Algorithms - Spring '25

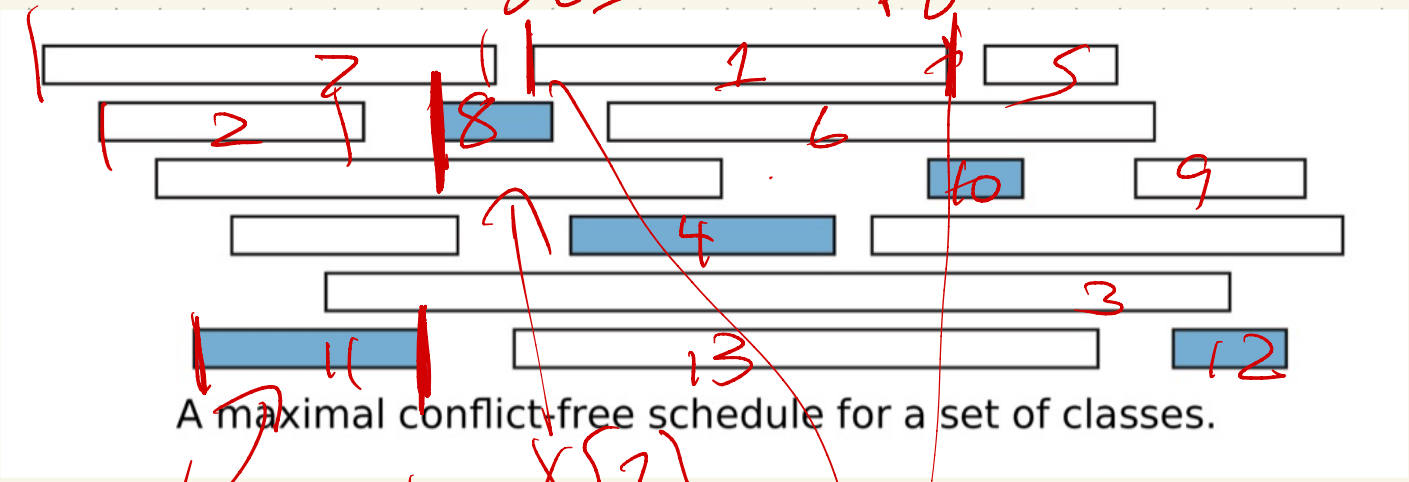
Greed:
Intervals
Huffman codes

Recap

- Office hours: today at 2pm (me) & 3:15 pm (TA)
- HW3 due
- HW4 — oral grading next Thurs/Friday
- Midterm: March 4, 8am

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.

X[1] = 11

X[2]

More formally:

Two arrays

S[1..n]

F[1..n]:

X[1] interval

↳ S[X[1]]

↑ of F[X[1]]

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 programing approach?

Recursive structure:

Consider job 1:

take it
↳ add to X
recurse on 2..n

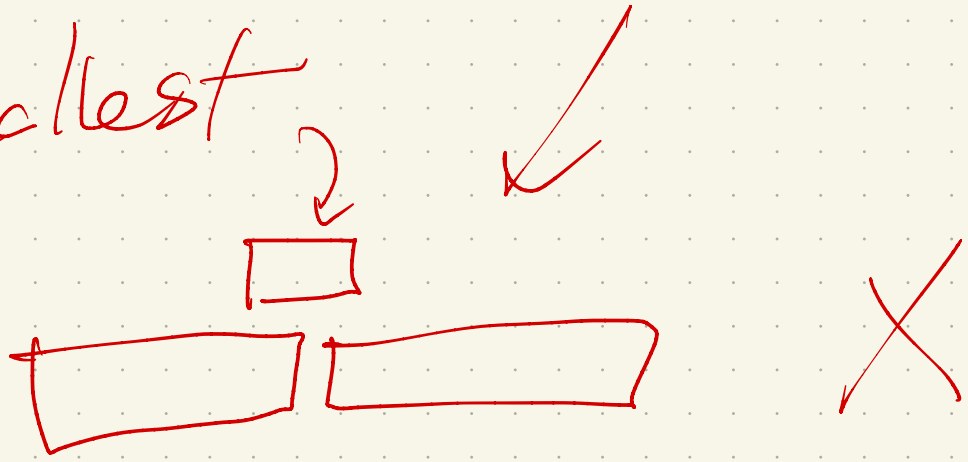
don't
recurse on 2..n

Intuition for greedy:

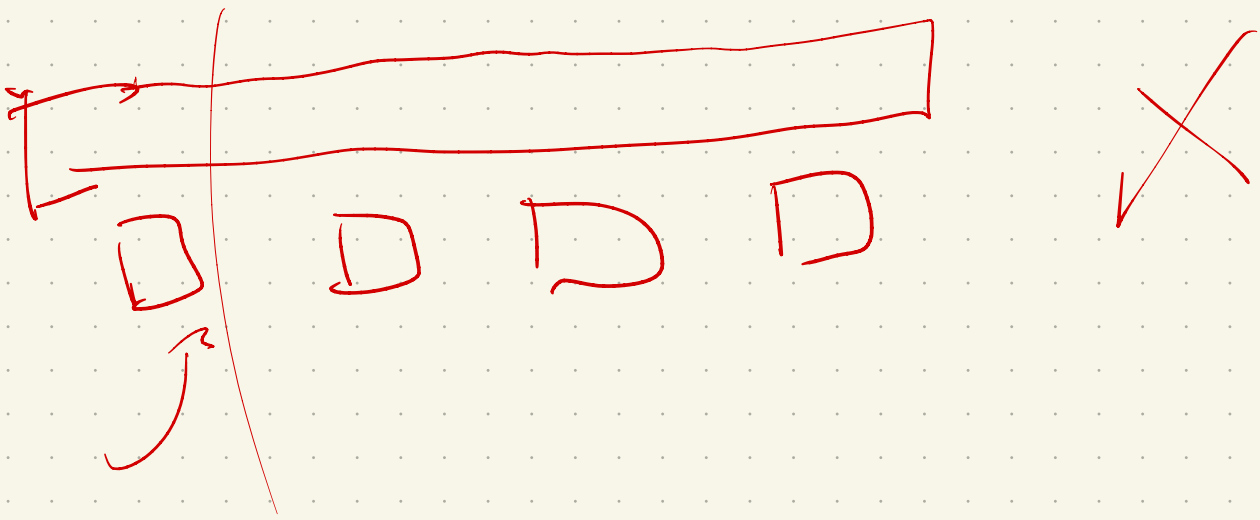
Consider what might be a good first one to choose.

Ideas?

Smallest



Earliest



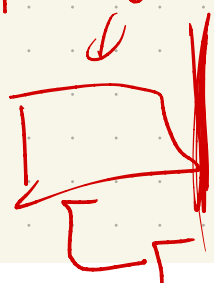
Key intuition:

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

So - strategy:

Sort by finish time
$F[1..n]$ ← sorted

→ Take interval 1, eliminate overlaps, & go on



The code:

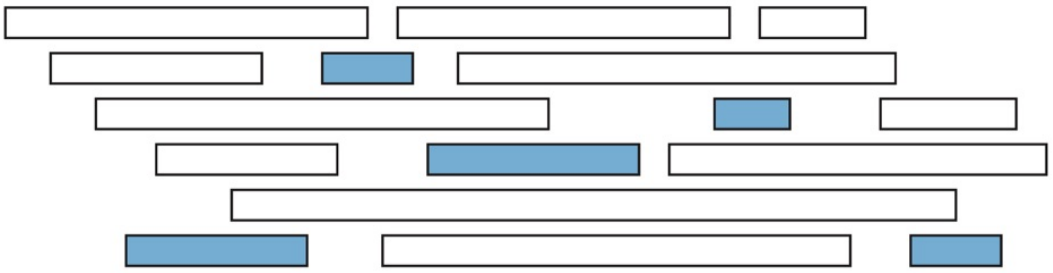
```
GREEDYSCHEDULE(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```

nlogn

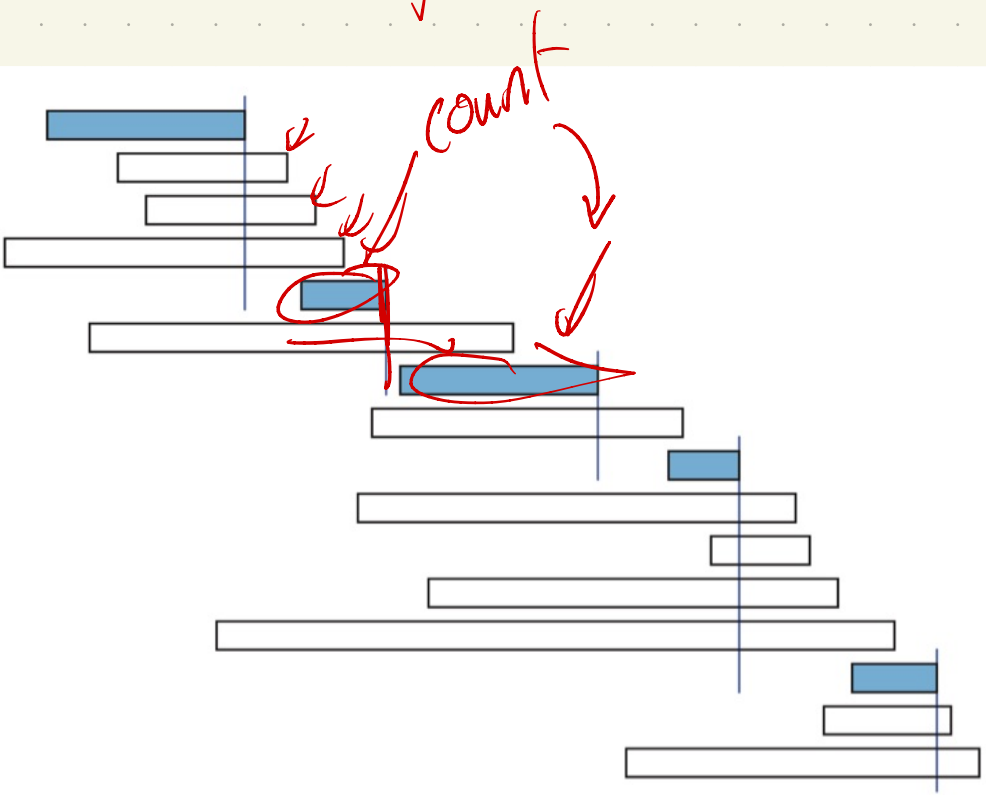
O(n)

time: O(n log n)

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.

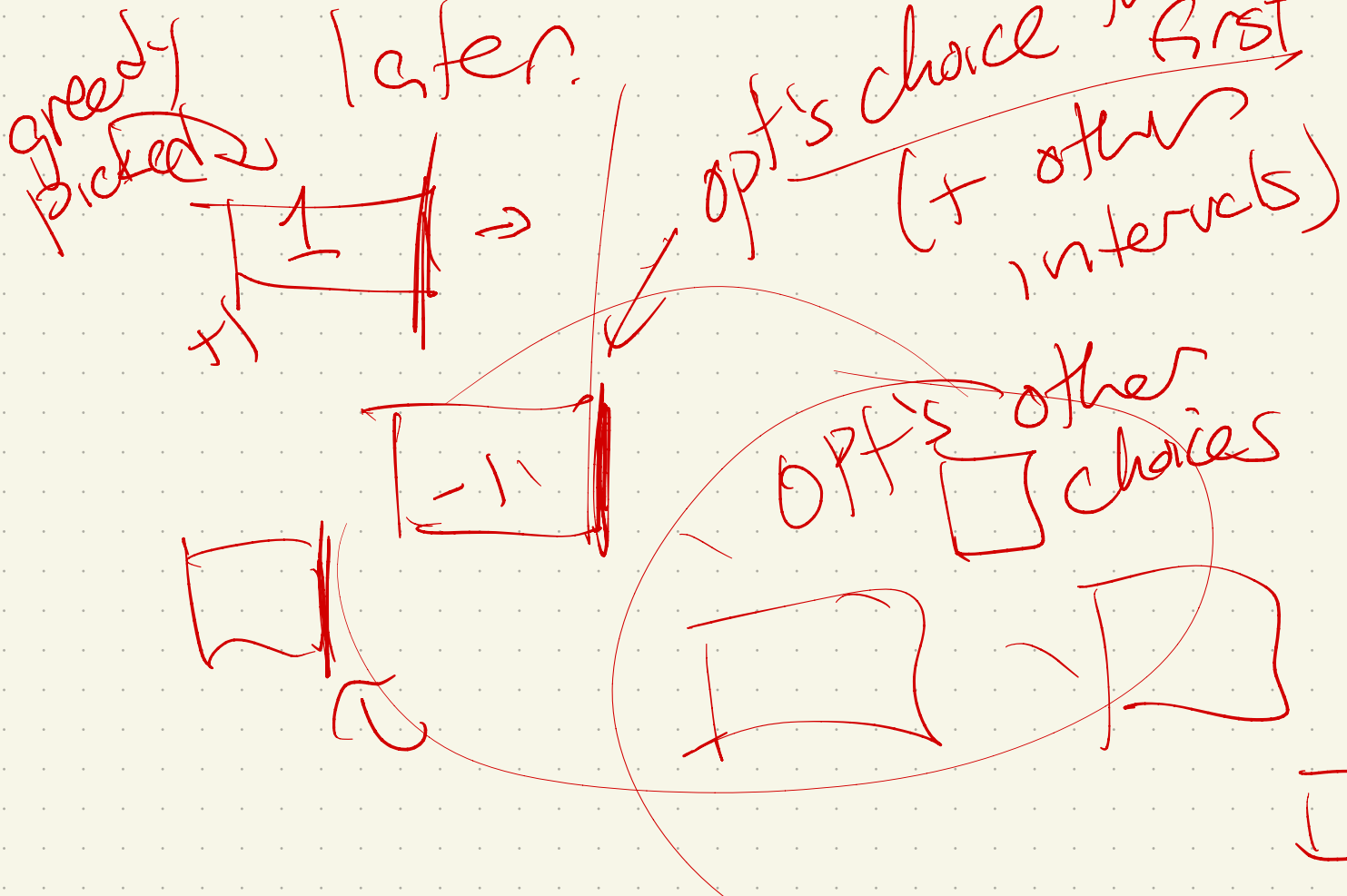
(Clearly, intuition can be wrong!)

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

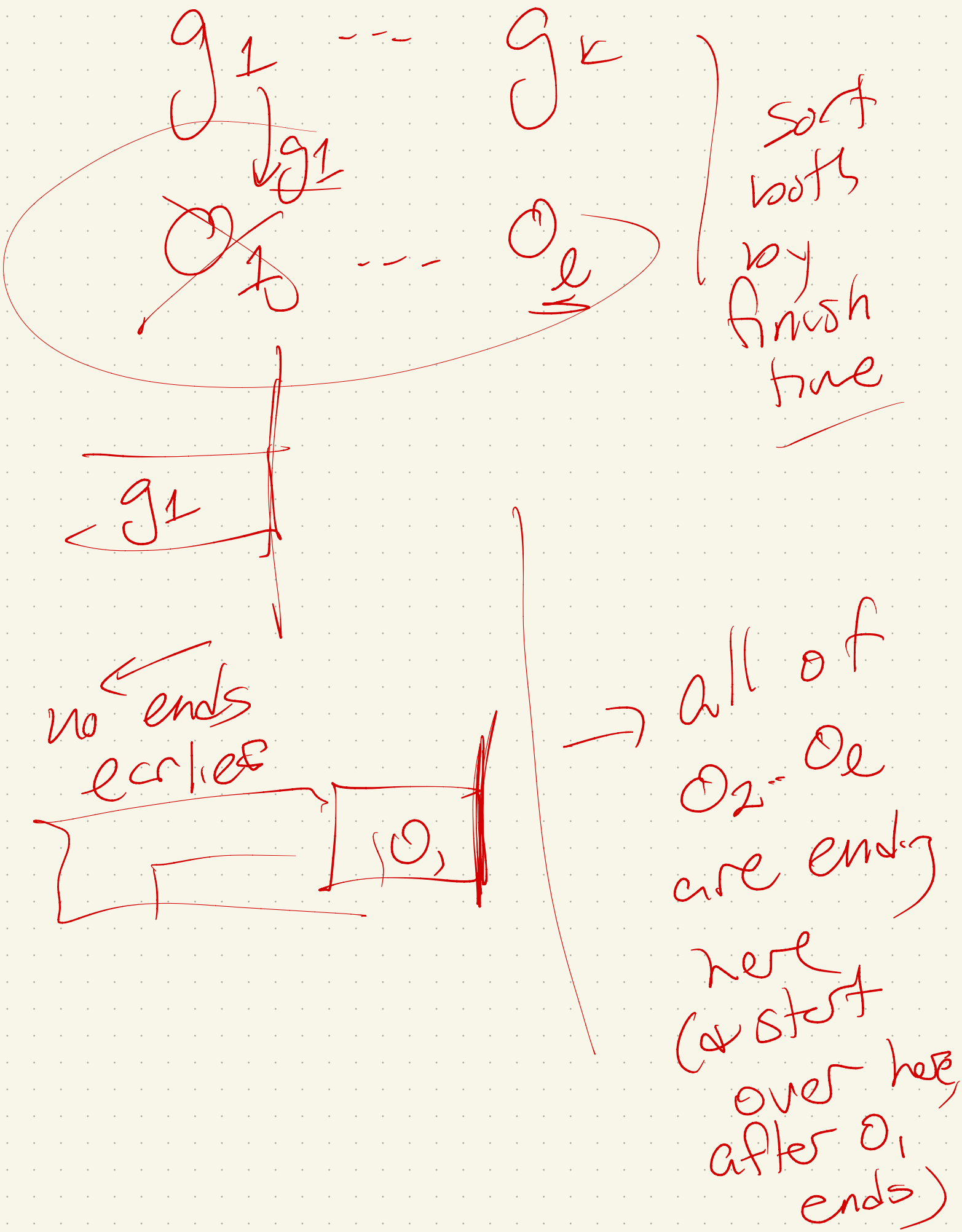
pf: by contradiction

Suppose it doesn't:

then picks some other interval, which finishes later.



□


g_1 --- g_k
g_1
O_1 --- O_ℓ
sort both by finish time
g_1
no ends earlier
O_1
→ all of $O_2 \cdots O_\ell$ are ending here (& start over here after O_1 ends)

Thm: The greedy schedule is optimal. (can optimal)

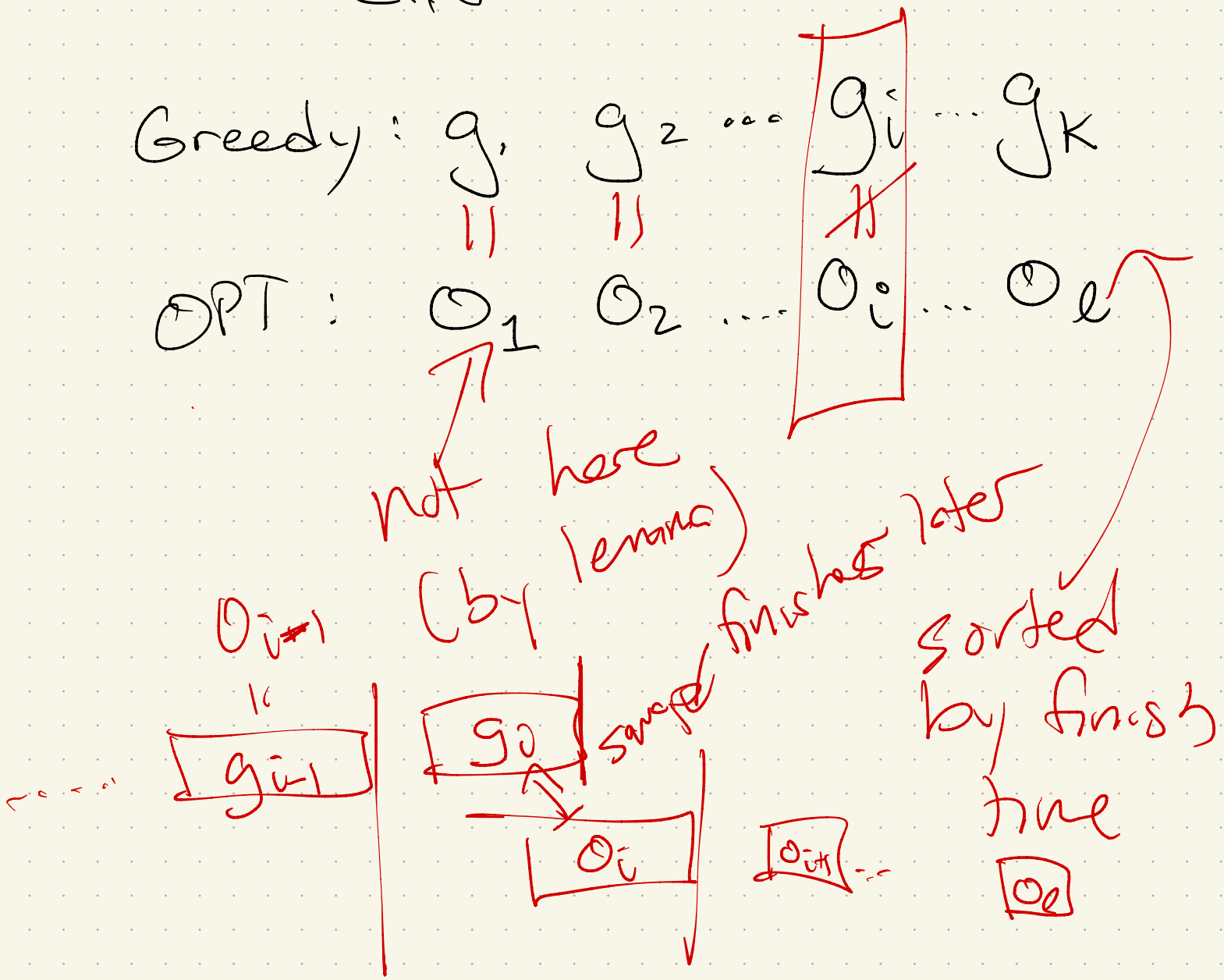
Pf: Suppose not.

Then ∃ (exists) an optimal schedule that has more intervals than the greedy one.

Consider first time they differ:

Greedy: g_1 g_2 ... g_i ... g_k

OPT: o_1 o_2 ... o_i ... o_ℓ



Example: Huffman trees

Many of you saw this in data structures.

Why?

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

Really - it's greedy!

Idea: Want to compress data, to use fewest possible bits.

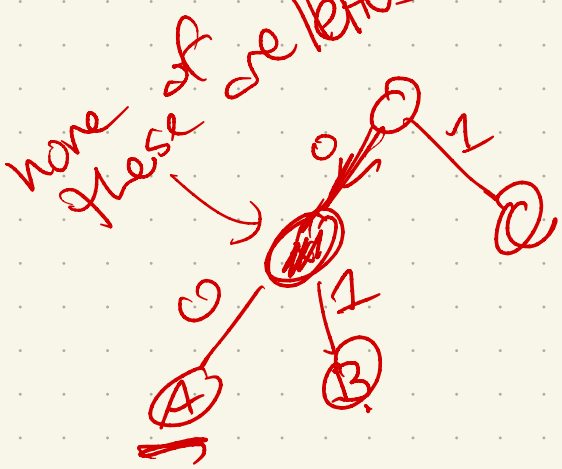
Goal: Minimize cost

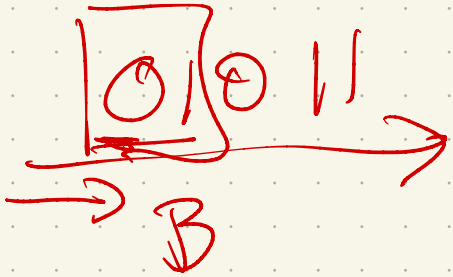
↳ here, minimize total length of encoded message:

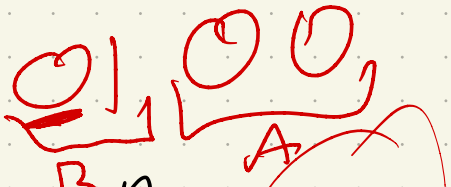
Input: frequency counts $f[1..n]$ } one per letter

Compute: binary tree

Leaves: are letters







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

Let's be greedy:

To do this, we'll need to use the array f:

This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

If we ignore punctuation & spaces (just to keep it simple), we get:

f:

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

Which letters should be deeper (or shallower)?

(ie: How to be greedy?)

2 least common

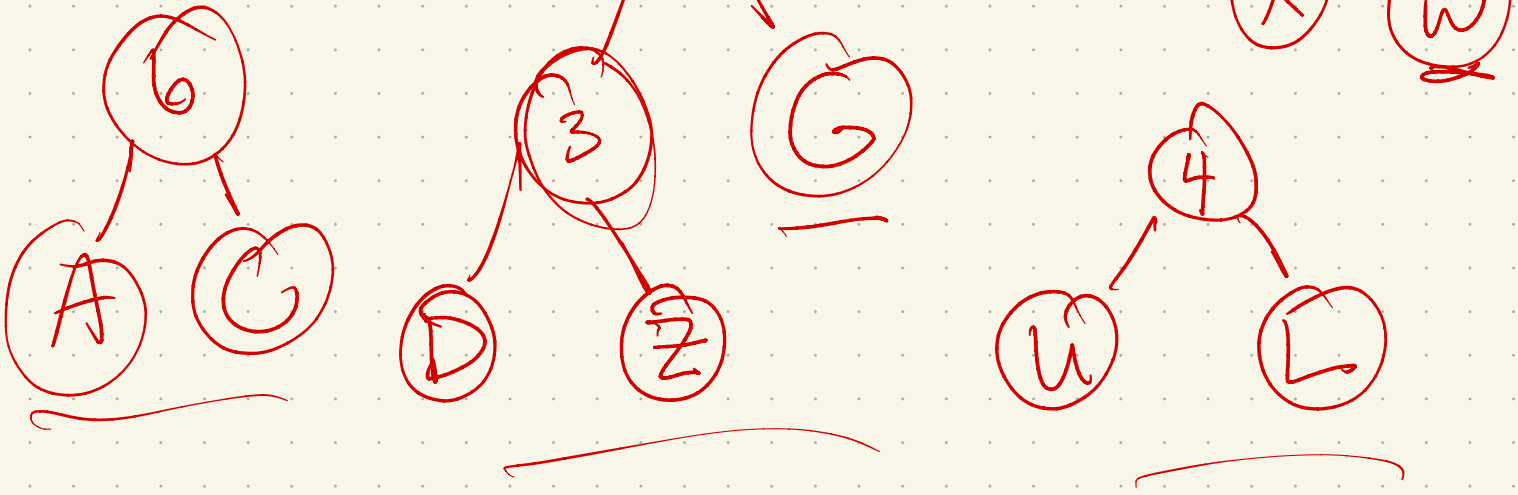
Huffman's alg:

Take the two least frequent characters.

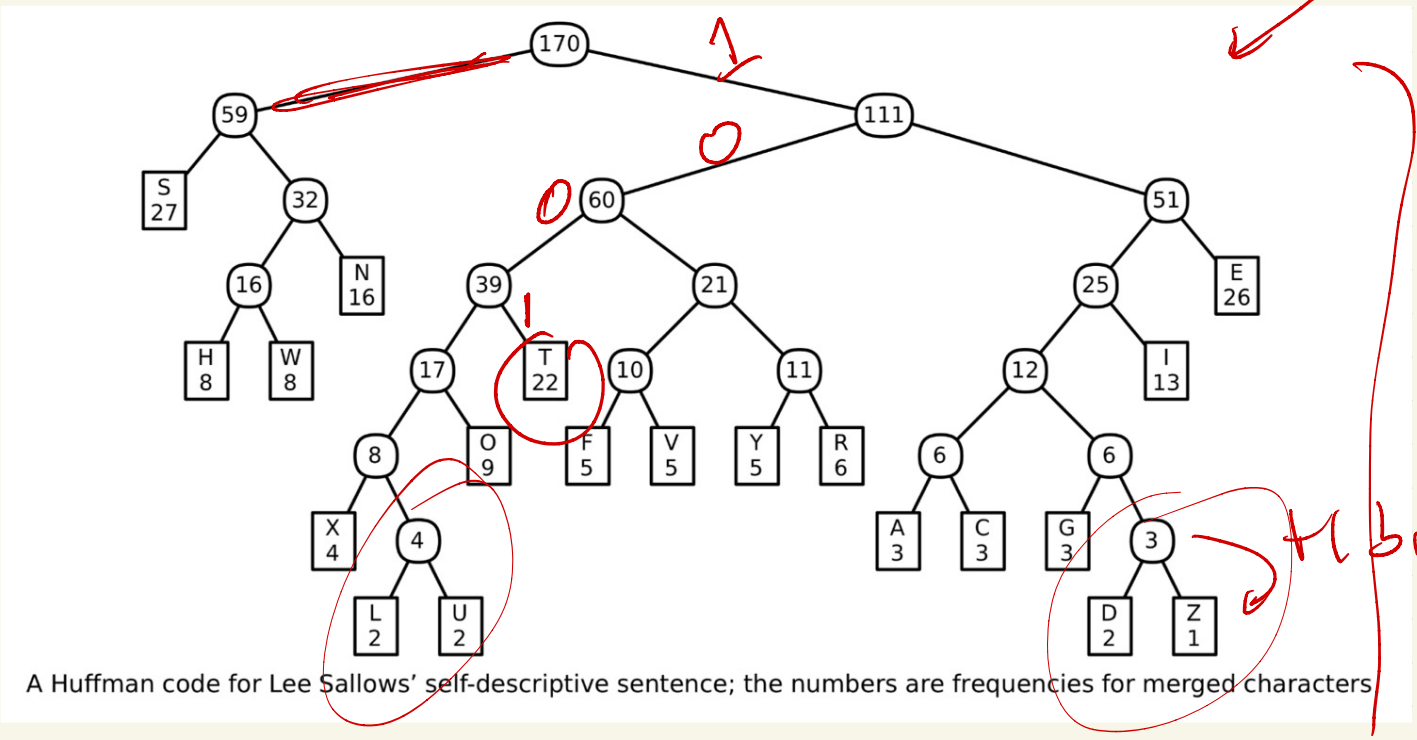
Merge them in to one letter, which becomes a new "leaf":

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

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



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 O N T A

How many bits?

char.	A	B	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq.	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	7	3	4	4	2	4	7	5	4	6	5	7
total	18	18	14	78	25	18	32	52	14	48	36	24	54	88	14	25	32	24	25	7

Total is $\sum f[i] \cdot \text{depth}(i)$

$= 646$ bits here

How would ASCII do on these 170 letters

8 bits per letter

$\hookrightarrow 170 \times 8 = 1350$ bits

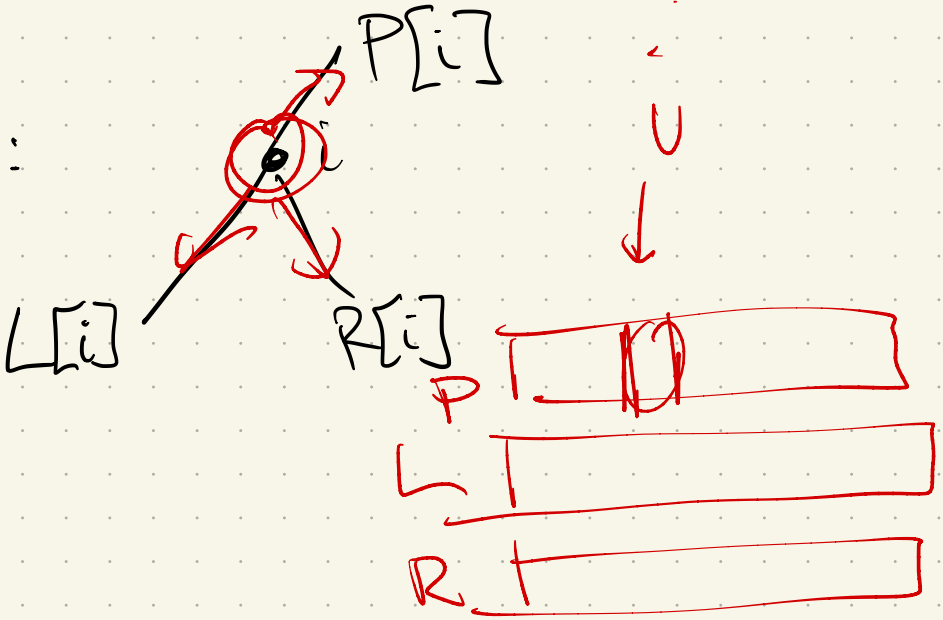
Implementation: use priority queue

```
BUILDHUFFMAN(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      INSERT(i, f[i])

  for i ← n to 2n − 1
      x ← EXTRACTMIN( )
      y ← EXTRACTMIN( )
      f[i] ← f[x] + f[y]
      L[i] ← x; R[i] ← y
      P[x] ← i; P[y] ← i
      INSERT(i, f[i])

  P[2n − 1] ← 0
```

heap
O(log n) per add/delete

3 arrays: L, R, P
to encode the tree

node i:



So: BANANA

index:	1	2	3	4	EOM	6
letters:	B	A	N	EOM		
freq: f:	1	3	2	1	2	4

n = 5

x = 1

y = 4

n leaves

n-1 interna

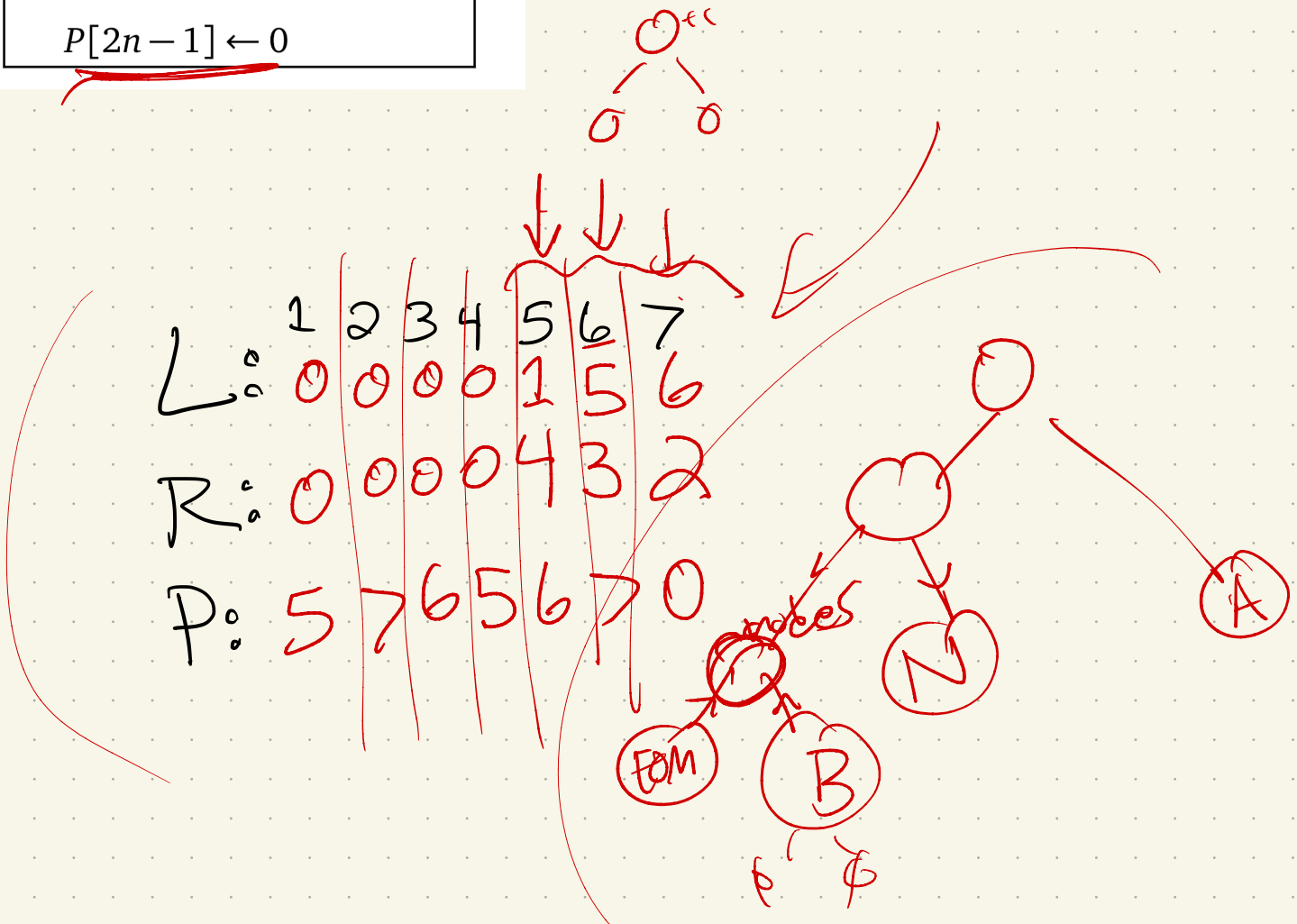
```
BUILDHUFFMAN(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      INSERT(i, f[i])
  for i ← n to 2n − 1
      x ← EXTRACTMIN( )
      y ← EXTRACTMIN( )
      f[i] ← f[x] + f[y]
      L[i] ← x; R[i] ← y
      P[x] ← i; P[y] ← i
      INSERT(i, f[i])
  P[2n − 1] ← 0
```

n log n

n times

n log n

	1	2	3	4	5	6	7
L:	0	0	0	0	1	5	6
R:	0	0	0	0	4	3	2
P:	5	7	6	5	6	7	0



Runtime?

```
BUILDHUFFMAN(f[1..n]):
    for i ← 1 to n
        L[i] ← 0;  R[i] ← 0
        INSERT(i, f[i])

    for i ← n to 2n − 1
        x ← EXTRACTMIN( )
        y ← EXTRACTMIN( )
        f[i] ← f[x] + f[y]
        L[i] ← x;  R[i] ← y
        P[x] ← i;  P[y] ← i
        INSERT(i, f[i])

    P[2n − 1] ← 0
```

$O(n \log n)$

$O(n \log n)$

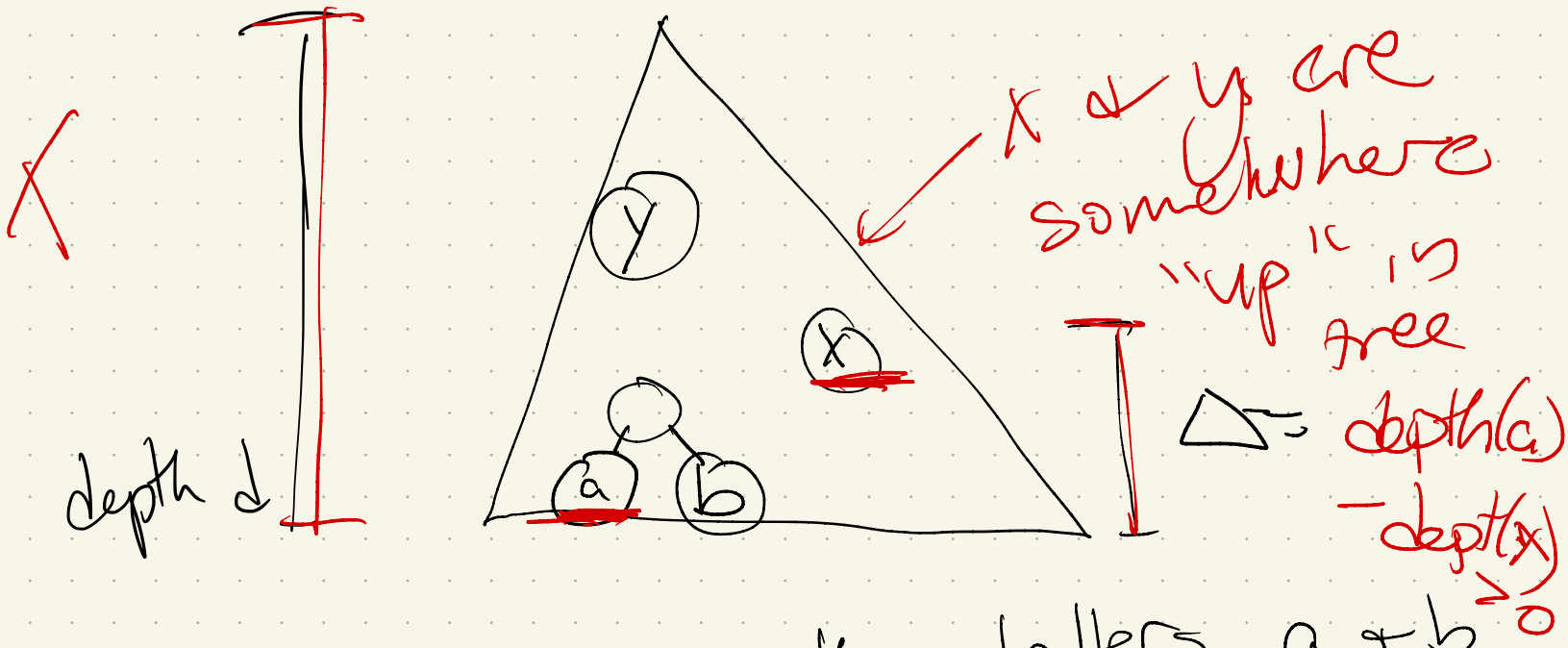
$\Rightarrow O(n \log n)$

w/ $O(n)$ space

Correctness:

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

pf: Spps not. Then optimal tree T has some depth d, but 2 least common letters x & y are not at that depth.



Note some other letters a & b are deepest

pf cont:

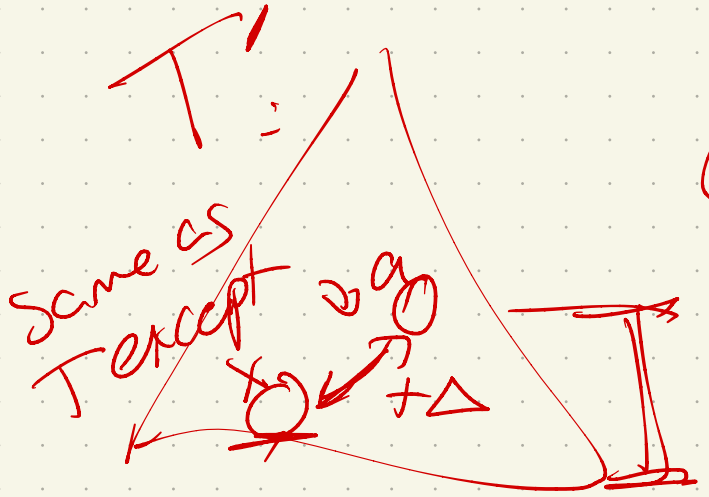
Know $f[x] \leq f[a]$, (least frequency) $f[x] - f[a] \leq 0$

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':



$$\text{cost}(T') = \text{cost}(T) + f[x] \cdot \Delta - f[a] \cdot \Delta$$

$$= \text{cost}(T) + \Delta (f[x] - f[a])$$

$\Delta > 0$ $\quad > 0$ $\quad \leq 0$

so T' is better! $\rightarrow$ less!

∎

Thm: Huffman trees are optimal.

pf:

Use induction (+ swap).

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

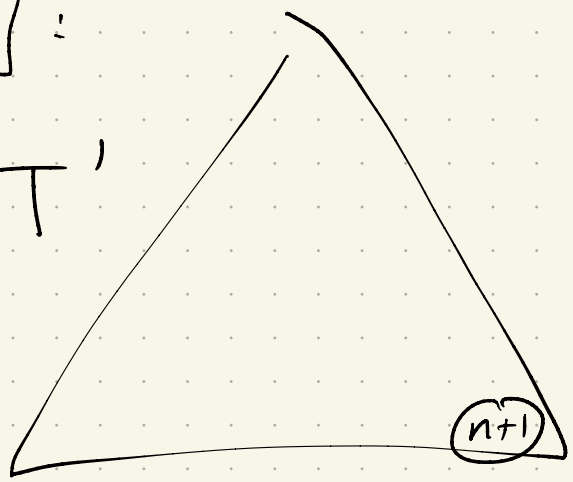
Why?

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

IS: Input $F[1..n]$, + spps $F[1]$ + $F[2]$ 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 optimal → IH!)

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

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

↳

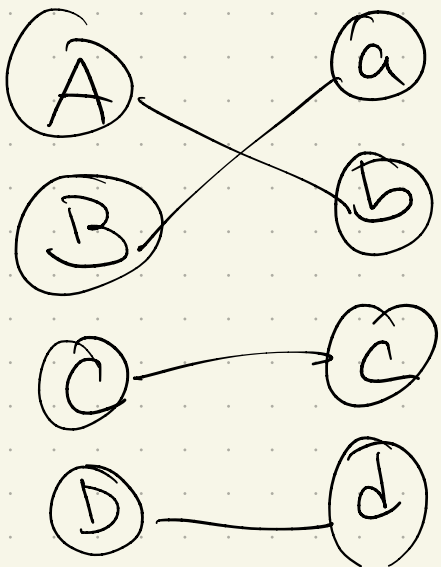
Stable matching

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"

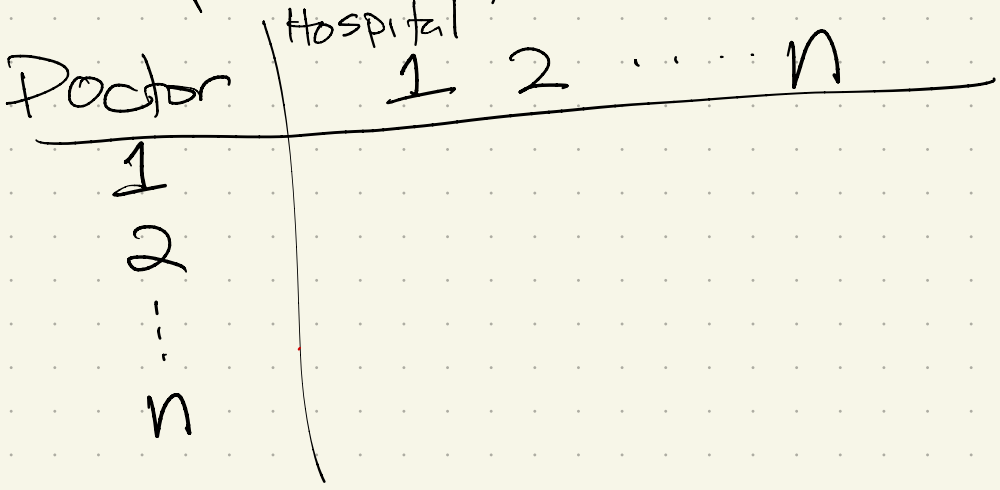
(long history of strange papers & variants.)

Algorithm: (Wikipedia)

Algorithm [edit]

```
algorithm stable_matching is
    Initialize all m ∈ M and w ∈ W to free
    while ∃ 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"?