STANFORD ALGOS 3

WEEK 1 L1

GRAPHS AND THE INTERNET

Claim: the Internet is a graph
[vertices = end hosts and routers, directed edges = direct connections]

Other graphs related to the Internet:
- Web graph
- Social Networks

INTERNET ROUTING

Suppose Stanford needs to send Cornell some data

Which route to use? We'd want to use the shortest path (of links)

Issue? Too many vertices (and changing weighted edges) in X.

Can't use Dijkstra's, but can use Bellman-Ford, which is non-centralized and can deal with negative edge lengths (but you just said it was useless?)

2)

WEEK 2.2

SEQUENCE ALIGNMENT

Input: two strings over the alphabet {A, C, G, T} (portions of on or more genes)

Problem: How "similar" are they?

Application examples: ① Extrapolate function of genome substrings
② Similar genomes to infer proximity in evolutionary tree

Example:
A G G G C T
A G G C A
(one more match after gap; mismatches after gap; insert gap here)

"Penalties for mismatches and gaps!"

Input

PROBLEM

Input: 2 strings over {A,C,G,T}, penalties $pen_{gap} \geq 0$ and $pen_{AT} \geq 0$

Output: alignment of strings that minimizes the total penalty

total penalty is called Needleman-Wunsch score

We need an efficient way to find it, and we can't Brute Force it

A 500 letter string has more alignments than atoms in the universe (10^{125})

Some Algorithm designs:

- Divide and conquer
- Randomized algorithms
- Eg. Greedy algorithms
- Dynamic Programming.

Greedy algorithms make "myopic" decisions and hope everything works out at the end. Process every data point once

[redacted]

GREEDY VS DIVIDE & CONQUER

(1) Easy to propose multiple greedy algorithms for problems

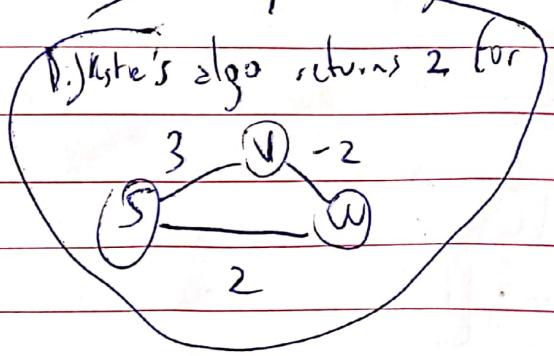
(2) And to calculate their running time too

(3) Hard to establish correctness, even ~~though~~ if you ~~might~~ find a correct one you'll have a hard time understanding why.

Dijkstra's algorithm ~~was~~ is easy to prove ish with positive lengths

It's easy to prove incorrectness!

Dijkstra's algo returns 2 for



Three strategies for Correctness:

(1) Induction

(2) Exchange (show it can be exchanged for an optimal solution)

(3) Whatever works!

1st WEEKLY

Recall: THE CACHING PROBLEM

- small fast memory (the cache)
- big slow memory
- process sequence of "page requests"
- on a fault, what do you evict from cache?

Example: Cache | a | b | c | d |

Request sequence: c ✓

d ✓

e ✗ what to do? We have to load e, but what do we evict?

Let's say we evict a: | e | b | c | d |

f ✗

we evict b: | e | f | c | d |

a ✗

b ✗

So in the end we had request sequence c d e f a b and 4 fails we could have gotten only 2 if we ~~evicted~~ evicted c and d

What eviction algorithm to choose?

THE OPTIMAL CACHING ALGORITHM

Theorem: [Belady 1960] The "furthest-in-future" algorithm is optimal (but what is it?)

Good benchmark for caching algorithms. LRU (least recently used) should work well enough as long as data exhibits locality of reference

WEEK 1 LS

SCHEDULING

Setup: - One shared resource (example: processor)
- many jobs to do

Question: What order do we do the jobs in?

Assume that each job j has a
- weight w_j
- length l_j

Completion Time c_j of job j = sum of job lengths up to and including j
Example: 3 jobs 1,2,3 with l_j 1,2,3.
c_j = 1, 3, 6.

We are trying to optimise the weighted sum of completion times:

$$\min \sum_{j=1} w_j c_j$$

Back to our (1,2,3) l_j (1,2,3) w_j (3,2,1) example:
3+6+15 = 24

How to do this in general?

WEEK 1 L6

ALGO FOR SCHEDULING

Intuition → do bigger weight smallest length first

These are decent greedy algorithms on their own, better if combined.

Idea: assign jobs a score, say $f: w \to 3 \cdot l$ to order them by

Guess ① $= w_j - l_j$ → Easy to find examples contradicting first

Guess ② $= w_j / l_j$

→ Always correct!

WEEK 1 L7 AND L8

PROOF: by Exchange Argument

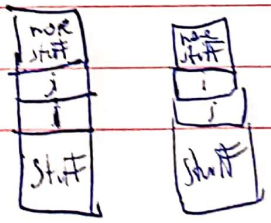
Fix n jobs, proceed by contradiction

Let σ = greedy schedule, σ^* = optimal schedule

Assume: all $\left(\frac{w_j}{l_j}\right)$ s distinct. Assume you order them $\frac{w_1}{l_1} > \frac{w_2}{l_2} \dots$

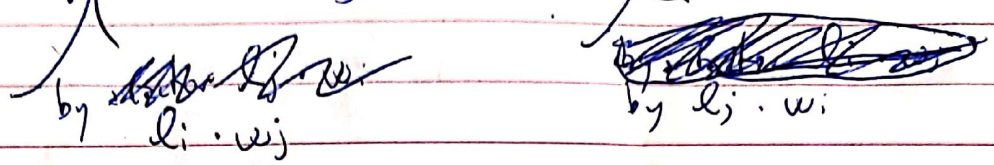
σ is 1, 2, 3, ... n.

if $\sigma^* \neq \sigma$, then a consecutive jobs (at least a pair) where i > j



What is the effect of swapping these two jobs is that

C_j goes up and C_i goes down



As we said, $\frac{w_i}{l_i} < \frac{w_j}{l_j} \Rightarrow l_i w_j > l_j w_i$

So if σ^* wasn't equal to σ, it would be a contradiction to it being optimal

WEEK 1 L10

MST PROBLEM.

Goal: Connect a bunch of vertices together as cheap as possible

Applications: clustering, networking

Blazingly fast Greedy Algorithms:

- Prim's Algorithm
- Kruskal's Algorithm

($O(m \log(n))$, really fast, almost linear!

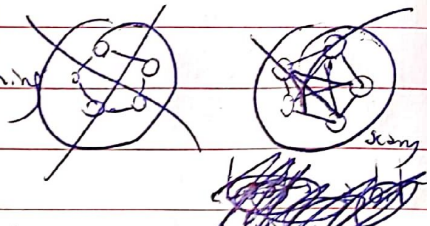
DEFINITION OF PROBLEM

Input: undirected graph, (we won't talk about directed graph branching) $G(V,E)$ cost c_e for each edge

- Assume adjacency list representation, no need to exclude negative edges

Output: minimum sum of edge costs cost tree $T \subseteq E$ that spans all vertices

Meaning



Skinny

Disconnected

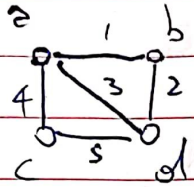
STANDING ASSUMPTIONS

#1 Input graph G is connected (which is easy to check in pre processing with DFS)

#2 Edge costs are distinct. (Prim + Kruskal remains correct though)

WEEK 1 L11

EXAMPLE OF PRIM-KRUSCAL



Like with Dijkstra's, we'll grow our tree vertex by vertex

~~Starting from b → (b, a) → (b, a, d) → (b, a, d, c)~~

Starting from b → $\underbrace{ba}_{1}$ → $\underbrace{ba + bd}_{3}$ → $\underbrace{ba + bd + ac}_{6}$

DEFINITION OF PRIM'S MST ALGO

- Initialize $X = \{s\}$, (s chosen arbitrarily, $s \in V$)
- $T = \emptyset$ (~~vertices explored by~~ tree so far.) pg)
- while $X \neq V$:
  - let $e = (u, v)$ be the cheapest edge between (X) and $(V - X)$
  - add v to X
  - add e to T

Theorem: PRIM'S MST ALGO IS CORRECT

Proof part I: Computes a spanning tree T^*

Proof part II: T^* is a MST

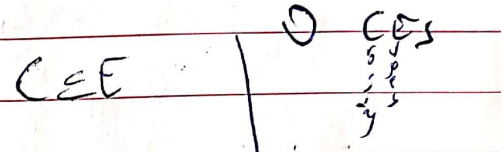
WEEK 1 L12

CUTS

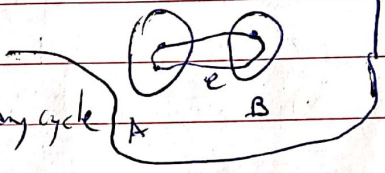
Claim: Prim's Algo outputs a spanning tree T^*.

Definition: a cut of a graph $G=(V,E)$ is a partition of V into 2 non-empty sets.

Empty cut Lemma: a graph is not connected $\iff$ $\exists$ cut A,B with 0 CEs



Double crossing Lemma: Suppose a cycle $C \subseteq E$ crosses a cut, it has to cross it twice



Lonely cut corollary: i.e only CE, not in any cycle

PROOF:

(1) It maintains invariant that T spans X

(2) Can't get stuck with $X \neq V$, otherwise the cut $(X, V-X)$ would be empty. By Empty cut Lemma, input graph G is disconnected.

(there no CEs to proceed on)

(3) No cycles ever get created in T because you only ever choose one edge leading to a new vertex that gets added to X

WEEK 1 L13

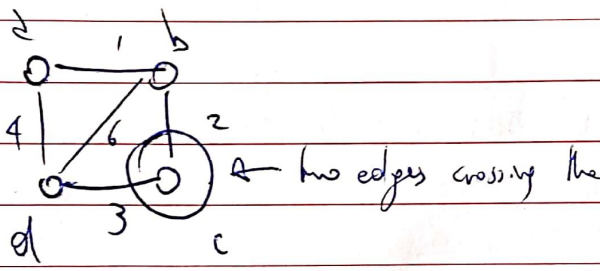
CORRECTNESS OF PRIM PROOF PART 2

Theorem 2: Prim's Algo computes minimum-cost T^*

Key Question: When is it "safe" to include an edge in the tree-safe?

CUT PROPERTY consider an edge e of G:

Suppose there is a cut (A, B) with the property that e is the cheapest edge of G that crosses it. e is part of the one and only MST



← two edges crossing the cut, edge bc with length 2 is minimum.

PROOF IN L14

12)

WEEK 1 L105

RUNNING TIME OF PRIM'S ALGO

- Init $X = \{s\}$
- $T = \emptyset$
- while $X \neq V$
  - let e be cheapest $(X)(V-X)$ edge
  - add e to T, add v to X

Running Time of naive implementation:

- $O(n)$ iterations
- $O(m)$ time per iteration

$O(nm)$ this is nothing to sneeze at, but worse than $O(n \log n)$

(can we do better?)

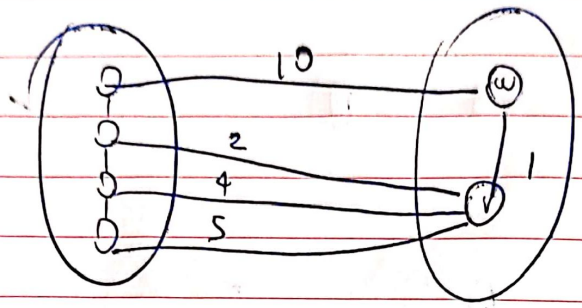
Yes, with heaps, just like with Dijkstra's

Heapify runs in $O(m \log n)$, allows us to run Prim's in $O(m \log n)$

WEEK 1 L16

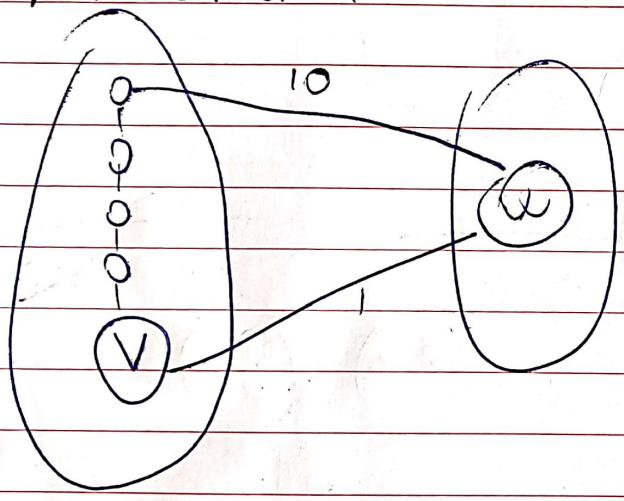
MAINTAINING INVARIANT #2

Heap with



Key[u] = 10
Key[v] = 2

After one iteration:



Key[u] = 1

We need to recompute some keys every step of the main while loop

Pseudo code: when v added to X:
- foreach edge $(v,w) \in E$
  - if $w \in (V - X)$ (the keys might have changed for these):
    - Delete w from heap
    - recompute $Key[w] := \min\{Key[w], c_{vw}\}$
    - re-insert w into heap.

(pre-processing) $< (n-1)$ Inserts
$< (n-1)$ Extract mins
$< m$ Delete-Insert combos
(Heap operations $O(\log(n))$)

N of operations at most $O(m \log(n))$ ish

14

Prob scribbling

s_i, t_i ....

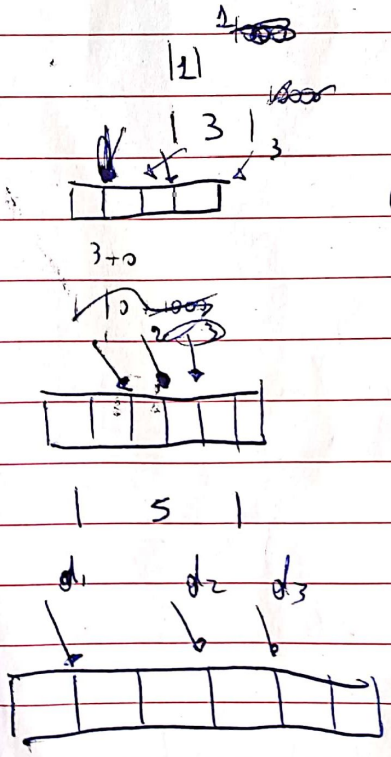
(0, 1) (1, 3) (2, 4) (4, 5), (4, 9)

(0, 1) (3, 4) (2, 5) (4, 430) (430, 5)

ditto not earliest start time

(0, 1) (2, 3) (2, 30, 330) 3

earliest finish time



(2, 2) (3, 2)

$C_j - d_j$

ditto not necessary order

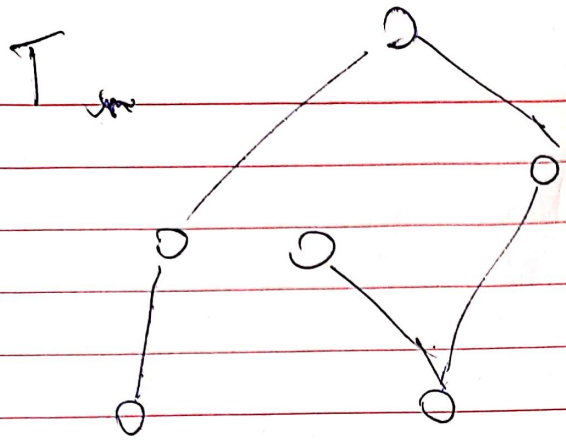
$(p_j + p_{j-1} + p_{j-2}) - d_i$

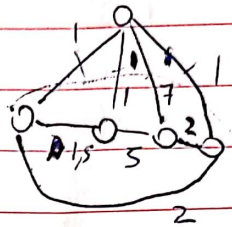
$p_i - d_i = x_i$, $p_j - d_j$ ~~$- p_i - d_i$~~

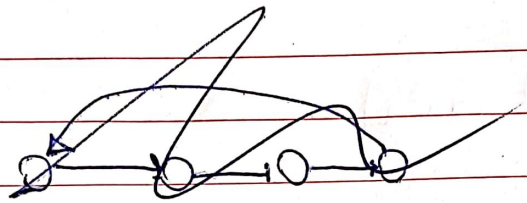
$= (p_j + p_i) - d_j - d_i + d_i$

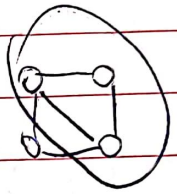
$p_j + x_i + (d_i - d_j)$ max

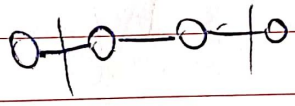
$= p_j + x_i - d_j + d_i = x_j + x_i + d_i$

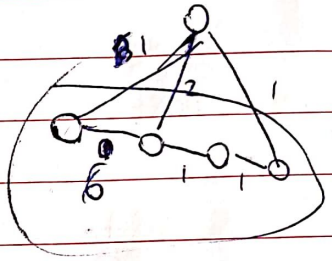


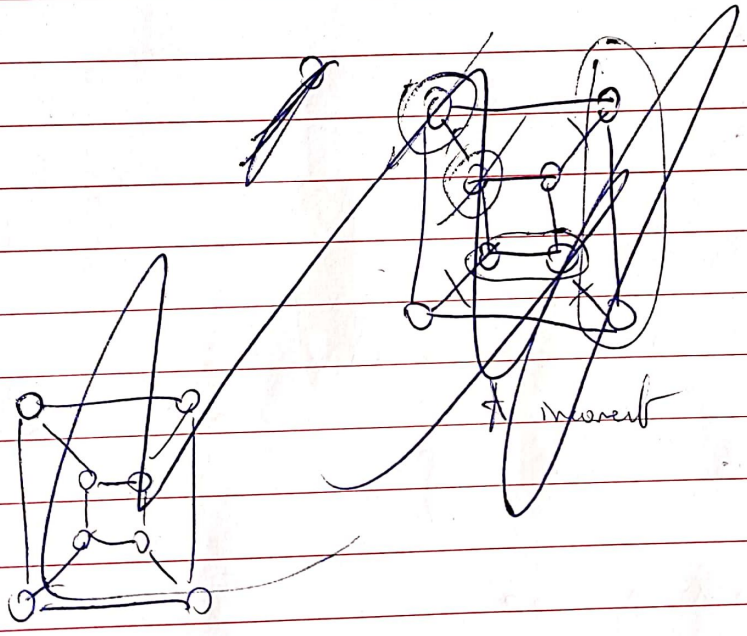


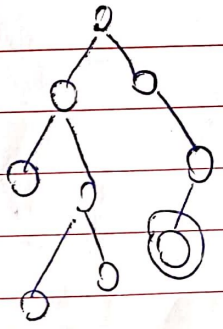


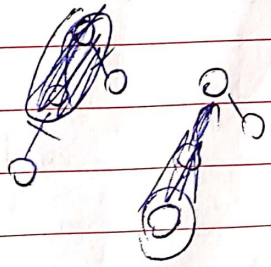


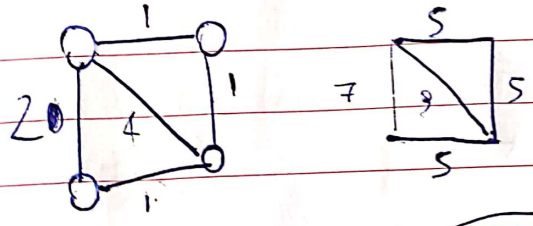


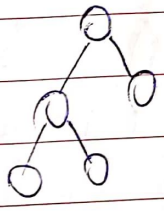


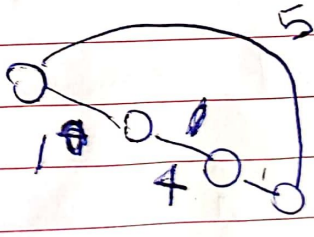












WEEK 2 L1

KRUSCAL'S MST ALGORITHM

- Uses union-find data structure

Cut Property Reminder: if e is the cheapest edge crossing any cut (A,B) of $G(V,E)$ then ~~e belongs to T~~ $e \in T^*$ (the mst)

- Sort edges in order of increasing cost (rename edges $1,2,3,\ldots,m$ so that $cost[1] < cost[2] < \ldots < cost[m]$)
- $T = \emptyset$
- For $i = 1$ to m:
  - if $T \cup \{i\}$ has no cycles
    - add i to T
- return T

WEEK 2 L2: Proving Correctness of Kruscal

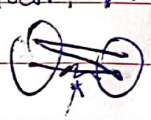
Theorem: Kruscal's algo is correct

① No cycles, we check in loop

② T^* is connected (duh you're adding edges with one vertex outside of X)

③ more rigorous proof of ②:

Fix cut (A,B), since G connected, there's one edge at least. Kruscal will include first edge crossing (A,B) that it sees.



④ every edge in T^* is justified by the Cut Property (it really is an MST)

④ In the adding step, since $T \cup e(u,v)$ has no cycle, there is no u,v path in T ~~therefore~~ ② there exists a cut A,B with $u \in A$, $v \in B$

③ by T^* being connected, no edges crossing (A,B) were previously considered by Kruscal so (u,v) is the cheap

WEEK 2 L2

KRUSCAL IMPLEMENTATIONS

why not m? m at most n^2 $\log m \leq 2\log(n)$

NAIVE / UNION FIND

- Sort edges in order of increasing cost $O(m \log(n))$
- $T = \emptyset$
- for $i = 1$ to m → $O(m)$ iterations
  - if $T \cup \{i\}$ has no cycles
    - add i to T
- return T

→ How much work do we do every iteration?

- $O(n)$? Checking for a cycle boils down to finding if there is a (v,w) path in T. DFS or BFS, $O(n)$, if you find a path you're done.

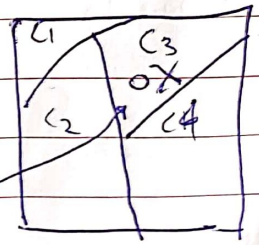
~~Summing to $O(mn)$~~ So $O(mn) + O(m\log(n)) = O(mn)$

CAN WE DO BETTER?

Plan: data structure for $O(1)$-time cycle checks $\Rightarrow O(m\log(n)) + O(m) = O(m\log(n))$ time

THE UNION-FIND DATA STRUCTURE

Raison d'être of a union-find data structure: maintain partition of a set of objects



If we hand it a value, we want to know what partition that value is in

find(x): return partition x belongs to

Union(C_i, C_j): fuse groups containing C_i and C_j into a single one

It is useful for Kruskal's algorithm:

objects = vertices
adding new edge (u,v) to T $\Leftrightarrow$ fusing partitions (components) of u, v

WEEK 2 C3

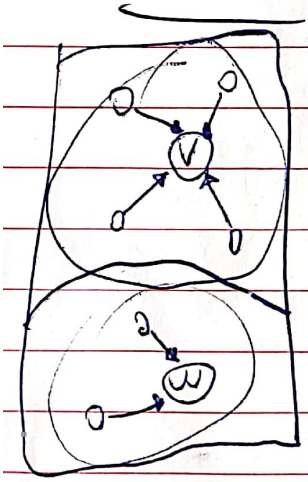
BASICS OF UNION-FIND

Idea #1 - maintain one linked structure per connected component of (V,T)

- Each component has an arbitrary leader vertex

Invariant: each vertex points to the leader of its component

"name" of a component is inherited from leader vertex



Key point: given edge (u,v), can check if u,v is already in the same component in $O(1)$ time:
$\{\text{Find}(u) = \text{Find}(v)\}$?

HOW TO MAINTAIN THE INVARIANT (when fusing)

Note: when new edge (u,v) added to T, connected components of u and v merge

Question: At most how many $O(n)$ leader pointer updates for n vertices
wait. we could just change w to v?
points

We're happy already with $O(n \log n)$ work. Every time two sets merge size at most doubles and it can happen $< \log n$ times

WEEK 2 @ L5

CLUSTERING

MSTs have multiple greedy algorithms which are correct.

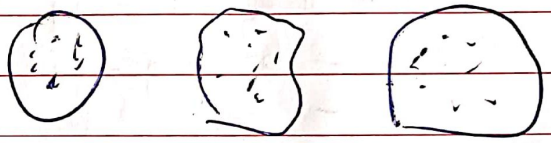
Informal Goal: cluster connected objects into coherent groups

Assumptions: ① as input, given a (dis)similarity measure, e distance $d(p,q)$ between each pair of objects

② $d(p,q) = d(q,p)$ (symmetric)

Examples: Euclidean distance, genome similarity, etc.

Goal: same Cluster $\Leftrightarrow$ "nearby"



MAX-SPACING K-CLUSTERING

Assume: you want K clusters in total

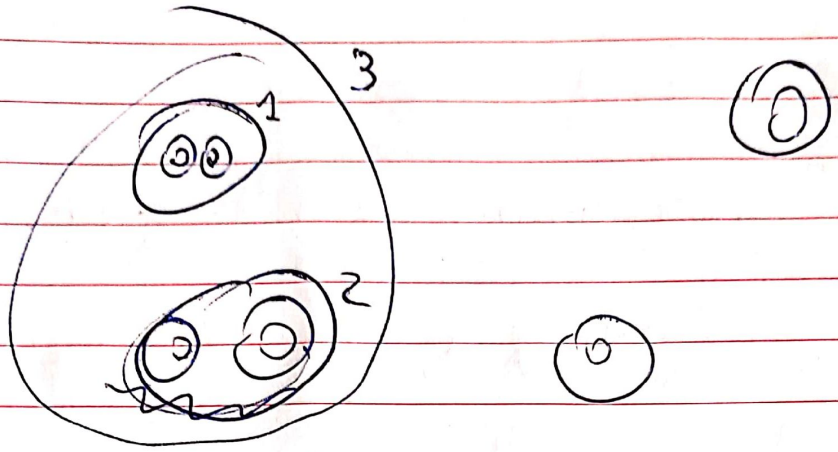
Call points p & q separated if they are assigned to different clusters.

Definition: The spacing of a K-clustering is the minimum distance after which two objects end up in different clusters. The bigger, the less clusters there are.

~~Problem statement~~

Problem Statement: Given a distance measure d and K ~~compute them~~ compute the K clustering with maximum spacing.

A GREEDY ALGORITHM



How to decrease # clusters? 1, 2, 3, keep increasing distances

Pseudocode:

- Initially, each point in separate clusters
- Repeat until only K clusters:
  - let p, q = closest pair of separated points
  - merge clusters containing p & q into a single one

Reminds you of Kruscal's, but aborted early

Objects are the vertices, distances are like edge costs.

This type of fusing clusters is called single link clustering.

WEEK 2 L6

CORRECTNESS CLAIM

Theorem: single-link clustering finds the max-spacing k-clustering

Proof: Let $C_1, \ldots, C_k$ = greedy clustering with spacing S

Let $\hat{C}_1, \ldots, \hat{C}_k$ = arbitrary other clustering

Need to Show: spacing of $\hat{C}_1, \ldots, \hat{C}_k$ is $\leq S$

CORRECTNESS PROOF

Case 1: Our $\hat{C}_i = C_i$, spacing $= S \Rightarrow \leq S$

Case 2: Otherwise, can find a point pair p, q such that:

(A) p, q in the same greedy cluster C_i

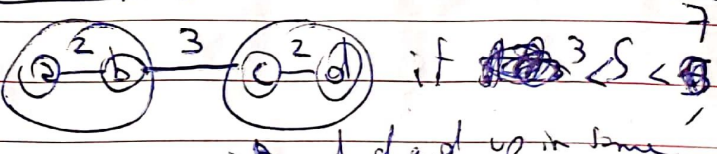
(B) p, q in different clusters $\hat{C}_i, \hat{C}_j$

Property of greedy algorithm: if two points x, y were merged, $d(x, y) \leq S$ (since S is ever increasing)

Easy Case: if p, q were merged, $S \geq d(p, q) \geq$ the spacing in $\hat{C}_1, \ldots, \hat{C}_k$

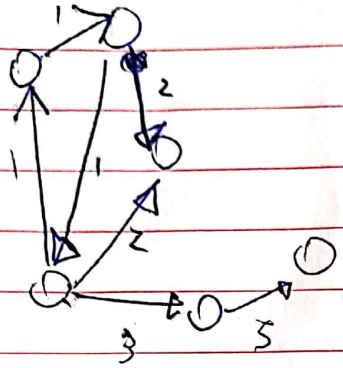
Tricky Case: p, q undirectly merged

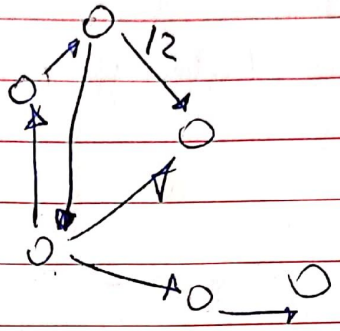
Since $p \in \hat{C}_i$ and $q \notin \hat{C}_i$ $\exists$ a consecutive pair a_j, a_{j+1} with $a_j \in \hat{C}_i$, $a_{j+1} \notin \hat{C}_i$

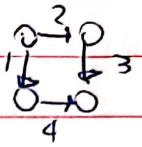


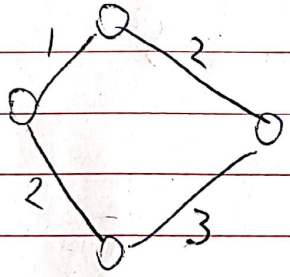
if $3 < S < 7$, a & d end up in same group without ever having been merged directly

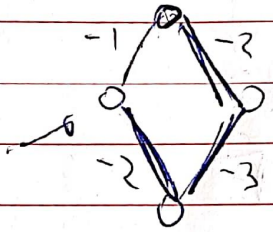
Problem tribbling

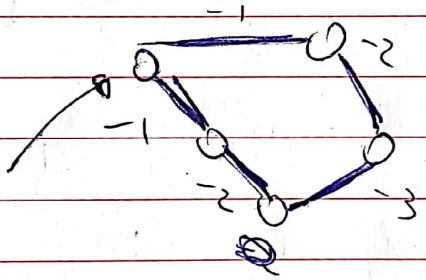


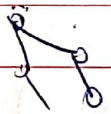


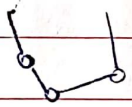


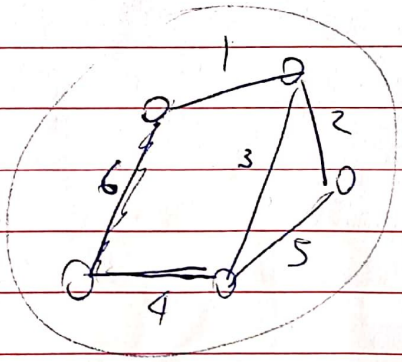


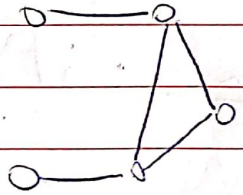


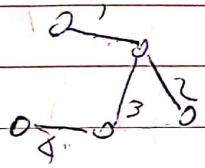


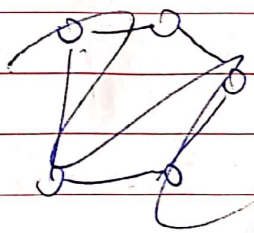


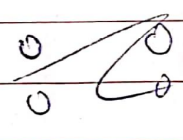


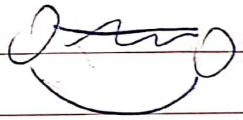




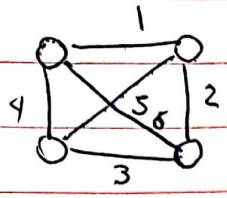


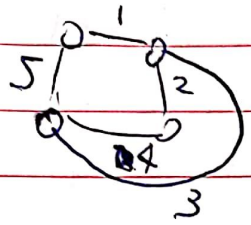


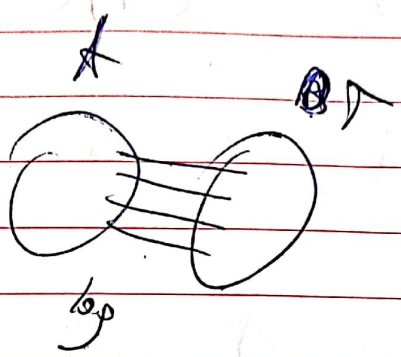


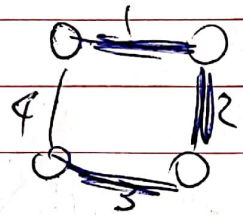


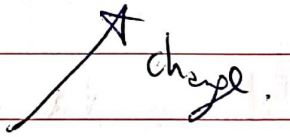
~~edge~~











then $O(1)$ to ~~find~~ lookup

WEEK 3 L1

BINARY CODES

Binary code: maps each character of an alphabet Σ to a binary string.

Example: $\Sigma = \{a - z\}$, $\overbrace{00000}^{a}, \overbrace{00001}^{b}$ .....

AMBIGUITY

Suppose $\Sigma = \{A, B, C, D\}$. Fixed length encoding is $\{0, 01, 10, 1\}$

Can't be $\{0, 01, 10, 1\}$! 001 could be AB or AAD

PREFIX-FREE CODES

Problem: with variable length codes it's never clear where one character ends and another begins

Solution: prefix-free codes - make sure that for every pair $(i, j) \in \Sigma$, neither of the encodings is a prefix of the other)

Example: $\Sigma \{0, 10, 110, 111\}$

BUT WHY WOULD WE WANT TO DO THIS?

		Frequency		
Suppose	A	60%	00	0
	B	25%	01	10
	C	10%	10	110
	D	5%	11	111

→ Less space to store, on average 1.55 bits

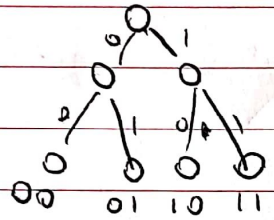
WEEK 3 L2

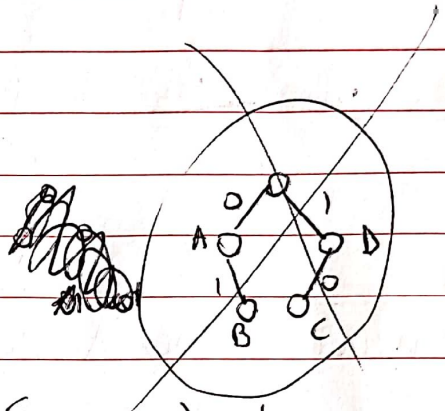
HUFFMAN CODES, PROBLEM DEFINITION

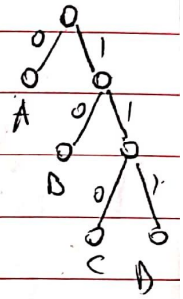
CODES AS TREES

Goal: best binary prefix-free encoding
Useful to think of it as a binary tree.

Examples:







$\{00, 01, 10, 11\} = \{A, B, C, D\}$!= NOT $\{0, 01, 10, 1\}$, ambiguous

$\{0, 10, 110, 111\}$

You need i endpoints/leaves to represent i values

To decode: repeatedly follow path from the root until you hit a leaf (so I assume $\log(n)$ time? $\leftarrow$ height)

Problem definition

Input: probability p_i for each character $i \in \Sigma$

Notation: if T is the binary tree with leaves $\Leftrightarrow$ symbols of Σ,

$$L(T) = \sum_{i \in \Sigma} p_i \cdot (\text{depth of } i \text{ in } T)$$

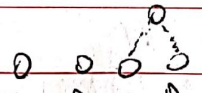
Output: T with minimum $L(T)$

A GREEDY ALGORITHM

Question: best approach to building T?

Natural idea: (1) Partition Σ (top-down) into Σ_1, Σ_2 each with $\simeq$ of total frequency.
(2) put one side on the left the other on the right

Huffman's (optimal) idea: build tree bottom up using successive merges



→ merge ~~least~~ 2 lowest frequencies?

WHAT GREEDY APPROACH TO USE?

Question: which pair of symbols is "safe" to merge?

Observation: final encoding length of $i \in \Sigma$ = # of mergers its subtree endures

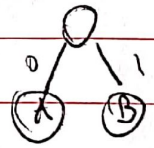
Reuse? How?

Suppose: 1st iteration of algorithm merges symbols a & b p_a p_b

Idea: Replace symbols a, b by meta-symbol ab with $p_{ab} = p_a + p_b$

- Proceed right!

Huffman's Algorithm(Σ):

- If $|\Sigma| = 2$, return 

- Let $a, b \in \Sigma$ have the smallest frequencies
- Let $\Sigma' = \Sigma$ with a, b replaced by (ab), with $p(ab) = p(a) + p(b)$
- Recursively compute T' (for Σ')
- Extend T' (with leaves $\leftrightarrow \Sigma'$) to tree T with leaves $\leftrightarrow \Sigma$ by splitting leaf ab back into a and b.
- Return T

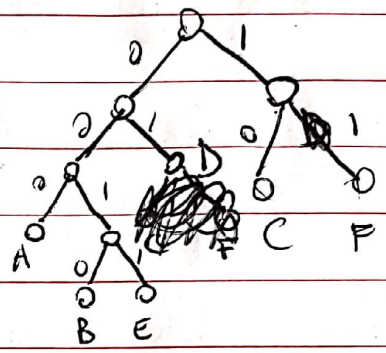
Input A B C D E F
3 2 6 8 2 6 (if you don't like to use fractions)

Step 1 : Merge B and E : A BE CD F
3 4 6 8 6

Step 2 : Merge A and BE → A(BE) CD F
7 6 8 6

Step 3 : Merge C and F → (A(BE)) (CF) (D)
7 12 8

Step 4 : Merge A(BE) and D → (A(BE))D (CF)



A : 000
B : 0010
C : 10
D : 01
E : 0011
F : 11

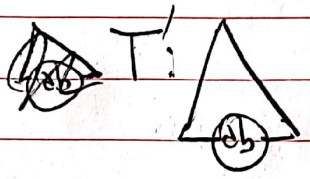
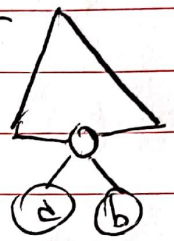
WEEK 3 CS16

PROOF OF CORRECTNESS OF HUFFMAN'S

We prove it by induction on $n = |\Sigma|$

Base case: when there are 2 elements, algorithm outputs the optimal tree.

Inductive hypothesis: Algorithm does solve smaller subproblems optimally.

Consider T':  ⟷ T: 

a, b = symbols with smallest frequencies

$$p(ab) = p(a) + p(b)$$

$$L(T) - L(T') = \text{offset} + p[a\text{'s depth in } T] - p(a) + p[b\text{'s depth in } T] - p(b)$$
$$- \text{offset} + p(a) + p(b) = p(a) + p(b)$$

$$L(T) - L(T') = p_a[a\text{'s depth in } T] + p_b[b\text{'s depth in } T] - p(ab) \dots$$
$$= p_a(d+1) + p_b(d+1) - (p_a + p_b)d = p_a + p_b$$

NOTES ON RUNNING TIME

Naive implementation is $O(n^2)$

A heap is perfect for this, (extract min twice + reinsert new meta symbol) $\cdot n$:

- $O(n \log(n))$

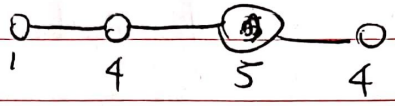
BUT WAIT 2 QUEUES DOES IT IN SORT + $O(n)$ time

WEEK 3 L7

DYNAMIC PROGRAMMING

PROBLEM: FINDING WIS, WEIGHT INDEPENDENT SETS

Input: a path graph $G(V, E)$, non-negative ~~edges~~ vertex weights



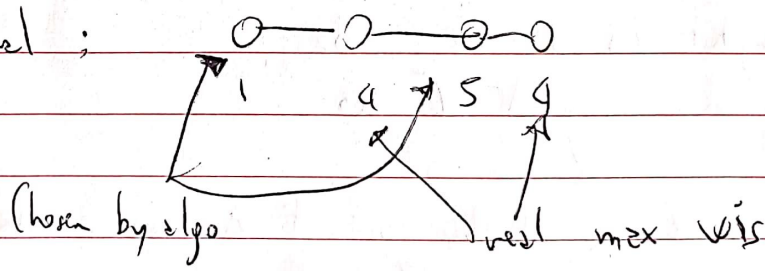
Desired output is a subset of non adjacent vertices of maximum total weight.

Brute Force is unfeasible as it requires exponential time

GREEDY APPROACH

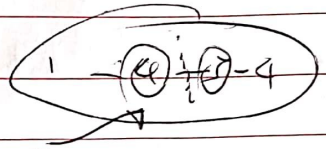
Idea: Iteratively choose max weight vertex not adjacent to any previously chosen vertex

Non optimal:



D&C APPROACH

Idea: split in two and then combine



but recursive subsolutions might conflict

WEEK 3 L8

THE OPTIMAL SUBSTRUCTURE OF WIS

Last lecture we tried to guess a G_i quickly, but often to find a good one it is necessary to reason about the structure that an optimal solution would have.

We can narrow down the set we have to brute force through

Notation: let $S \subseteq V$ be a max weight (IS)
let v_n = last vertex of a path.

Case 1: $v_n \notin S$. $G' = G - \{v_n\}$

After deleting $\{v_n\}$, S is still the mwis in G'

Case 2: if $v_n \in S$, $S - \{v_n\}$ is the mwis in G'' (v_n deleted too)

Upshot: A max weight IS must be either:

- a max weight IS of G' or
- v_n + a max weight of G''

Corollary: if we knew whether or not v_n was in the mwis, we could recursively compute the max weight IS of G' and be done.

Could we try both possibilities and return the better solution?

YES

WEEK 3 L8

WHAT TIME WOULD THE RECURSIVE ALGO RUN IN?

$\Theta(n)$! worst case scenario each iteration removes one vertex, best case two.

But it runs n times, sometimes on the same subproblems more than once.

To eliminate redundancy, cache every subproblem in a hashtable to get $O(1)$-time lookup later on.

Or better: bottom-up iterative algorithm: instead of pluck**up**, add!

G_i = 1st (i) vertices of G

Populate left to right with A[i] = value of the MWIS of G_i

$$A[i] = \max\left(A[i-1],\ A[i-2] + w_i\right)$$

Runtime is $O(n)$!

OPTIMAL

L10

We can trace back through the array to reconstruct the optimal solution.

Let A = filled in array

0	1	2	3	...	n
0	4	4	7	...	184

Runs in $O(n)$

- Let $S = \emptyset$
- While $i \geq 1$ (from right to left)
  - If $A[i-1] \geq A[i-2] + w_i$
    - decrease i by 1
  - Else
    - Else
    - Add v_i to S, $i -= 2$
- Return S

WEEK 3 L11

PRINCIPLES OF DYNAMIC PROGRAMMING

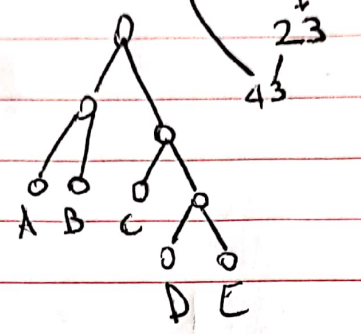
(1) Identify a small number of subproblems (example: MWIS for $G_i = 0, 1, \ldots, n$ — not 2^n!)

(2) Can quickly solve larger subproblems given the solutions to smaller subproblems (usually through recurrence)

(3) After solving all subproblems, quickly compute the final solution.

Problem Scribblings

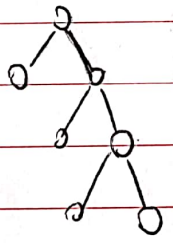
A	B	C	D	E
32	25	20	18	5



$32 \cdot 2 + 25 \cdot 2 + 20 \cdot 2 + 18 \cdot 3 + 5 \cdot 3 =$

$= 64 + 50 + 40 + 54 + 15 =$

$223 \cdot \frac{1000}{100} = 2230$



n-1

0.33, 0.33, 0.33

0,4, 0,45, 0,

WEEK 4 L1

THE KNAPSACK PROBLEM

- Input: n items each with a value:
  - value v_i (non-negative)
  - size w_i.

As much value as possible, size limited set $S \subseteq \{1, 2, 3, \ldots, n\}$

DEVELOPING A GA FROM OPT. SUBSTRUCTURE

Let S be a max-value solution to an instance of Knapsack.
Delete item n.

Case 1: item $n \notin S$, S still optimal for first $n-1$ items

Case 2: item $n \in S$, S still optimal for first $n-1$ items with capacity decreased by w_n

WEEK 4 L2

RESUMING FROM LAST TIME

Notation: $V_{i,x}$ = value of solution that uses first i items, size $\leq x$

$$V_{i,x} = \max\left\{V_{(i-1),x},\ v_i + V_{(i-1),x-w_i}\right\}$$

+ Recurrence

- Let A = 2D Array
- Initialize $A[0,x] = 0$ for $x = 0, 1, \ldots, W$
- For $i = 1, 2, \ldots, n$
  - For $x = 0, 1, \ldots, W$
    - $A[i,x] = \max\left\{A[i-1,x],\ A[i-1, x-w_i] + v_i\right\}$

Available for O(1) time lookup from hashtable

Return $A[n, W]$

RUNNING TIME IS $O(nW)$

Example:

$W = 6$

$v_1 = 3,\ w_1 = 4$

$v_2 = 2,\ w_2 = 3$

$v_3 = 4,\ w_3 = 2$

$v_4 = 4,\ w_3 = 3$

prefer to skip number 2

W \ i	0	1	2	3	4
6	0	3	3	7	8
5	0	3	3	6	8
4	0	3	3	4	4
3	0	0	2	4	4
2	0	0	0	4	0
1	0	0	0	0	0
0	0	0	0	0	0

Then: Start from (4,6) = 8.

Remove item 4. $6 - w_4 = 3$.

What now?

Remove item 3, $3 - w_3 = 1$

0, 0, 0

Optimal: 8, {item 3, item 4}

WEEK 4 L4

SEQUENCE ALIGNMENT

Example: AGGGCT
AGG-CA

Needleman-Wunsch score, total penalty = $\alpha_{gap} + \alpha_{xy}$

Input: two strings over some alphabet Σ, penalties α_{gap} and $\alpha_{mismatch}$

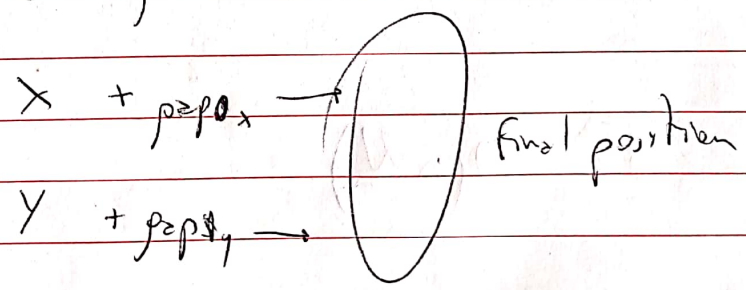
Try different gaps. Insert gaps to equalize lengths of the strings

Output: Alignment with minimum penalty

A DP APPROACH

Identify subproblems from optimal substructure

An optimal alignment:



Either $gap_x = gap_y = 0$ or gap_x = ith or gap_y = sth

1. x_m & y_n
2. x_m & gap
3. y_n & gap

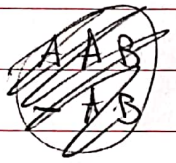
WEEK 4 L5

RECURSION APPLYING

P_{ij} = penalty of optimal alignment of X_i and Y_j

Recurrence: for all $i = 1, \ldots, m$ and $j = 1, \ldots, m$

$$P_{ij} = \min \begin{cases} (1)\ \alpha_{x_i y_j} + P_{i-1, j-1} \\ (2)\ \alpha_{gap} + P_{i-1, j} \\ (3)\ \alpha_{gap} + P_{i, j-1} \end{cases}$$



$P_{i,0}$ and $P_{0,j}$ have penalty $i \cdot \alpha_{gap}$.

ALGORITHM PSEUDO CODE (Similar to knapsack)

A = 2D Array

$A[i,0] = A[0,i] = i \cdot \alpha_{gap}, \forall i \geq 0$

For i = 1 to m:
  For j = 1 to n:

$$A[i,j] = \min \begin{cases} A[i-1, j-1] + \alpha_{x_i y_j} \\ A[i-1, j] + \alpha_{gap} \\ A[i, j-1] + \alpha_{gap} \end{cases}$$

All available for O(1)-time lookup.

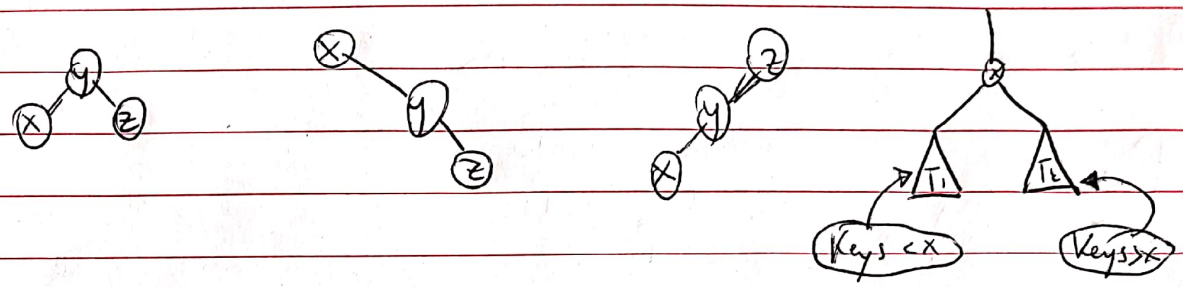
Running time is $O(mn)$

YOU CAN RECONSTRUCT THE SOLUTION in $O(m+n)$. From A[i,j]
- IF A[i,j] filled by (1), match x_i & y_j and go to A[i-1, j-1]
- IF A[i,j] filled by (2), match x_i with a gap and go to A[i-1, j]
- IF A[i,j] filled by (3), match y_j with a gap and go to A[i, j-1]

WEEK 4 L6

OPTIMAL BINARY SEARCH TREES

For a given set of keys there are lots of valid search trees



Question: What is the "best" search tree for a given set of keys?

A good answer: balanced search tree, like a red-black tree.

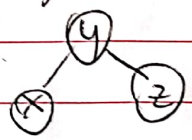
An RBT has height $\leq \log(n)$ so $\Theta(\text{height}) = \Theta(\log(n))$

EXPLOITING NON-UNIFORMITY

Question: Suppose we have keys $x < y < z$ and we know that

80% of searches are for x
10% of searches are for y
10% of searches are for z

Answer: We want to optimize the search time:



Same as Huffman codes.
The cost changes because we're looking for min depth instead of E(the whole weighted sum)
Also constraint symbols only at leaves → search tree property

WEEK 4 L7

OPTIMAL SUBSTRUCTURE OF BSTs

Input: frequencies $p_1, p_2, \ldots, p_n$ for items $1, 2, \ldots, n$.

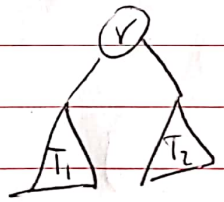
Goal: Compute a valid search tree that minimizes the ~~weighted average search time~~

Ideally, we'd want the ~~most~~ least frequently searched items at the bottom

Both bottom up and top down are faulty: ~~choosing~~ a chosen root may not always be the median and fuck everything up.

What if we calculated it or tried all possibilities?

For ~~a generic~~ an optimal BST:



T_1 and T_2 are optimal for respectively items $\{1, \ldots, r-1\}$ and $\{r+1, \ldots, n\}$

WEEK 4 & 8

OPTIMAL SUBSTRUCTURE LEMMA

RELEVANT SUBPROBLEMS

Let $\{1, \ldots, n\}$ = original items. For which subsets $S \subseteq \{1, \ldots, n\}$ might we need to compute the optimal BST for S?

$$S = \{i, i+1, \ldots, j-1, j\} \text{ for every } i \leq j$$

So we'll recurse on $A[i,j]$:

THE ALGORITHM

Important: solve smallest subproblems first.

DPBST (frequencies[ ]):

- A = 2D Array with $A(i,j)$ representing opt BST value for items $\{i, \ldots, j\}$
  - $j - i$
- For $s = 0$ to $n-1$:
  - For $i = 1$ to n
    - $A[i, i+s] = \min_{r = i, \ldots, i+s} \left(\sum_{k=i}^{i+s} p_k + A[i, r-1] + A[r+1, i+s] \right)$
      - $r = j$

$O(n^2)$ subproblems, $O(j-i)$ times, $O(n^3)$ overall, which is slow

but we can optimize to $O(n^2)$ if we really want to do

(By remembering previous subproblems and avoiding repetition)

PROBLEM SCRIBBLING

{1, 2, 1, 5, 3}

3 3 6 7 5

1, 2 3

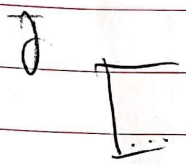


Table will still be filled okay

1 2 3 4 5 6 7

5, 40, 8, 4, 10, 10, 23

Runs in O(mn) time

O(mn?) like subsets?

MWIS only stores 2 values ever O(1)

SA ~~was table 2~~ 2 columns O(n)

BFS needs the whole table