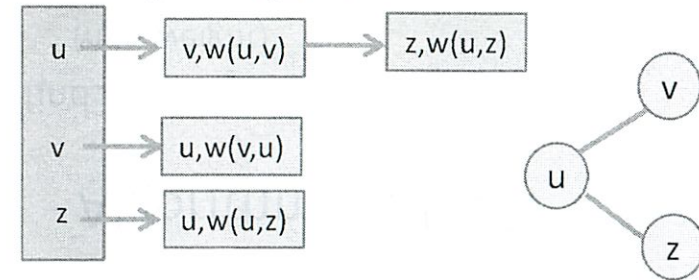
6.046 Fall 2012

## Trees

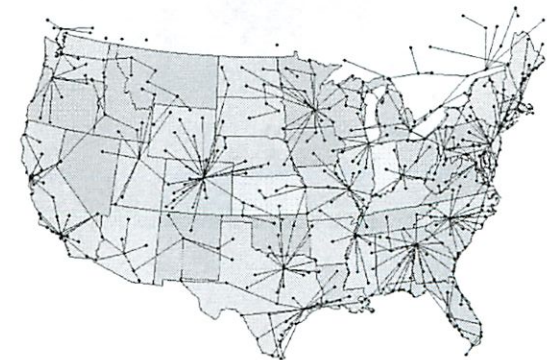
- Connected graph with no cycles
- Spanning tree of a graph  $G=(V,E)$ :
  - Subset  $T$  of edges that form a tree touching all vertices
  - Fact: Spanning tree has  $n-1$  edges

## Quick review

- Undirected graph  $G=(V,E)$ 
  - Weight function  $w: E \rightarrow \mathbb{R}$
  - Representation:
    - Adjacency list: for each  $u \in V$ ,  $Adj[u]$  is a linked list containing  $u$ 's neighbors



## Spanning tree of airports

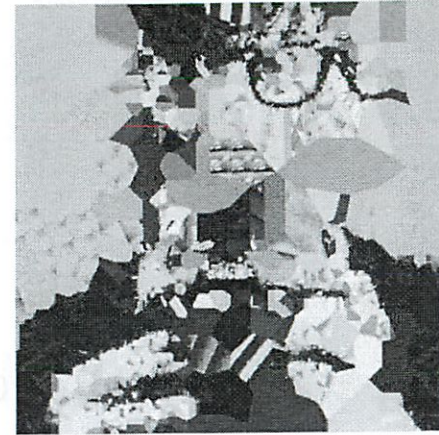




## MST of a city



## MSTs and art



## Is MST unique?

- If edge weights distinct, then YES!
- If not distinct, then MAYBE

## Algorithm for MST:

- Try to find:
  - Best (min weight)
  - Subset (T)
  - Of a given set (E)
  - That is legal (connected, acyclic...a spanning tree!)



## Greedy approach

- Locally optimal choices might lead to globally optimal solution
- Works if:
  - Can identify some edge of an MST
  - After committing to that edge, remaining problem has same form, i.e., “optimal substructure”

## Moral

- If find ANY edge  $e$  in ANY MST can commit to it and solve smaller problem on  $G' = G \setminus e$ 
  - “supernodes” correspond to connected components found so far
- At end output contracted edges