

Algorithms - Spring '25

Greedy Algs:
File Access
Scheduling



Dynamic Programming vs Greedy

Dyn. pro: try all possibilities
→ but intelligently!

In greedy algorithms, we avoid building all possibilities.

How?

- Some part of the problem's structure lets us pick a local "best" and have it lead to a global best.

But - be careful!

Students often design a greedy strategy, but don't check that it yields the best global one.

Overall greedy strategy:

- (• Assume optimal is different than greedy)
 - Find the "first" place they differ. ~~differ~~
 - Argue that we can exchange the two without making optimal worse.
- ⇒ there is no "first place" where they must differ, so greedy, in fact is an optimal solution.

First example in the book:

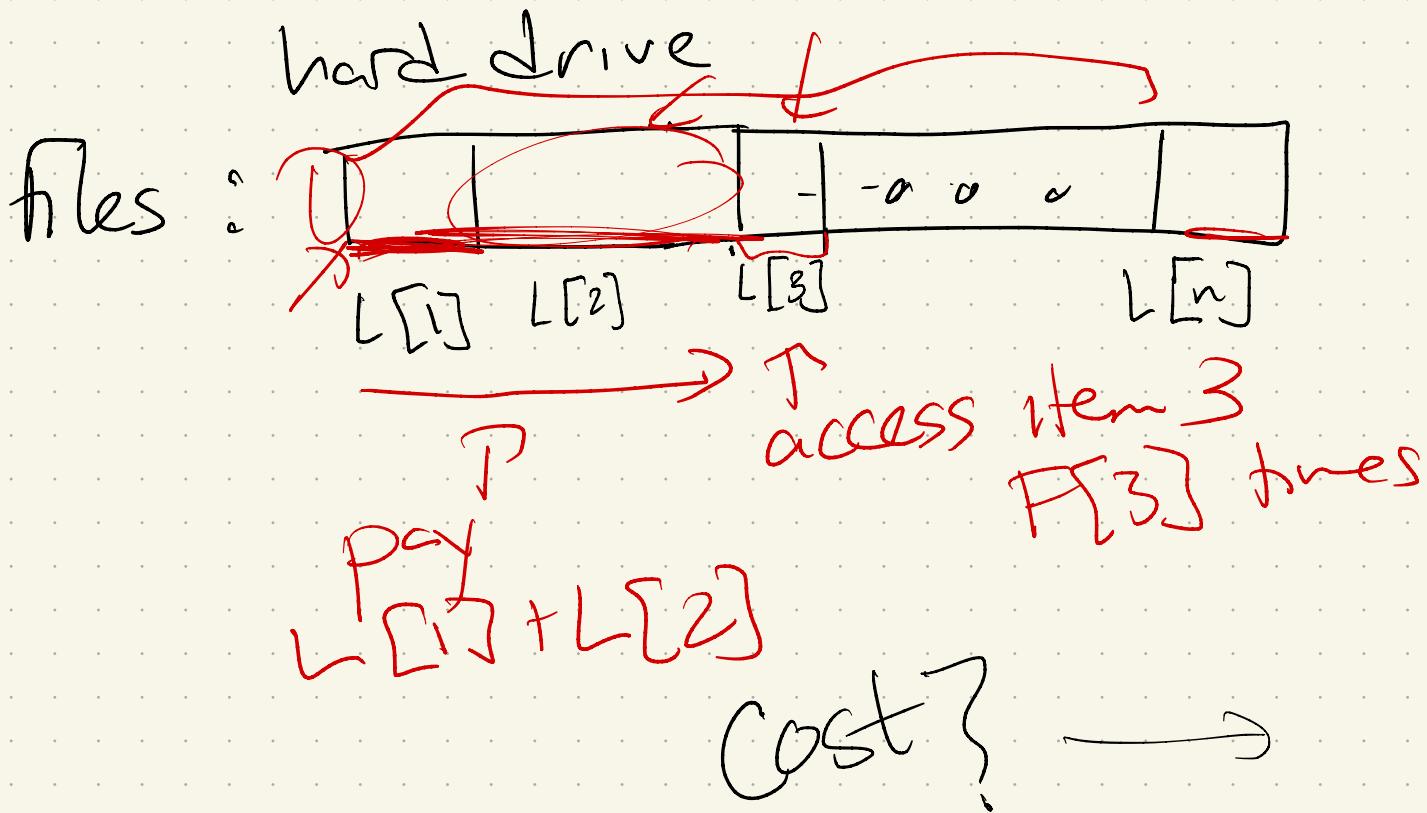
Storing files on tape.

Input: n files, each with
a length & # times
it will be accessed:

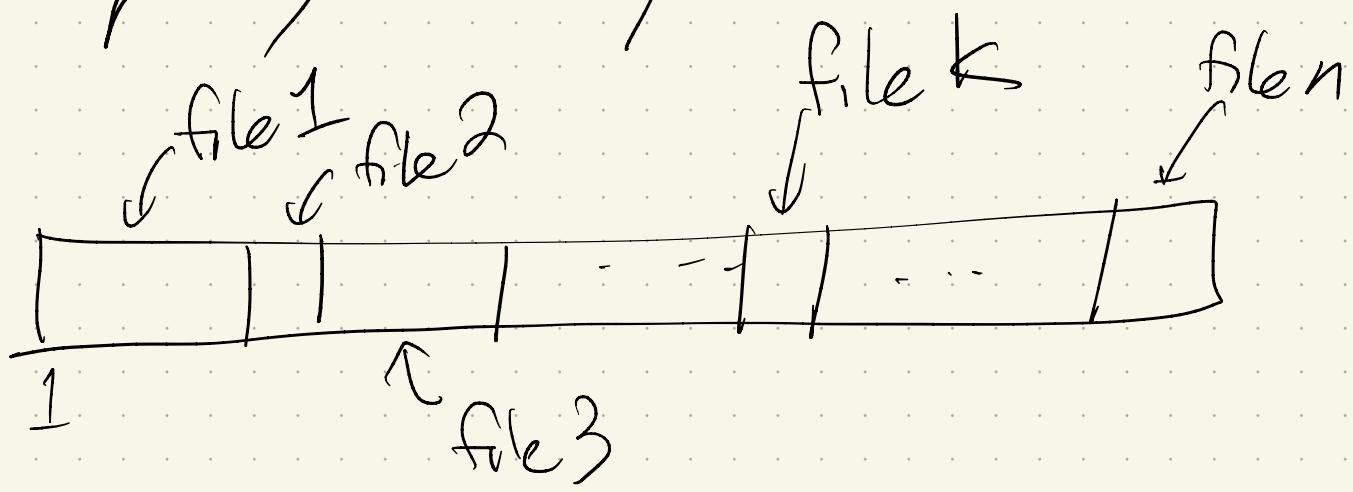
$L[1..n] \rightarrow F[1..n]$

length frequency

Goal: Minimize access time:



If equally likely:



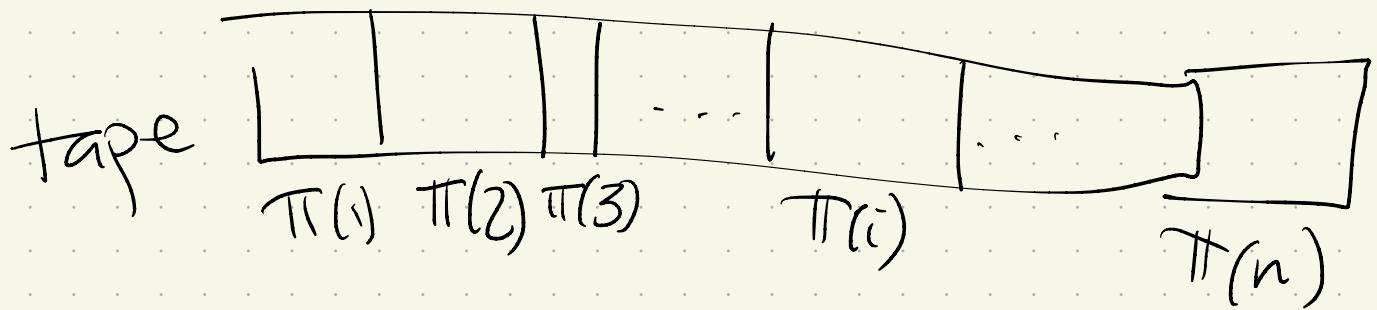
Cost to access file k:

If equally likely to access
any file:

$$E[\text{cost}] = \sum_{k=1}^n (\text{prob of file } k) \cdot (\text{cost of file } k)$$

=

Files: We can re-order:
Permutation π



Cost to access k^{th} one:

And: how many times do I
access it?

Total:

$$\Sigma \text{cost}(\pi) = \sum_{k=1}^n \left(F[\pi(k)] \cdot \sum_{i=1}^k L[\pi(i)] \right) = \sum_{k=1}^n \sum_{i=1}^k (F[\pi(k)] \cdot L[\pi(i)]).$$

How to be greedy?
(Not immediately clear!)

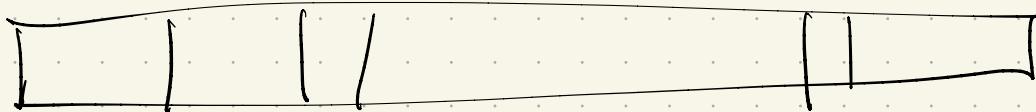
Try smallest first:

Try most frequent first:

Lemmas: Sort by $\frac{L[i]}{F[i]}$

& will get optimal schedule.

Pf: Suppose we sort:

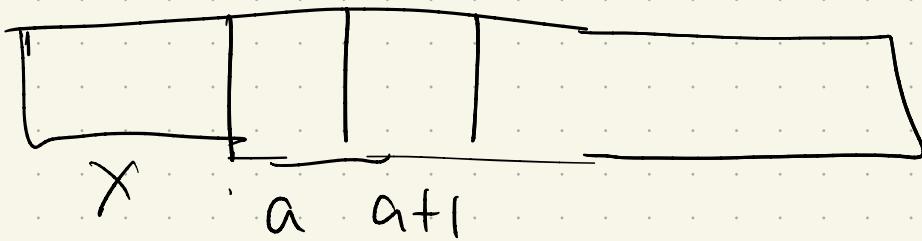


$$\forall i, \frac{L[i]}{F[i]} \leq \frac{L[i+1]}{F[i+1]}$$

Suppose this is not optimal.
What does that mean?

Well, OPT must be different
so \exists out of order pair.

OPT



with $\frac{L[a]}{F[a]} > \frac{L[a+1]}{F[a+1]}$

If OPT, must beat our
"sorted" solution.

What if we swap a & a+1?

Before:

After:

Difference?

Pf (cont):

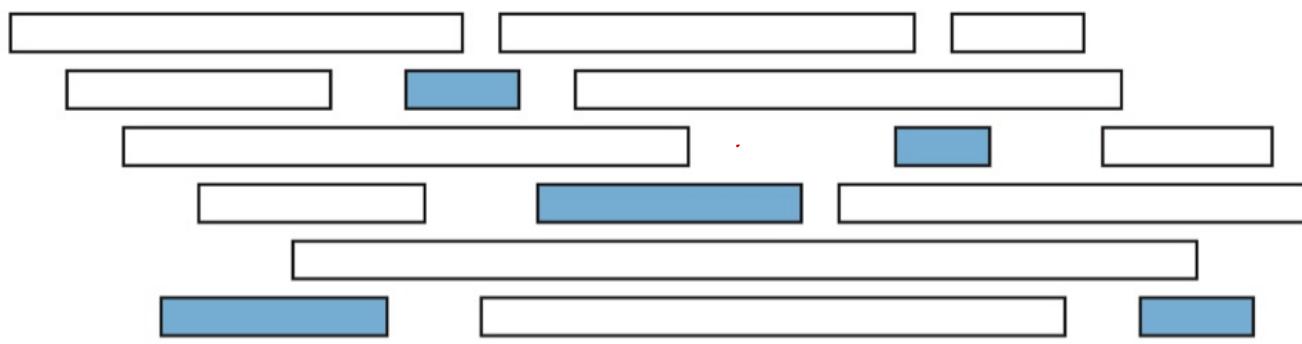
18: algorithm

- Calculate $\frac{L[a]}{F[a]}$ for all a .
- Sort & permute order of jobs to match.

Runtime:

Problem: Interval Scheduling

Given a set of events (ie intervals, with a start and end time), select as many as possible so that no 2 overlap.



A maximal conflict-free schedule for a set of classes.

More formally:

Two arrays

$S[1..n]$:

$F[1..n]$:

Goal: A subset $X \subseteq \{1..n\}$ as big as possible s.t $\forall i$

$$F[i] \leq S[i+1]$$

How would we formalize a dynamic programming approach?

Recursive Structure:

Intuition for greedy:

Consider what might be
a good first one to
choose.

Ideas?

Key intuition:

If it finishes as early as possible, we can fit more things in!

So - strategy:

The code:

GREEDYSCHEDULE($S[1..n], F[1..n]$):

sort F and permute S to match

$count \leftarrow 1$

$X[count] \leftarrow 1$

for $i \leftarrow 2$ to n

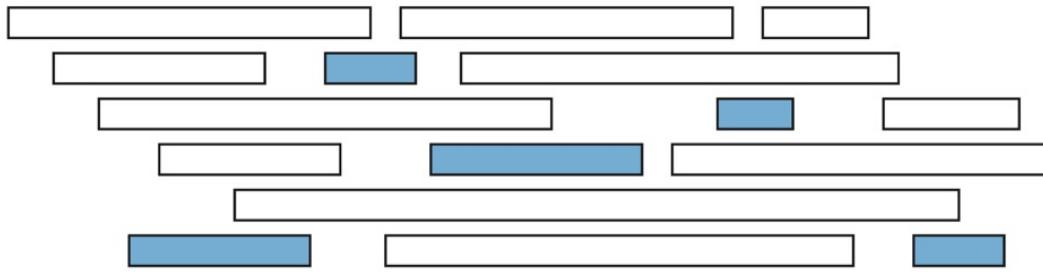
 if $S[i] > F[X[count]]$

$count \leftarrow count + 1$

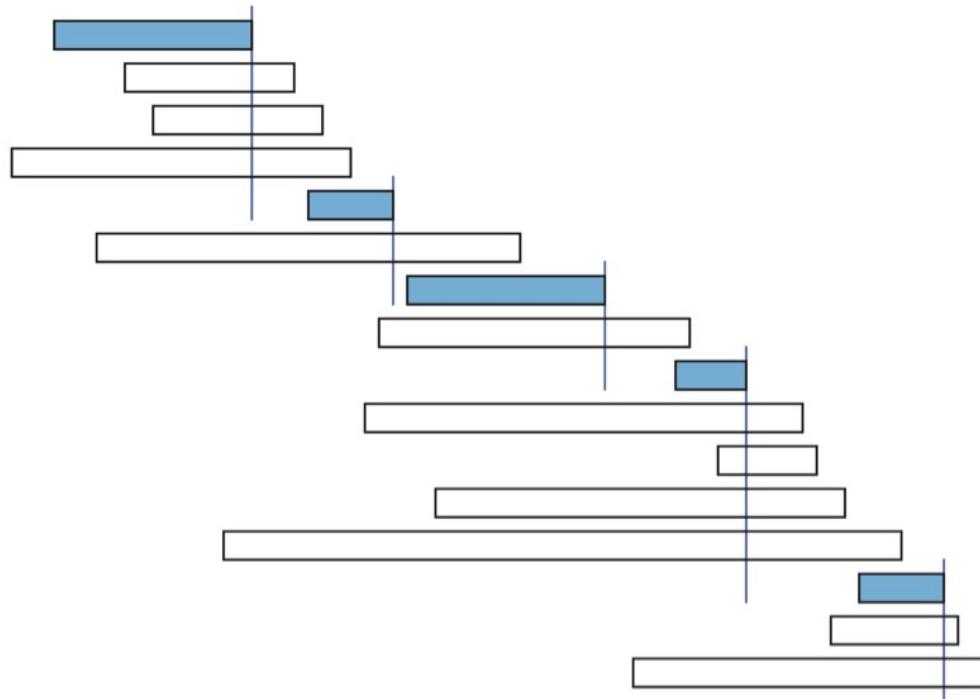
$X[count] \leftarrow i$

return $X[1..count]$

Picture:



A maximal conflict-free schedule for a set of classes.



The same classes sorted by finish times and the greedy schedule.

Correctness:

Why does this work?

Note: No longer trying all possibilities or relying on optimal substructure!

So we need to be very careful on our proofs.

(Clearly, intuition can be wrong!)

Lemma: We may assume the optimal schedule includes the class that finishes first.

Pf:

Thm: The greedy schedule is optimal.

Pf: Suppose not.

Then \exists an optimal schedule that has more intervals than the greedy one.

Consider first time they differ:

Greedy: $g_1 \ g_2 \dots g_i \dots g_k$

OPT: $o_1 \ o_2 \dots o_i \dots o_l$

Example: Huffman trees

Many of you saw this in data structures.

Why?

- cool use of trees
- non-trivial use of other data structure

Really - it's greedy!

Goal: Minimize Cost

↳ here, minimize total length of encoded message:

Input: Frequency counts
 $f[1..n]$

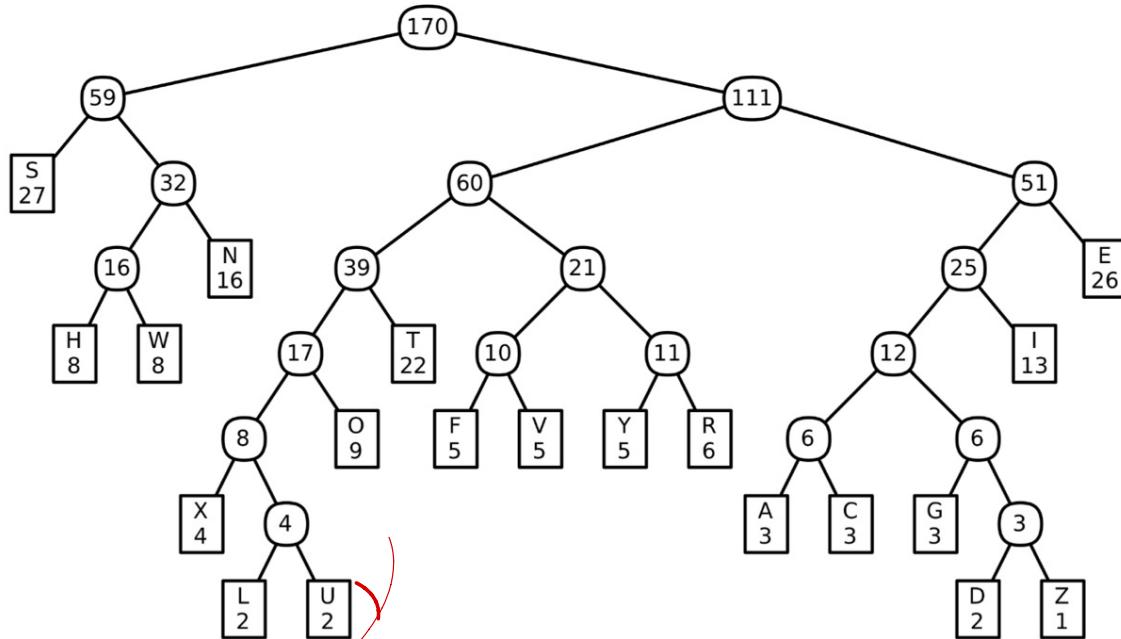
Compute:

$$\text{cost}(T) = \sum_{i=1}^n f[i] \cdot \text{depth}(i)$$

Strategy:

- Pick 2 least common letters & make them leaves
- "Merge them": remove letters, & add a new letter with sum of their frequencies
- Reverse!
 ↑ well, a bit imprecise...

In the end, get a tree with letters at the leaves:



A Huffman code for Lee Sallows' self-descriptive sentence; the numbers are frequencies for merged characters

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

If we use this code, the encoded message starts like this:

1001 0100 1101 00 00 111 011 1001 111 011 110001 111 110001 10001 011 1001 110000 ...
 T H I S S E N T E N C E C E C O N T A ...

Implementation: use priority queue

BUILDHUFFMAN($f[1..n]$):

for $i \leftarrow 1$ to n

$L[i] \leftarrow 0; R[i] \leftarrow 0$

INSERT($i, f[i]$)

for $i \leftarrow n$ to $2n - 1$

$x \leftarrow \text{EXTRACTMIN}()$

$y \leftarrow \text{EXTRACTMIN}()$

$f[i] \leftarrow f[x] + f[y]$

$L[i] \leftarrow x; R[i] \leftarrow y$

$P[x] \leftarrow i; P[y] \leftarrow i$

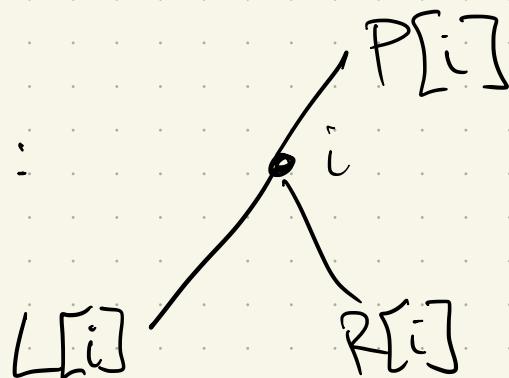
INSERT($i, f[i]$)

$P[2n - 1] \leftarrow 0$

3 arrays: L, R, P

to encode the tree

node i :



So:

BANANA

index: 1 2 3 4 \nwarrow^n

letters: B A N EoM

Freq: f,

BUILDHUFFMAN($f[1..n]$):

for $i \leftarrow 1$ to n

$L[i] \leftarrow 0; R[i] \leftarrow 0$

INSERT($i, f[i]$)

for $i \leftarrow n$ to $2n - 1$

$x \leftarrow \text{EXTRACTMIN}()$

$y \leftarrow \text{EXTRACTMIN}()$

$f[i] \leftarrow f[x] + f[y]$

$L[i] \leftarrow x; R[i] \leftarrow y$

$P[x] \leftarrow i; P[y] \leftarrow i$

INSERT($i, f[i]$)

$P[2n - 1] \leftarrow 0$

L: 1 2 3 4 5 6 7

R:

P:

Runtime?

BUILDHUFFMAN($f[1..n]$):

for $i \leftarrow 1$ to n

$L[i] \leftarrow 0; R[i] \leftarrow 0$

INSERT($i, f[i]$)

for $i \leftarrow n$ to $2n - 1$

$x \leftarrow \text{EXTRACTMIN}()$

$y \leftarrow \text{EXTRACTMIN}()$

$f[i] \leftarrow f[x] + f[y]$

$L[i] \leftarrow x; R[i] \leftarrow y$

$P[x] \leftarrow i; P[y] \leftarrow i$

INSERT($i, f[i]$)

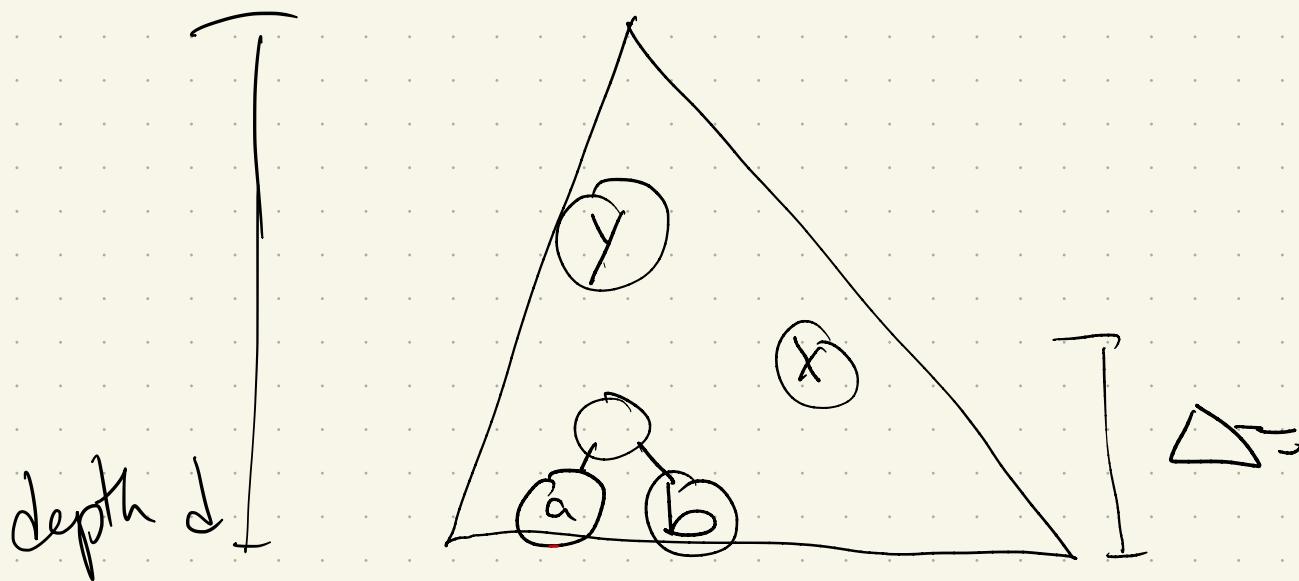
$P[2n - 1] \leftarrow 0$

Correctness :

1st Lemma: There is an optimal prefix tree where the two least common letters are siblings at the largest depth.

Pf: Suppose not. Then

optimal tree T has some depth d , but at least 2 common letters $x + y$ are not at that depth.



Note some other letters $a + b$ are deepest

Pf cont:

Know $f[x] \leq f[a]$,

but $\text{depth}(a) = \text{depth}(x) + \Delta$

+ recall that:

$$\text{cost}(T) = \sum_{i=1}^n f[i] \cdot \text{depth}(i)$$

Build T' :

Thm: Huffman trees are optimal.

Pf:

Use induction (+ swap).

BC: For $n=1, 2$, or 3 , Huffman works

Why?

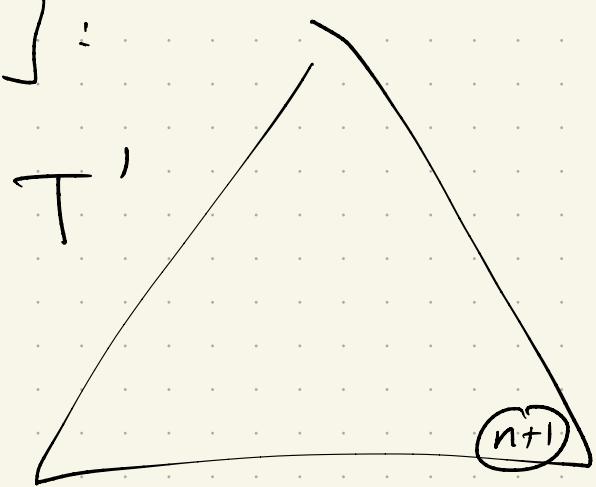
IH: Assume Huffman works on $\leq n-1$ characters

IS: Input $F[1..n]$, + spp's

$F[1] + F[2]$ are min freq.

↳ create a smaller array

IS : optimal tree T' of
 $F[3..n+1]$:



Note: $n+1$
is in tree

Build a tree T for $F[1..n]$:

Claim: T is optimal.

Why?

Why is T optimal??
(we know T' is $\rightarrow IH!$)

Cost(T) =

$$\sum_{i=1}^n F[i] \cdot \text{depth}[i]$$

$$= \text{cost}(T') + \underbrace{\text{changes we made}}$$

↗

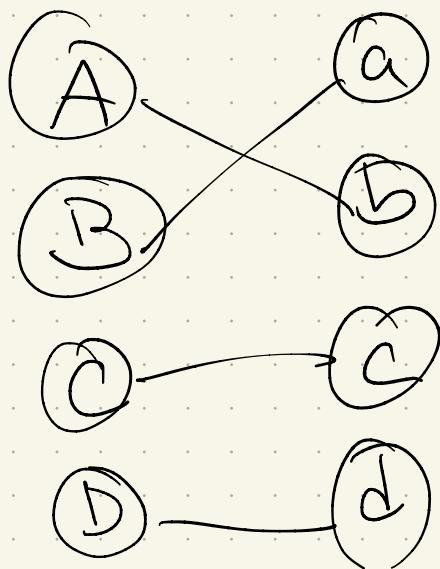
Stable matching

Really useful! Many variants:

- ties
- incomplete preference lists
- one side picks many from the other
- "egalitarian" matchings
- minimizing "regret"

Really a lot of choices to be made.

First: "unstable":



- (A, a) is unstable
 - If A prefers a to current match
 - and a prefers A to current match

In a sense: if put together + realizing they both prefer each other, would (A, a) leave current matches?

↳ unstable!

History: used to be "stable marriage"

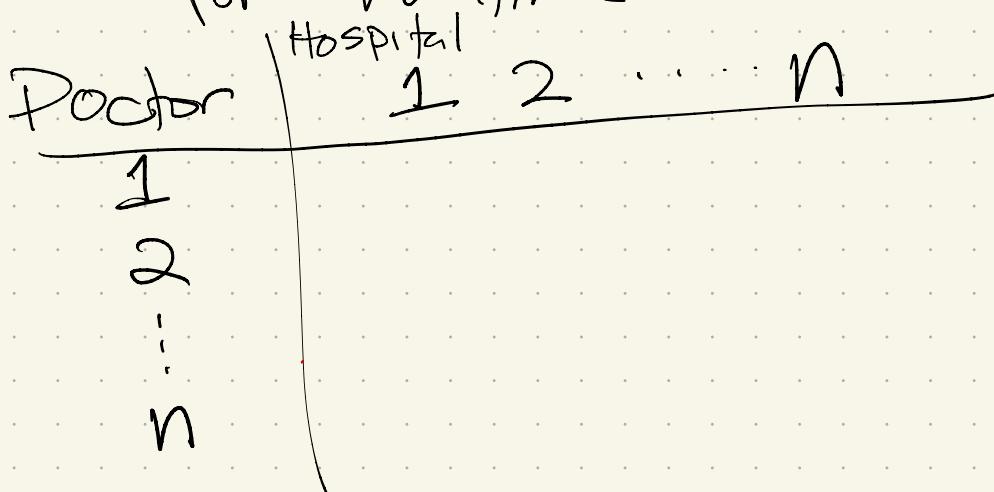
(long history of strange papers + variants.)

Algorithm: (wikipedia)

Algorithm [edit]

```
algorithm stable_matching is
    Initialize all  $m \in M$  and  $w \in W$  to free
    while  $\exists$  free man  $m$  who still has a woman  $w$  to propose to do
         $w :=$  first woman on  $m$ 's list to whom  $m$  has not yet proposed
        if  $w$  is free then
             $(m, w)$  become engaged
        else some pair  $(m', w)$  already exists
            if  $w$  prefers  $m$  to  $m'$  then
                 $m'$  becomes free
                 $(m, w)$  become engaged
            else
                 $(m', w)$  remain engaged
            end if
        end if
    repeat
```

In book, data structures matter
for runtime:



Not obvious why it works.
(or even how to be greedy!)

Good example of why the
proof matters.

Nice example of fairness:

This algorithm sucks for
one side.

(Not all solutions are equal!)

How to even define "fair"?

↳ EC reading if
interested??