6.006 Lecture 19

4/24

Quiz tomorrow

Will have small qv on Dynamic Programming

Today: More DP

Going to improve our $O(n^4)$

DP ≈ Recursion + Memoization

Fibonacci, Crazy 8s, SPPs

When the sol can be produced by combining sols to subproblems

$F_n = F_{n-1} + F_{n-2}$

Subproblems when combo of sub-subproblems

$F_{n-1} = F_{n-2} + F_{n-3}$

Efficient when # subproblems is polynomial

$F_1, \ldots, F_n$

②

Try to think if you meet the 2 conditions!

All-Pairs Shortest Path

want $\delta(i, j)$

Assume no $\ominus$ weight cycles

DP Approach: Build Adj Matrix A

Also $d_{ij}(m)$ weights of shortest path that uses at most m edges

Since we want $d_{ij}(n-1)$

So (missed analysis)

For $m = 1, 2, \ldots, n-1$

$$d_{ij} = \min_k \left(d_{ik}(m-1) + a_{kj} \right)$$

(Don't fully get...)

But - missing the immediate path

$k = j$ so $a_{kj} = 0$ - must make sure to include

Basically what Bellman Ford "relaxation" is

└ still n iterations

How long to compute $n-1$?

$O(n^4)$ each step $O(n^3) \cdot n$ times

↑ since iterating over ks → $O(n^2)$ over the i,js

Basically Bellman Ford from all possible vertices.

Need to define the right subproblem

$d_{ij}(m)$ weight of shortest path $i \to j$ that

Only uses intermediate vertices from set $\{1, \ldots, m\}$

Want $d_{ij}(n)$

└ $O(n^3)$ things need to compute

Must justify there is a recursion subproblems meet

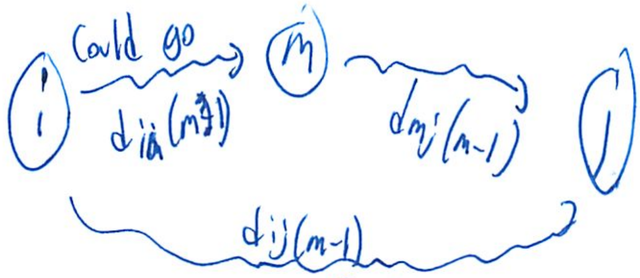
④

$d_{ij}(0) = a_{ij}$ ← no intermediate nodes

$d_{ij}(m-1), d_{ij}(m-2), \ldots, d_{ij}(0)$ $\forall i,j$

How do we get $d_{ij}(m)$

↑ same nodes as before, but
can also use node m



↑ but could also be

$d_{ij}(m) = \min\left(d_{ij}(m-1),\ d_{im}(m-1) + d_{mj}(m-1)\right)$

↑ must be the best path
i→m

So running time $O(n^3)$

- Only 1 possible intermediate node → so $O(1)$

Called Floyd-Warshall

Longest Common Subsequence

popular in bio-informatics - trying to find long matches in genome
or in diff

Given 2 seq X[1.... m] and y[1...n]
and find a LCS(x,y) common to both

X: A B C B D A B
Y: B D C A B A

Brute Force Sol

~~Given X[1...m]~~

~~Check~~ For every sub seq in X, check if in y

2^m subseq of x

n to check each one

so $(2^m)n$

Or check letter by letter?

Need to find substructures that are useful

What sounds like a subproblem that is useful to solve?

- Prefix of x $x[1, \dots, i]$ = ith prefix of $x[1 \dots m]$
- Prefix of y $y[1, \dots, j]$ = jth prefix of $y[1, \dots, n]$

Subproblem define $c[i, j] = LCS(x[1 \dots i], y[1 \dots j])$

How do we compute this?

1. LCS and end w/ $x_i = y_j$

x_1	x_2	$\dots$	x_{i-1}	x_i
y_1	y_2	$\dots$	y_{j-1}	$y_j = x_i$

So $c(i, j) = c(i-1, j-1) + 1$ if $x_i = y_i$

⑦

B AN ANA

A T A ANA = ④

$\longleftarrow$ work

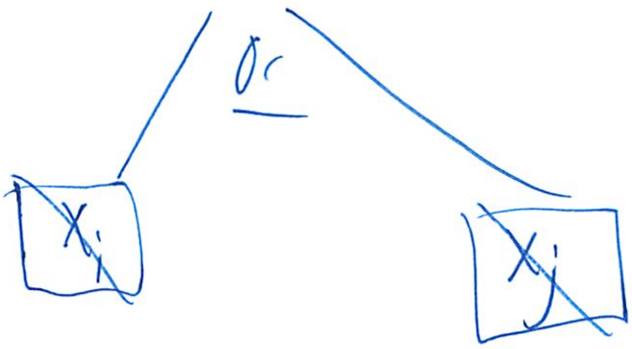
2. $x_i \neq x_j$

$$\boxed{x_1\ x_2\ \cdots\ x_{i-1} \mid x_i}$$

$$\boxed{y_1\ y_2\ \cdots\ y_{j-1} \mid x_j \neq x_i}$$

Can't match - since nothing before

Must drop something

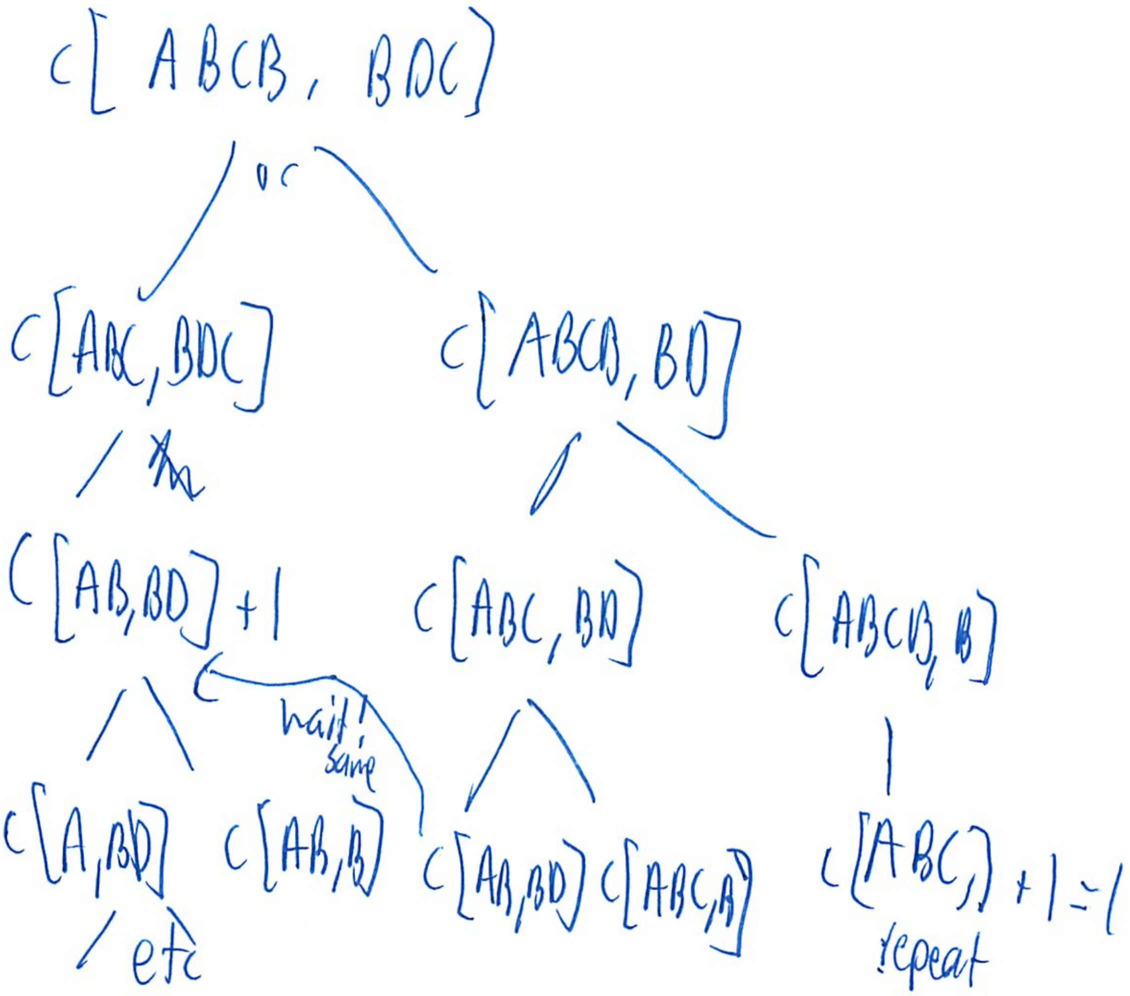


$c(i,j) = \max\{c(i,j-1), c(i-1,j)\}$ if $x_i \neq y_i$

A recurrence, summary

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

So

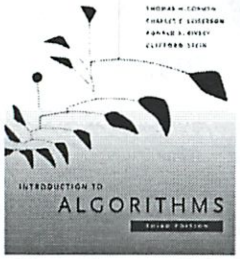


Don't recompute what did before

(memoize

Won't do more than $O(nm)$ times

6.006- *Introduction to Algorithms*



Lecture 19

Prof. Constantinos Daskalakis

Lecture overview

- review of last lecture
  - key aspects of Dynamic Programming (DP)
  - all-pairs shortest paths as a DP
- a smarter DP for all-pairs shortest paths
- longest common subsequence

CLRS 15.3, 15.4, 25.1, 25.2

Dynamic Programming Definition

- DP ≈ Recursion + Memoization
- Typically, but not always, applied to optimization problems – so far: Fibonacci, Crazy eights, SPPs
- DP works when:
  - the solution can be produced by combining solutions to subproblems; e.g. $F_n=F_{n-1}+F_{n-2}$
  - the solution to each subproblem can be produced by combining solutions to sub-subproblems, etc;

$F_{n-1}=F_{n-2}+F_{n-3}$ $F_{n-2}=F_{n-3}+F_{n-4}$

moreover it's efficient when….

- the total number of subproblems arising recursively is polynomial. $F_1, F_2, \dots, F_n$

Dynamic Programming Definition

- DP ≈ Recursion + Memoization
- Typically, but not always, applied to optimization problems – so far: Fibonacci, Crazy eights, SPPs
- DP works when:

Optimal substructure
The solution to a problem can be obtained by solutions to subproblems. $F_n=F_{n-1}+F_{n-2}$

Overlapping Subproblems
A recursive solution contains a "small" number of distinct subproblems (repeated many times)
$F_1, F_2, \dots, F_n$

4/24

All-pairs shortest paths

- **Input:** Directed graph $G = (V, E)$, where $V = \{1, \ldots, n\}$, with edge-weight function $w : E \to \mathrm{R}$.
- **Output**: An $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

Assumption: No negative-weight cycles

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that uses <u>at most</u> m edges
- Want: $d_{ij}^{(n-1)}$
- $d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

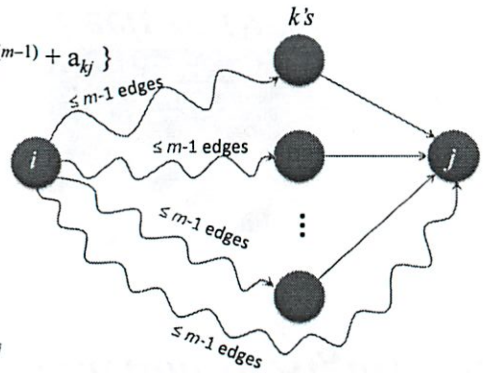
Claim: For $m = 1, 2, \ldots, n-1$,

$$\mathbf{d}_{ij}^{(m)} = \min_k\{\mathbf{d}_{ik}^{(m-1)} + \mathbf{a}_{kj}\}.$$

Proof of Claim

$$d_{ij}^{(m)} = \min_k\{d_{ik}^{(m-1)} + a_{kj}\}$$

```
for $k \leftarrow 1$ to $n$
    if $d_{ij} > d_{ik} + a_{kj}$
        $d_{ij} \leftarrow d_{ik} + a_{kj}$
```



"Relaxation" (recall Bellman-Ford lecture)

Dynamic Programming Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)} =$ weight of a shortest path from i to j that only uses <u>at most</u> m edges
- Want: $d_{ij}^{(n-1)}$
- $d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

Claim: For $m = 1, 2, \ldots, n-1$,

$$\mathbf{d}_{ij}^{(m)} = \min_k\{\mathbf{d}_{ik}^{(m-1)} + \mathbf{a}_{kj}\}.$$

$O(n^4)$ - similar to n runs of Bellman-Ford

Another DP Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)}$ = weight ~~of a shortest path from~~ i to j that only uses at most ~~m edges~~
- Want: $d_{ij}^{(n-1)}$
- $d_{ij}^{(0)} = 0$, if $i = j$, and $+\infty$, if $i \neq j$;

Claim: For $m = 1, 2, \ldots, n-1$,

$$\mathbf{d}_{ij}^{(m)} = \min_k \{\mathbf{d}_{ik}^{(m-1)} + \mathbf{a}_{kj}\}.$$

O(n^4) - similar to n runs of Bellman-Ford

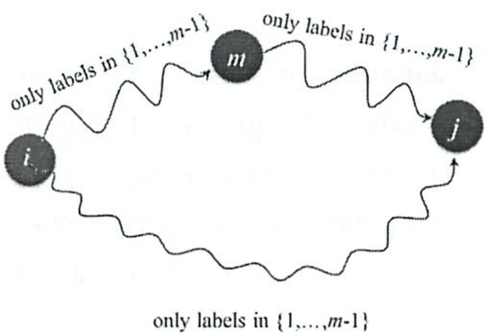
Another DP Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)}$ = weight of a shortest path from i to j that only uses intermediate vertices from set $\{1,\ldots,m\}$
- Want: $d_{ij}^{(n)}$
- $d_{ij}^{(0)} = a_{ij}$;

Claim: For $m = 1, 2, \ldots, n$,

$$\mathbf{d}_{ij}^{(m)} = \min\{\mathbf{d}_{ij}^{(m-1)}, \mathbf{d}_{im}^{(m-1)} + \mathbf{d}_{mj}^{(m-1)}\}.$$

Proof of Claim



$$\mathbf{d}_{ij}^{(m)} = \min\{\mathbf{d}_{ij}^{(m-1)}, \mathbf{d}_{im}^{(m-1)} + \mathbf{d}_{mj}^{(m-1)}\}$$

Another DP Approach

- Consider the $n \times n$ matrix $A = (a_{ij})$, where:
  - $a_{ij}=w(i,j)$, if $(i,j) \in E$, 0, if i=j, and $+\infty$, otherwise.
- and define:
  - $d_{ij}^{(m)}$ = weight of a shortest path from i to j that only uses intermediate vertices from set $\{1,\ldots,m\}$
- Want: $d_{ij}^{(n)}$
- $d_{ij}^{(0)} = a_{ij}$;

Claim: For $m = 1, 2, \ldots, n$,

$$\mathbf{d}_{ij}^{(m)} = \min\{\mathbf{d}_{ij}^{(m-1)}, \mathbf{d}_{im}^{(m-1)} + \mathbf{d}_{mj}^{(m-1)}\}.$$

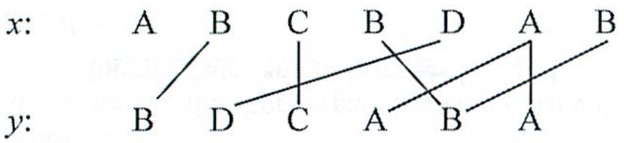
O(n^3) running time (Floyd-Warshall)

Lecture overview

- review of last time
  - key aspects of Dynamic Programming (DP)
  - all-pairs shortest paths as a DP
- another DP for all-pairs shortest paths
- **longest common subsequence**

Longest Common Subsequence

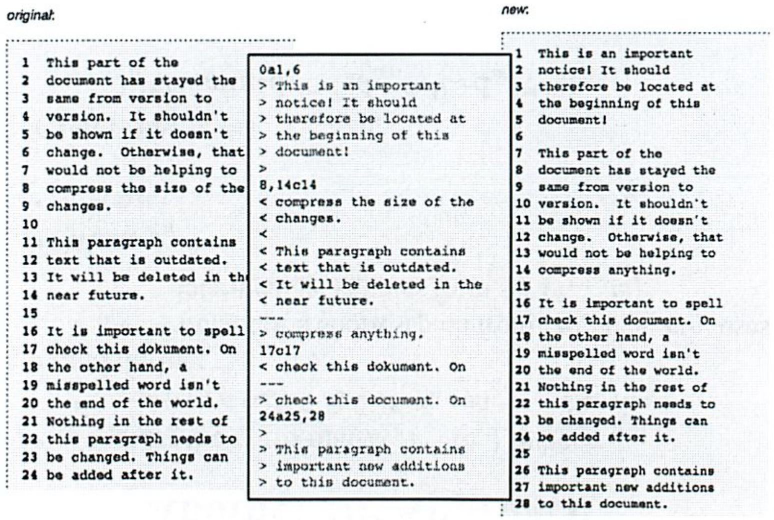
- given two sequences x[1..*m*] and y[1..*n*], find a longest subsequence LCS(x,y) common to both:



- denote the length of a sequence *s* by |*s*|
- let us first try to get |LCS(x,y)|

Applications of LCS

- Tons in bioinformatics, e.g. long preserved regions in genomes
- file comparison, e.g. **diff**



original:

```
1  This part of the
2  document has stayed the
3  same from version to
4  version.  It shouldn't
5  be shown if it doesn't
6  change.  Otherwise, that
7  would not be helping to
8  compress the size of the
9  changes.
10
11 This paragraph contains
12 text that is outdated.
13 It will be deleted in th
14 near future.
15
16 It is important to spell
17 check this dokument. On
18 the other hand, a
19 misspelled word isn't
20 the end of the world.
21 Nothing in the rest of
22 this paragraph needs to
23 be changed. Things can
24 be added after it.
```

```
0a1,6
> This is an important
> notice! It should
> therefore be located at
> the beginning of this
> document!
>
8,14c14
< compress the size of the
< changes.
<
< This paragraph contains
< text that is outdated.
< It will be deleted in the
< near future.
---
> compress anything.
17c17
< check this dokument. On
---
> check this document. On
24a25,28
>
> This paragraph contains
> important new additions
> to this document.
```

new:

```
1  This is an important
2  notice! It should
3  therefore be located at
4  the beginning of this
5  document!
6
7  This part of the
8  document has stayed the
9  same from version to
10 version.  It shouldn't
11 be shown if it doesn't
12 change.  Otherwise, that
13 would not be helping to
14 compress anything.
15
16 It is important to spell
17 check this document. On
18 the other hand, a
19 misspelled word isn't
20 the end of the world.
21 Nothing in the rest of
22 this paragraph needs to
23 be changed. Things can
24 be added after it.
25
26 This paragraph contains
27 important new additions
28 to this document.
```

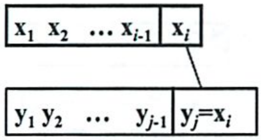
Brute force solution

- Given x[1..*m*] and y[1..*n*], how do we get the |LCS(x,y)| ?
- For every subsequence of x[1..*m*] , check if it is a subsequence of y[1..*n*]
- Analysis:
  - 2^m subsequences of *x*
  - each check takes *O(n)* time ...
  - (two finger algorithm)
  - worst-case running time is $O(n2^m)$

Using prefixes

- consider prefixes of x and y
  - x[1..*i*] ith prefix of x[1..*m*]
  - y[1..*j*] jth prefix of y[1..*n*]
- subproblem: define c[*i,j*] = |LCS(x[1..*i*],y[1..*j*])|
- so c[*m,n*] = |LCS(x,y)|
- recurrence?

1) x[1..*i*] and y[1..*j*] end with $x_i=y_j$



x_1 x_2 ... x_{i-1} | x_i

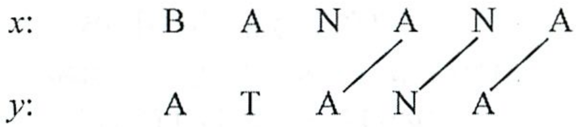
y_1 y_2 ... y_{j-1} | $y_j=x_i$

I might as well match x_i and y_j and look for LCS of x[1..*i*-1] and y[1..*j*-1].
So

$$c(i,j) = c(i-1,j-1)+1, \text{ if } x_i=y_j$$

where recall c[*i,j*] = |LCS(x[1..*i*],y[1..*j*])|

Example - use of property 1

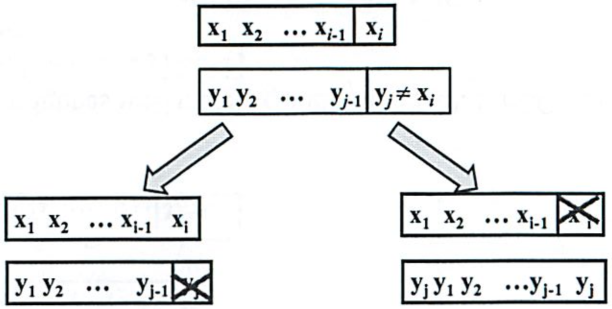


x: B A N A N A

y: A T A N A

by inspection LCS of B A N and A T is A
so |LCS(x,y)| is 4

2) x[1..*i*] and y[1..*j*] end with $x_i \neq y_j$



LCS of x[1..*i*] and y[1..*j*-1]

LCS of x[1..*i*-1] and y[1..*j*]

$c(i,j)=\max\{c(i,j-1), c(i-1,j)\}$, if $x_i \neq y_j$

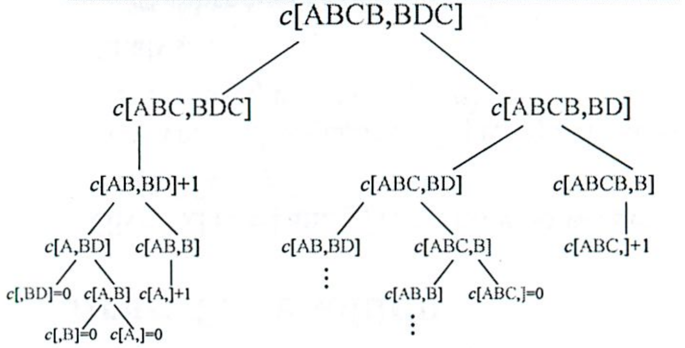
A recurrence, summary

- consider prefixes of x and y
  - x[1..*i*] ith prefix of x[1..*m*]
  - y[1..*j*] jth prefix of y[1..*n*]
- define c[*i,j*] = |LCS(x[1..*i*],y[1..*j*])|
  - so c[*m,n*] = |LCS(x,y)|
- recurrence:

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$

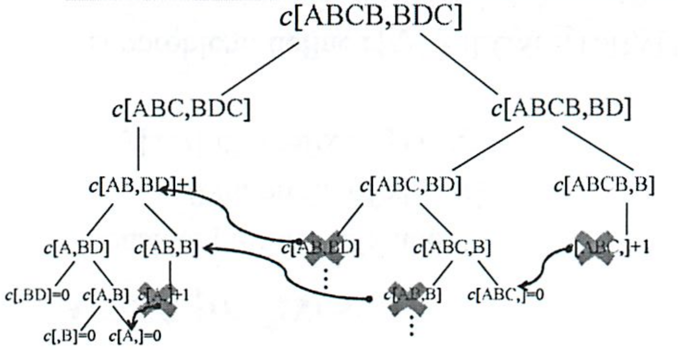
Solving LCS with Recursion

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$



Solving LCS with Recursion+Memoization

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$



Solving LCS with Recursion+Memoization

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$

```
memo = { }
c(i, j):
    if (i, j) in memo: return memo[i, j]
    else if i=0 OR j=0: return 0
    else if $x_i = y_j$: f = c(i-1, j-1)+1
    else f = max{c(i, j-1), c(i-1, j)}
    memo[i, j]=f
    return f
return c(n,m)
```

Solving LCS with Recursion+Memoization

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise} \end{cases}$$

- and the running time is $O(n \times m)$

6.006 Recitation

4/25

(2 min late)

Quiz grading

30% blank
30 + 10% correct more

} on marked analytical problems

Percentile where you are is what matters

Counting sort : not comparison
Radix sort
Only sort small int
Potentially faster - but destroys cache

	CS	RS	
Deterministic	✓	✓	Only Quicksort not - Since picks random int
Stable	✓ Does not have to be	✓	
In-place	X	X	

↑ open problem to have all 3

②

CS/ Sort n integers in $\{0, 1, \ldots, k\}$ in $O(n+k)$ time

- Create an array of size k	$O(k)$
- Move elements into their appropriate bucket (using chaining)	$O(n)$
- Pull everything out of bucket in order	↳ $O(n+k)$

Radix Sort

921
883
623
841

- Write #s in base b
- CS them on least sig digit
  ok thats it (should have seen)
- Then 2nd least, etc

range $\{0, 1, \ldots, m\}$

$\log_b m$

③

Inital A	1st	2nd	3rd
921	921	921	623
883	841	623	841
623	883	841	883
841	623	883	921

Runtime

b = # of slots

So CS takes $O(n+b)$

$d = \log_b m$

for range $\{0, \cdots, m\}$

$O((n+b)\log_b m)$

$O(n \log_n m)$

④

List your possibilites

BFS] unweighted — finds shortest path (paths of a certain lenght) — ~~B~~BFS kinda finds cycles - undirected esp

DFS] — find cycles, topo sort

Dijstra - ~~neg~~ ~~[illegible]~~ - non neg

Bellman - Ford] weighted - w/ neg

(incomplete list)

Topo sort → Bellman Ford on DAG $O(V+E)$

Scheduling

shortest paths

↙

paths in a DAG

~~#~~na DP

~~S~~ubproblem → # of paths into a vertex

memoize that

Must look at which edges can be used in a shortest path

Compute the distance $\delta(s,t)$ to every vertex t

(u,v) is on shortest paths

$$d(s,u) + 1 = d(s,v)$$

check if edge in

delete other edges

find # paths in a graph

Often subproblems form a DAG

(study more)

Huristic Search

Bidirectional Dijstra

Best on graphs that expand exponentially

? Actually less used than you might think in practice

ie graphs that look like a tree

Want to alternate entire levels - not vertices

⑥

A*

Modification to Dijstra

$w'(u,v) = w(u,v) - h(u) + h(v)$

$d(v) = \min(d(v), d(u) + w(u,v) - h(u) + h(v))$

Works best in geometric graphs

$h(v)$ = distance from v to t

e huristic ; crows fly distance

2 diff huristics - work in very diff scenarios

DP

* Always write subproblem first *

- In English (ie no math)
- Looks like [illegible] problem, but more general
  - ↳ ie subseq that ends at that char
  - answer to main problem is answer to 1 of them
- Should be useful to solve actual problem
- Count them

⑦

Crazy 8s

$dp[i]$ = the lenght of the longest crazy 8s subseq ending at $A[i]$ $O(n)$

- *not* a recurrence

Longest increasing subseq

$dp[i]$ = longest ↑ sub seq ending at $A[i]$ $O(n)$

Floyd-Warshal

$dp[i,j,k]$ = lenght of shotest path from i to j using vertices $\{1, 2, \dots, k\}$ $O(n^3)$

Longest Common Subseq

$dp[i,j]$ = longest common sub seq of $A[:i]$ and $B[:j]$ $O(n^2)$

⑧

Remember we are looking for $< i$, left than i

└ why we built the BST

6.006 Lecture

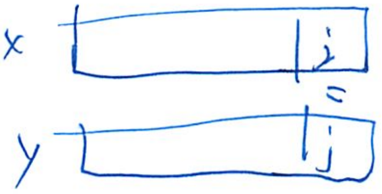
4/26

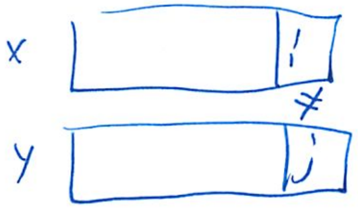
More DP

Remember Longest Common Subsequence

Subproblem $c(i,j)$ longest common subseq $i \to j$

Produce recursion





$c(i,j) = c(i-1,j-1)+1$ So $i = i-1$ or $j = j-1$

take max

Memoization

$O(nm)$

only exceed this $n \cdot m$ times

②

or <u>bottom up</u> - solve subproblems in order so don't have to recurse

What order?

eg ABCB
BDC

		j=0	=1	2	3
	-		B	D	C
i=0		0	0	0	0
1	A	0			
2	B	0			
3	C	0			
4	B	0			

How do dependencies work?

Solve $c[1,1]$

$L = \max\left(c[0,1], c[1,0]\right)$

From $c[0,1] \rightarrow 0$

(remember the parent!

	j=1	2	3
i=1	0↑	0↑	0↑

③

$c(2,1) = C(1,1) + 1$

	j=1	2	3
i=2	1 ↖	1 ←	1 ←
3	1 ↑	1 ↑	2 ↖
4	1 ↑	1 ↑	2 ↑

(don't get)

Bottom up - no sols of subproblem have not seen

L complete in very specific order

No memozation

Indices store which was binding

So $c(3,3) =$

- have everything to the left and above

Look at recurrence

Does last char match? Yes C=C

$c[i-1, j-1] + 1$

$c[2,2] + 1$

$1 + 1 = 2$

arrow to show is diagonal (since they were the same)

④

So why parent pointers?

So can reconstruct longest common subseq

$c[i-1, j-1]$ or $c[i-1, j]$ or $c[i, j-1]$

Starting at $c[m, n]$ look at parent pointer $p(m,n)$

if diagional $x[m] = y[n]$

So trace it back

Print the diagonal (red arrows)

What case is binding is related to the structure of the problem

DP DAG

Generalize subproblem so don't recurse

dependencies implies (missed)

edge $x \to y$ if x depends on y

need order where x follows y if $x \to y$

- Topo sort!
- Only if DAG
- otherwise cyclic problem dependency

Knapsack Problem

Get as much as possible in the bag

bag size S

n items of size s_i and value v_i

Goal: choose subset w/ $\sum_i s_i < S$

maximize $\sum_i v_i$

? Max. value while fitting in

⑥

Nieve: try all possible subsets 2^n

~~greedy?~~

Greedy idea?

maximize value

No - won't produce optimal

or optimize $\frac{\text{value}}{\text{volume}}$ - pick greedily

(very clever - and so obvious!)

Actually this is NP-hard

- no polynomial time algo
- many related problems

But if sizes are an int in a small range

Can do

Subproblem

$Val[i]$ = Best value obtained if only items $[i:n]$ were avilable to choose from

- So decide include item i or not

= best of $val[i+1]$ or ~~val~~ $v_i + Val[i+1]$

↑ but not enough info — Should be different!, so not well defined recurrency

Obvious subproblem is <u>not</u> Correct

Often need to solve bigger/more general problem (which on)

└ more complicated

initial problem is a special case

Subproblem 2

$Val[i, X]$ = max value ~~an~~ if one can choose $[i:n]$ and avilable size is X

↑ ranges 1-5

⑧

if $s_i > X$ then can't include i, so $val[i, x]$
$= Val[i+1, x]$

Otherwise
$val[i, X] = max(Val[i+1, x], v_i + Val[i+1, X-s_i])$

(Working w/ suffixes)

$Opt = Val[1, S]$

Is a DAG?

Yes, each subproblem depends on bigger i and smaller x
Compute by ↓ i and ↑ X

# of subproblems

$n \cdot S$ subproblems
$O(1)$ for each
$= O(n \cdot S)$ = polynomial in n, S?

↓

⑨

It takes $\log_2 S$ bits to discribe S

So <u>not</u> polynomial

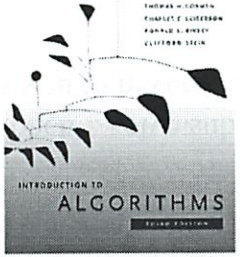
(very confused)

Is pseudo-polynomial time

Dijstra is really polynomial in description of #s

This is not polynomial in description of #s

6.006- *Introduction to Algorithms*



Lecture 20

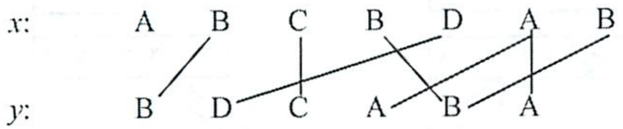
Prof. Constantinos Daskalakis

Lecture overview

- longest common subsequence:
  - the bottom-up approach
  - reconstructing the LCS: back-pointers
- knapsack

Longest Common Subsequence

- given two sequences x[1..*m*] and y[1..*n*], find a longest subsequence LCS(x,y) common to both:

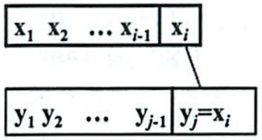


x: A B C B D A B

y: B D C A B A

- find the length of a longest common subsequence
- DP subproblem:
- c(*i*, *j*)= length of longest common subsequence between strings x[1..*i*] and y[1..*j*]

1) x[1..*i*] and y[1..*j*] end with $x_i = y_j$



$x_1\ x_2\ \ldots\ x_{i-1}$ | x_i

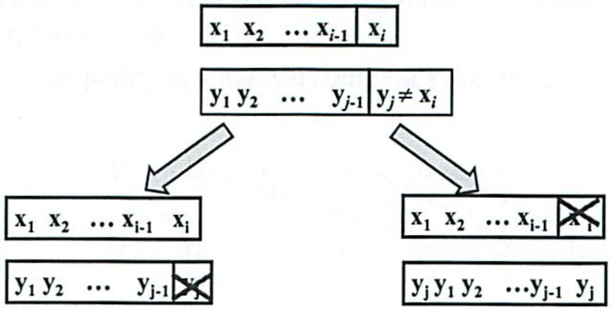
$y_1\ y_2\ \ldots\ y_{j-1}$ | $y_j = x_i$

I might as well match x_i and y_j and look for LCS of x[1..*i*-1] and y[1..*j*-1].
So

$$c(i, j) = c(i-1, j-1)+1, \text{ if } x_i = y_j$$

4/26

2) x[1..*i*] and y[1..*j*] end with $x_i \neq y_j$



LCS of x[1..*i*] and y[1..*j*-1] **LCS of x[1..*i*-1] and y[1..*j*]**

$c(i, j)=\max\{c(i,j-1), c(i-1,j)\}$, if $x_i \neq y_j$

Solving LCS with Recursion+Memoization

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise}\end{cases}$$

```
memo = { }
c(i, j):
    if (i, j) in memo: return memo[i, j]
    else if i=0 OR  j=0: return 0
    else if x_i=y_j: f = c(i-1, j-1)+1
    else f = max{c(i, j-1), c(i-1, j)}
    memo[i, j]=f
    return f
return c(n,m)
```

The Bottom-Up Approach

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise}\end{cases}$$

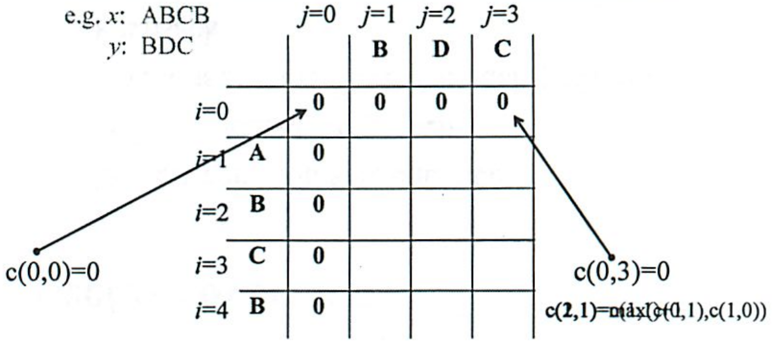
"bottom-up approach": solve sub-problems in an order that allows you to never recurse

---

The Bottom-Up Approach

$$c[i,j]=\begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j],c[i,j-1]\} & \text{otherwise}\end{cases}$$

e.g. *x*: ABCB
y: BDC



		j=0	*j*=1	*j*=2	*j*=3
			B	**D**	**C**
i=0		**0**	**0**	**0**	**0**
i=1	**A**	**0**			
i=2	**B**	**0**			
i=3	**C**	**0**			
i=4	**B**	**0**			

c(0,0)=0

c(0,3)=0

c(2,1)=max{c(1,1),c(1,0))

The Bottom-Up Approach

```
Length of Longest Common Subsequence(x,y)
$m \leftarrow length[x]$
$n \leftarrow length[y]$
for $i \leftarrow 1$ to $m$
    do $c[i, 0] \leftarrow 0$
for $j \leftarrow 0$ to $n$
    do $c[0, j] \leftarrow 0$
for $i \leftarrow 1$ to $m$
    do for $j \leftarrow 1$ to $n$
        do if $x_i = y_j$
            then $c[i, j] \leftarrow c[i-1, j-1] + 1$
                $p[i, j] \leftarrow$ "↖"
            else if $c[i-1, j] \geq c[i, j-1]$
                then $c[i, j] \leftarrow c[i-1, j]$
                    $p[i, j] \leftarrow$ "↑"
                else $c[i, j] \leftarrow c[i, j-1]$
                    $p[i, j] \leftarrow$ "←"
return $c$ and $p$
```

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases}$$

parent pointers

Why parent pointers?

- **Goal:** finding a longest common subsequence
  - p remembers if the c[*i*, *j*] computation used c[*i*-1, *j*-1], c[*i*, *j*-1], or c[*i*-1, *j*]
- Here is how they are used:
- Starting at c[*m*, *n*], look at parent pointer p[*m*,*n*]
  - if it points to (m-1,n-1), then x[m]=y[n] is part of *opt*
    - put x[m] at end of output string, jump to square (m-1,n-1) and continue building *opt* from there
  - else, build *opt* from squares (m-1,n) or (m,n-1) depending on where p[*m*,*n*] points
  - repeat

Constructing an LCS

```
PRINT-LCS ($p, x, i, j$)
if $i = 0$ or $j = 0$
    then return
if $p[i, j]$ = "↖"
    then PRINT-LCS($p, x, i-1, j-1$)
        print $x_i$
    elseif $p[i, j]$ = "↑"
        then PRINT-LCS($p, x, i-1, j$)
else PRINT-LCS($p, x, i, j-1$)
```

initial call is PRINT-LCS (*p*, *x*, *m*, *n*)
running time: $O(m+n)$

Example

x: A B C B
y: B D C

	y_j	B	D	C
x_i	0	0	0	0
A	0	↑0	↑0	↑0
B	0	↖1	1 ←	1 ←
C	0	↑1	↑1	↖2
B	0	↖1	↑1	↑2

Lecture overview

- longest common subsequence:
  - the bottom-up approach
  - reconstructing the LCS: back-pointers
- **the DP DAG**
- knapsack

Generalization: Bottom-Up DP

- we've seen DP recurrences
  - which suggest recursive implementation
  - …with memoization to avoid re-computing intermediate results
- we've also seen "bottom up" implementations
  - order sub-problems in a way that allows answering bigger sub-problems using already computed solutions to smaller sub-problems
- how to get a good ordering?

The DP DAG

- define a graph representing DP
  - sub-problems are vertices
  - edge x → y if problem x depends on problem y
- what order of problem solving works?
  - need order where x follows y if x → y
  - Topological Sort!
  - can do so if graph is a DAG
  - what if not?
    - cyclic problem dependency
    - can't use DP

Lecture overview

- longest common subsequence:
  - the bottom-up approach
  - reconstructing the LCS: back-pointers
- the DP DAG
- **knapsack**

Knapsack Problem

- Knapsack (or cart) of size S
- Collection of n items; item i has size s_i and value v_i
- Goal: choose subset with:
  - □ $\Sigma_i s_i < S$ (feasible, i.e. fits in knapsack)
  - maximize $\Sigma_i v_i$
- Ideas?
  - try all possible subsets: 2^n
  - greedy?
    - choose items maximizing value ?
    - choose items maximizing value/size
      - great, but what if items don't exactly fit (non-divisible items)?

Some bad and better news

- For arbitrary sizes, Knapsack is hard (NP-hard)
  - no polynomial time algorithm in 40 years of trying
  - it's exactly as hard as several thousand other important problems
  - and we haven't been able to find polynomial time algorithms for them for 40 years of trying either
  - most people think there is none
- Better news:
  - There is a DP algorithm if sizes are integers from a small range

---

First attempt for DP Algorithm

- subproblem?
  - Val[i] = Best value obtained if only items[i:n] were available to choose from
- recurrence?
  - Val[i] = best of Val[i+1] or v_i + Val[i+1],
- not a well-defined recurrence: doesn't have enough info

namely, in a correct recursion these should be different values

---

Second Attempt

- Solve a more complicated problem
  - of which the initial problem is a special case
- Val[i, X] = max value if one can choose from items[i : n] and available size is X
- Recurrence for Val[i, X]:
  - if $s_i > X$, then can't include i, so Val[i, X] = Val[i+1, X]
  - otherwise:
    - Val[i, X] = max(Val[i + 1, X], v_i + Val[i + 1, $X - s_i$])
- OPT = Val[1, S]

Analysis

$$\text{Val}[i, X] = \begin{cases} \text{Val}[i+1, X], & \text{if } s_i > X \\ \\ \max(\text{Val}[i+1, X], v_i + \text{Val}[i+1, X - s_i]) \end{cases}$$

- Is the recurrence a DAG?
  - yes, each problem depends on bigger i and smaller X
  - compute by decreasing i and increasing X
- Runtime?
  - O(n S) subproblems and work per subproblem is O(1)
  - So total time: O(n S)
- Is this polynomial?
- Looks polynomial but it isn't: to describe S need $\log_2 S$ bits
- "Pseudo-polynomial time": exponential dependence on numerical inputs, but polynomial dependence on everything else

6006 Recitation

4/27

Canceled

Problem Set 6

This problem set is due **Wednesday, May 2** at **11:59PM.**

Solutions should be turned in through the course website. **You must enter your solutions by modifying the solution template (in Python) which is also available on the course website.** The grading for this problem set will be largely automated, so it is important that you follow the specific directions for answering each question.

For multiple-choice and true/false questions, no explanations are necessary: your grade will be based only on the correctness of your answer. For all other non-programming questions, full credit will be given only to correct solutions which are described clearly and concisely.

Programming questions will be graded on a collection of test cases. Your grade will be based on the number of test cases for which your algorithm outputs a correct answer within time and space bounds which we will impose for the case. Please do not attempt to trick the grading software or otherwise circumvent the assigned task.

---

1. **Dynamic programming analysis (20 points)**

For each of the following recursions, all of which take exponential time to compute naively, answer the following questions. You may assume that the functions can be computed in constant time when any of the arguments are 0.

i. In terms of n, how many distinct subproblems are ever solved to evaluate the function with arguments bounded by n?

ii. If we use memoization to speed up the computation of the recurrence, what is time needed to evaluate the function?

(a) A function defined by the Fibonacci recursion:

$$\text{FIB}(n) = \text{FIB}(n-1) + \text{FIB}(n-2)$$

(b) A function defined by Pascal's recursion:

$$\text{CHOOSE}(n,k) = \text{CHOOSE}(n-1,k) + \text{CHOOSE}(n-1,k-1)$$

where $\text{CHOOSE}(n,k) = 0$ if $k > n$.

(c) A function defined by the Bell numbers recursion:

$$\text{BELL}(n) = \sum_{k=0}^{n-1} \text{CHOOSE}(n,k) \cdot \text{BELL}(k)$$

where the binomial coefficients are already computed (using the recursion above).

(d) A function defined by the recursion:

$$\text{GAME}(n,k) = \max_{\frac{k}{2}\le i\le k} (-1)^n \cdot \text{GAME}(n-1,i)$$

where $\text{GAME}(n,k) = 0$ if $k > n$.

(e) A function defined by the recursion:

$$\text{HALF}(i,j) = \left(\sum_{k=0}^{\frac{j-i}{2}} \text{HALF}\left(i+k, i+k+\frac{j-i}{2}\right)\right)^2$$

where $0 \le i \le j \le n$. (Hint: think of $[i,j]$ as an interval. What do the recursive calls look like?)

Solution Format:

Your choices for this problem are:

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n)$
D. $\Theta(n \log n)$
E. $\Theta(n^2)$
F. $\Theta(n^2 \log n)$
G. $\Theta(n^3)$

So your solution to each part of this problem should be a single character in the set `set(['A', 'B', 'C', 'D', 'E', 'F', 'G'])`.

2. **A game of DAGs (30 points)**

   You are given a directed acyclic graph, in the same adjacency list format as the graph from Problem Set 4. Silvio and Costis are playing a game on this graph.

   The game begins at a node s in the graph. The two players alternate taking turns, with Silvio going first. On a player's turn, he chooses a vertex which is a direct descendent of the vertex chosen in the previous round (e.g., on Silvio's first turn, he chooses any vertex which s has an edge to). A player loses if he has no legal moves, which happens when the other player chooses a sink.

   Fill in the code for a function `find_winning_nodes(graph)` which returns a list of the start nodes s in the graph from which Silvio wins, assuming that both players play optimally.

```
graph = {0: [1, 2], 1 : [2, 3], 2 : [3], 3 : []}
# If s is 0, Silvio chooses 1 or 2, so Costis chooses 3 and wins.
# If s is 1 or 2, Silvio chooses 3 and wins.
# If s is 3, Silvio immediately loses.
set(find_winning_nodes(graph)) == set([1, 2])
```

~~even one win or loss~~

~~leftes is after any~~

~~who can I pick~~

~~— top down / bottom up~~
~~sub to top - not optimal~~

~~go big to small~~

3. **Optimal parenthesization (40 points)**

Given an array of n positive (but not necessarily integral) numbers, your goal is to determine the largest value that can be obtained by interspersing parentheses, multiplication signs, and addition signs between them.

Fill in the code for a function `find_largest_value(numbers)` for this problem. Your code should be able to handle a 100-element list in about a second and pass the following test cases:

```
# An optimal parenthesization: (1 + 2) * (3 * (4 * 5)) = 180
abs(find_largest_value([1, 2, 3, 4, 5]) - 180) < 0.001

# An optimal parenthesization: 0.8 + (0.5 + (0.3 + 0.5)) = 2.1
abs(find_largest_value([0.8, 0.5, 0.3, 0.5]) - 2.1) < 0.001

# An optimal parenthesization: (0.8 + 1.5) * (1.6 + 0.5) = 4.83
abs(find_largest_value([0.8, 1.5, 1.6, 0.5]) - 4.83) < 0.001
```

4. **MIT's football team (40 points)**

The Institute wants to develop a set of robots that can defeat Harvard's football team in a head-to-head comparison. Harvard's team is composed of n players, each of which have a strength a_i and a speed b_i.

A robot majorizes a human if it is at least as strong and at least as fast as the human. It costs MIT $a \cdot b$ thousand dollars to create a robot which has strength a and speed b, and MIT would like to have at least one robot that majorizes each player on Harvard's team. Describe an efficient algorithm to compute the minimal amount of money needed to create a set of robots that satisfies this condition.

Example: If Harvard's team has three players with strength-speed pairs $(10, 1)$, $(2, 9)$, and $(1, 10)$, the most cost-efficient team of robots is the team of two: $(10, 1)$ and $(2, 10)$. This team costs $30,000.

Solution Format:

Fill in the string `answer_for_problem_4` with your solution.

6.006 P-Set 6

4/30

w/ Crystal + Alanna

Last pset ?!?

1. DP analysis

a) Fib (n)

n subproblems

each takes $O(1)$ since just adding

$\Theta(n)$ overall

b) Pascal

$\text{Choose}(n, k) = \text{Choose}(n-1, k) + \text{Choose}(n-1, k-1)$

$n > k$

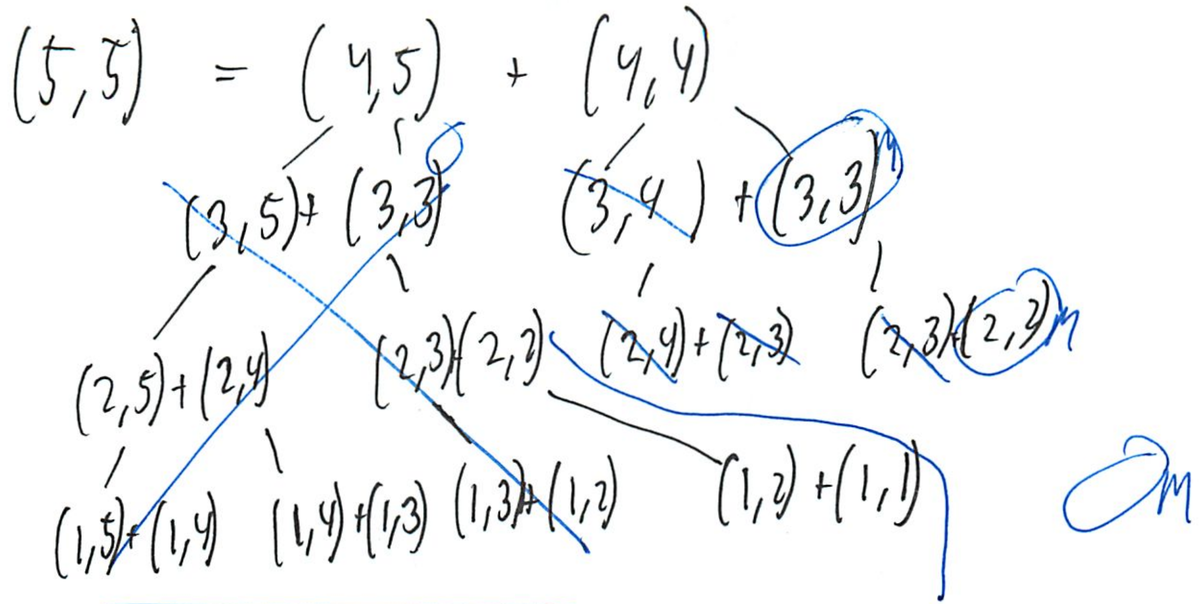
So $(5,5) = (4,5) + (4,4)$ 2

$(4,5) = (3,5) + (3,3)$ 2

$(4,4) = (3,4) + (3,3)$ 1

Having trouble picturing

②



but 0 if $k > n$

So if $k = n \rightarrow n$

but if (10, 5) then diff → bigger

n^2

but branching factor of 2 → 2^n

Did we do in 6.042

How does memozation help?

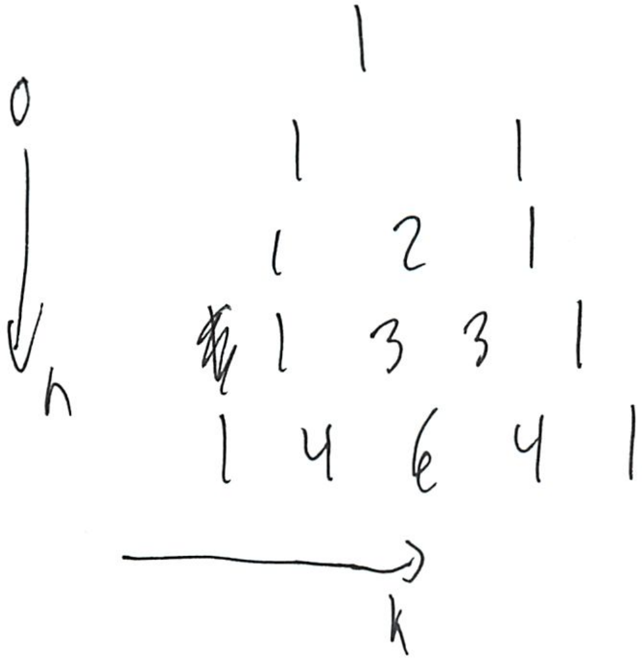
$\binom{n}{k}$ thing?

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

~~[illegible]~~ then solve?

What about memozation?

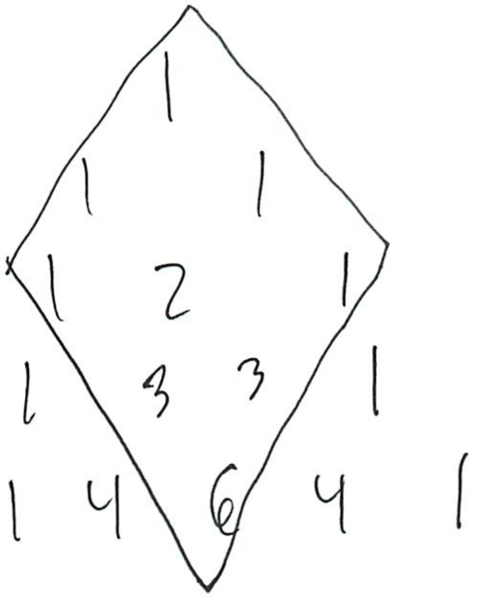
Pascalls triangle



But repeat?

④

So everything in the square



↑ all the things you need to calculate

Area → so n^2

Fib is a line → so n

ii Just adding → so n^2

⑤

c) Bell #

$$\mathrm{Bell}(n) = \sum_{k=0}^{n-1} \underbrace{\mathrm{Choose}(n,k)}_{\text{done already}} \cdot \mathrm{Bell}(k)$$

So n?

With each taking 1

So function in total O(n)

d) $\mathrm{Game}(n,k) = \max\limits_{\frac{k}{2} \leq i \leq k} (-1)^n \cdot \mathrm{Game}(n-1, i)$

$k > n \rightarrow = 0$

?notation?

?Range the i from $\frac{k}{2}$ to k

take the max

and multiply by n-1

k each subproblem?

Must be a multiple of n from set

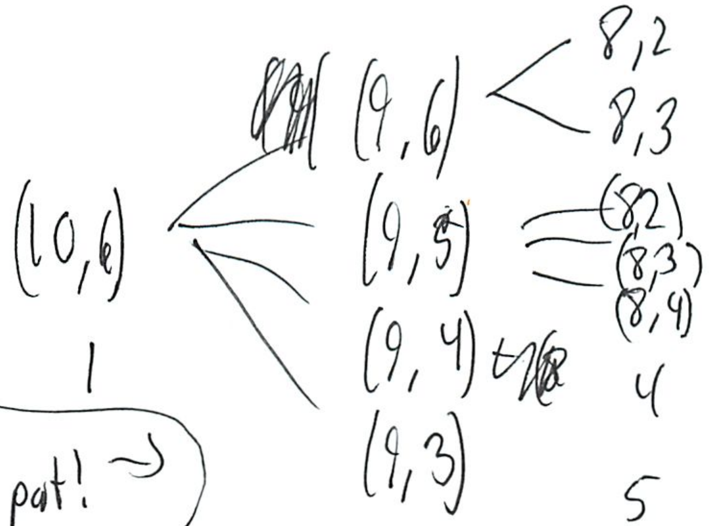
⑥

So game(5,4) is what

$$\max_{2 \le i \le 4} (-1)^n \cdot \text{Game}(4, i)$$

Game$(\overset{n}{10}, \overset{k}{6})$

$$\max_{3 \le i \le 6} (-1) \cdot \text{Game}(9, i)$$



follow the biggest pat! →

$\frac{k}{2}$

(7,1) to (7,6)
5

(6, 1-6)
5

↓

(5, 1-5)
(4, 1-4)
(3, 1-3)
(2, 1-2) (1,1)

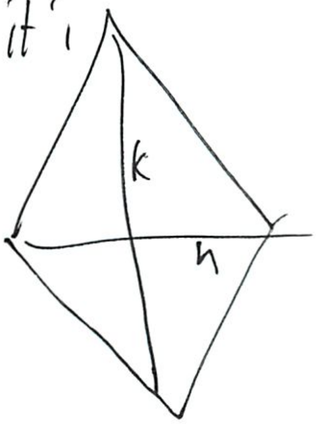
diamonds also

①

Remember Memoization!

10	9	8	7
1	$\frac{k}{2}$	$\frac{3k}{4}$	$\frac{7k}{8}$
1	$1-\left(\frac{k}{2}\right)n$		

Is it?



$$\#\text{ subproblems} = 1 + \sum_{i=1}^{n} k\left(1 - \frac{1}{2^i}\right)$$

each level goes all the way to k
but k/2 keeps halfing

6 → 3-6 → 2-6 → 1-6

⑧

$$k\left(\sum_{i=1}^{n} 1 - \frac{1}{2^{i}}\right)$$

$$k\left(\sum_{i=1}^{n} 1 - \sum_{i=1}^{n} \frac{1}{2^{i}}\right)$$

what is this recurrence in closed form?

$$kn - \sum_{i=1}^{n} \left(\frac{1}{2}\right)^{i}$$

$$kn - 2\left(1 - \frac{1}{2}^{n}\right)$$

(I should read the book on this)

Skip

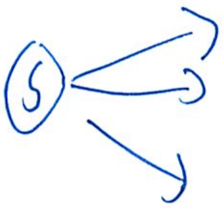
e) $\text{Half}(i,j) = \left(\sum_{k=0}^{\frac{j-1}{2}} \text{HALF}\left(i+k, i+k+\frac{j-i}{2}\right)\right)^{2}$

Skip

2. A game of DAGs

more graphs!

adj list



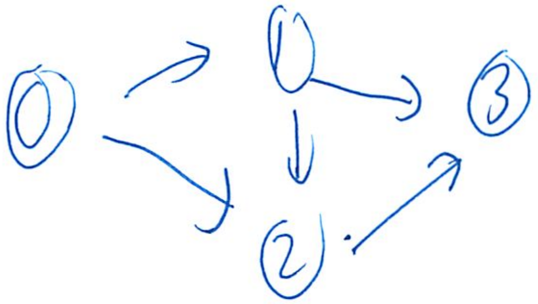
Lose = no legal move

ie chooses a sink

* Only 1 piece

So if I choose a sink → I win

Return a list of nodes from which Silvio wins



⑩

So is this an all pairs shortest path ?
↳ solved w/ DP before

Is a coding problem

We know where sink is
↳since []

Want to be an even # away

Reverse adj list → O(V + E) right?

BFS from that
everything even

But players can avoid
Can you find that working backwards?

And prob not the time to reverse the graph...

⑪

Paths b/w anything and a sink

Easy to get sinks → [ ]

Subproblem space ↓ from that

(skip)

---

3. Optimal parenthesization

n positive #s

Find largest # made by inserting parentheses

Similar to to exam problem I didn't really get...

Longest football

What does it mean to have another el on the list?

≤ 1 add

> 1 multiply

⑫ So if $c \leq 1$ $c > 1$

$(a+b) \rightarrow$ ~~[illegible]~~ $a+(b+c)$ $(a+b) \cdot c$

$(a \cdot b) \longrightarrow a \cdot (b+c)$ $(a \cdot b) \cdot c$

↑ So cases depending on the previous ↑ ~~prev~~ prev ·c

only previous weird case?

Do we store expression?

Or store what we just did

-plus previous #

.8 1.5 1.6 .8

left
.8

~~[illegible]~~

$(.8 + 1.5) \cdot 1.6$

left right

$((.8 + 1.5) \cdot 1.6)$

left

but can't just be # -since want $(.8+1.5) \cdot (1.6 \cdot .8)$

(13)

So need to also store prev right
prev op

$$\underbrace{\left[(0.8 + 1.5) \overset{\downarrow\text{prev op}}{\cdot} \overset{\swarrow\text{prev right}}{1.6}\right]}_{\text{left}} \qquad \underbrace{1.5}_{\text{new right}}$$

new ~~ri~~ right

So $\dfrac{\text{left}}{\text{prev right}} = \text{left}$

? to undo

prev right + new right → new right

Then $\text{left} \cdot \overset{\text{new}}{\text{right}} = \text{left}$

~~prev right~~

prevright = new right

? What is it when we go 1 more?

⑭

Optimal to add it → so add it in the future

prev op = 0

When does prev op change?
L when you do something else

new light = light
L what you are currently working w/

---

4. Football Robots

a_i b_i — Harvard

Majorize $a_H > a_R$ $b_H > b_R$

Costs a · b
Minimal amt of \$

⑮

Strategy

DP

∟ not ~~short~~ longest substring ?

~~or~~ well some minimization

1. Build identical robots
2. Try combining them

Or 1 player

∟ 1 robot

2 players

∟ 1 or ~~on~~ 2

try both, minimize the cost

When ever add player to H team

1. Check if existing robots majorize
2. Find the lowest cost improvement so it majorizes
   or add a matching robot

Sounds correct + speedy

Whats the trick?

What can you memoize?

robots or H players

If new H player majorized by another H player
→ skip

But list of robots is not worth using

So look at previous
— not strictly memoization

Could do in a loop

But memozation is an improvement on...

⑰

min (improving robots i , add new robot)
for all i
prev robots

$O(n^2)$

is that good enough?

my inkling is no
↳ scan list - memoized
to scan through prev robots

$1.3 + 1.3 > 1.3 \cdot 1.3$
So cases are wrong
So $2 \cdot 2 = 2 + 2$ is where
it matches
just try both

there got to be a better
way...

give up...

OH

5/1

On reverse

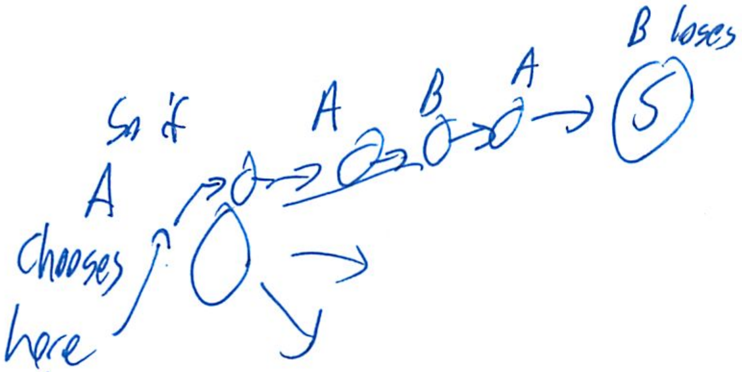
lead to sink

go no where else

odd distance

So odd distance from sink
exclusive

So if



Than B loses

Propagating the sink forward

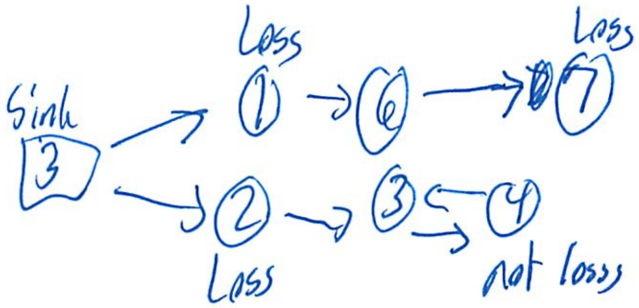
Working backwards is a bottom up DP

What is correct ans?

└ The ones you choose that will be correct

②

Backwards BFS

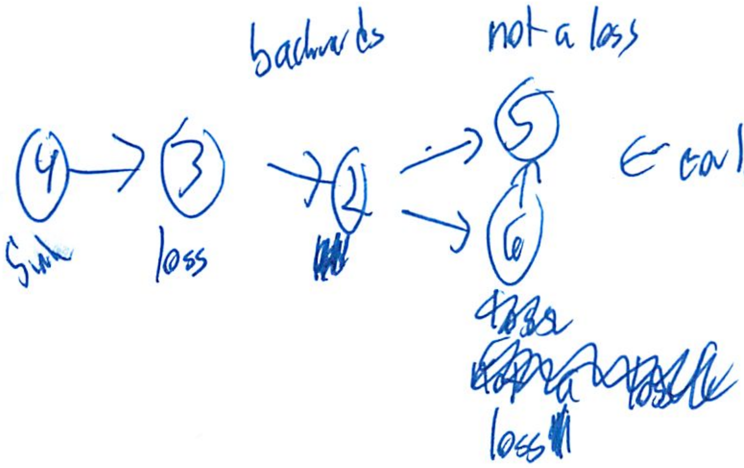


are there any other odd exclusive paths

But its a DAG!

What does it mean to that its a DAG

i can't avoid a sink



← could have

③

TA
GH

(he hates this problem)

Thinks of game/chess

~~So like chess~~

So like flip DAG ↓

Game is asy

Regardless of what you do you lose

[illegible]

Hyp: ~~eva~~ if you start - and play perfectly you win

? game is asy

Tricky to do odd/even since not exclusive

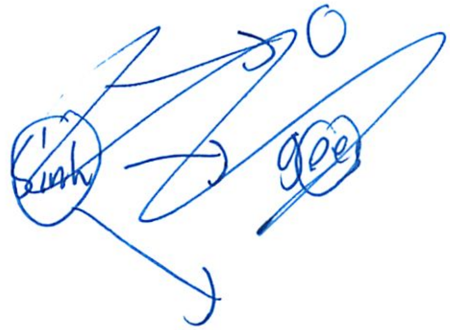
Start from end is reasonable time

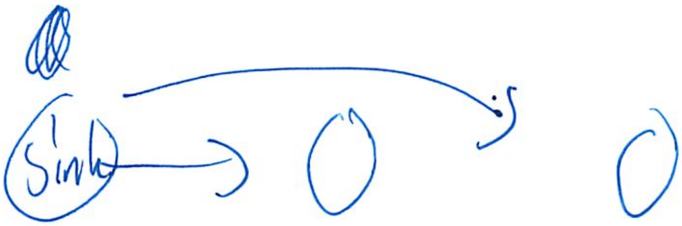
TA: ~~Focus on correct~~ Don't focus on run time

④

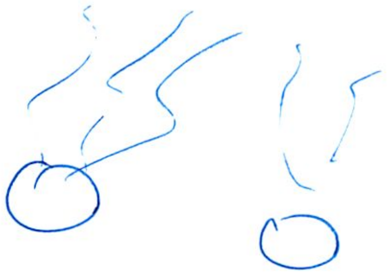
How to solve if not a sincy

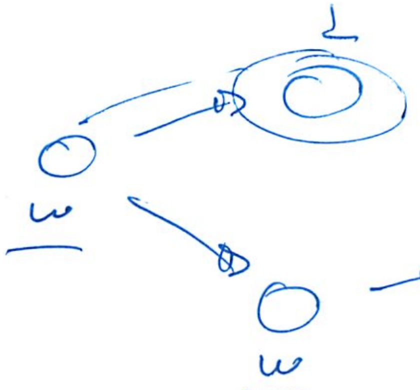
Topo sort





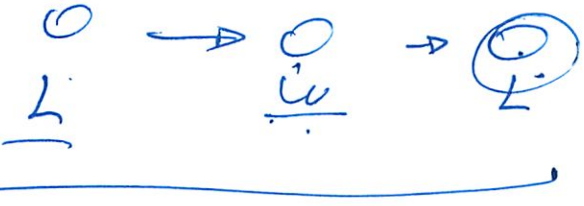
Instead pick a node and see where it goes

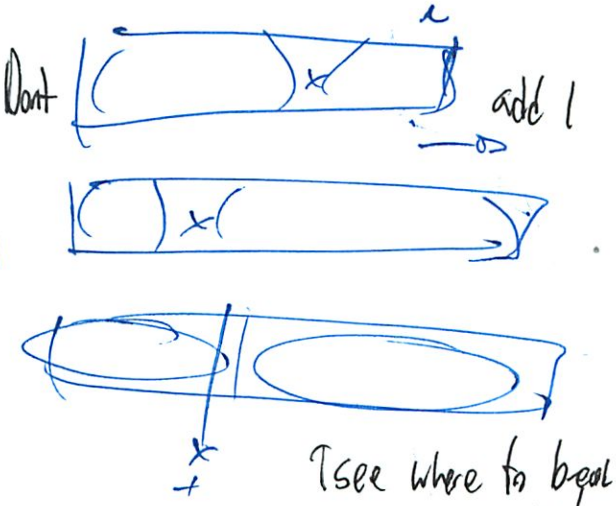




Don't work back from sinks

need to update multiple times

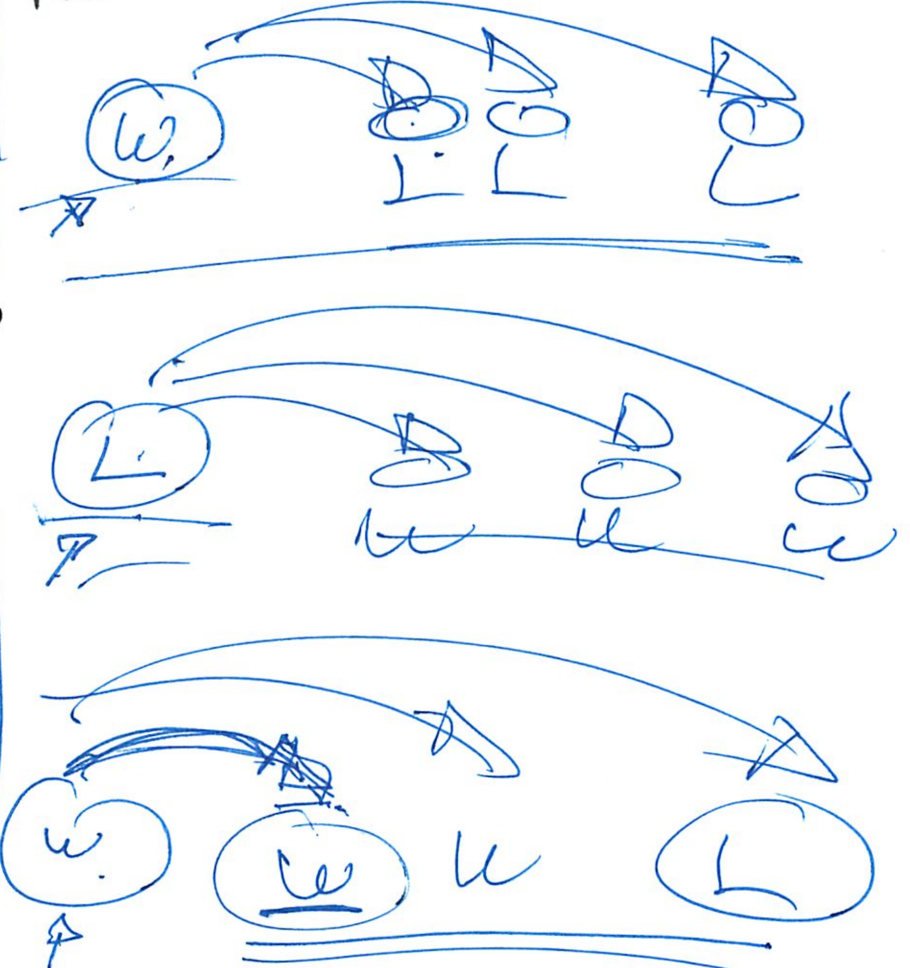




I see where to break up best

⑤

Patterns



See whee it points to

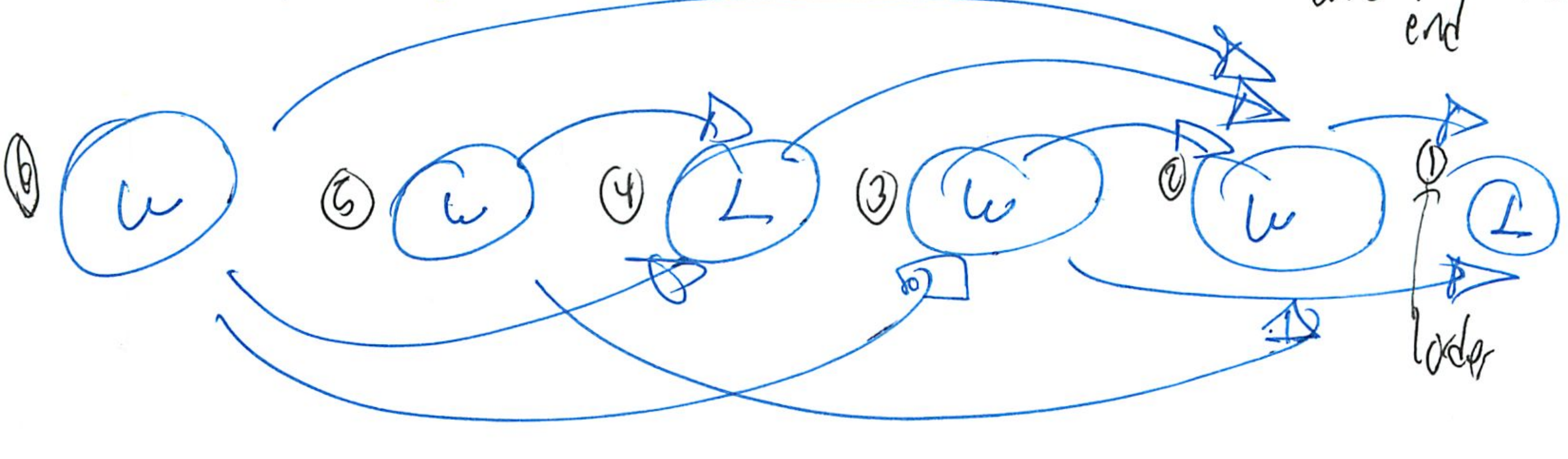
every one is a win
or loss
not neigher

Know whom is winning
or losing

~~[illegible]~~ ↳ where letter is
other guy (Silvio)

Whom can I pick?
if I'm at a node

Solve from the
end



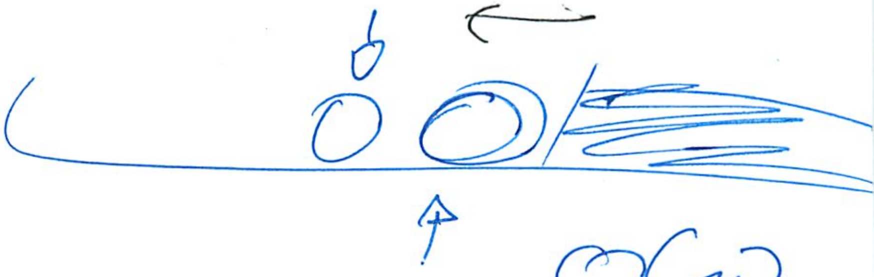
How can I know how to solve it

How to reduce big problem to small problem to solve

⑥

LW

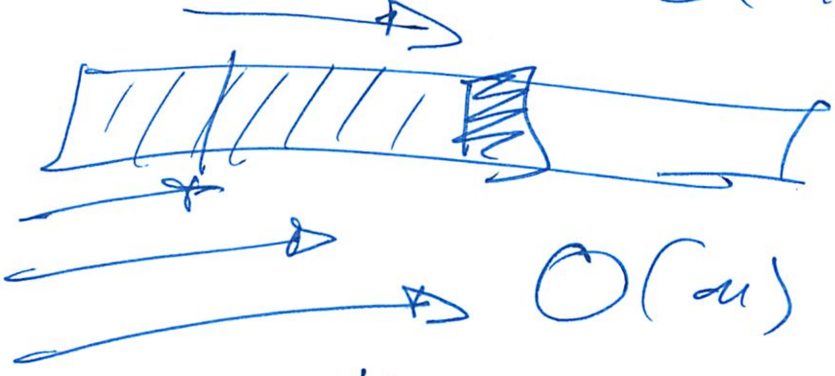
this one



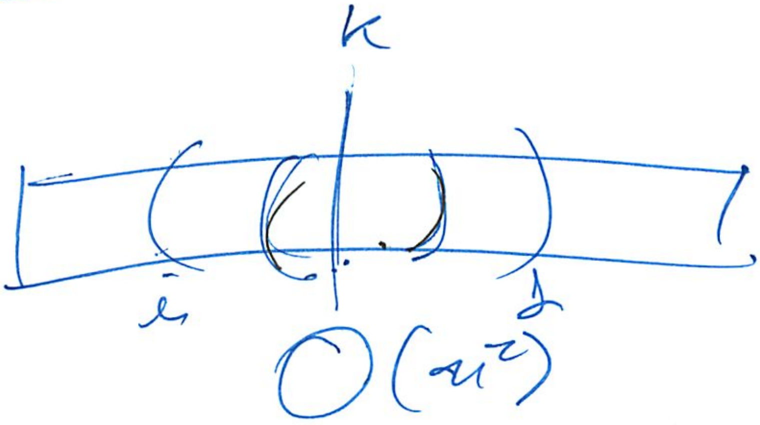
$O(n)$

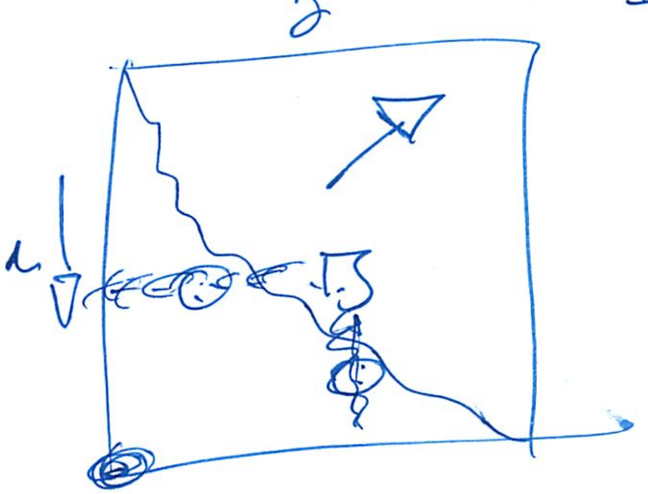
crazy 8

football

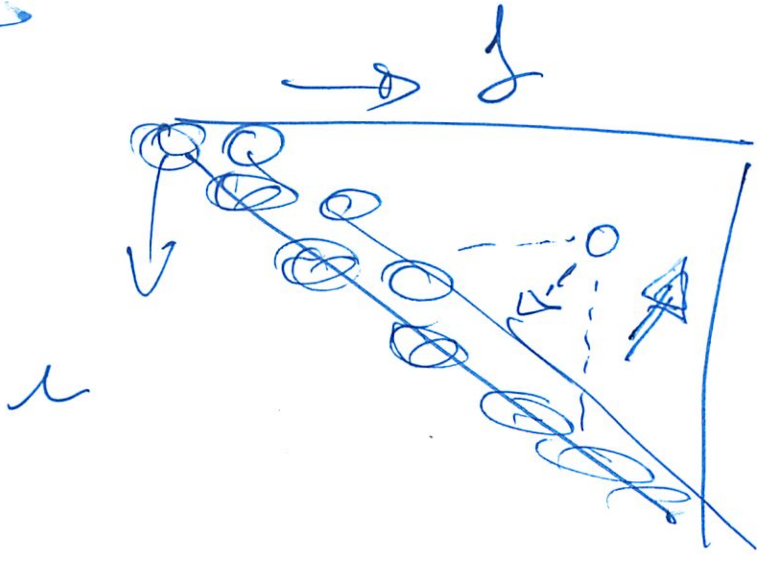


i, j



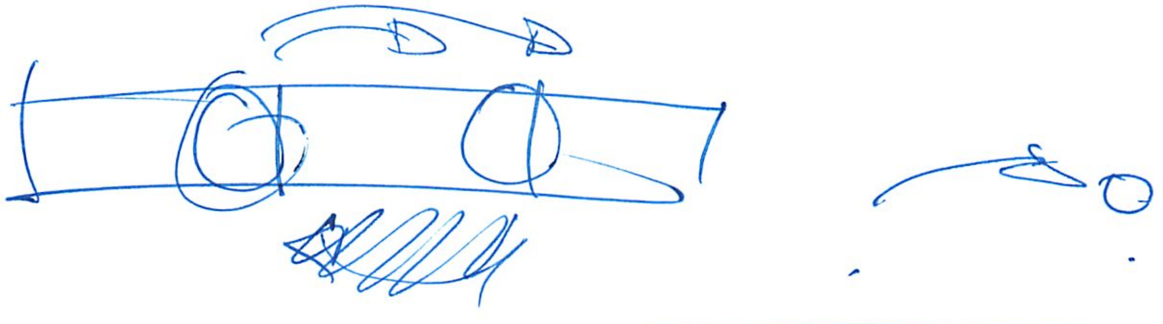


$i < j$

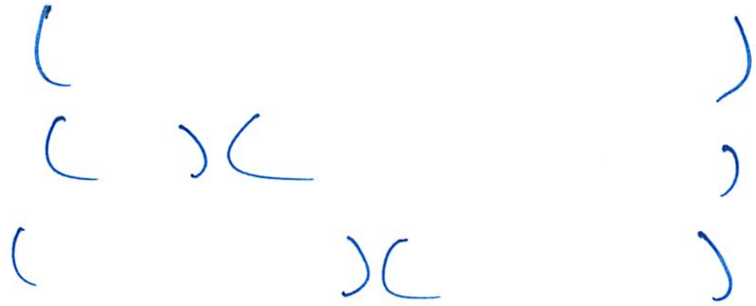


⑦

This game



#s



For optimality can't go from small problem up to big problem

Need to go big to small

idea of recursion here (in game) still bigger to smaller

Not working backwards from sink

Top-down vs bottom up are ways to solve
recursion — coding

each type (on prev pg) not the diff types

⑧

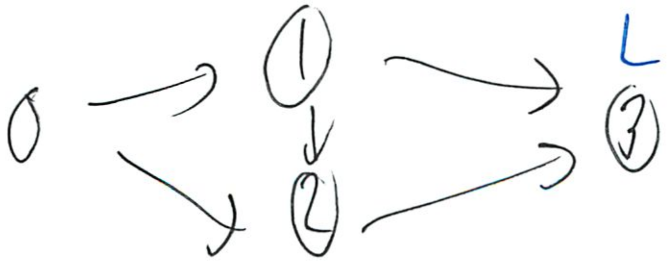
If Silvio can send his opponent by picking to an L
he wins

If Silvio is facing all Ws he loses
~~face of his Ws~~
but starting here

Since he has to put it on a W
Then his opponent

Ⓦ ← who ever turn it is on that node will
either W or L
We want to return all Ws
Since Silvio goes 1st

So our model



⑨

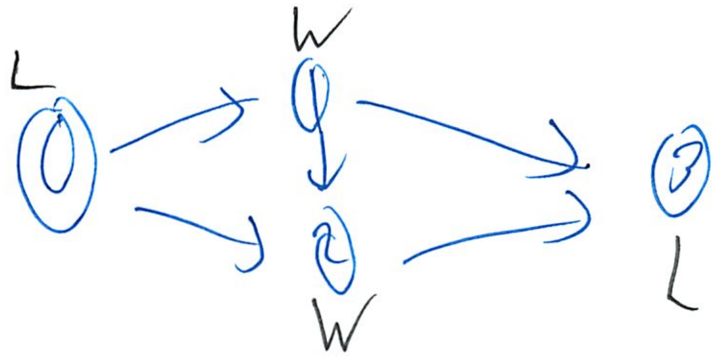
Blanks ?

No - not if $\overset{\text{rev}}{\text{topo}}$ sorted!

Would do order

3 2 1 0

reverse topo sort

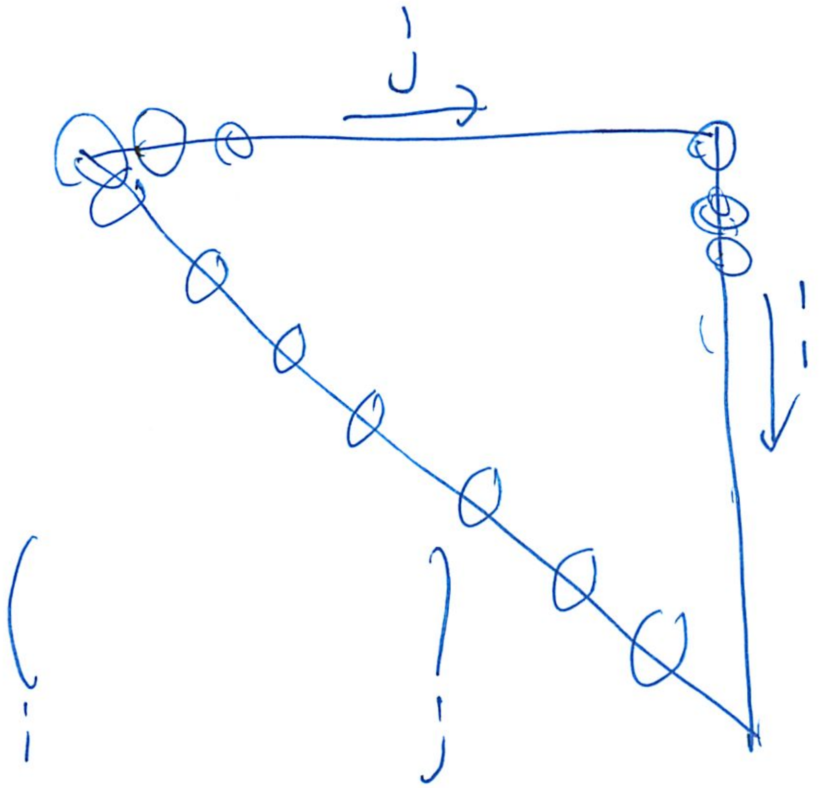


Return all Ws

So cool
Be able to apply this technique to all PP

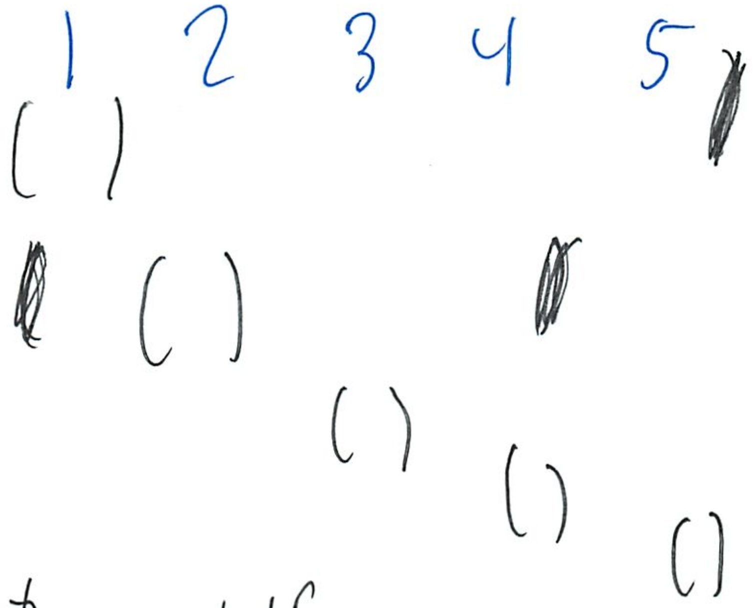
B. Multiplication

So the hint he gave us



try to see where is the best

So



But then what?

And this is worthless

⑪

Or is it

↓try each

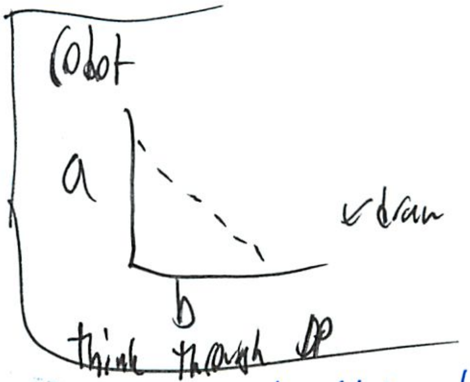
$(1) \overset{+}{\times} (2 \quad 3 \quad 4 \quad 5)$

$(1 \quad 2) \overset{+}{\times} (3 \quad 4 \quad 5)$

$(1 \quad 2 \quad 3) \overset{+}{\times} (4 \quad 5)$

$(1 \quad 2 \quad 3 \quad 4) \overset{+}{\times} (5)$

↑then recurse here



subproblem only thinks about himself

but this is still top down

exponentially big

fill 1 #

w/ memoization it works

but stack of recursion

└ if multiple threads overlap

⑫

largest = answer

after

	7	4	.5	10	1	
j	0	1	2	3	4	i
	7	28	ⓐ	ⓓ	ⓗ	0
		4	4.5	ⓒ	ⓖ	1
			.5	ⓑ 10.5	ⓕ	2
				10	ⓔ	3
					1	4

(7)

(7 $\overset{+}{\times}$ 4)

↑ + or ×

7 | 4 .5 or so 7 $\overset{+}{\times}$ 4.5 ← better

7 4 | .5 or 28 $\overset{+}{\times}$ .5

weird for loop - but can be done

⑬

So in ⓐ can do either

7 | 4 .5 or 7 4 | .5

7 $\overset{+}{\times}$ 4.5 .28 $\overset{+}{\times}$ .5

Tans from prev block

So best of the 4 is

31.5

write in ⓐ

Then for ⓑ .5 $\overset{+}{\times}$ 10 = 10.5

ⓒ [illegible]

4.5 $\overset{+}{\times}$ 10 ~~[illegible]~~

~~4 | $\overset{+}{\times}$ (.5 $\overset{+}{\times}$ 10)~~ or

4 | 10.5

(14)

(h)

7 ‖ g

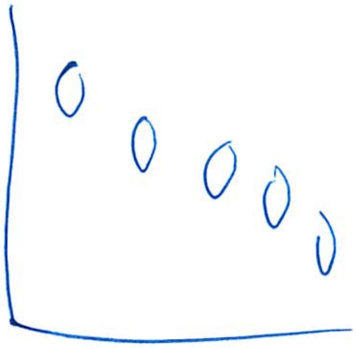
28 | f   or   take max

a ‖ e

d | e

the previous one before

Football



majorized players

Add robots

A's move down slope

└ add robot or improve

↑ generally

↑ unless on same line vert or horiz

Can't have i try to go to i+1

So get new i+1

Then figure out where to break

don't try to ↑

⑯

If n, compare to n-1
  └not ness.

Exponential # of groupings

Instead Take a guy you don't know
  See where it could have come from
That Transitions into the 1 you are looking at

) Foot ball problem

For multiply

$$(\quad) \overset{+}{+} (\quad)$$

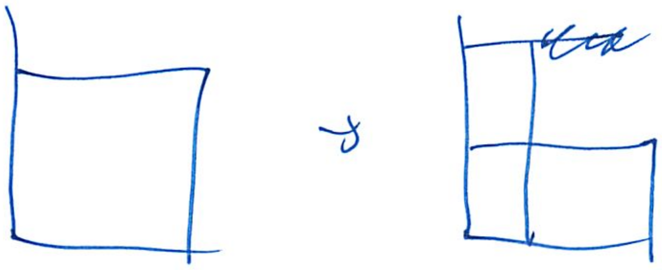
Go From biggest poblem
Till get to small prob
Answer then and go back up

So its still actually bottom up

⑰

First majorize everyone

Then split up



TA Works but not optimal

This is n^3

L n each for n^2 times

Can do in $O(n^2)$

Find a $O(n)$ subproblem

Goal: ~~Make big problem into~~

Make big into small

Go from big to small

⑱

Before

$$\underbrace{(x_1 \quad x_2)}_{\text{elim}} \quad \underbrace{x_3}_{\text{find optimal}}$$

?

must find optimal inside

Don't need the other here

If remove highest speed or strenght

Or if 1 robot fore each

Then try to combine

still $O(n^2)$ problems

This is diff

Since if maximize lot 10 - they are good

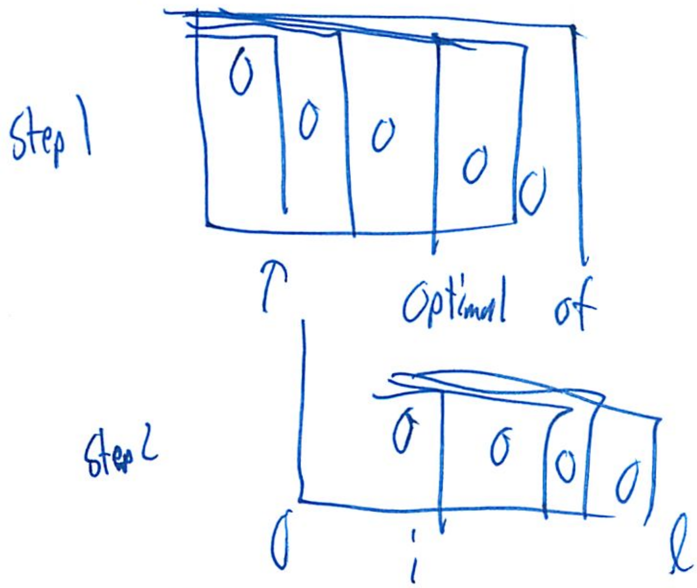
Theke

Have a bunch of cases

Not 1 robot and combine

Pick a guy
  - robot covers him an someone else

Start w/ k people
  - don't add or remove guys



Subproblem Optimal from i to l

S.O n subproblem

20

So max of inc 1 + optical rest
2 + "
3 + ") n

So n subproblems w/ memozation
n each Compute
5 → 1
4 → 1 Once
3 → 1

Can't say if part is optimal - rest is

~~(Always combined optimal of smallest~~

lots ? initution of this

but wording

- ( ) allow a place to break def,
) has to end some where

Must consider all poss
Found optimal on all of them
So found optimaly

㉑

Football

Find the optimal of the 6 possible transitions

$x-2$ $y-2$
$x-3$ $y-3$
$x-7$ $y-7$

1st robot must end somewhere
Consider all $O(n)$

Then 2nd robot must end somewhere

Bottom up way

last robot includes that last player

Look at next player
- is it optimal for same robot to improve

Must only consider these cases

order is this

Same as game

but from here

look this way

[redacted]

Not ness always in DP

(he doesn't know

aka'i

~~include or add~~ or improve

don't think of this

Which robot to improve; then

- don't get in

for k
↑ # of duties the 1st robot covers

bottom up

coding

look here - then look down at the possibilities

(I think I am ~~write~~ right here - need to think about)

So its same coding

treat sub problem as blackbox
just combine it simple

Subproblem = price of all robots.

(24)

Never bump a robot

So no list. need to know

6.006 Reading

5/1

Chap 15 DP

~~Subproblems~~ (skipping stuff I know)

Best on optimization

1. Characterize structure of optimal sol
2. Recursively define the value of optimal
3. Bottom up find
4. Put it all together

~~What is this~~ bottom up or top down? both?
When use when?

Rod cutting

For max price
as many qty as you want
Can decide not to cut
2^{n-1} diff ways at each int, cut or not

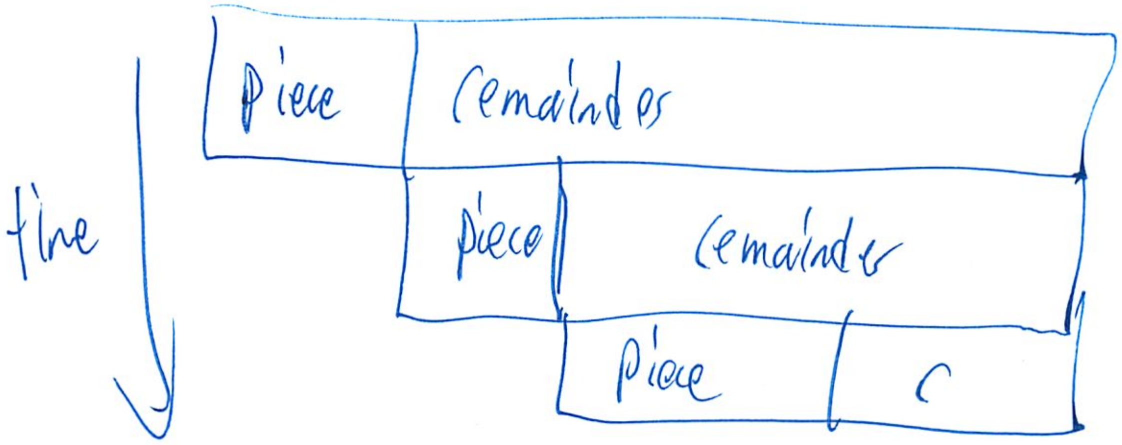
②

$r_n = \max(p_n, r_1 + r_{n-1} \ldots)$

must consider all possible cuts

Once make 1 cut → rest of problem is same structure as before

↑optimal substructure



$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

↑only 1 subproblem

↑test all i

↑find max of current price and max price of remainder

So tree

test n at each level

n levels max $O(n^2)$ correct?

③

Top-down

```
Cut rod(p, n)
    if n==0
        return 0

    $q = -\infty$ ← max revene
    for i=1 to n
        $q = \max(q, p[i) + \text{cutrod}(p, n-1))$
    return q
```

So method we saw before

But for $n = 40$ → more than an hr

L since recursivly calls over + over

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) \qquad j = n - i$$

Oh opps one less on each level

So 2^n

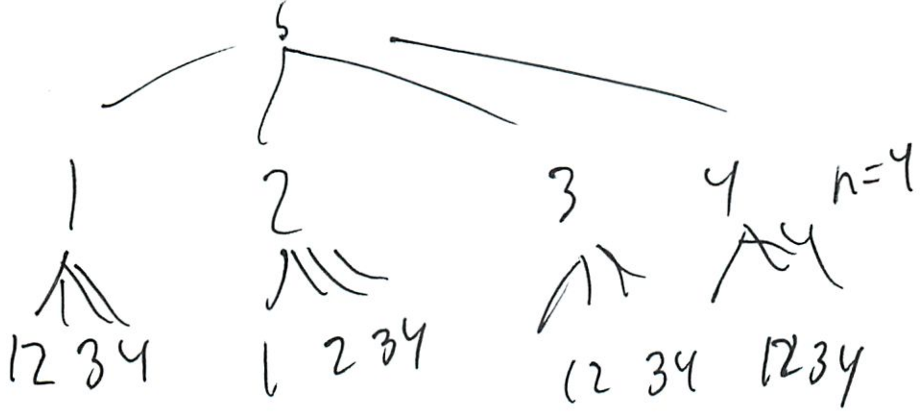
? how do we get this?

its an execise - so they don't tell you

But draw it

at each level branch out w/ n-i items

If was simple



Would be ∞ if forever

n^{n} ?

⑤ OH

$T(n)$ is time

to decide 4 need 3, 2, 1, 0

then need to solve that recursion

$T(0) = \underline{\underline{1}}$

try it $T(1) = \underbrace{1}_{\text{static}} + \underbrace{1}_{T(0)} = \underline{2}$

$T(2) = 1 + \underbrace{2}_{T(1)} + \underbrace{1}_{T(0)} = \underline{4}$

$T(3) = 1 + \underbrace{4}_{T(2)} + \underbrace{2}_{T(1)} + \underbrace{1}_{T(0)} = \underline{8}$

↑ see just 2^n

---

math way

$$1 + \boxed{\sum_{i=0}^{n-1} 2^i} = 2^n - \cancel{1} + \cancel{1} = 2^n$$

geometric series

$T(0) = 4$

$T(1) = 16$ at each level $= 4^k$ $k = \#$ of levels

$T(2) = \downarrow 64$

↑ remember this is # of levels

plus need to add the prev levels

which is less than double

---

eventually problems get small!

need to estimate height

So if divide by 4

height $\lg_4 n$

Usually last level has most on leaves

or nodes

Master theorm ↗ read theorm 3 cases

↑ not that important

TA: Try smaller cases. Find pattern

(Ok I think I feel better about this)

Back to cut cod

continue

Doubles when n ↑ by 1

2^k branching factor of 2

k levels

though here it includes the previous

⑦

So DP → save solution

time-memory tradeoff

this is top-down w/ memoization

~~So have solved~~

also bottom up

have already solved all of the smaller/prereq sub problems

both are same asy running time

but bottom up usually better - less recursive over head

Add memoization to top down

~~bottom~~

Or bottom up even simpler

```
r[0 ... n] new array
r[0] = 0
for j = 1 to n
    q = $-\infty$
    for i = 1 to j
        q = max(q, p[i] + r[j-i])
    r[j] = q
return r[n]
```

* Uses the natural ordering of the sub problems
Solves j = 0 → n in that order

Both memoized + bottom up = $O(n^2)$

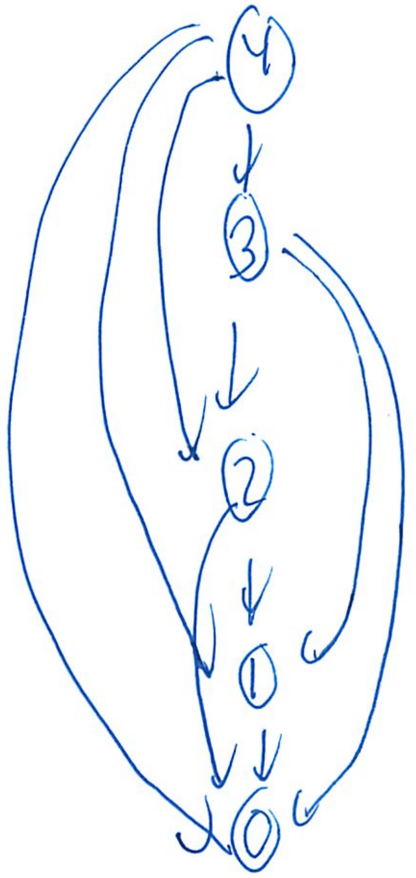
- solves each size n just once ← memo, top down
  - requires n iterations

or it iterates across the n problems once ← bottom up

⑨

Subproblem graph

For n = 4



So ⑨ → 3, 2, 1, 0

③ → 2, 1, 0

② → 1, 0

① → 0

⓪ → nil

this shows that subproblem ② requires sol of subproblem ①, ⓪

So we've collapsed our subgraph tree into 1 node per object

Bottom up is like the reverse topo sort

Solve each node once so

$\#$ nodes $\cdot$ time per node

↳ usually the degree of outgoing edges

Extend bottom up to compute optimal size for 1st piece s

↳ details...

15.3 ~~Optimal~~ ~~Substructure~~ Elements

Optimal substructure

Subproblems

1 Making a choice leaves subproblems

2. You are given the choice that leads to an optimal sol

3. Determine subproblems

4. "Cut and paste" subproblems
ie cut out non optimal sol
paste in better one

Ah oh

So ~~path~~ the subpaths ~~of~~ an optimal shortest paths are the shortest path b/w 2 points

can't sub them for anything

↑ not true for longest paths

problem must be independent

⑫

Overlapping subproblems

Sub problem small + repeated over + over
Can store in table

15.4 Longest Common Sub Seq

from lecture

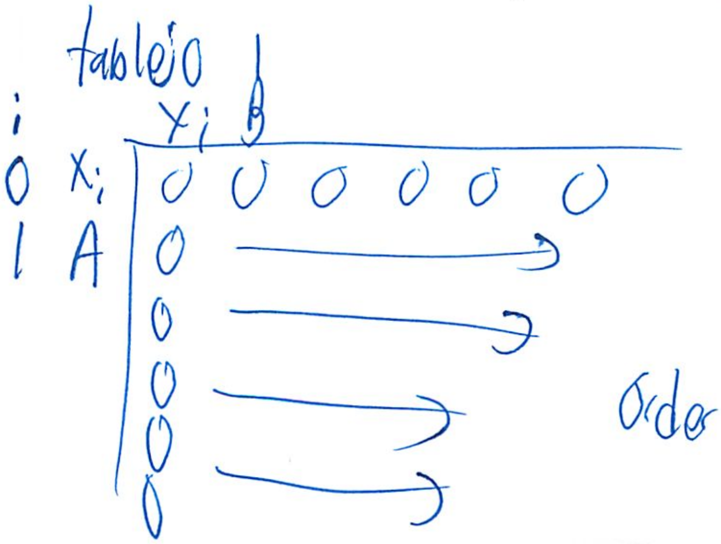
S_1 = ACCG ...
S_2 = GTCGT ...

in order, but not consecutive

2^m sub seq in X

(they made the notation way too complex!)

The table



Order process table

⑬

if $x_i = y_j$

++, ↖ so if same, increment 1

else if $c[i-1, j] \geq c[i, j-1]$

↑ ~~the~~ skipping on ~~top~~ left

else

← stipping on top

Then find ~~largest in the table~~ last node in table (lower right)

Print out backwards $O(m+n)$

Since its largest up to this point!

Can make tweaks (no change asy)

- elim b and figure it out
- only keep last 2 rows

(14)

Exam

Work →

C[i] longest seq that ends at i

look at C[1] → C[i-1]

then look for max C[j] to append

So sample

find C[1] (2,7) (0,0) (7,0) (0,3) (0,6) (7,6)

max (find C(2) find C(3) find C(4) find C(5) find C(6))

↳ (find C[3] C(4) C(5) C(6))

↳ [illegible]

↳ [illegible]

↳

think I did no memo

$O(n^2)$

⑮

Exam b

Single linear scan

6 possible terms that could imm precede S[i]

(2,0)	(0,2)
(3,0)	(0,3)
(7,0)	(0,7)

S[i]=(a,b) = leng of longest foot ball seq ending here

Compute C[i] by searching for all 6 possible in a chash table

~~Work~~

But it says look for preceding 6 values

$\longleftarrow$ Work ???

No do S[1] first

(16)

(2,7) (0,0) (7,0) (0,3) (0,6) (7,6)

S[1]

(what is d)

↳ the hash table

but what values is stored

$d[(a,b)] \leftarrow \max\{d[(a,b)], c^* + 1\}$

$\uparrow c^* = \max_{p \in P} d[p]$

P = the 6 values

So S[1]

look for the 6

must precede → so no

S[2]

nothing preceds

S[3]

(0,0) preceds

has d = 1

new d = max(1, 1+1) = 2

for (7,0) is 2

⑰

S[4]

match ~~(1,0)=2 2 pick~~

(0,0)=1

~~insert~~

insert as 2

S[5]

(0,6)

get (0,3) = 2

(0,0) = 1

Store as 3

So basically work

$\longrightarrow$

O(h)

I did this convoluted work backwards

Though got 14/28 on this

Basically I confused the top down and bottom up... which are 2 methods to do the same thing. Just diff to code

The recursive + non recursive

```
ie  fib = fib(~~[illegible]~~ n-1) + fib(n-2)
or
    fib(k)
        while ~~m~~ 1 to k
            fibi = fib[i-1) + fib[i-2]
        return fib k
```

[Need to review Floyd Warsal too

In their longest common sub seq they did work backwards thus essentially adding a letter when working back ↑

⑲

(This whole forwards/backwards thing ...)

So what is the bottom up

just go through table

Top Down is gout out the whole tree

Knapsack

$\max\left(Val[i+1] \text{ or } v_i + Val[i+1]\right)$

So not a well defined recurrence

$\max\left(Val[i+1, x], v_i + Val[i+1, x-s_i]\right)$

(skipping)

⑳

Text Justification

$$DP[i] = \min_{j \text{ in } (i+1,n)} \text{badnes}\,[w[i:j] + DP(j+1)]$$

↑notation is weird

$DP[i]$ - min barness for $w[i:n]$
↑best layout

So decide where to end 1st line

Add that bareress and move on

words 1 2 3 4 5

1

1 + DP[2]

2 + DP(3)

3+ DP[4)

etc

So $n + n-1 + n-2 + \dots$ n times

$n2$ total

What is the bottom up?

Last word badness = 1

But can't tell that

or marginal badness?

Ask on Piazza

Is top down since big recursive tree ...

(Think I am better - but not good at DP)

Wiggle More

5/2

Implement #3

7 4 .5 10 1

7
4
.5
10
1

Now what for to do
did

main diagonal

No do

②

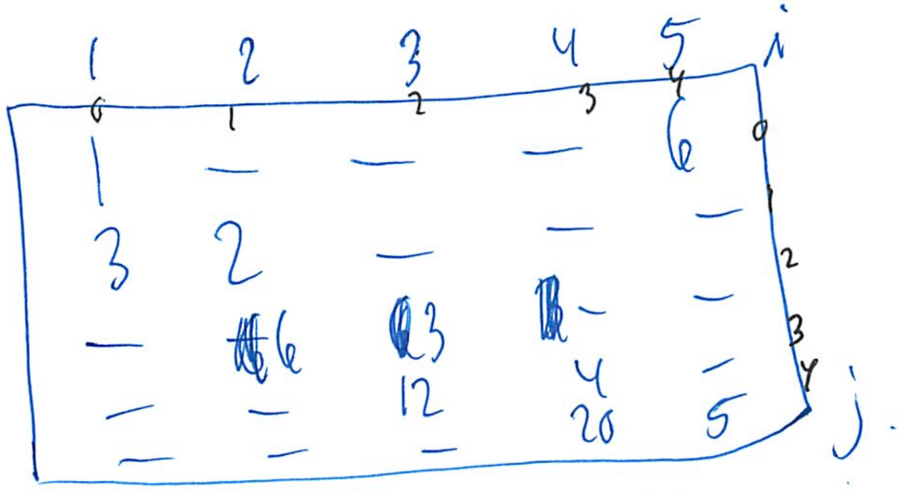
So numbers i j

1 0 → look at 0 0
1 1

2 1

3 2

4 3



? wrong side

So add back > 0 check on i

The l and r are right

1 and 2

~~Should~~ i -1

Should write in i, i-1
which it is!

Oh I draw it

③

So	0	1	2	3	4	j
0	1					
1		2				
2			3			
3				4		
4					5	
i						

Change the rep back to not confuse me

Well print function

Want it to print

j=0 $\overset{i=}{1}$ 2 3 4

Work print [i][i]

④

Now the next level is

2 0
L checks 00 and 2 1
+1 ↳ 10 and 22 ↲ +1

3 1 — checks 11 and 32
21 33
4 2

Stop

Find the pattern!

diff

2 i-2 i-2 vs i-1 i-2
i-1 i-2 i i

Same

3 i-2 i-2 i-1 i-2
i-1 i-2 i i

Order messed up

(so lost in the weeds

⑤

(✓) wont got another layer

now need generic layer

(✓) Finished

└ but done

Wrong in round 3

Should be 1 .4

should

1 3 4 2004 find

2 6 is 24

3 12

9

(✓) Fixed

(✓) Works on test case 1

(✓) Test case 2

(✓) Test case 3

Must actually return data

(✓) 40/40 on #13

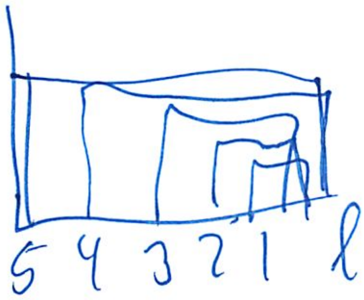
ⓔ

Why is Football n^2

do each one l times only

from i to l
↑fixed

5 → l
4 → l
3 → l



Call optimal on rest

Only do each once

but must scan n - each time

①

#2 – Fall 2011 lecture on topo sort subcode
Then super easy to code

#1d | $\text{Game}(n,k) = \max_{\frac{k}{2} \le i \le k} (-1)^n \cdot \text{Game}(n-1, i)$

i) How many subproblems

ii) Total time w/ memoization

max

$\frac{k}{2} \to k$ so k

then each one subtract 1

$n \cdot k$

since $k < n$

n^2

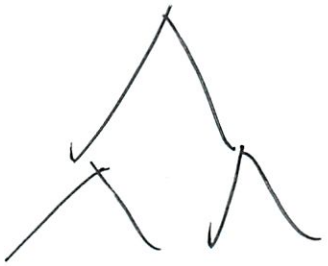
⑧

~~Think forwards~~

Or n subproblems
each n time

So n^2 over all

#1e) $\text{Half}(i,j) = \sum_{k=0}^{\frac{j-i}{2}} \text{Half}\left(i+k, i+k+\frac{j-i}{2}\right)\Big)^2$



log n subproblems

n log n total

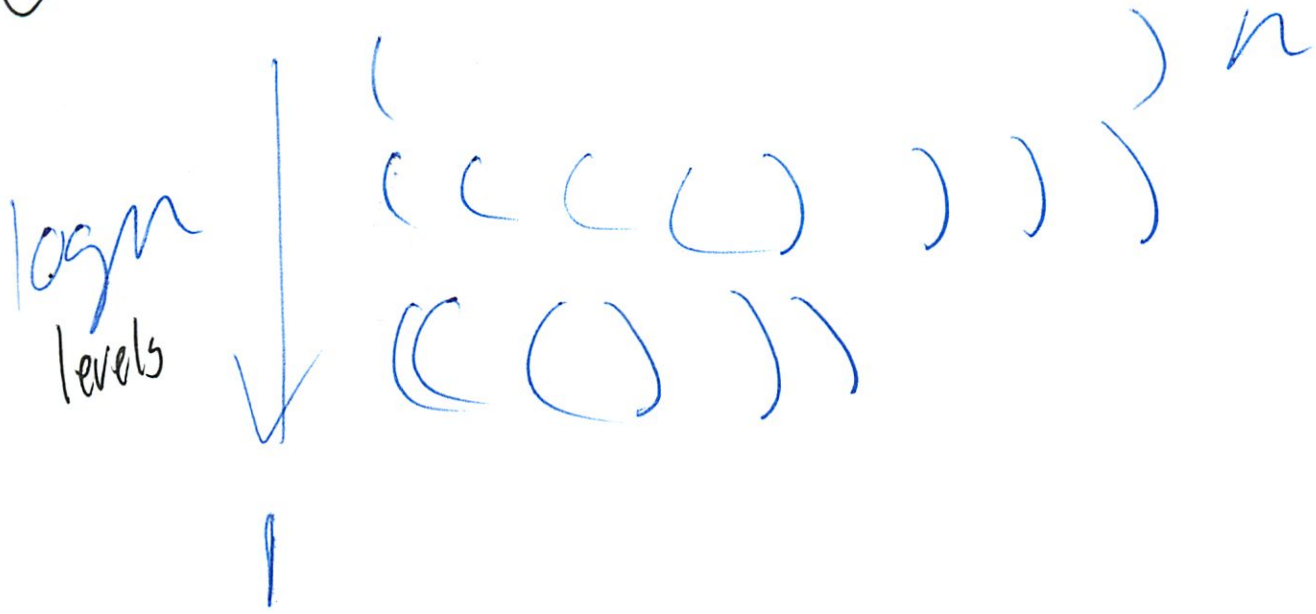
reasoning of name

But each one takes n time?

i+k + half interval

log n intervals

⑨

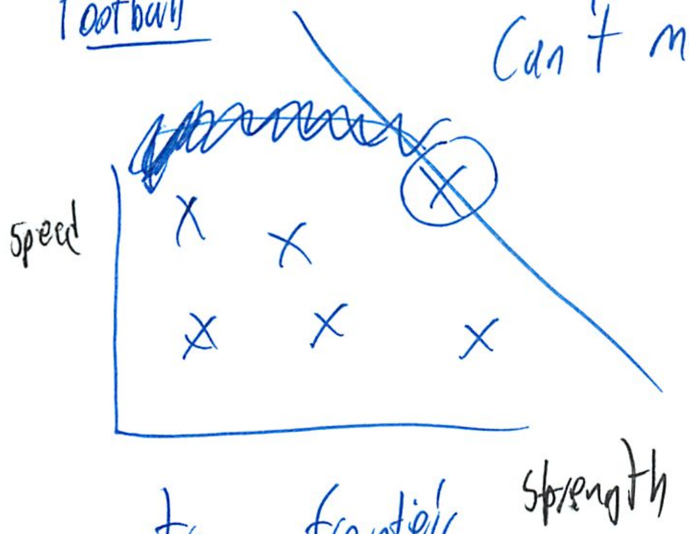


n log n subproblems

each O(1)

So n log n total

⑩ Football

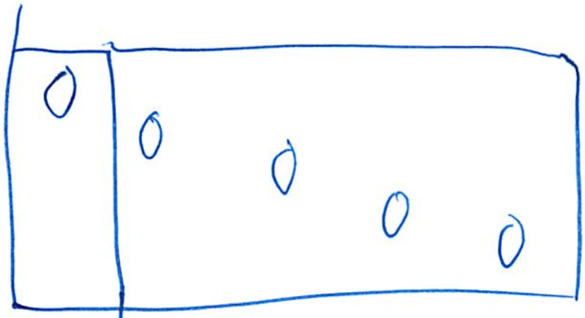


Can't move

- unmajorized by any other Harvard player

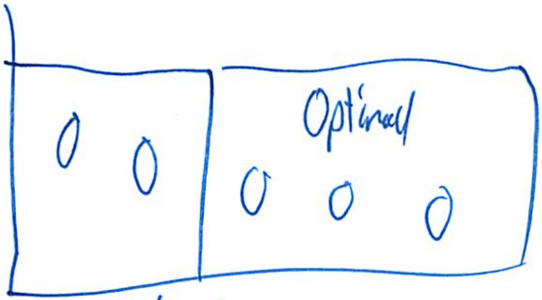
trace frontier

Only majorized players on frontier



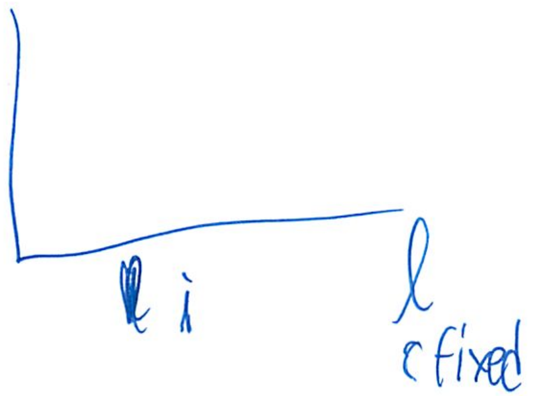
cost of that + optimal of rest

Or



cost of that

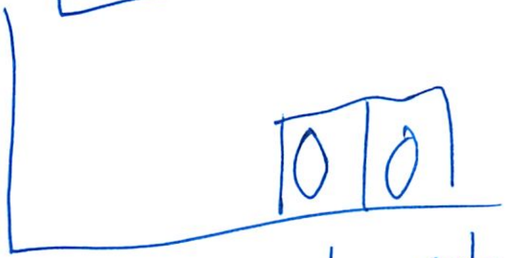
Or etc



Optimal(i, l)
↑fixed

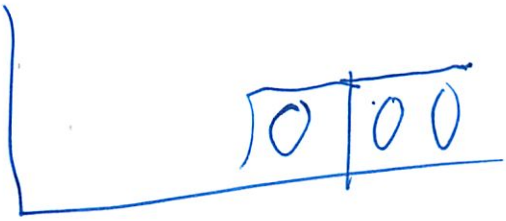
So its optimal(i)

= Cost ← just return



"bottom up"

min cost of 2 robots ← adding
or cost of 1 robot ↑ improving

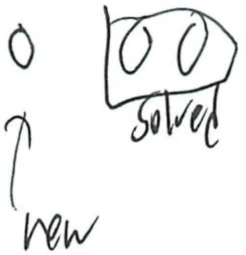


min cost of adding a robot
or improving
↑ despite TA hating this verbage

Black box sol for the 2 players = black box
– don't change it!

So how does this work then

So then



So can't improve

So what is the choice??

(bad pictures...)

Subproblem = piece of robot

Is choice including 1 more robot or 2 more?

So top down

⑬

This is bottom up

Optimal of 2 ~~(top down)~~ ① ②

Choice is

- 1 robot majorize both ① ②
- 1 robot majorizes 1 + opticamal of 2 ① ②

returns the best of ~~the~~ those

Still qv is bottom up sol...

For 3

X X X

? robot for 3rd only + optimal of 2 from above

or robot for ~~all~~ the first 2 + optimal of last (ie its own robot)

㉔

or a robot for all 3.

✓ I like it

Le. OCK 5/3

Implement

#2 1. Reverse topo sort

└ no real good code online
don't have my book...
└ don't futz around w/

#4

How to ignore the other players?
When go through the list

Oh min, not max

The TA didn't ~~describ~~ describe his help particularly well...
Or react well to what we were describing
(makes stuff worse...)

(Good to write it up well
└ (clearly understand)

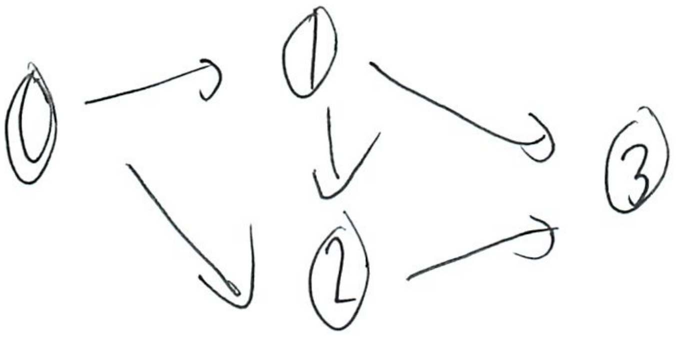
②

Reverse topo sort
- do it
- then reverse arrows?

Do we have DFS done?

When vertex finished (black) add to linked list

No DFS block



topo

3 2 1 0

Where to put the line to add

Insert at start is bad
↳ just reverse the list

③

Screwed it up big time!

 L Fixed

Now the other part

Work backwards

 what is it pointing to?

 Lables directory

What about unknowns?

 Should be none

Ok working - but not done

 If [illegible] any [illegible] Losses is a win

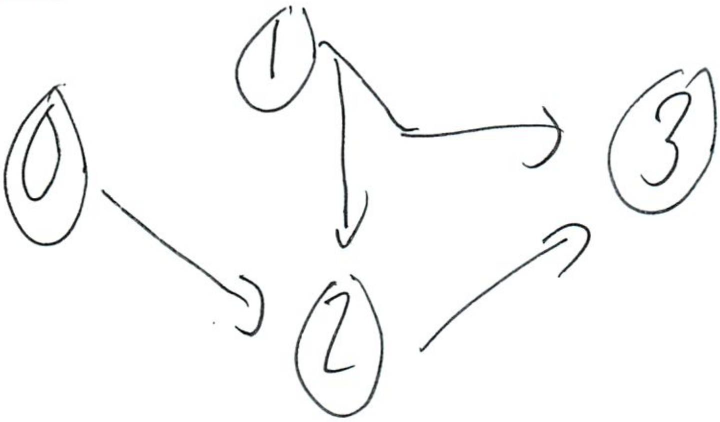
✓ solved

 now need to [illegible] return

✓ Done

0

Failed



So

3 L

2 W is it? ans (1,2)

1 L

0 L

It never ~~did~~ visited 1

Why not?

Need to add some BFS style queue...

No should iterate over all

The topo is wrong

⑤ Oh it needs multiple start points!

Just try all...

Or any we have not visited...

have not seen, but try it....

(I should think clearer)

✓ Done

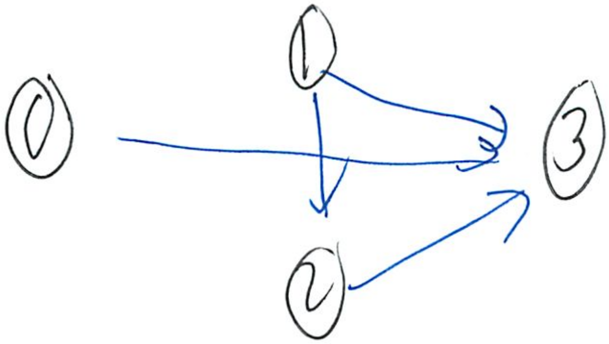
Oh ~~[illegible]~~ what order?

reverse <u>within</u> the sort

✓ Fixed

Grrr error

(It passed like 8)

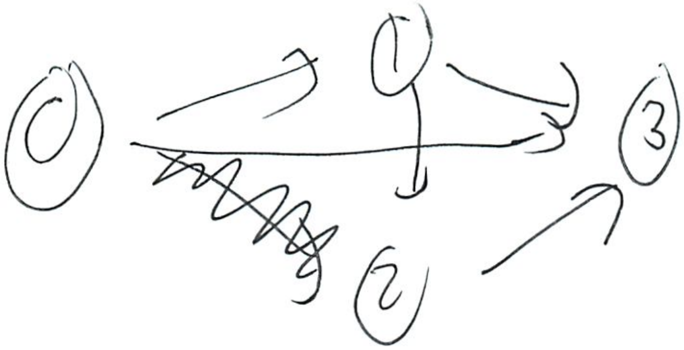


Save!

5/4

Now diff key error

(I did this all wrong - now figuring from scratch)



Yeah is reasonable ...

No in order that returns at start

I only reverse once

Then reverse all ?

correct ⓪ ① ② ③

it's not when return

not a problem till now!

That's in CLRS - which don't have handy

But

①

Got back

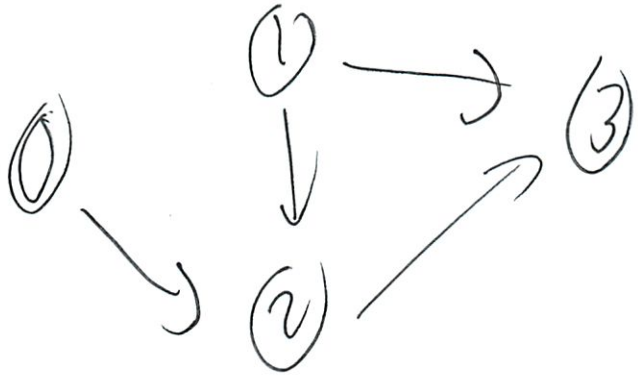
put color at after for each visit

They - have sep visit function ...

if unvisited

Move append back one

So error



appends

1, 3 2 0

↑sep

but how does it get to 1

Sep - It should append to end

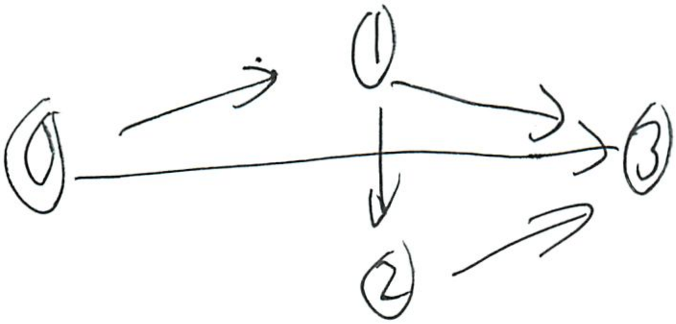
(this has been so much trouble!)

⑧

Oh a bad reverse

① Fixed

Now



2 3 0

Never adds 1

visits but why not finishes

They visit immedatly

here adds to queue

is that the difference?

Redo the stack of items...?

└ but then non standard

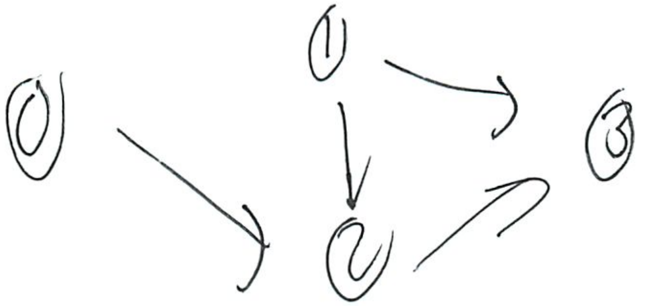
mixing ideas...

Doing like book w/ sep fn (makes more sense to me

So switched to CLRS DFS

No source..

Recheck



3 2 0, 1

(we never reverse...)

Oh we do want it reversed...

No no non visited BS

Is this correct now?

✓ Passed all little test cases

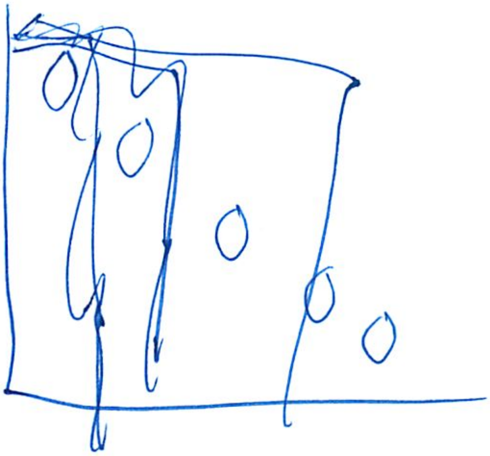
✓ Passes large test cases 30/30

Should have used book from begining

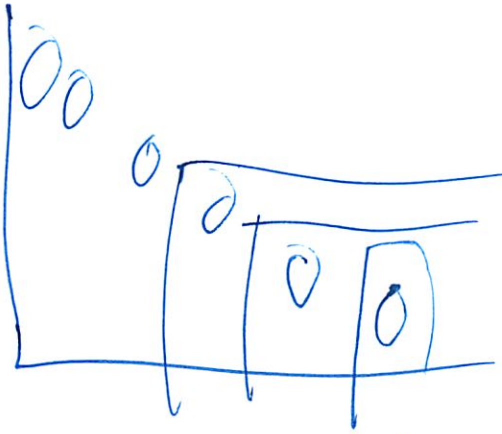
Done

Explain to Shri

5/15



Optimal(n, ~~[illegible]~~ Fixed)



optimal (1 → n) = n

$$\frac{n \text{ entries}}{n^2}$$

"Convex hull"

```
""" Problem 1 """

answer_for_problem_1_part_a_i  = 'C' # Theta(n)
answer_for_problem_1_part_a_ii = 'C' # Theta(n)
answer_for_problem_1_part_b_i  = 'E' # Theta(n^2)
answer_for_problem_1_part_b_ii = 'E' # Theta(n^2)
answer_for_problem_1_part_c_i  = 'C' # Theta(n)
answer_for_problem_1_part_c_ii = 'E' # Theta(n^2)
answer_for_problem_1_part_d_i  = 'E' # Theta(n^2)
answer_for_problem_1_part_d_ii = 'G' # Theta(n^3)
answer_for_problem_1_part_e_i  = 'D' # Theta(n log n).
""" Let L = j-i.  In the worst case, L = n, and there are n subproblems for each of the
log(n) interval lengths L, L/2, L/4, ..., 1 """
answer_for_problem_1_part_e_ii = 'E' # Theta(n^2).
"""
Naive analysis gives O(n^2 log n), but it only takes S time to evaluate a subproblem of
size S.
So since there's at most n subproblems for each of the sizes S = L, L/2, L/4, ..., 1,
we have that the total time needed is n * (L + L/2 + L/4 + ... + 1) = n * 2L = Theta(n^2)
"""


""" Problem 2 """

# Reverse topological sort
def reverse_topo_sort(graph):
  visited = set()
  order = []

  def single_source_DFS(source):
    visited.add(source)
    for neighbor in graph[source]:
      if neighbor not in visited:
        single_source_DFS(neighbor)
    order.append(source)

  for node in graph:
    if node not in visited:
      single_source_DFS(node)
  return order

# Run the DP, going in reverse topological order.
# Subproblems are of the form "Does player 1 win if he is currently at node x?"
# A node is a winning node iff there exists a losing node which it points to
def find_winning_nodes(graph):
  order = reverse_topo_sort(graph)
  answer = set()
  for cur in order:
    if any(next not in answer for next in graph[cur]):
      answer.add(cur)
  return list(answer)

# A faster implementation of the above, which does the DP while doing the topological sort
```

```
def find_winning_nodes(graph):
  winning = {}
  answer = set()

  def find_if_winning(source):
    for neighbor in graph[source]:
      if neighbor not in winning:
        find_if_winning(neighbor)
      if not winning[neighbor]:
        winning[source] = True
        answer.add(source)
        return
    winning[source] = False

  for node in graph:
    if node not in winning:
      find_if_winning(node)
  return answer

""" Problem 3 """

# An O(n^3) DP which works, even with negative values
def find_largest_value_n3(numbers):
  memo = {}
  for i in range(len(numbers)):
    memo[(i, i+1)] = numbers[i]

  for d in range(2, len(numbers) + 1):
    for i in range(0, len(numbers) - d + 1):
      memo[(i, i+d)] = max(max(memo[(i, j)] + memo[(j, i+d)] for j in range(i + 1, i + d)), \
                           max(memo[(i, j)] * memo[(j, i+d)] for j in range(i + 1, i + d)))

  return memo[(0, len(numbers))]

# An O(n^2) DP.  It uses the fact that with only positive values, all parenthesizations are
products of sums.
def find_largest_value_n2(numbers):
  n = len(numbers)
  sum = 0
  prefixes = [0]

  for i in range(n):
    sum += numbers[i]
    prefixes.append(sum)

  def sum_from(i, j):
    return prefixes[j] - prefixes[i]

  best_to = [1]
  for i in range(n):
    best = max(best_to[j] * sum_from(j , i + 1) for j in range(i+ 1))
    best_to.append(best)
```

```
  return best_to[n]

def find_largest_value(numbers):
  if all(x >= 0 for x in numbers):
    return find_largest_value_n2(numbers)
  else:
    return find_largest_value_n3(numbers)

""" Problem 4 """

answer_for_problem_4 = '''
DESCRIPTION:

Without loss of generality, assume there are no majorized players. (If there are, simply
eliminate them.)

We then sort the remaining players in order of decreasing strength (and thus increasing
speed).
So we can assume that a_1 > a_2 > ... > a_n and b_1 < b_2 < ... < b_n.

Now simply do the following:
  Initialize P[0] = 0.
  For i = 1, ... , n:
      Let P[i] = max_(j=1,...,i) ( P[j-1] + a_j * b_i )
  Return P[n]

CORRECTNESS:

We have subproblems of finding the lowest cost P[i] for a robot team majorizing players 1
through i
(i.e. the problem where Harvard's team only has their first i strongest players).

Now suppose we know P[1], P[2], ... , P[i-1] and would like to compute P[i].
In order to majorize the first i players, we need a robot which is as fast as player i.
Furthermore, we will not use a robot any faster, since player i is the fastest of the first
i players.
There are i potential strengths to consider for this robot:  a_1, a_2, ..., a_i
If we use the strength a_j, we will have majorized Harvard players j through i.
The remaining robot team to use should simply be that which we used for P[j-1]
Thus taking the max over j = 1,...,i of P[j-1] + a_j * b_i gives us the value P[i].

Clearly P[n] is the value we would like to return.

RUNTIME:

Removing majorized players takes O(n log n), as seen in 'Team Selection' on Quiz 1.  (The
naive O(n^2) algorithm also suffices.)
Sorting takes O(n log n) as well.
The remaining loop takes O(n^2), since it takes O(n) time to compute each of n subproblems.
'''
```

L21 6.006

5/1

(Last lecture on DP)

Application: Text Justification

Obvious greedy put as much on 1st line
then continue to lay out rest

but he thinks the layout is suboptimal

Need to formalize layout as optimization problem
- so can optimize
- assigns a score to each layout

One possibility -1 pt for each blank space
but this is the same in both
"linear"

Other: $\text{badness}(L) = (\text{page width} - \text{total Lenght}(L))^3$

$\sum_L \text{badness}(L)$ $\in$ minimize

②

Solve w/ DP!

Proposals for subproblems.

- Smaller problems can shoot for 1st

$dp[i]$ = min badness for words $w[i:n]$

n = # of words

decision: where to end 1st line in optimal layout

$$DP[i] = \min_{j \text{ in range}(1+i,\, n)} \left[\text{badness } w[i:j] + DP[j+1]\right)$$

↑ badness of 1st line + ↑ best we can do w/ the rest

$DP[n+1] = 0$

Optimal = $DP[1]$

Runtime $O(n^2)$

n subproblems

n time each since running over the range

③

Could we do bottom up?

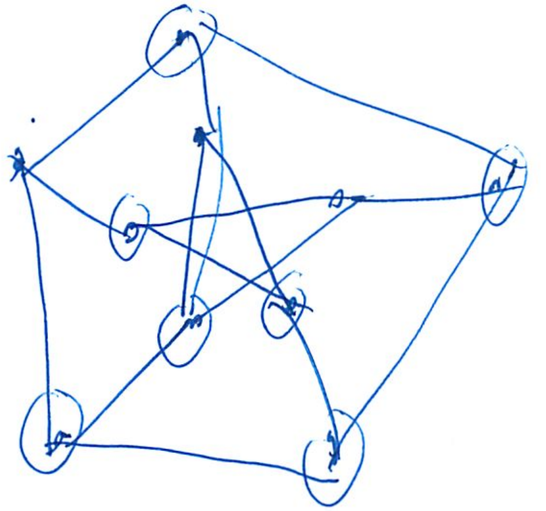
- no recurse
- high to low

- Smaller to larger subproblem

(guess we won't do it...)

Structural Dynamic Programming

ie Vertex Cover on trees



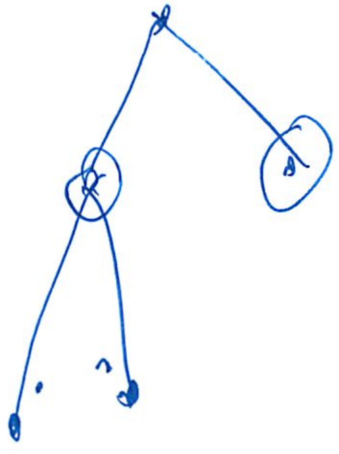
Smallest # of guards

So at least one endpoint of each edge covered

NP-hard in general
- L no polynomial time alg

So stick to trees
- L which are solvable

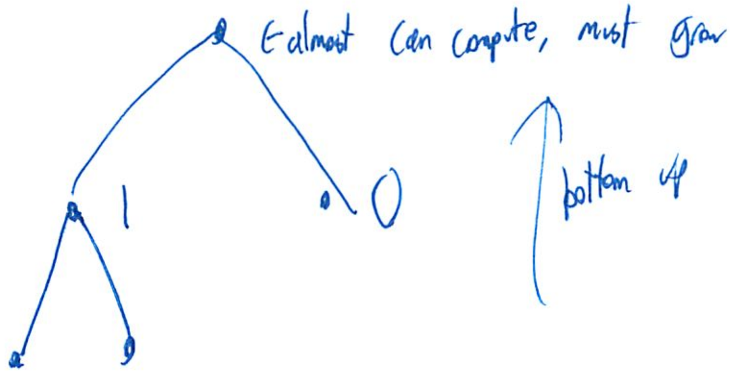
④



Can do bottom up or top down

There are no cycles ← good about trees

Ideas?



← almost can compute, must grow problem

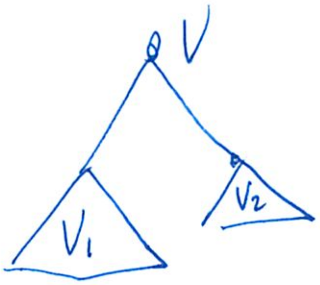
$\text{cost}(v, b)$ = min cost solution of subtree rooted at v

assuming v's status is $b \in \{\text{Yes}, \text{No}\}$

Yes = included
No = not included

Recurrence:

⑤



$$\text{Cost}(v, \text{True}) = 1 + \min_{b_1, b_2}\left(\text{cost}(v_1, b_1) + \text{cost}(v_1, b_2)\right)$$

$$\text{Cost}(v, \text{False}) = \text{cost}(v_1, \text{TRUE}) + \text{cost}(v_2, \text{True})$$

Base case

$\text{Cost}(v, \text{Yes}) = 1$

$\text{Cost}(v, \text{No}) = 0$

2n subproblems

└ time for each [illegible] (missed)

So $O(n)$

└ $O(d)$ if degree d

but still constant for each edge

edges n → $O(n)$

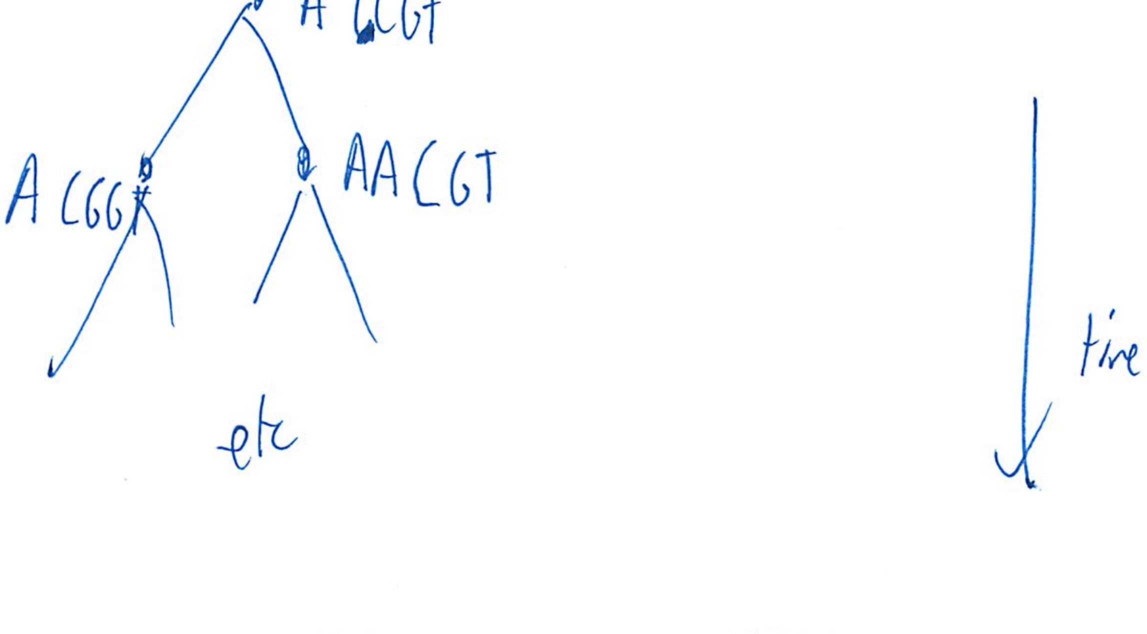
⑥

If the graph not a tree?

- if small size seperators are good enough
- guess all the possible joint states in the vertex cover
- can break graph into 2 parts + repeat
- more in 6.046

Parsimony

recovering the tree of life



⑦

We can sequence the leaves
  L the living organisms

But how can we build the tree w/ dead species?

(wait not all the ones that have branched off are extinct)

How to score a proposed tree?

Parsimony

Given n leave strings w/ letters {A, C, G, T}
and a tree

Goal: Find "inner node sequences"

parsimony score = 5

So that the sum of mutations along edges is minimize

⑧

We will use Of course DP!

Obs 1: We can consider 1 letter at a time
? the position thereof

Define $D(a, b) = 0$ if $a = b$ and $= 1$ otherwise

For any node v of the tree and labeled L
define $\text{cost}(v, L)$

This is the min cost for subtree rooted at v, if v is labeled L

$$\text{Sol} = \min_{L} \text{cos}(\text{root}, L)$$

recurrence for $\text{cost}(v, L)$

$$\text{Cost}(v, L) = \min_{L_1, L_2} \left(D[(L, L_1) + D(L, L_2) + \text{cost}(u_1, L_1) + \text{cost}(u_2, L_2) \right)$$

Best case : v is a leaf

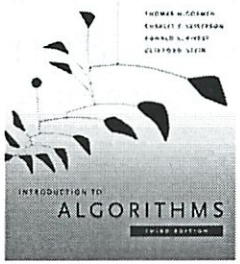
0 if use sure

$-\infty$ if use otherwise

↑penalty

(missed last slide)

6.006- *Introduction to Algorithms*



Lecture 21

Prof. Constantinos Daskalakis
CLRS 15

Menu

- Text Justification
- Structured Dynamic Programming
  - Vertex Cover on trees
  - Parsimony: recovering the tree of life

Menu

- **Text Justification**
- Structured Dynamic Programming
  - Vertex Cover on trees
  - Parsimony: recovering the tree of life

Text Justification – Word Processing

- A user writes stream of text
- WP has to break it into lines that aren't too long
- obvious algorithm => greedy:
  - put as much on first line as possible
  - then continue to lay out rest
  - used by MSWord, OpenOffice
- Problem: suboptimal layouts !!

e.g.			
	blah blah blah		blah blah
	b l a h	vs	blah blah
	reallylongword		reallylongword

5

A Better Approach

- formalize layout as an optimization problem
- define a scoring rule
  - takes as input partition of words into lines
  - measures how good the layout is
- it's not an algorithm, just a metric
- find the layout with best score
  - here's where you think of algorithm

Layout Function

- Want to penalize big spaces. What objective would do that?
  - sum of leftover spaces?
  - then

```
blah blah  blah               blah          blah
b    l    a   h   as good as  blah          blah
reallylongword                reallylongword
```

  - i.e. it's the same for two layouts with the same number of lines (just total space minus number of characters)
- should penalize big spaces "extra"
  - (LaTeX uses sum of cubes of leftovers)

Formally

- input: array of word lengths w[1..n]
- split into lines $L_1, L_2 \dots$
- **badness of a line:**

  badness(L) = (page width – total length(L))^3

  – (or ∞ if total length of line > page width)
- **objective:** break into lines $L_1, L_2 \dots$ minimizing Σ_i badness(L_i)

Can We DP?

- Subproblems?
  - DP[*i*] = min badness for words w[*i*:*n*]

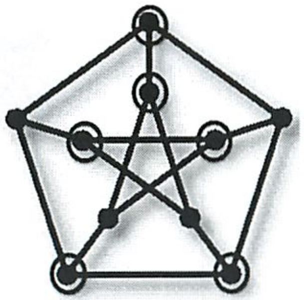
  (i.e. the score of the best layout of words w[*i*],…,w[*n*])
  - *n* subproblems where n is number of words
- Decision for problem *i*?
  - where to end first line in optimal layout of words w[*i*:*n*]
- Recurrence?
  - $DP[i] = \min_{j \text{ in range}(i+1,n)} (\text{badness}(w[i:j]) + DP[j+1])$
  - DP[*n*+1]=0
  - OPT = DP[1]
- Runtime? O(n^2) ?

Menu

- Text Justification
- **Structured Dynamic Programming**
  - Vertex Cover on trees
  - Parsimony: recovering the tree of life

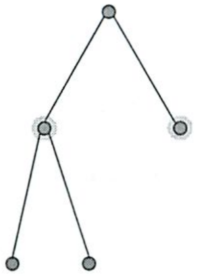
Vertex cover

- Find a minimum set of vertices that contains at least one endpoint of each edge
- (like placing guards in a house to guard all corridors)
- NP-hard in general

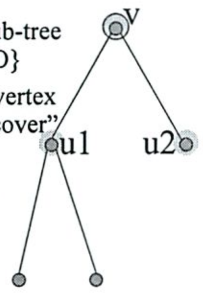


Vertex cover

- Find a minimum set of vertices that contains at least one endpoint of each edge
- (like placing guards in a house to guard all corridors)
- NP-hard in general
- We will see a polynomial (in n) time algorithm for trees of size n
- Ideas ?



Vertex cover: algorithm



- Let cost(v,b) be the min-cost solution of the sub-tree rooted at v, assuming v's status is b∈{YES,NO}
- where YES corresponds to "v included in the vertex cover" and NO to "not included in the vertex cover"
- Recurrence for cost(v,b)

  cost(v,YES)= $1+\min_{b1} \text{cost}(u1,b1)+ \min_{b2}\text{cost}(u2,b2)+\ldots$

  cost(v,NO)= $\text{cost}(u1,\text{YES})+\text{cost}(u2,\text{YES})+\ldots$

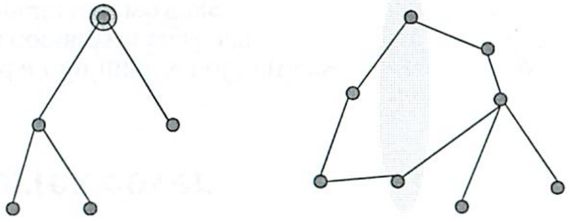
  include a term for all children of v
- Base case v=leaf:

  cost(v,YES)=1

  cost(v,NO)=0
- Running time ? O(n)
- Because constant amount of work per edge of the tree in the execution of the algorithm

What if graph is not a tree?

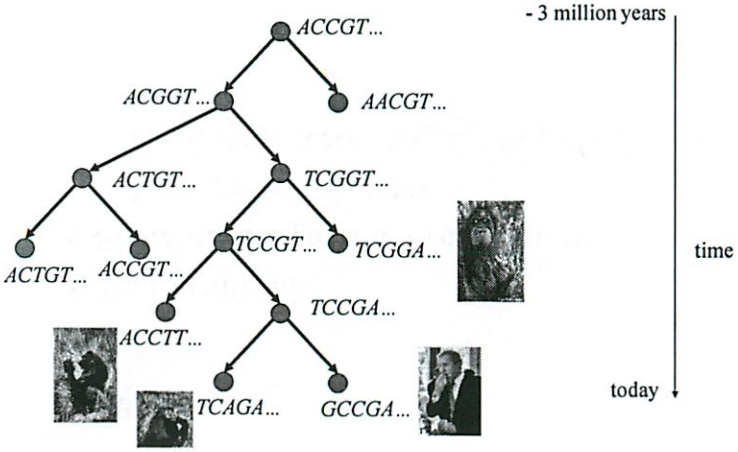
- For trees, we had two subproblems corresponding to whether we included a vertex in the vertex cover or not..
- For general graphs, the existence of small separators is good enough.
- We can have a DP subproblem for all possible joint states of the vertices in the separators.
- Notion of "treewidth" of a graph (advanced material)



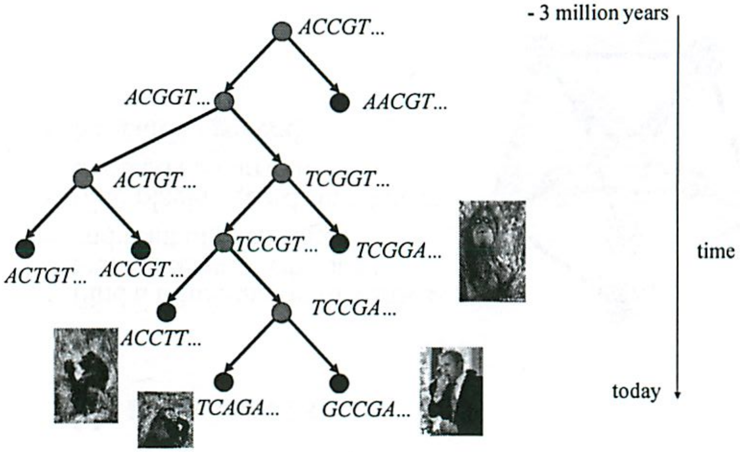
Menu

- Text Justification
- Structured Dynamic Programming
  - Vertex Cover on trees
  - **Parsimony: recovering the tree of life**

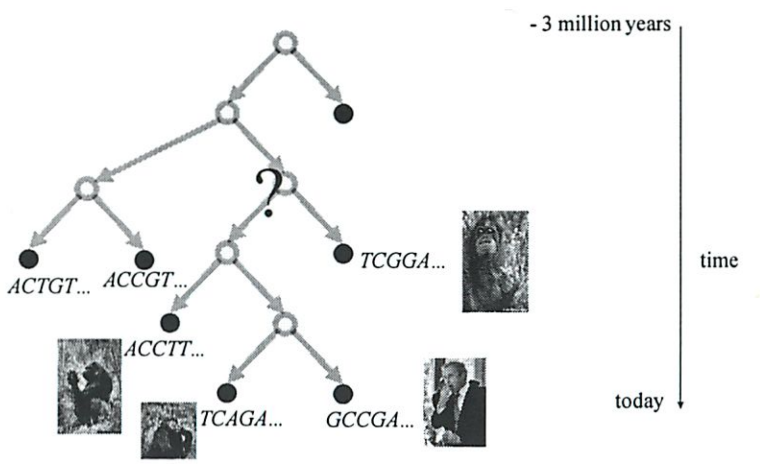
The Tree of Life



The Computational Problem

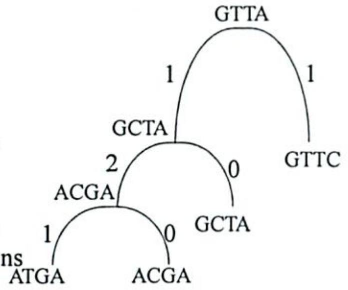


The Computational Problem



Useful Subroutine: Scoring a proposed tree

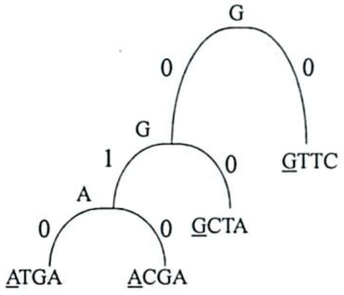
- A desired property of a plausible tree:
  Explains how the observed DNA sequences came about using few mutations.
- Such tree has **"high parsimony"**.
- Algorithmic problem. Given:
  - n "leaf strings" of length m each, with letters from {A, C, G, T}
  - a tree
- Goal: find **"inner node" sequences** that minimize the sum of all mutations along all edges
- This is the **parsimony** of the tree.
- Algorithmic Ideas ?



parsimony = 5

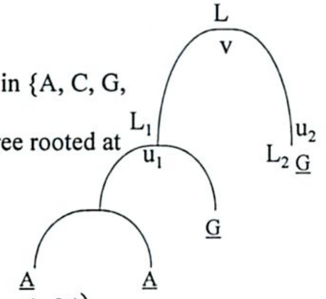
Parsimony: algorithm

- Observation I: we can consider one letter at a time
- Observation II: can use dynamic programming to find the best inner-node letters



Parsimony: dynamic program

- Define letter distance as follows
  D(a,b)=0 if a=b and =1 otherwise
- For any node v of the tree and label L in {A, C, G, T}, define cost(v,L)
- This is the minimum cost for the subtree rooted at v, if v is labeled L
- solution=$\min_L$ cost(root,L)
- Recurrence for cost(v,L) ?



$$\text{cost}(v,L)=\min_{L1,L2}\left(D(L,L_1)+D(L,L_2)+\text{cost}(u_1,L_1)+\text{cost}(u_2,L_2)\right)$$

- Base case: if v is a leaf
  cost(v,L)= ∞*D(L,given_label(v))

Parsimony: analysis

- We have

$\text{cost}(v,L)=\min_{L1,\,L2} D(L,L_1)+D(L,L_2)+\text{cost}(u_1,L_1)+\ \text{cost}(u_2,L_2)$

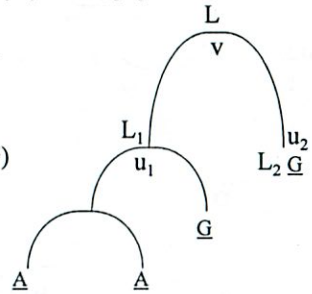
- Equivalently

$$\text{cost}(v,L) = \min_{L1}D(L,L_1)+\text{cost}(u_1,L_1) \ + \\ \min_{L2}D(L,L_2)+\text{cost}(u_2,L_2)$$

- Running time?

$$O(n\ k) * O(k) = O(nk^2)$$

where k is the alphabet size



<u>6.006</u>
<u>Recitation</u>

5/2

(Skipped)

Missed: More DP examples
Handing back test

6.006

(Chalkboard lecture)

5/3

Computational # Theory

L will be on Final

L will distribute resources

Somewhat difficult

L no pictures

and no real life relevancy

Very abstract

Z_m = mod m

(this again!?)

a, b are rel prime when $\gcd(a,b) = 1$

Z_n^* = Multiplictive group of integers $< n$

and rel prime w/ n

$1 \leq x \leq n$

So if $Z_n = 15$, $Z_n^* = 1, 2, 4, 7, 8, 11, 13, 14$

②

What is a group?

- closed
- identity
- inverse

$\forall a \ \exists \ a^{-1}$ s.t. $aa^{-1} = 1 \bmod m$

$\equiv 1 \pmod{m}$

$$n = \underbrace{1\,0\,1\,0\,1\,1 \ \ldots \ 1}_{k \text{ bits}}$$

2^k bits is normal

2^{1000} is not even galactic

- does not exisit

③

Euclid

$a \in \mathbb{Z}_n^*$

$\gcd(a, m) = 1$

Extended Euclid

find x, y

$ax + yn = 1$

↖ smallest int combo of a b

$\gcd(a, b) =$ smallest pos int of $xa + by$

So we have alg that finds x, y

↳ in polynomial time

④

Done since

$$ax = 1 - ym$$
$$ax \equiv 1 \pmod{m}$$

$\mathbb{Z}_q^*$ q is prime

$$\phi(n) = |\mathbb{Z}_n^*|$$

↑ cardonality

↑ Eulers' totnant function

$$\varphi(p_1 \cdot p_2) = (p_1 - 1)(p_2 - 1)$$

Problem

input: a, b, c ← but integers > 0

output: $a^b \bmod c$

↑

$$\underbrace{a \cdot a \cdot a \cdots a}_{b \text{ times}}$$

⑤

Remember bit shift to multiply

But slow → b multiplications
and way too large

Can take mod c all the time
at each multiplication step
not just at the end

But still too many multiplications

Assume "b" is $\underbrace{100\ldots0}_{k}$

$a^{\text{"}b\text{"}} \bmod c$

$$a^2 \to a^{2^2} \to a^{2^3}$$

$$a \to a^2 \to a^4 \to a^6 \pmod{c}$$

⑥

$$b = \overbrace{1\,0\,1\;0\,0\,1\;\dots\;0}^{k}$$

$$\begin{tikzcd}
= 2^k + \arrow[rrrrrrd, bend right] & 2^{k-2} + \arrow[rrrrd, bend right] & 2^{k-5} \arrow[rrd, bend right] & \cdots & & & \\
a \arrow[r] & a^2 \arrow[r] & a^{2^2} \arrow[r] & \cdots \arrow[r] & a^{2^{k-2}} & a^{2^{k-1}} & a^{2^k}
\end{tikzcd}$$

So multiply the elements that have been previously computed

$= a^b \pmod{c} < 2k$ $\quad k =$ bit mod multiplications

(I don't like his handwriting !)

<u>Trick</u> repeated squaring

⑦

Def

$a \cdot \mathbb{Z}^*_p$ is a square mod p

if $\exists x \in \mathbb{Z}_n^*$ $a \equiv x^2$ [mod p)

Problem

given a, p guarenteed

is "a" a square mod p?

You could brute force it

but exponential time → too many ints to check

instead binary search

but modular aramithic kills binary search

$\mathbb{Z}_p^*$ is cyclic

when there exists a generator

such that $g: g^1, g^2, g^3 \ldots g^{p-1}$

all distinct

⑧

Prove by example

Z_{11}^* need a good guess for generator

$?_{11}$ try 2

$2^1 \pmod{11} = 2$

2^2 '' $= 4$

$2^3 = 8$

$2^4 = 5$

$2^5 = 10$

$2^6 = 9$

2^7 7

2^8 3

2^9 6

2^{10} 1

There is no repetition, each # appears exactly once

But any el to power of size group is I

Proof by example QED

Can you tell if a is square of p (mod n)?

g^{2i} is a square?

g^i is in the group

so $(g^i)^2$

$$\left(g^{\frac{p-1}{2}}\right)^2 = g^{p-1} = 1$$

$$g^{2i}$$

$$g^i \qquad \left(g^{i+\frac{p-1}{2}}\right)^2 = g^{2i}$$

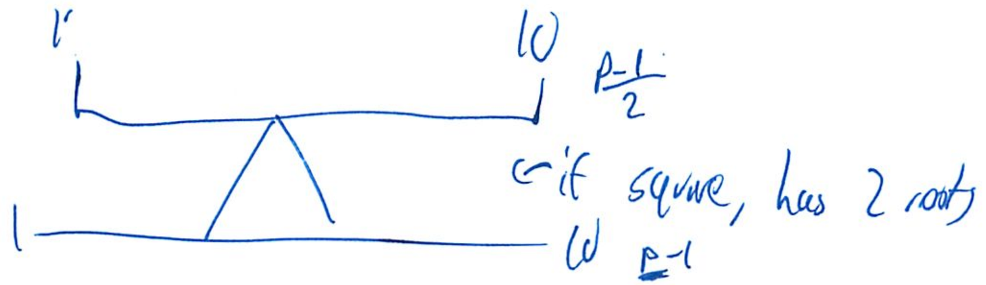
(10)

Any quadratic (missed) has exactly 2 roots

g^{2i+1} ↑ $g^{\text{odd \#}}$ could be a square

↑ is a square

Think about writing the #s



Can't be the root of 2 diff guys

- no room for anything else

Does this help us? Decide if a is a square of p?

(missed

↳ Next week

5/3

Lectures 22 and 23

Algorithmic Number Theory

(Silvio Micali, Jeff Wu)

1 Review: Number Theory and Notation

Here we review some basic concepts in number theory.

- $\mathbb{Z}_n$: the additive group of integers modulo n.
- Two integers a, b are relatively prime if $\gcd(a, b) = 1$.
- $\mathbb{Z}_n^*$: the multiplicative group of integers less than n and relatively prime to n. Every element $a \in \mathbb{Z}_n^*$ has an inverse that is easy to compute. (Because, since $\gcd(a, n) = 1$, we know that there exist integers x and y such that $ax+ny = 1$, so that $ax = 1 \mod n$; thus, $x \equiv a^{-1} \mod n$. We can find x using the Extended Euclidean Algorithm.)
- $\mathbb{Z}_p^*$: a special case of $\mathbb{Z}_n^*$ where $n = p$ is a prime. $\mathbb{Z}_p^*$ is a cyclic group of order $p - 1$ (See Dana Angluin's notes for the proof.) This means that there exists an element $g \in \mathbb{Z}_p^*$ such that $\mathbb{Z}_p^* = \{g, g^2, \ldots, g^{p-1}\}$. For instance, 2 is a generator of $\mathbb{Z}_{11}^*$. We have:

$$\mathbb{Z}_{11}^* = \{2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 5,\ 2^5 = 10,\ 2^6 = 9,\ 2^7 = 7,\ 2^8 = 3,\ 2^9 = 6,\ 2^{10} = 1\}$$

- Fermat's Little Theorem states that $a^{p-1} \equiv 1 \mod p$ for all $a \in \mathbb{Z}_p^*$. (For example, note that $2^{10} = 1 \mod 11$ in $\mathbb{Z}_{11}^*$ above is a consequence of Fermat's Little theorem).
- $\phi(n)$: The Euler totient function is defined by $\phi(n) := |\mathbb{Z}_n^*|$. Equivalently, $\phi(n)$ is the number of integers between 1 and n that are relatively prime to n.
- When p is prime, all integers between 1 and $p-1$ are relatively prime to p. Thus, $\phi(p) = p-1$.
- If p^k is a prime power, then all integers between 1 and p^k are relatively prime to p^k except $p, 2p, 3p, \ldots, p^{k-1}p$. Thus $\phi(p^k) = p^k - p^{k-1}$.

2 Modular exponentiation

Problem: "Given a, x, and n, efficiently compute $a^x \mod n$."

An obvious (naive) approach is to repeatedly multiply by a, and then take the remainder modulo n:

```
1. Set $y = 1$
2. Repeat $x$ times: $y = y \cdot a$
3. Return $y \bmod n$
```

This number y gets extremely large. To avoid this, we can simply use properties of modular arithmetic:

```
1. Set $y = 1$
2. Repeat $x$ times: $y = (y \cdot a) \bmod n$
3. Return $y$
```

However, this still requires x multiplications. Can we do better?

The answer is (of course) yes! We will use a well-known trick called repeated squaring. The important observation is that squaring a number doubles the exponent. That is, $(a^k)^2 = a^{2k}$. This means that $(a^{(2^i)})^2 = a^{2 \cdot 2^i} = a^{(2^{i+1})}$. So by repeatedly squaring, we can obtain $a^{(2^i)}$ in only i multiplications. Thus if x is a power of 2, we can obtain a^x in only $\log(x)$ multiplications.

But if x is not a power of two, we can write it as a sum of powers of 2 in the following manner: $x = \sum_{i=1}^{k} b_i 2^i$ where $b_k b_{k-1} \ldots b_0$ is the binary representation of x. Here, $k = \lfloor \log_2(x) \rfloor$. This means

$$a^x = a^{\sum_{i=1}^{k} b_i 2^i} = \prod_{i=1}^{k} a^{b_i 2^i} = \prod_{i=1}^{k} \left(a^{(2^i)}\right)^{b_i}.$$

This is simply the product of the $a^{(2^i)}$ for i where $b_i = 1$. All of this analysis holds modulo n, as well.

So we have the following algorithm, which uses $O(k) = O(\log x)$ multiplications:

```
1. First make a matrix so that $A[i] = a^{(2^i)}$.
    i. Set $A[0] = a$.
    ii. For $i = 1, \ldots, k$: $A[i] = (A[i-1])^2 \bmod n$
2. Obtain the binary representation $b_k b_{k-1} \ldots b_0$ of $x$.
3. Let $y = 1$
4. For $i = 1, \ldots, k$: If $b_i = 1$, set $y = (A[i] \cdot y) \bmod n$
5. Return $y$
```

3 Quadratic residues modulo p

We can define a quadratic residue modulo p as follows:

Definition 1 *We say that a is a* quadratic residue *(or simply* square*) modulo p if there exists $x \in \mathbb{Z}_p^*$ such that $a \equiv x^2 \bmod p$.*

Note that $\mathbb{Z}_p^*$ has a generator g. Thus, we can write any $x \in \mathbb{Z}_p^*$ as $x = g^k$. Not only that, but the set $\{g^1, \ldots, g^{p-1}\}$ is precisely $\{1, \ldots, p-1\}$, the set of elements of $\mathbb{Z}_p^*$.

3.1 Deciding if a number is a quadratic residue modulo p

Problem: "Given (a, p), decide if a is a quadratic residue modulo p."

Let's first suppose we have a generator g. Which of the elements of $\mathbb{Z}_p^*$ are quadratic residues? There is a very simple answer. We'll assume $p > 2$, so it is odd.

We know that the set of elements is $\{g^1, \ldots, g^{p-1}\}$. Thus the set of squares is simply the elements of the sequence $g^2, g^4, \ldots, g^{2p-2}$. However, we are counting elements twice. By Fermat's little theorem, $g^{p-1} \equiv 1 \bmod p$. So the sequence was equivalent to $g^2, g^4, \ldots, g^{p-3}, g^0, g^2, \ldots, g^{p-3}, g^0$. From this, we see that the set of squares is simply $\{g^2, g^4, \ldots, g^{p-3}, g^{p-1} = g^0\}$.

So we have shown that the quadratic residues modulo p are precisely the even powers of g. That is, a is a square modulo p if and only if $a = g^{2k}$ for some k.

Note that this also shows that every quadratic residue $a = x^2 \in \mathbb{Z}_p^*$ has two square roots (which are $+x$ and $-x$). And precisely half the elements of $\mathbb{Z}_p^*$ are quadratic residues.

This gives an immediate approach to the problem posed: Compute $\log_g(a) \bmod p$ and check if the result is even. Unfortunately, we don't know of any algorithms for efficiently computing discrete logarithms. (In fact, we believe that doing so is hard!)

What can we do when the main road is blocked? We find another road! In this case, we can use *Euler's Criterion.*

Proposition 2 (Euler's criterion) *The element a is a square modulo p if and only if $a^{\frac{p-1}{2}} \equiv 1 \bmod p$. Moreover, a is not a square modulo p if and only if $a^{\frac{p-1}{2}} \equiv -1 \bmod p$.*

Proof: If a is a square modulo p, then $a \equiv x^2 \bmod p$ for some $x \in \mathbb{Z}_p^*$. Let g be a generator of $\mathbb{Z}_p^*$ and write $x = g^k$ for some $k \in [p-1]$. Then $a = g^{2k}$, and we can write:

$$a^{\frac{p-1}{2}} \equiv g^{k(p-1)} \equiv (g^{p-1})^k \equiv 1^k \equiv 1 \bmod p \ .$$

For the other direction, suppose a is *not* a square modulo p. Then write $a = g^{2k+1}$, so that

$$a^{\frac{p-1}{2}} \equiv g^{(2k+1)(\frac{p-1}{2})} \equiv (g^{p-1})^k g^{\frac{p-1}{2}} \equiv 1 \cdot g^{\frac{p-1}{2}} \equiv g^{\frac{p-1}{2}} \bmod p \ .$$

To conclude the proof, we need to show is that $g^{\frac{p-1}{2}} \equiv -1 \bmod p$.

Recall that g is a generator of $\mathbb{Z}_p^*$, which has $p-1$ elements. Hence, $g, g^2, \ldots, g^{\frac{p-1}{2}}, \ldots, g^{p-1}$ must *all* be *distinct* elements of $\mathbb{Z}_p^*$. In particular, $g^{\frac{p-1}{2}} \not\equiv g^{p-1} \equiv 1 \bmod p$. (Where $g^{p-1} \equiv 1 \bmod p$ follows from Fermat's Little Theorem.)

But $g^{\frac{p-1}{2}}$ is a square root of $g^{p-1} \equiv 1 \bmod p$. As we showed above, this square root is not 1. Since $\mathbb{Z}_p$ is a field, there are only two square roots of unity: +1 and -1. Thus, we must have $g^{\frac{p-1}{2}} \equiv -1 \bmod p$.
□

In summary, if p is prime, we can efficiently decide whether a given a is a square modulo p.

3.2 Finding square roots modulo p

Problem: "Given (a, p), find a square root of a modulo p."

We assume that p is a prime such that $p \equiv 3 \bmod 4$ and that a is a square modulo p. Since a is a square modulo p, Euler's Criterion tells us that

$$a^{\frac{p-1}{2}} \equiv 1 \bmod p \ .$$

Writing $p = 3 + 4k$ for some integer k, we obtain

$$\begin{aligned} a^{\frac{2+4k}{2}} &\equiv 1 \bmod p \\ a^{2k+1} &\equiv 1 \bmod p \\ a^{2k+2} &\equiv a \bmod p \\ (a^{k+1})^2 &\equiv a \bmod p \ . \end{aligned}$$

We deduce that $\sqrt{a} \equiv a^{k+1} \bmod p$, provided $p = 3 + 4k$, for some integer k. See Dana Angluin's notes (chapters 20 and 21) for general (randomized polynomial time) procedures to find a square root of a modulo p for any odd prime p.

4 Quadratic residues modulo n (composite)

The definition of a quadratic residue modulo a composite number n remains the same:

Definition 3 *We say that a is a* quadratic residue *(or simply* square*) modulo n if there exists $x \in \mathbb{Z}_p^*$ such that $a \equiv x^2 \bmod n$.*

6.006 Recitation

5/4

Today's Practical # Theory

Given a prime #, find a generator

└ $(1 << 19) - 1$

$(1 << 31) - 1$

We want as little time as possible

$\mathbb{Z}_p^* = \{1, \ldots, p-1\}$

↑ set

$\mathbb{Z}_n^* = \{i \text{ in } \{1, 2, \ldots n-1\}\}$

= subset of #s i rel prime to n

= $\gcd(i, n) = 1$

$\mathbb{Z}_n^*$ is closed under multiplication mod n

└ $\boxed{a, b \in \mathbb{Z}_n^\times \rightarrow \underset{\text{mod } n}{ab} \in \mathbb{Z}_n^*}$

$1 \in \mathbb{Z}_n^*$

②

$\forall a \in Z_n^* \quad \exists a^{-1} \in Z_n^*$

such that $aa^{-1} = 1$

In lecture saw $\exists a, b$ such that

$ax + ny = 1$

$x = a^{-1}$

Facts to know about the multiplicative factors for all n

For primes

- something more specific is true

$\exists$ generator g s.t. $\{g, g^2, g^3, \cdots g^{p-1}\} = Z_p^*$

every # in multiplicative group can be written as a power of the generator

③

$g^{\log_g a} = a$

example Z_7^* is generated by 3

$3 = 3$

$3^2 = 2$

$3^3 = 6$

$3^4 = 4$

$3^5 = 5$

$3^6 = 1$

↑all unique

if it wasn't - the powers would repeat

kinda proof

$g^{p-1} = 1$ For any generator g

If a is in the set $(a \in Z_n^*)$ then

$a = g^i$ for some i

$a^{p-1} = g^{(p-1)i} = (g^{(p-1)})^i = 1$

④

Code to find a generator

```
def find generator (p)
    for g in range(2, p):
        if is_generator(g, p):
            return g
```

↑ sets slow, use a lot of memory

dict slow

So unique if last one to appear - just track last one

Must do the slow exponentiant to check each one

Check that none of the powers are 1

```
def is_generator(g, p):
    ~~power = 1~~
    for ~~i~~ in range(p-1):
        ~~power~~ = (power * g) % p
```

⑤

```
power = g
for i in range(p-2):
    if power == 1:
        return False
    power = (power * g) % p
return true
```

Is $O(n^2)$ since call is_gen n times
which takes n time.

Can't time since it takes python a while
to start up

Other time measurement 10^{-5}

But was lucky since answer was 3

But its not n^2

Its $n \log n$

randomly

Generators mod p are evenly distributed

One every $\log n$ #s is randomly distributed

$$\# \text{ generators} \approx \frac{p}{\log p}$$

$$= \Omega\left(\frac{p}{\log p}\right)$$

Its not provably $n \log n$

~~avg case $n \log n$~~

Can change so $p \log p$

By picking a rand int $[1, p-1]$ to check for g instead of increasing

⑦

$\mathbb{Z}_p^* = \{1, \dots, p-1\}$

Def: If a is a # in $\mathbb{Z}_p^*$ the order of a is the min i s.t. $a^i = 1$

g is gen ↔ ord $g = p-1$

Ord. a $\|$ $p-1$

(divides for all a

$a = g^i$

$\text{ord } a = \frac{p-1}{\gcd(i, p-1)}$

d of p-1 is maximal if $\frac{d}{p-1}$

$\nexists d'$ s.t $d/d'/p-1$ $(d' < p-1)$

(Totally don't get)

⑧

$p=31$

$p-1=30$

maximal divisors

2 not max div $2 | 10 | 30$

10 is max. divisor

for 30 max divisors : $\frac{30}{2}$ $\frac{30}{3}$ $\frac{30}{5}$

So factor 30

Can't do anything lin time → or alg will be lin time

Fast than lin → only check up to $\sqrt{n}$

Since only d or $\frac{n}{d} < \sqrt{n}$

⑨

√ factor (but not all the way - only the prime divisors)

```
def list_primes(n, result = []):
    last_d = 1
    for d in range(2, n):
        If n% d = 0          ← smallest divisor is a prime
            result.append(d)
            while n%d ==0:   ← as many times as possible
                n/=d

            last_d = d
           ~~breaks~~ break
        elif d > sqrt(n)
            ~~break~~
            result.append(n)
    return result
```

~~Does not work~~

10x faster than it used to be

Have not done fast exponentiation

6.006

5/8

Board

$\mathbb{Z}_n$

$c = \gcd(a, d)$ = min pos int s.t. $\exists x, y$ ints

$ax + by = c$

$\mathbb{Z}_n^*$

$a^b \bmod c$ easy to compute

$a \in \mathbb{Z}_n^* \rightarrow a^{-1} \pmod n$ easy to

$\mathbb{Z}_n^*$ ← prime a's cyclic

eg $\mathbb{Z}_{11}^* = \{2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 5$

$2^5 = 10,\ 2^6 = 9,\ 2^7 = 7,\ 2^8 = 3.$

$2^9 = 6\ \ 2^{10} = 1$

"a" Squares mod p iff $a \equiv g^{2i}$

– proved last time

other side

"a" non-square mod p iff $a \equiv g^{2i+1}$

②

(I don't get this at all..)

$$x^2 \equiv a$$

$$x\, g^{\frac{t-1}{2}} \equiv a$$

Problem: Given a,p ~~a square 2 mod p~~

↳ $a \in \mathbb{Z}_p^*$

Is "a" square mod p?

Need to understand problem better

↓

Helps us make a better alg

So how do we decide square or not?

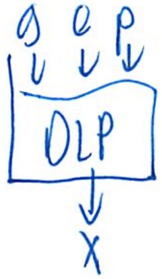
1. Find generator
2. Find discrete log that gives exponent
3. Even or odd?

③

But bad news:

- How to efficently find a generator

DLP give $g, p, e \in \mathbb{Z}_p^*$ find $x \in [1, p-1)$ and $g^* = q$



But very hard to actually solve

(if you do auto A+ and PhD

Euler

prolhific in math

and having kids ... 18 of them

$\forall a$

$$a^{\frac{p-1}{2}} \bmod p = \begin{cases} +1 & a \text{ is square} \\ -1 & a \text{ is not square} \end{cases}$$

+ only 2 possiblitites

④

Prove

$\frac{1}{2}$ of theorm: a squared mod p $\Rightarrow e^{\frac{p-1}{2}}(p) = +1$

How to prove things?

Some tricks

- by contridiction (works in 50% of cases)
- rewrite hyp (49%)
  - if you have writers block

p prime $\mathbb{Z}_p^*$ cyclic g

$a = g^{2i}$

i don't know, but is even

$$a = \left(g^{2i}\right)^{\frac{p-1}{2}}$$

$$= g^{i(p-1)}$$

$$= \left(g^{(p-1)}\right)^{i}$$

$= 1^i$

$= 1$

✓

$a \ \overline{sq} \Rightarrow e^{\frac{p-1}{2}} \equiv -1 \mod p$

Can prove by Contridiction

but try something else

$a = \left(g^{2i+1}\right)^{\frac{p-1}{2}}$

$= \left(g^{2i}\right)^{\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}}$

$= 1 \cdot g^{\frac{p-1}{2}}$

$\uparrow = -1$ since $\sqrt{1}$

euler was right

---

if you can really find sol, easy to find

Problem given a sq mod p, find a sqrt mod p

Binary Search does not work mod p

So need new idea

Proof $\frac{1}{2}$ cases

$p \equiv 3 \mod 4$

If we assume this

Need to re-write problem

Prof; Proofs is deterministic

- recycle stuff
- don't think!

⑦

Rewrite "a" Square mod p

$\downarrow$

$$a = \left(g^{2i}\right)^{\frac{p-1}{2}} \equiv 1$$

What can we recycle/rewrite?

$$p = 4\underset{?n}{k} + 3 \rightarrow p-1 = 4k+2 \rightarrow \frac{p-1}{2} = 2k+1$$

$$a = \left(g^{2i}\right)^{2k+1}$$

Now we can think

if want $\sqrt{a}$ what can we have

blah x^2 so x

$$\left(g^{2i}\right)^{2k+1}$$

$$a^{2k+1} \equiv 1$$

$$a \equiv x^2$$

So multiply both sides by a

⑧

$$\equiv a^{k+2} \equiv a$$

$$\equiv \left(a^{(k+1)}\right)^2$$

$$a^{k+1} = \sqrt{a} \mod p$$

Lec006 is about efficiently finding the sol

6.006 Recitation

5/[illegible]

Last time: Given a prime, find a generator

- Pick random #
- Test if generator

But there is a bug

Is $O(n)$

Not $O(\sqrt{n})$ since bug

Its a line of python

~~[illegible]~~

`range( last_d + 1, n+1)`

↑ Constructs entire list

Xrange gives you next # w/o building whole list

But also ran of memory

Since range( )

requested 1 GB

Finding generators is trival once have factors

Crypto

Things we learned so far

p = prime

g ~~ma~~ gen mod p

} we can compute these efficiently

Given a we can compute $g^a \bmod p$

L in polynomial time

Given g^a we <u>can't</u> compute a

L not known how to in polynomial time
Conjectured to be hard

We can't easily find a generator for a given prime
But we can compute a generator + a prime randomly
L Don't need to know how

We can't easily find $\log_g$

example Goal $A \xrightarrow{x} B$

1011 = 11 securly infront of C

A tells B a prime p, gen g_u

A picks a and annonces g^a

⊗ But doesn't work. B doesn't know anything C doesn't know!

B picks b and announges g^b

✓ Basically done. Only need to comm the message

Note b does not know a

a b

A knows a, g^b, $\overset{\text{also}}{g}$, g^a

Can find $g^{(ab)} = (g^b)^a$

④

B knows g^a, b

Can find $g^{ab} = (g^a)^b$

C knows g^a, g^b

↳ can't find g^{ab}

↳ Diffe Hellen assumption

(This seems simpler than last year - whats different?)

Now

$$A \xrightarrow{g^{(ab)} \overset{\text{XOR}}{\oplus} \overset{\text{message}}{x}} B$$

[This is Diffe Hellman key exchange

Runs in polynomial time

No guarentees

↳ proving it would prove $P = NP$

⑤

GM security

high bar

if no poly time algorthm to determine
if x is a square mod pq

And the weak pat of crypto
are
- small primes
- implemented wrong
- Or User error

512 bit is 512 bit prime #
↳ unbreakable
256 prob Gov or Google could break

⑥

HW Review

What info is useful?

Football Robots

Given (a_i, b_i) ...

Want robots (c_i, d_i)

Such that $\exists$ robot j for each ~~[illegible]~~ player i

$c_j \geq a_i$
$d_j \geq b_i$

Min $\sum c_j d_j$

Have to use DP

L sometimes it easier put things on tree in proper order

⑦

1. Elimahate majaried players $O(n \log n)$

2. Sort players by strengh or speed

$a_1 \leq a_2 \leq \dots \leq a_n$

$b_1 \geq b_2 \geq \dots \geq b_n$

If a robot majaries players i and j
it majaizes $i, i+1, \dots, j-1, j$

So only need to majorize consecuitive subsets
(much eaiser than just subsets...

$dp[i]$ = opt cost for robots $i \dashrightarrow n$

$O(n)$ subproblems

$dp[n] = a_n b_n$ ← last guy

$$dp[i] = \min_{i \leq j \leq n} b_i a_j + dp[j+1]$$

(did n't write in this format)

$O[n]$

dp[1] is the answer

$O(n^2)$

Something really tricky in $O(\log n)$. Kishore doesn't know off the top of his head

Often Greedy doesn't work on DP

DP w/o sorting won't work

6.006 L24

5/10

NP-Completeness

One of the biggest exports of CS

Many tractable problems

polynomial time

- almost all the algos we saw before

So mild dependence on input

↳ not astronomical amt of data

Some intractable problems

Can't solve in polynomial time

eg Count from 1 to n for combo lock

Since O(log n) digits input

Output is exponential

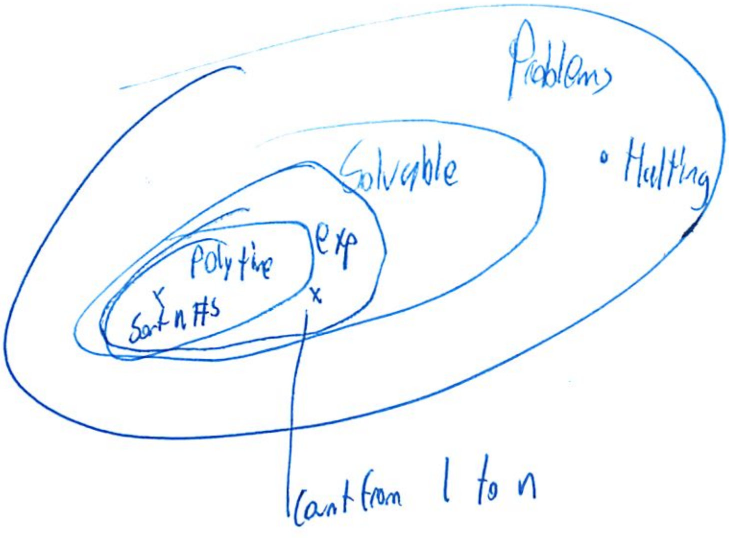
Unsolvable problems

is the program is syntatically correct?

Will the program terminate

So for some programs easy to tell

But in general very hard to tell



like knapsack problem

Size S

Collection of n items

input size $\log S + \sum_i (\log s_i + \log v_i)$

Goal i fit maximum value into knapsack

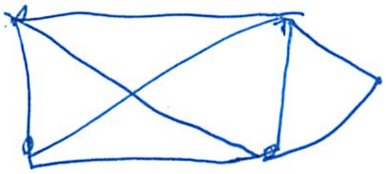
Gave a DP $O(ns)$

Is there a algo that runs in poly time?

③

Traveling Salesperson Problem

Input undirected weighted eges



Output : Shortest tour that visits each vertex once

Can try each route $n!$

Best is $O(n^2 2^n)$

Is there a poly time algorithm?

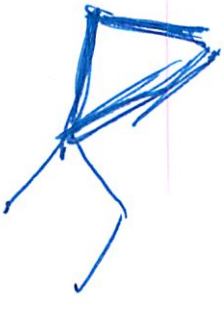
Clique problem

Input: undirected graph $G = (V,E)$

Output: Largest subset C of V so that every pair of vertices

$O(1.1888^n)$

Is there a poly time alg?



How do we prove problems are hard?

taught many polynomial designs
But can we prove not working?

We don't know!

Proving a Negative → the Science way

How to prove no ~~[illegible]~~ perpetual motion machine
L try to build it?
~~[illegible]~~ but can't prove you can't do it
You tried + failed
preponderance of people who tried
Are they idiots?

Stronger proof: That "laws" of physics preclude its evidence
(which were similarly proved through obs.

Lots of smart people have tested these laws

If PMM were possible, it would violate this wall

So a very large # of smart people must be wrong

Hardness "Proof"

Through reduction

Problem Q - want to prove

Problem P - that CS don't think is possible

Prove that if had sol to Q, could solve P

So if no poly time sol for P, there is no poly time alg for Q

Partition

Given n #s that sum to S

Is there a subset of #s Σ to $S/2$

If had poly time alg for napsack

Could use it to solve partition

If there is a partition, can fill knapsack
and get $S/2$

Otherwise best is $< S/2$

(details on slides)

If was poly time for napsack, would have one for partition

~~If~~ Since we belive no poly time alg for Partition
↳ none could exist for napsack

Decision problems

Yes or No answer

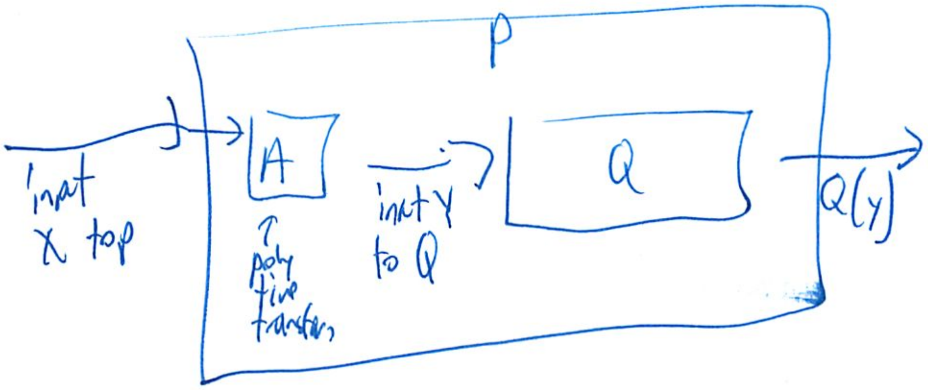
- Given an array is it sorted in ↑ order
- Given a list of #s are there duplicates
- etc

⑦

Reduction

takes in X and transforms to Q so that

$P(x) = \text{Yes}$ iff $Q(y) = \text{Yes}$



transform must be poly time

Suppose X has size n

Then $Y = A(x)$ has size $\text{poly}(n)$

↳ because A is poly-time

So overall runtime is $\text{poly}(|A(x)|) = \text{poly}(\text{poly}(n))$

$= \text{poly}(n)$

Can modularize

If all modules poly → its poly

Cosequence

If is poly time for Q the redction $P \to Q$ gives 1 for P

If no poly time for P, we can conclude none for Q

Reduce $P \to Q$ "Q is at least as hard as P"

Order is important!
 └ don't screw it up

Find a whole family of hard problems
 that can be reduced to P

NP

Problem belongs to NP

- poly-sized sol
  - └ rel to input size
- can be verited in poly-time

non deterministic (solve) in polynomial time (NP)

We can guess the path, then check if its lenght is $> L$

Too many guesses to simulate deteministically

Problems in NP vary by difficulty

<u>NP-hard</u> - if every problem in NP can be reduced to it

<u>NP-complete</u> - NP hard and belongs to class

"The hardest proble in NP"

not clear if one exists

Cook 73 showed one

Lots of problems have been shown ~~to~~ to be like this

6.006- *Introduction to Algorithms*

Lecture 24

NP-completeness

(The Dismal Computer Science)

Prof. Constantinos Daskalakis

Tractable Problems

- We have seen many problems that can be solved in **polynomial time.**
- e.g. finding the shortest path in a graph
- e.g.2 sorting *n* numbers
- e.g.3 finding the exit in a maze
- e.g.4 finding the square root of an integer
- etc
- These problems are **tractable.**
- Polynomial dependence in the input ≈ mild dependence on the input

true for reasonable input size [but not for "Internet-size", or "galaxy-size" inputs]

$^{\text{In}}$tractable Problems

- ~~We have seen many~~ problems that can$^{\text{not}}$ be solved in **polynomial time.**
- e.g. ?
- Suggestion: Count from 1 to a given *n*
- *?!?!?!???!?!?*
- OK, what if I phrase the problem as follows:

- Observation: to represent *n* I just need to provide O(log *n*) digits.

Unsolvable Problems?

- Are there computational problems that cannot be solved?
- let's try PYTHON: is program P syntactically correct?
- PYTHON can be solved; in particular the python compiler solves it.
- HALTING: Does python program P on input I terminate ?
- e.g.
  - ➢ while True: continue
    - does not terminate for any input
  - ➢ print "Hello World!"
    - terminates for any input
- Suppose there exists an Algorithm A solving HALTING, i.e.
- A(P,I)=1 if program P on input I terminates
- A(P,I)=0, if program P on input I runs forever

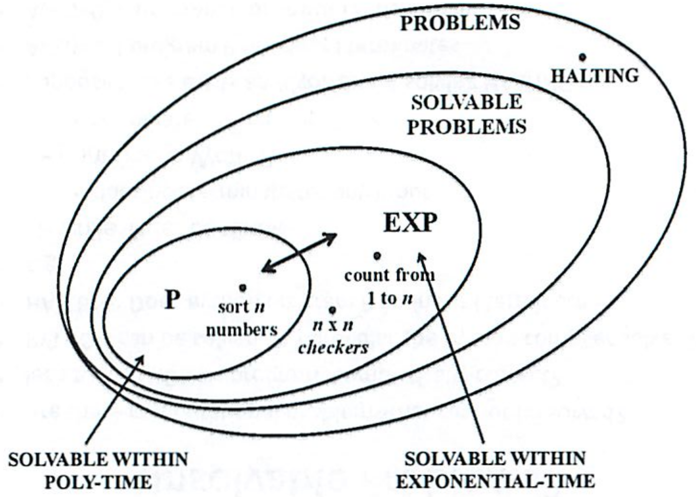
5/10

HALTING PROBLEM

- A(P,I)=1 if program P on input I terminates
- A(P,I)=0, if program P on input I runs forever

- COSTAS
  - ➢Input: Program P
  - ➢if A(P,P)=1, then enter infinite loop
  - ➢else if A(P,P)=0, then stop

- Question: Does COSTAS(COSTAS) terminate?
  - ➢suppose it does, then …
  - ➢suppose it does not, then…
- Contradiction! So there is no algorithm that solves HALTING. More in 6.045



PROBLEMS

HALTING

SOLVABLE PROBLEMS

EXP

count from 1 to *n*

P

sort *n* numbers

n x *n* checkers

SOLVABLE WITHIN POLY-TIME

SOLVABLE WITHIN EXPONENTIAL-TIME

Menu

- Classification of problems: P, EXP, unsolvable
- **Problems for which we can't decide yet**
- the class NP
- the P vs NP question

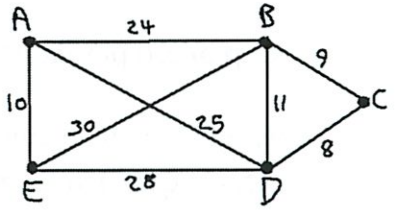
Knapsack Problem

- **Input:**
  - ➢Knapsack of (integer) size *S*
  - ➢Collection of *n* items
    - Item *i* has (integer) size s_i and (integer) value v_i
  - ➢Input size: $\log S + \Sigma_i (\log s_i + \log v_i)$
- **Goal:** Fit maximum value into knapsack
  - ➢i.e., choose subset of items with $\Sigma_i s_i < S$ maximizing $\Sigma_i v_i$
- Gave a DP algorithm running in time O(*n S*).
- Is there an algorithm that runs in time

  $\text{poly}(\log S + \Sigma_i (\log s_i + \log v_i))$?

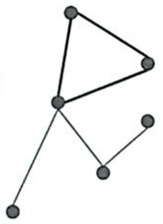
Traveling Salesperson Problem (TSP)

- **Input:** Undirected graph with lengths on edges.
- **Output:** Shortest tour that visits each vertex exactly once.
- Best known algorithm: $O(n^2 2^n)$ time.
- Is there a poly-time one?



The CLIQUE problem

- **Input:** Undirected graph G=(V,E)
- **Output:** Largest subset C of V such that every pair of vertices in C has an edge between them.
- Best known algorithm: $O(1.1888^n)$ time
- Is there a poly-time one?



Menu

- Classification of problems: P, EXP, unsolvable
- Problems for which we can't decide yet
- **Proving hardness of problems**
- the class NP
- the P vs NP question

What problems are in P?

- We've taught you ways to design polynomial-time algorithms for many problems
- So when faced with a new problem, you'll try applying them in various ways
- What if you don't succeed?
- When can you give up?
- Are there ways for you to know not to waste time trying in the first place?
- Can you prove there's no poly-time algorithm?
- In the problem of counting from 1 through n, the proof is easy as the output itself is exponential in the input.
- But what if the output is polynomial in the input?

Proving a Negative

- How prove there is no polynomial time algorithm for a problem whose solution is polynomial in length?
  - ➢i.e show that there is no algorithm of time O(n), OR O(n^2), OR O(n^3), …

Short Answer: We don't know how to prove such statements.

Don't even have general technique to show that there is no O(n) algorithm

"Proving" a Negative: the Science Way

- How prove no perpetual motion machine?
  - ➢We can prove one exists by building it.
  - ➢We can't prove none exists.
  - ➢Especially if only "evidence" is that we tried and failed.
- Many have tried to build one and failed
  - ➢A preponderance of evidence that is impossible
- But maybe only idiots tried to build PMMs
  - ➢Maybe possible if someone from MIT tries?

A Stronger "Proof"

- Prove that the "laws of physics" preclude its existence.
- Lots of smart people have tested these laws.
  - ➢Gives a real preponderance of evidence the laws are correct.
- If a PMM was possible, it would prove those laws false.
- So unless a very large number of smart people are all wrong, there is no perpetual motion machine.

Menu

- Classification of problems: P, EXP, unsolvable
- Problems for which we can't decide yet
- Proving hardness of problems
  - ➢**hardness via algorithms**
- the class NP
- the P vs NP question

Algorithmic Hardness "Proof"

- Suppose you want evidence that there is no poly-time algorithm for your problem ***Q.***
- Take a problem ***P*** where many scientists have tried and failed to find a poly-time algorithm.
- Prove that if you have a poly-time algorithm for ***Q***, you can use it to build a poly-time algorithm for ***P.***
- Contrapositive: if there is no poly-time algorithm for ***P***, there is no poly-time algorithm for ***Q.***
- All the evidence from those scientists that ***P*** is hard becomes evidence that ***Q*** is hard.

Example: Knapsack

- A "believed hard" problem is **Partition**:
  - ➢Given a set of *n* numbers summing to *S*.
  - ➢Is there a subset of numbers summing to *S*/2?
- We can use this to show **Knapsack** is hard
  - ➢Suppose we have an algorithm A for **Knapsack.**
  - ➢Want to use it to solve **Partition**. How?
  - ➢Given an input $\{s_1,\ldots,s_n\}$ to **Partition.**
  - ➢Consider **Knapsack** problem where item *i* has size s_i and value s_i, and knapsack size is *S*/2.
  - ➢If there is a partition, you can fill the knapsack and get value S/2.
  - ➢Otherwise, best achievable value is < S/2.

Example: Knapsack

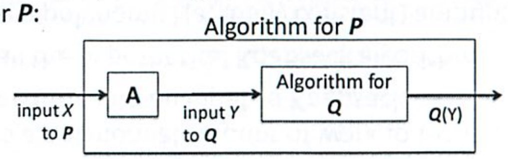
- We now have an algorithm for **Partition**:
  - ➢Do a polynomial amount of work to turn the input to **Partition** into an input to **Knapsack**.
  - ➢Call the hypothetical **Knapsack** algorithm.
  - ➢Do a polynomial (actually constant) amount of work to turn the **Knapsack** answer into a **Partition** answer.
    - If Knapsack result is S/2, return YES, else return NO
- If there is a polynomial-time algorithm for **Knapsack**, get one for **Partition**.
- Since we believe no polynomial-time algorithm for **Partition**, conclude none exists for **Knapsack**.

Formalizing our ideas

- We will concentrate on **Decision Problems**
  - ➢These are problems that have a YES or NO answer
- Examples:
  - ➢Given an array, is it sorted in increasing order?
  - ➢Given a list of numbers, are there any duplicates?
  - ➢Given a Knapsack problem, is there a solution (that fits) with total value at least V?
  - ➢Given a graph with positive edge lengths, is there an s-t path of length less than L?
  - ➢Given a graph with edge lengths, is there an s-t path of length greater than L?
  - ➢Given a graph with edge lengths, is there a traveling salesman tour of cost at most C?
  - ➢Given a graph, does it have a clique of size K?

Reductions

- Define a **reduction** from problem ***P*** to problem ***Q***
  - ➢A polynomial-time algorithm A that takes an input X to problem ***P*** and transforms it into an input Y to problem ***Q*** such that:
    P(X)=YES if and only if ***Q***(Y)=YES.
- If there is a poly-time algorithm for ***Q***, the reduction gives one for ***P***:



- Suppose X has size n.
- Then Y=A(X) has size poly(n) (because A is poly-time)
- So overall runtime is poly(|A(X)|) = poly(poly(n)) = poly(n).

Consequence

- If there is a poly-time algorithm for ***Q***, the reduction from ***P*** to ***Q*** gives one for ***P***.
- Contrapositive: If we believe there is no poly-time algorithm for ***P***, we can conclude there is none for ***Q***.
- Reduce ***P*** to ***Q*** → "***Q*** is at least as hard as ***P***"
- Order is important!
  - ➢On the final, at least one person always reduces Q to P and concludes Q is harder than P.

Summary so far

- If problem ***P*** is reduced to problem ***Q***,..
- this shows that ***Q*** is at least as hard as ***P***.
- If people think ***P*** is hard, they'll believe ***Q*** is hard.
- Problem: what is a plausibly hard ***P***?
  - ➢Is there a problem that everyone agrees is hard despite not being able to prove it?
- Solution: Find a whole family of hard problems that can be simultaneously reduced to Q.

Menu

- Classification of problems: P, EXP, unsolvable
- Problems for which we can't decide yet
- Proving hardness of problems
  - ➢hardness via reductions
- **the class NP**
- the P vs NP question

NP

- A decision problem belongs to the class **NP** if:
  - ➢ it always has a poly-size solution;
  - ➢ whether a proposed poly-size solution is truly a solution can be checked in polynomial-time.
- We say that such problem can be solved in **nondeterministic polynomial time (NP)**.
- In the following sense: We can (non-deterministically) **guess** the solution, then in polynomial-time **check** whether our guess is truly a solution.
- E.g., LONG PATH: Is there an s-t path of length greater than L?
- We can guess a path, then check if its length is larger than L.
- Obstacle: too many possible guesses to simulate deterministically.

The hardest problems in NP

- A problem ***Q*** is **NP-hard** if every problem in NP can be reduced to it
  - ➢ i.e., a deterministic polynomial-time algorithm for ***Q*** can be turned into a polynomial-time algorithm for any other NP problem
  - ➢ "At least as hard as any NP problem"
- A problem is **NP-complete** if it is in NP and is NP-hard
  - ➢ "The hardest problem in NP"
- Cook '73: There is an NP-complete problem!
- Such problem is a good starting point for showing other problems are hard, as it carries with it the hardness of all problems in NP.

Menu

- Classification of problems: P, EXP, unsolvable
- Problems for which we can't decide yet
- Proving hardness of problems
  - ➢ hardness via reductions
- the class NP
- **the P vs NP question**

P vs NP

- Many problems have been shown NP-complete
  - ➢ Clique, Independent Set, TSP, Graph Coloring, 4-way matching, Vertex Cover, Hamiltonian Path, Longest path, Multiprocessor Scheduling, Max-Cut, Constraint Satisfaction, Quadratic Programming, Integer Linear Programming, Disjoint Paths, Subset Sum...
  - ➢ So not just one, but many "hardest problems in NP"
- In 50+ years, scientists haven't found a polynomial-time algorithm for any of them.
- (A poly-time algorithm for one of them, implies a poly-time algorithm for all, as all are reducible to each other)
- The "P vs NP" problem, i.e. answering whether or not there is a poly-time algorithm for any of these problems, is one of the seven millennium prize problems.
- The Clay Mathematics Institute offers $1million for its answer.

Is P=NP?

6.006 Recitation

5/11

Goal: Find c_n the # of correct parenthes
w/ n "(" and n ")"

↑ 5 Parens

Goal 2: The allowed edits to a string are

- deleting a char (cost 1)
- inserting a char (cost 1)
- changing one car to another cost (x)

CATC

↓ Costs 1+x

CTT

minimal cost seq of edits from one to other

$S \to A$

both given

②

Not a seq $(())\overset{\times}{)}\overset{\times}{(}$

Find the # of valid

C_1 ( ) $=1$

C_2 ( )( ) or (( )) $=2$

C_3 ( )( )( ) or ((( ))) or (( ))( ) or ( )(( )) $=5$

or (( )( ))

So is this the l → r count # of open parens

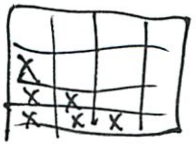
make sure never goes < 0

Or just a generic DP way

~~mi~~ (+ rest ( )

Again returning # of open

Or like the PS6 problem



③

Now 2. This is very similar to the Disney form qv
But different

What was the subproblem?

X = One of the inputs
So alg works for whatever x is
$x > 0$

'longest match substring?'
Or n One by other matching?

Solutions

1. O(n) sol

$C_n = 3C_{n-1} - 1$

Ohh you are not given a string
Want to answer C_n
TA: dont think it works

④

$S_n = (S_n - 1)$

or $()\, S_{n-1}$

or $S_{n-1}\, ()$

TAiWo (())(())

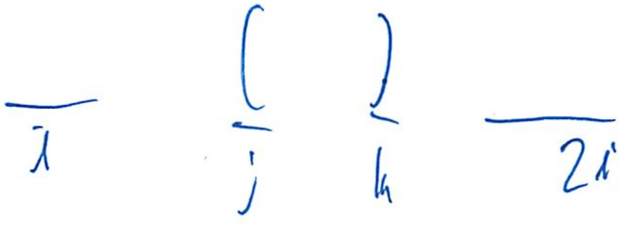
---

1. $O(n^3)$ sol

DP

Subprobmens C_i $i \le n$

$C_i = \quad 0 \le j \le k \le 2i$



Sum all possible j, k

← all 3 sections must be valid

③

$$C_i = \sum C_{j/2} \, C_{\frac{k-j}{2}} \, C_{i-\frac{k}{2}}$$

$$0 \le j < k \le 2i$$

Subproblems $O(i^2) \rightarrow$ round up $O(n^2)$

n sub probms so $O(n^3)$

Is it correct?

Think so

But must know something else

Does it count everything exactly one?

No!

() () ()
  j k

() () ()
j k

both count! duplicate

Need a way to combine something else ...

Having trouble writing a recurrance

⑥

$$C_i = \frac{\displaystyle\sum_{0\le j\le k\le 2i} C_{j/2}\, C_{\frac{k-j}{2}}\, C_{i-k/2}}{i}$$

(~~[illegible]~~ TA ; correct

each one gets counted once

( )

j k

$C_0 = 1$

$C_1 = 1$

$C_2 = 3$ but how do we find?

1	()_ _	_()_
0	(_)_	_(_)
1	(_ _)	_ _()

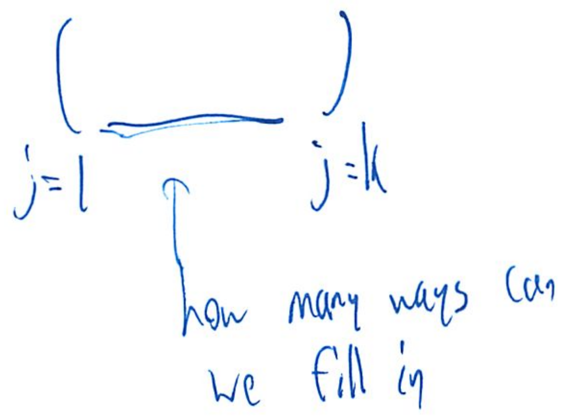
Filling in the remainder w/ valid parentheses

⑦

but don't count ___(())__

So it actually does not work

$C_i =$



j = # of left parens b/w the 2 parenthesis

So

$(\underline{\ j\ })\ \underline{\underline{i-j-1}}$

$$C_i = \sum_{0 \le j \le i-1} C_j \, C_{i-j-1}$$

But how are we sure it does not count?

⑧

$$(\underbrace{()()}_{j=2})\,()$$

$$C_4 = C_2 \cdot C_1$$

$$C_0 = 1$$

Note: 1st parenthese fixed at 1

Runtime

n subproblems

each $O(n)$

So $O(n^2)$

Closed form takes n times to calculate

$$\frac{1}{n+1}\binom{2n}{n}$$

(But the DP is the important one

⑨

TA: Test your recurrences for small cases
Do they work?

This was called The Cataline Problem

---

2. The Levenshein Problem

Useful to find the diff b/w files

But Not the alg used by diff since X

Subproblems?

Search problems very large

↑my original idea

$S =$ length n

$A =$ lenght m

Plausable: goal-directed search would work

Oh double ended BFS :)

⑩

Transform prefixes

dp[i] = min cost

If subproblems are cost, can almost always
find cost of min seq
then can extract the seq later

dp[i] = min cost of edits s[:i] to t[:i]

What is the recurrance?
Like finding max len common sub seq

~~ABCEF~~

A B C F G

A B D F F G

②

Could delete everything in ① not in max subseq ①
and insert everything in ② but not in max subseq ②

Case 1

$dp[i] = s[i] + t[i]$

$dp[i-1]$ ← memoized

Case 2 $s[i] \neq t[i]$

Could also use the substitution

$\min(x + dp[i-1])$

Instead $dp[i,j]$

Case 1 $dp[i,j]$

$s[i] + t[j]$

$dp[i-1, j-1]$

Case 2 $s[i] \neq t[i]$

$\min(1 + dp[i-1, j],$

$1 + dp[i, j-1],$

$x + dp[i-1, j-1])$

⑫

So 1. delete ith char of s, match $s \to i-1$
$t \to j$

2. add t_j to end of $s \to i$, then chars are matches
rest as $s \to i$
$t \to j-1$

3. Substitute $s[i]$ w/ $t[j]$. Match range

__Runtime__ $O(n \cdot m)$

$n \cdot m$ subproblems, each $O(1)$

Can modify slightly to find actual seq

parent $[i, j]$ = last transition to get to that subproblem

parent $[i, j]$ = Deletes s_i
Insert t_j
Swap s_i w/ t_j

So when filled out whole table, need to return an ans

dp[n,m]

Parent[n,m]

← go backwards in Parent table

A B ~~C~~ ~~E~~ F ~~G~~

A B D ~~E~~F

6.006

5/15

Almost last lecture

Today: Something cool

Thur: Something else cool

Recall $\text{sqrt}(a, p) = a^{k+1} \bmod p$ if $p = 3 + 4k$

Fact mod p squares have 2 roots $x, -x$

$\llcorner p > 2$

Same mod $2p$ $\quad p^n \quad 2p^n$

Fact n = not prime

$= p_1^{n_1} p_2^{n_2} \cdots p_k^{n_k}$

some power $\quad p_i > 2$

so distinct primes

How many roots: - prime? 2

prime power 2

②

any square 2^k roots

Why? Chinese remainder theom

$$x \overset{1\text{ to }1}{\leftrightarrow} (x_1, \ldots, x_k)$$

$$(\pm x_1, \ldots, \pm x_k)$$

Primality

Given a # > 2

Is it prime?

L try dividing $\frac{n}{2, 3, 4, \ldots \sqrt{n}}$

not polynomial

1000 bits means 500 bits

n $\sqrt{n}$

Fermet's Test

$x \in Z_n^+$

$x^{n-1} \bmod = 1$

But doesn't work for Carmichael #s

└ those composite #s look like prime

Miller

relies on assumption

└ no one can prove

Solovey + Stressen
Robin

} fast alg w/ a coin (ie has random bit input)

eaiser varient

(missed)

Wird(m): ← don't look up in books

1. If $m=2$ output "prime"
   else if m is even output "composit"
2. If $\exists\, a,b>1$ s.t. $m=a^b$ output "composite"

3. Randomly choose $x \in [1, \ldots, m-1]$

If $\gcd(x, m) \neq 1$, output composite

4. $x \to x^2 \bmod m \to z$

SQRT(z, m) If $y \neq x, -x$ output "composite"

Else output "prime"

(very hard to distinguish n, m)

Running time

1. $O(1)$ if have fast way to end

2. Is it easy to see $m \overset{?}{=} 2^b$

(missed discussion)

easy → polynomial time

3. Polynomial time

4. " "

⑤

$\forall$ prime m $P[\text{weird}(m) = \text{composit}] = 0$

If $z \in \mathbb{Z}_n^*$

if compute $\sqrt{\ }$

its either x or $-x$

So no prob of composite

$\forall$ composite n ~~[illegible]~~ $\rightarrow P_a[\text{weird}(n) = \text{composite}] \overset{?}{=} \frac{1}{2}$

(missed discussion)

(Oh he meant n, not m - his handwriting sucks)

(something about conditional prob)

Some uncertainty

tells Comp → is comp

tells prime → can be either prime or composite

6.006 Recitation

5/16

Last one, so hardest problem

Circular Hashing

Lexicographic order

If A, B are arrays of lenght n, then
A < B in lex. order if A[i] < B[i]
at first location i at which the
arrays differ

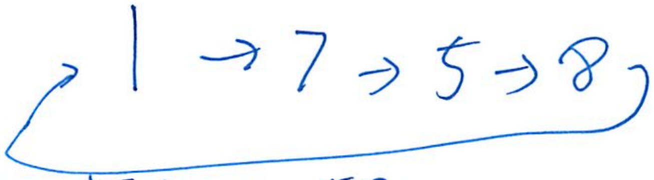
			$i=2$		
A	1	7	5	8	A < B
B	1	7	6	3	

Circular Array

The circular rotation of A are the arrays

$A[i]\ A[i+1] \ldots A[n]\ A[1]\ A[2] \ldots A[i-1]$

example



4 circular rotations

1 7 5 8
7 5 8 1
5 8 1 7
8 1 7 5

n possible rotations for lengh n

Define canotical rotation of an array A to be the circular rotation of A that is smallest in lexigraphic order

1. Why does this help hash circular arrays?
2. How do you compute Canontical rotation?

③

1. Normally have to find each + compare

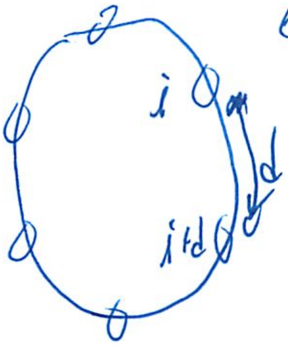
$< n^2$

If find canotical - its unique even for others

So to compare 2 → just compute + compare their ~~can~~ canonitcal arrays

2. Nieve is $O(n^2)$

Can we do better? Yes



← any point can be start

← smallest diff

any pointers that get knocked out (missed)

Not true that if advance d steps you are good

Unless you kill that pointer

㊾

"Rolling hash"

Circular substring$_{ij}$ = A[i] A[i+1] ... A[i+j]
since circular, indices loop around

Want to compare S_{1n} O(n)
S_{2n} O(1) ← since delete end 1 + add 1
⋮ ⋮
S_{nn} O(1)

Is there a "order preserving hash fn"
– he doesn't know

Also one that keeps like values close together

We can hash for any fixed k, and can hash
[illegible] ↓

⑤

S_{1k}

S_{2k} in $O(n)$ time

S_{3k}

$\vdots$

S_{nk}

TA: [Could work, but I don't know off the top of my head

For any k, these strings

S_{1k}

$\vdots$ are ordered

S_{nk}

$dp[i,j]$ = rank of S_{ij} among $\{S_{1j}, S_{2j} \dots S_{nj}\}$

The # of #s of the set that are strictly $< n$

⑥

Ranks preserve order
are in range 0 .. n

$dp[i,n]\ \forall i$

n^2 subproblems that have n choices

but don't need all of them

What is a recurrence that works?

$dp[i,j]$

Can compute $dp[i,j]\ \forall i$ by taking pairs $(dp[i,j-1], A[i+j])$ and sorting them

$$S_{ij} = (S_{i,j-1}, A[i+j])$$

⑦

1758

Lex subgraph of len 2

	rank
1 7	0
7 5	2
5 8	1
8 1	3

← can sort this instead

rank = # of lex substrings less than n

Can use counting sort

Actually

pairs $\left(dp\left[i, \frac{j}{2}\right], dp\left[i+\frac{j}{2}, \frac{j}{2}\right]\right)$

$S_{ij} = \left(S_{i,j/2}, S_{i+j/2, j/2}\right)$

⑧

$dp[i,1]$

$dp[i,2]$] $dp[i,3]$

$dp[i,4]$ — $dp[i,7]$

⋮

$dp[i,2^k]$

good if j even

bad if j odd

$n = \sum_i 2^{a_i}$

↑ get all powers of 2

$n = 2^k + 2^{k-1} + \dots + 1$

$dp[i,a]$

$dp[i,b]$] → $dp[i,a+b]$

$O(n)$ time

Q (missed)

No simple recurrence

$S_{i,a+b} \sim (dp[i,a], dp[i+a,b])$

⑨

This problem involves almost everything except
graphs

Review it before the final

6.006
L26 Flipping a Coin

5/17

(not on final)

Last time: alg Silvio for primality testing

Input n (w/ $\log n$ bits)

Desired behavior: Prime
Composit

Runs in time poly $(\log n)$

Chose random el in $\mathbb{Z}_n^*$

$P[A = \text{prime}] = 1$ if actually prime

$P[A = \text{prime}] \leq \frac{1}{2}$ if composit

? Do a bunch of time to see

Can we de-randomize this?

Was found many years later - but more complicated

The randomization alg was far more elegant

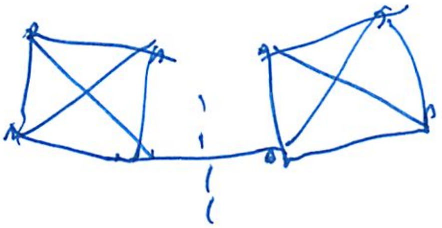
Minimum-cut

Undirected connected graph $G=(V,E)$

Split graph into L, R

So min # of edges b/w L and R

- ie find bottleneck



Can have multiple

- like tree can cut anywhere
  every edge is a bottleneck

③

But how does it find min cut

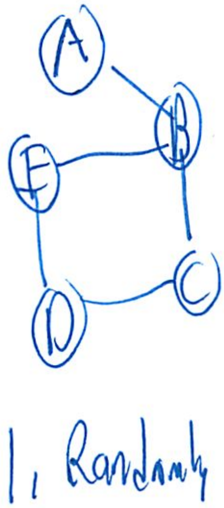
Deterministic $O\left(|V||E| \log \frac{|V|^2}{|E|}\right)$

Random $O(|V|^2 \log |V|)$

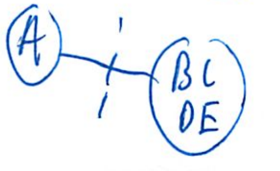
↙

Does not always work, but usually do

Intuition: Min cut is hopefully a small set of edges



1. Randomly pick an edge
2. Collapses nodes together
3. Repeat till ans



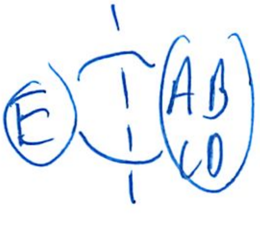
So alg was lucky + returned right cut

But it may fail!

Psuedo code

```
- while more than 2 nodes remain
- pick random edge $e = (u,v)$
- merge $u + v$
- (missed)
```

Failure



size 2 cut

worse than before

Claim $P(\text{good ex}) \geq \frac{2}{n^2}$

But this is really low

So repeat alg ~n^2 times

And pick the best

So alg likely to work

Lower-bounding prob of good execution

Graph may have many min-cuts

So fix one → C

$G_0 = G, G_1, G_2, \dots G_{n-2}$ = graphs created by Kargers alg

each step is a G_i

Want p(G_{n-2}) only contains edges of C

$P[\text{success}] = P[\text{none of chose edges belong to } C]$

$= P[e_0 \notin C] \cdot P[e_1 \notin C \mid e_0 \notin C] \cdot \dots$

Bayes' Rule

Finally a good lecture again!

$= P[e_0 \notin C \land e_1 \notin C \land e_2 \notin C \text{ etc}]$

not ind events

Now lets lower bound $P[e_0 \notin C]$?

$$P[e_0 \notin C] = \frac{|E| - |C|}{|E|}.$$

$$= 1 - \frac{|C|}{|E|}$$

$$\geq 1 - \frac{|C|}{|V||C|/2}$$

(missed)

$$\geq 1 - \frac{2}{|V|}$$

$$P[e_i \in C \mid e_0, .. e_{i-1} \notin C] = ?$$

? if then min cut of G_i has at most size C

⑦

Can it become smaller?

No - since then could draw back and find smaller graph

$$P\left[e_i \notin C \;\middle|\; c_0 \notin C, \vdots, e_{i-1} \notin C\right] \geq 1 - \frac{2}{|V| - i}$$

So

$$P[\text{success}] \geq \frac{|V|-2}{|V|} \cdot \frac{|V|-3}{|V|-1} \cdot \text{etc} \cdot \frac{4}{6} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}$$

$$= \frac{2}{|V|\,|V|-1} \geq \frac{2}{n^2}$$

Chance is reasonably small

So do it n^2

Random Walks in Graph

Squirl goes to random neighbor

Picks randomly each time

Where is he after t space?

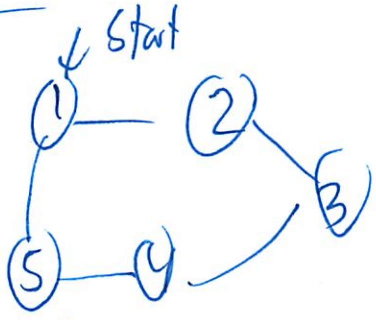
L somewhere random

So what is prob that he will be at each vertex

(Did we do in 6.042?)

He must move

Simple example



node	1	2	3	4	5

$X_0 = (1, 0, 0, 0, 0)$

$X_1 = (0, \frac{1}{2}, 0, 0, \frac{1}{2})$

$X_2 = (\frac{1}{2}, 0, \frac{1}{4}, \frac{1}{4}, 0)$

What is a more systematic way to do?

$$A = \frac{\text{adj matrix}}{\underset{\uparrow \text{degree}}{2}}$$

$$= \begin{pmatrix} 0 & 1/2 & 0 & 0 & 1/2 \\ 1/2 & 0 & 1/2 & 0 & 0 \\ 0 & 1/2 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 0 & 1/2 \\ 1/2 & 0 & 0 & 1/2 & 0 \end{pmatrix}$$

so ~~then~~

A_{ij} = prob squirl jumps to j if at i

Random Walks $\forall t$

$$X_t = X_{t-1} \circ A$$

$$X_t = X_0 A^t$$

How to compute?

Not A^t - but repeated squaring

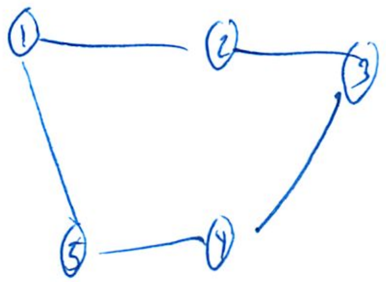
then vector-matrix product

A
A^2
A^4
etc

limiting distribution x_t as $t \to \infty$

What is x_∞ in 5-cycle

So back
to our
problem



$x_\infty = \left(\frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}\right)$

Can verify in MatLab

λ values

$\lambda = 1, \underbrace{.3090, .3090}_{=}, \underbrace{-.8090, -.8090}_{=}$

This holds for all directed graph

largest eig val $= 1$

all others abs < 1

$e_1 = (1/5, 1/5, 1/5, 1/5, 1/5)$ is the left e.vector for λ_1

Proof choose e_2, e_3, e_4, e_5 so evectors for a basis

So $x_0 = a_1 e_1 + a_2 e_2 + \dots + a_5 e_5$

$x_t = x_0 A^t = a_1 e_1 A^t + \dots + a_5 e_5 A^t$

$= x_0 \lambda_i^t = a_1 e_1 \lambda^t + \dots$

$\uparrow$ goes to 0

(like 18.06)

General Theorem

(see slide)

· PageRank

Google

From p of random links

(see slides)

6.006- *Introduction to Algorithms*

Lecture 26

How Flipping Coins Helps Computation

Prof. Constantinos Daskalakis

Coin Flips in Algorithms

- Last time we gave an algorithm *SILVIO* for **primality testing**
- **Input:** Number *n* (represented by O(log n) bits)
- **Desired Behavior:** "PRIME" if *n* is prime, "COMPOSITE" o.w.
- *SILVIO* run in time poly(log *n*), i.e. polynomial in the representation of *n*.
- *SILVIO* flipped coins (namely somewhere in its execution it chose a random element in Z_n^*)
- ***SILVIO*'S Behavior:**
  - Pr[A(n)="PRIME"]=1, if *n* is prime
  - Pr[A(n)="PRIME"]≤1/2, if *n* is composite
- By repetition can boost the probability of outputting a correct answer as much as we want.
- Can *SILVIO* be derandomized? Unknown as of yet
- There *is* a primality testing algorithm that is deterministic.
- It was discovered many years later and is more complicated.
- **Moral:** Flipping coins enables simpler, and (potentially) faster computation.

Menu

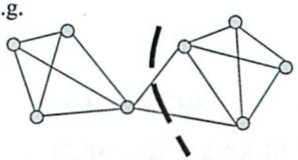
- Minimum-cut
- Random walks in graphs
  - Pagerank

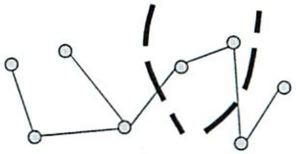
Menu

- **Minimum-cut**
- Random walks in graphs
  - Pagerank

MIN-CUT

- **Input:** Undirected connected graph G=(V,E).
- **Output:** Partition V into L and R minimizing the edges between L and R.
- i.e. find the **bottleneck** of a graph.
- E.g.



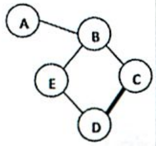


Any edge is a min-cut

- Best deterministic algorithm: $O(|V|\,|E| \log |V|^2/|E|)$.
- Fastest and simplest known algorithm: randomized; time $O(|V|^2 \log|V|)$
- Obtained by David Karger in 1993.
- **Intuition:** Minimum cut is (hopefully) a small set of edges.
- SO if I pick a random edge, chances are that it's not part of the minimum cut.

Karger's Algorithm

- Example execution:

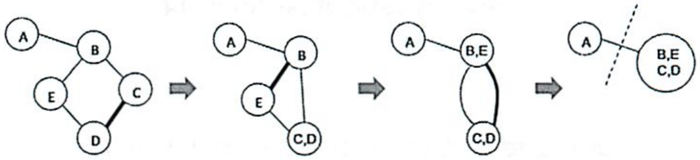


- Pseudocode:

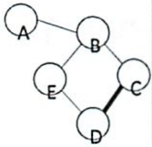
```
While more than two nodes remain:
    - pick random edge $e = (u, v)$;
    - merge $u$ and $v$.
      (called a contraction)
Output surviving edges.
```

Karger's Algorithm

- Good execution:



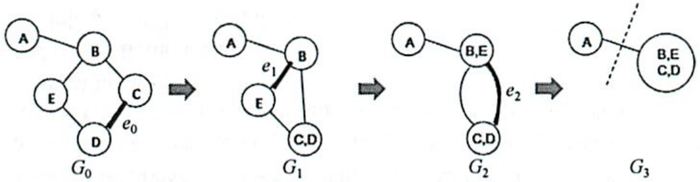
- Bad execution:



Claim: Pr[good execution] ≥ $2/n^2$ ➔ ~ n^2 repetitions suffice!

Karger's Algorithm

- Lower-bounding the probability of good execution.
- Graph may have many min-cuts (remember tree example).
- Let's fix one of them ***C***.
- Call $G_0=G, G_1, G_2, \dots, G_{n-2}$ the graphs created by Karger's algorithm.

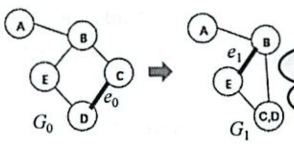


- Want to find probability that G_{n-2} only contains edges of ***C***.
- Pr[success] = Pr[none of chosen edges belongs to ***C***]

$$= \Pr[e_0 \notin C] \cdot \Pr[e_1 \notin C \mid e_0 \notin C] \cdot \ldots \cdot \Pr[e_{n-3} \notin C \mid e_0, \ldots, e_{n-4} \notin C]$$

Karger's Algorithm

- Let's fix a min-cut ***C***.
- Call $G_0=G, G_1, G_2,\ldots,G_{n-2}$ the graphs created by Karger's algorithm.



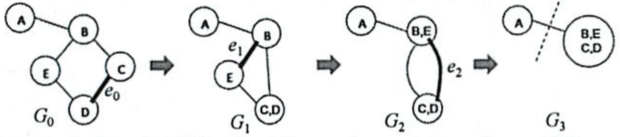
if the min-cut of a graph has size |C| then every vertex has degree ≥|C|

- Want to find probability that $G_{n\text{-}2}$ only contains edges of ***C***.
- Pr[success] = Pr[none of chosen edges belongs to ***C***]
  $= \Pr[e_0 \notin C] \cdot \Pr[e_1 \notin C \mid e_0 \notin C] \cdot \ldots \cdot \Pr[e_{n\text{-}3} \notin C \mid e_0,\ldots,e_{n\text{-}4} \notin C]$
- Warm-up: $\Pr[e_0 \notin C]$?

$$\Pr[e_0 \notin C] = \frac{|E|-|C|}{|E|} = 1 - \frac{|C|}{|E|} \geq 1 - \frac{|C|}{|V||C|/2} = 1 - \frac{2}{|V|}$$

Karger's Algorithm

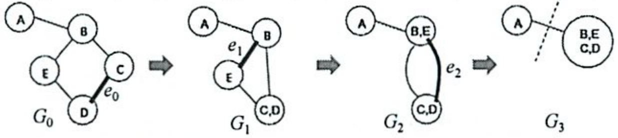
- Let's fix a min-cut ***C***.
- Call $G_0=G, G_1, G_2,\ldots,G_{n-2}$ the graphs created by Karger's algorithm.



- Want to find probability that $G_{n\text{-}2}$ only contains edges of ***C***.
- $\Pr[\text{success}] = \Pr[e_0 \notin C] \cdot \ldots \cdot \Pr[e_{n\text{-}3} \notin C \mid e_0,\ldots,e_{n\text{-}4} \notin C]$
- Warm-up: $\Pr[e_0 \notin C] \geq 1\text{-}2/|V|$
- $\Pr[e_i \notin C \mid e_0,\ldots,e_{i\text{-}1} \notin C]$?
- **Claim:** If $e_0,\ldots,e_{i\text{-}1} \notin C$, then the minimum cut of G_i has size |C|.
- **Proof:** All edges in C have survived. So min-cut at most size |C|.
- If there is a smaller cut in G_i, then that cut exists also in G_0.
- QED

Karger's Algorithm

- Let's fix a min-cut ***C***.
- Call $G_0=G, G_1, G_2,\ldots,G_{n-2}$ the graphs created by Karger's algorithm.

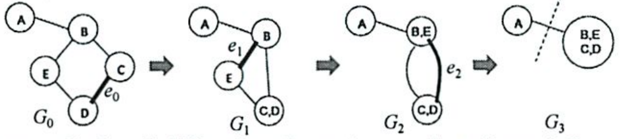


- Want to find probability that $G_{n\text{-}2}$ only contains edges of ***C***.
- $\Pr[\text{success}] = \Pr[e_0 \notin C] \cdot \ldots \cdot \Pr[e_{n\text{-}3} \notin C \mid e_0,\ldots,e_{n\text{-}4} \notin C]$
- Warm-up: $\Pr[e_0 \notin C] \geq 1\text{-}2/|V|$
- $\Pr[e_i \notin C \mid e_0,\ldots,e_{i\text{-}1} \notin C]$?
- **Claim:** If $e_0,\ldots,e_{i\text{-}1} \notin C$, then the minimum cut of G_i has size |C|.
- So:

$$\Pr[e_i \notin C \mid e_0 \notin C,\ldots,e_{i-1} \notin C] \geq 1 - \frac{2}{??}$$

Karger's Algorithm

- Let's fix a min-cut ***C***.
- Call $G_0=G, G_1, G_2,\ldots,G_{n-2}$ the graphs created by Karger's algorithm.

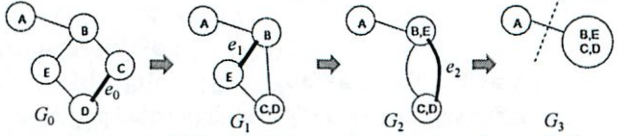


- Want to find probability that $G_{n\text{-}2}$ only contains edges of ***C***.
- $\Pr[\text{success}] = \Pr[e_0 \notin C] \cdot \ldots \cdot \Pr[e_{n\text{-}3} \notin C \mid e_0,\ldots,e_{n\text{-}4} \notin C]$
- Warm-up: $\Pr[e_0 \notin C] \geq 1\text{-}2/|V|$
- $\Pr[e_i \notin C \mid e_0,\ldots,e_{i\text{-}1} \notin C]$?
- **Claim:** If $e_0,\ldots,e_{i\text{-}1} \notin C$, then the minimum cut of G_i has size |C|.
- So:

$$\Pr[e_i \notin C \mid e_0 \notin C,\ldots,e_{i-1} \notin C] \geq 1 - \frac{2}{|V|-i}$$

Karger's Algorithm

- Let's fix a min-cut *C*.
- Call G_0=G, G_1, G_2,…,G_{n-2} the graphs created by Karger's algorithm.



- Want to find probability that G_{n-2} only contains edges of *C*.
- $\Pr[\text{success}] = \Pr[e_0 \notin C] \cdot \ldots \cdot \Pr[e_{n-3} \notin C \mid e_0, \ldots, e_{n-4} \notin C]$
- So:

$$\Pr[e_i \notin C \mid e_0 \notin C, \ldots, e_{i-1} \notin C] \geq 1 - \frac{2}{|V|-i} = \frac{|V|-i-2}{|V|-i}$$

- Hence:

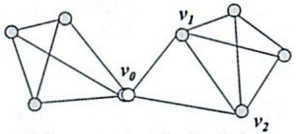
$$\Pr[\text{success}] \geq \frac{|V|-2}{|V|} \cdot \frac{|V|-3}{|V|-1} \cdot \frac{|V|-4}{|V|-2} \cdot \frac{|V|-5}{|V|-3} \cdot \ldots \cdot \frac{4}{6} \cdot \frac{3}{5} \cdot \frac{2}{4} \cdot \frac{1}{3}$$

$$= \frac{2}{|V||V|-1} \geq 2/n^2$$ ➔ repeat algorithm ~n^2 times and choose best cut

Menu

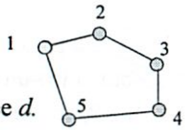
- Minimum-cut
- **Random walks in graphs**
  - Pagerank

Random Walks



- Given undirected graph $G = (V, E)$
- A squirrel stands at vertex v_0 :
- Squirrel ate fermented pumpkin so doesn't know what he's doing
- So jumps to random neighbor v_1 of v_0
- Then jumps to random neighbor v_2 of v_1
- etc
- Question: Where is squirrel after *t* steps?
- A: At some random location.
- OK, with what probability is squirrel at each vertex of the graph?
- Want to compute $x_t \in \mathbb{R}^n$, where
- $x_t(i)$: probability squirrel is at node *i* at time *t*.

$x_t \rightarrow x_{t+1}$?

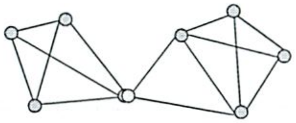


- Simplification: all nodes have same degree *d*.
- $x_0 = (1, 0, 0, 0, 0)$
- $x_0 \rightarrow x_1$?
- if $u_1, u_2, \ldots, u_d$ are the *d* neighbors of v_0, then
- $v_1 = u_i$ with probability $1/d$
- so $x_1 = (0, ½ , 0, 0, ½)$
- $x_2 = (½, 0, ¼, ¼, 0)$
- …
- $A = \begin{pmatrix} 0 & ½ & 0 & 0 & ½ \\ ½ & 0 & ½ & 0 & 0 \\ 0 & ½ & 0 & ½ & 0 \\ 0 & 0 & ½ & 0 & ½ \\ ½ & 0 & 0 & ½ & 0 \end{pmatrix}$ (adjacency matrix divided by *d*)

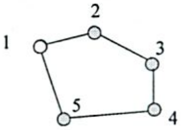
$x_1 = x_0 A$
$x_2 = x_1 A = x_0 A^2$
$x_3 = x_2 A = x_0 A^3$
$\vdots$
$x_t = x_0 A^t$

A_{ij} =probability of jumping to *j* if squirrel is at *i*

x_t



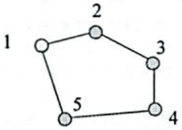
- More general undirected graphs?
- A =adjacency matrix where row i is divided by the degree d_i of i
- $x_t = x_0 A^t$
- Computing x_t ?
- Silvio will be disappointed if you don't use…
- repeated squaring!
- Compute $A \to A^2 \to A^4 \to \ldots \to A^t$ (if t is a power of 2; if not …)
- then do vector-matrix product
- How about limiting distribution x_t as $t \to \infty$?
- e.g. what is x_∞ in 5-cycle?
- $x_\infty = (⅕, ⅕, ⅕, ⅕, ⅕)$



Verifying $x_t \to$ (⅕, ⅕, ⅕, ⅕, ⅕)

- Recall

$$A = \begin{pmatrix} 0 & \tfrac12 & 0 & 0 & \tfrac12 \\ \tfrac12 & 0 & \tfrac12 & 0 & 0 \\ 0 & \tfrac12 & 0 & \tfrac12 & 0 \\ 0 & 0 & \tfrac12 & 0 & \tfrac12 \\ \tfrac12 & 0 & 0 & \tfrac12 & 0 \end{pmatrix}$$



- $x_0 = [1 \quad 0 \quad 0 \quad 0 \quad 0]$
- $x_1 = [0 \quad 0.5000 \quad 0 \quad 0 \quad 0.5000]$
- $x_2 = [0.5000 \quad 0 \quad 0.2500 \quad 0.2500 \quad 0]$
- $x_3 = [0 \quad 0.3750 \quad 0.1250 \quad 0.1250 \quad 0.3750]$
- $x_4 = [0.3750 \quad 0.0625 \quad 0.2500 \quad 0.2500 \quad 0.0625]$
- $x_5 = [0.0625 \quad 0.3125 \quad 0.1562 \quad 0.1562 \quad 0.3125]$
- $x_6 = [0.3125 \quad 0.1094 \quad 0.2344 \quad 0.2344 \quad 0.1094]$
- $x_7 = [0.1094 \quad 0.2734 \quad 0.1719 \quad 0.1719 \quad 0.2734]$
- $x_8 = [0.2734 \quad 0.1406 \quad 0.2227 \quad 0.2227 \quad 0.1406]$
- $x_9 = [0.1406 \quad 0.2480 \quad 0.1816 \quad 0.1816 \quad 0.2480]$
- $x_{10} = [0.2480 \quad 0.1611 \quad 0.2148 \quad 0.2148 \quad 0.1611]$
- $x_{11} = [0.1611 \quad 0.2314 \quad 0.1880 \quad 0.1880 \quad 0.2314]$
- $x_{12} = [0.2314 \quad 0.1746 \quad 0.2097 \quad 0.2097 \quad 0.1746]$
- $x_{13} = [0.1746 \quad 0.2206 \quad 0.1921 \quad 0.1921 \quad 0.2206]$
- $x_{14} = [0.2206 \quad 0.1833 \quad 0.2064 \quad 0.2064 \quad 0.1833]$

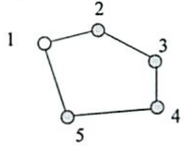
$x_{15} = [0.1833 \quad 0.2135 \quad 0.1949 \quad 0.1949 \quad 0.2135]$
$x_{16} = [0.2135 \quad 0.1891 \quad 0.2042 \quad 0.2042 \quad 0.1891]$
$x_{17} = [0.1891 \quad 0.2088 \quad 0.1966 \quad 0.1966 \quad 0.2088]$
$x_{18} = [0.2088 \quad 0.1929 \quad 0.2027 \quad 0.2027 \quad 0.1929]$
$x_{19} = [0.1929 \quad 0.2058 \quad 0.1978 \quad 0.1978 \quad 0.2058]$
$x_{20} = [0.2058 \quad 0.1953 \quad 0.2018 \quad 0.2018 \quad 0.1953]$
$x_{21} = [0.1953 \quad 0.2038 \quad 0.1986 \quad 0.1986 \quad 0.2038]$
$x_{22} = [0.2038 \quad 0.1969 \quad 0.2012 \quad 0.2012 \quad 0.1969]$
$x_{23} = [0.1969 \quad 0.2025 \quad 0.1991 \quad 0.1991 \quad 0.2025]$
$x_{24} = [0.2025 \quad 0.1980 \quad 0.2008 \quad 0.2008 \quad 0.1980]$
$x_{25} = [0.1980 \quad 0.2016 \quad 0.1994 \quad 0.1994 \quad 0.2016]$

$\boxed{x_t = x_0 A^t}$

Proving $x_t \to$ (⅕, ⅕, ⅕, ⅕, ⅕) ?

- Recall

$$A = \begin{pmatrix} 0 & \tfrac12 & 0 & 0 & \tfrac12 \\ \tfrac12 & 0 & \tfrac12 & 0 & 0 \\ 0 & \tfrac12 & 0 & \tfrac12 & 0 \\ 0 & 0 & \tfrac12 & 0 & \tfrac12 \\ \tfrac12 & 0 & 0 & \tfrac12 & 0 \end{pmatrix}$$

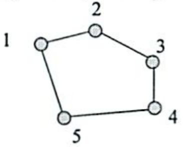


- Random idea: what are the eigenvalues of A ?
- A symmetric so 5 real eigenvalues
- $\lambda_1 = 1.0000$, $\lambda_2 = \lambda_3 = 0.3090$, $\lambda_4 = \lambda_5 = -0.8090$ (thanks Matlab)
- coincidence: $\lambda_2 = \lambda_3$ and $\lambda_4 = \lambda_5$ (5-cylce is a special graph)
- non-coincidence (holds for any undirected graph*):
  - largest eigenvalue =1
  - all others have absolute value <1
- left eigenvector corresponding to $\lambda_1 = 1.0000$?
- $e_1 = (⅕, ⅕, ⅕, ⅕, ⅕)$ is a left eigenvector for λ_1
- Wow. Why would $x_t \to e_1$ as $t \to \infty$?

Proving $x_t \to$ (⅕, ⅕, ⅕, ⅕, ⅕) ?

- Recall

$$A = \begin{pmatrix} 0 & \tfrac12 & 0 & 0 & \tfrac12 \\ \tfrac12 & 0 & \tfrac12 & 0 & 0 \\ 0 & \tfrac12 & 0 & \tfrac12 & 0 \\ 0 & 0 & \tfrac12 & 0 & \tfrac12 \\ \tfrac12 & 0 & 0 & \tfrac12 & 0 \end{pmatrix}$$



- $\lambda_1 = 1.0000$, $\lambda_2 = \lambda_3 = 0.3090$, $\lambda_4 = \lambda_5 = -0.8090$
- $e_1 = (⅕, ⅕, ⅕, ⅕, ⅕)$
- Proof: choose e_2, e_3, e_4, e_5 so that eigenvectors form a basis
- (guaranteed by the spectral theorem since A is symmetric)
- so $x_0 = a_1 e_1 + a_2 e_2 + a_3 e_3 + a_4 e_4 + a_5 e_5$, for some a_1, a_2, a_3, a_4, a_5
- Now $x_t = x_0 A^t =$

$$= a_1 e_1 A^t + a_2 e_2 A^t + a_3 e_3 A^t + a_4 e_4 A^t + a_5 e_5 A^t$$
$$= a_1 e_1 \lambda_1^t + a_2 e_2 \lambda_2^t + a_3 e_3 \lambda_3^t + a_4 e_4 \lambda_4^t + a_5 e_5 \lambda_5^t$$
$$\to a_1 e_1, \text{ as } t \to \infty$$

- since $e_1 = (⅕, ⅕, ⅕, ⅕, ⅕)$ is a distribution, it must be that $a_1 = 1$
- Hence $x_t \to (⅕, ⅕, ⅕, ⅕, ⅕)$, as $t \to \infty$

More General Theorem

- Given directed graph G
- Take A = adjacency matrix where row i is divided by the **out**-degree d_i of i
- (Under mild conditions*) A has eigenvalue 1 with multiplicity 1 and all other eigenvalues will have absolute value <1
- Moreover, if e_1 be the (unique) left eigenvector corresponding to eigenvalue 1,
- then e_1 will have all components positive.
- Normalize it so that it is a distribution.
- **Theorem:** A random walk on G **started anywhere** will converge to distribution e_1!
- e_1 is called the "**stationary distribution of *G***"

- (Fundamental Theorem of Markov Chains)

- Two obvious Questions:
  - why is x_∞ interesting?
  - how fast does $x_t \to x_\infty$?

Menu

- Minimum-cut
- Random walks in graphs
  - **Pagerank**

Pagerank

- No better proof that something is useful than having interesting applications ☺
- It turns out that random walks have a famous one: PageRank
- PageRank of a webpage $p \approx^*$ Probability that a web-surfer starting from some central page (e.g. Yahoo!) and following random weblinks arrives at webpage p in infinite steps.
- How compute this probability?
- Form graph G = the hyperlink graph;
- Namely, G has a node for every webpage, and there is an edge from webpage p_1 to webpage p_2 *iff* there is a hyperlink from p_1 to p_2.
- Compute stationary distribution of G, i.e. the left eigenvector of the (normalized by out-degrees) adjacency matrix A of G, corresponding to eigenvalue 1.
- How compute stationary distribution?
- Idea 1: Crawl the web, create giant A, solve eigenvalue problem.
- Runtime $O(n^3)$ using Gaussian elimination
- too much for n = size of the web

Pagerank

- Graph G = the hyperlink graph
- Compute stationary distribution of G, i.e. the left eigenvector of the (normalized by out-degrees) adjacency matrix A of G, corresponding to eigenvalue 1.
- How compute stationary distribution?
- Different (better?) idea:
- Forget linear algebra;
- Start at some central page and do random walk for a few steps (how many?);
- Restart and repeat (how many times?);
- then take PageRank(p) ≈ empirical probability that random walk ended at p.
- If web-graph is well-connected*, hope that empirical distribution should be good approximation to stationary distribution for the right choice of "how many" above…
- or at least for the top components of the eigenvector, which are the most important for ranking the top results.
- *caveat: In reality, Pagerank corresponds to the stationary distribution of a random surfer who does the following at every step: with probability 15% jumps to a random page (called a restart), & with probability 85% jumps to a random neighbor.
- Same theory applies.

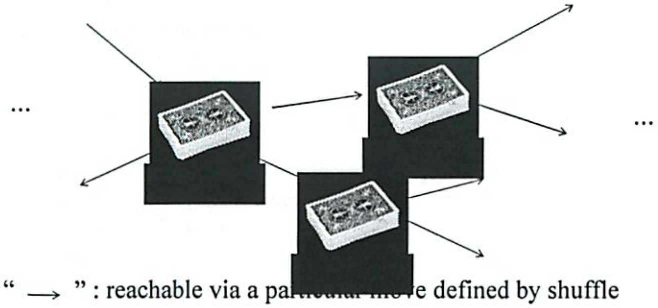
Menu

- Minimum-cut
- Random walks in graphs
  - Pagerank
  - **How fast does $x_t \rightarrow x_\infty$?**

"Mixing Time"

- Captures the speed at which $x_t \rightarrow x_\infty$
- Speed depends on connectivity of G.
- Sometimes G is given to us and we can't change it.
- But sometimes we design G .
- e.g. in card shuffling
- type of shuffle defines connectivity of the graph between deck configurations…

Card Shuffling Graph



" ⟶ " : reachable via a pa[illegible]e defined by shuffle

while performing the shuffle we jump from node to node of this graph

stationary distribution of a correct shuffle?

probability 1/52! on each permutation

Effect of Shuffle to Mixing Time

Different shuffles have different mixing times. Examples:

- Top-in-at-Random: *take top card and stick it to random location*

Number of repetitions to be close to uniform permutation?

~300 repetitions

- Riffle Shuffle:

Number of repetitions? ~10

So different shuffles have different graphs with different mixing times.

Summary

Randomness is useful
As are the other techniques we saw in this class
When facing an algorithmic problem:

- understand it
- try brute force first
- then try to improve it using:
    - a cool data structure such as an AVL tree/heap/hash table
    - a cool algorithmic technique such as Divide and Conquer, or DP
    - map it to a graph problem and use off the shelf algorithm such as BFS/DFS/Dijkstra/Bellman-Form/Topological Sort
    - or modify these algorithms
- If everything else fails, maybe NP-hard? Try to reduce an NP-hard problem to your problem.
- Look at a catalog of NP-hard problems, find a similar problem to your problem and try to reduce that problem to your problem.
- Great hanging out every Tuesday and Thursday
- Evaluate class: http://web.mit.edu/subjectevaluation/evaluate.html