Algorithms - Spring '25

Greedy:
- Huffman
- Stable Matching

Graphs intro

Recap

- Oral grading (HW4)
  - Join a canvas group
  - Sign up for calendar slot
  - What to expect: see webpage!
- Midterm exam
  - → Sample posted
- Last HW: group issue for some
- Regrades: Apologies for delay

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.

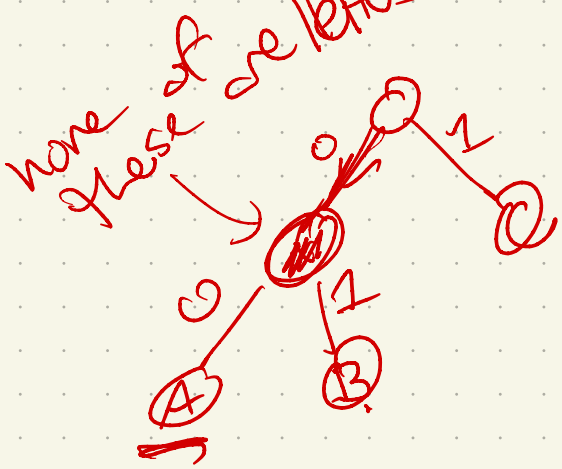
Goal: Minimize cost

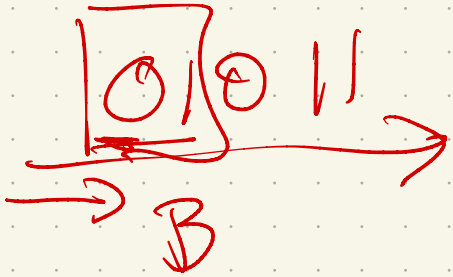
↳ here, minimize total length of encoded message:

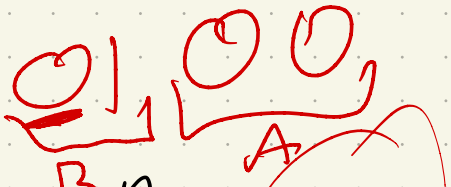
Input: frequency counts $f[1..n]$ } one per letter

Compute: binary tree

Leaves: are letters







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

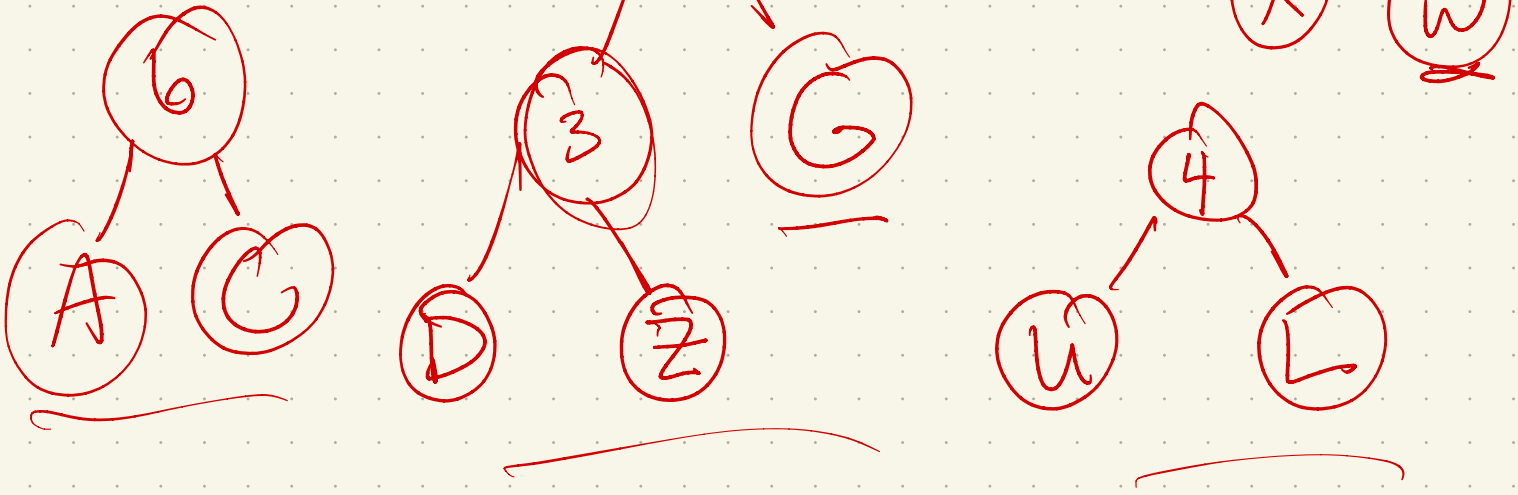
Huffman's alg:

Take the two least frequent characters.

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

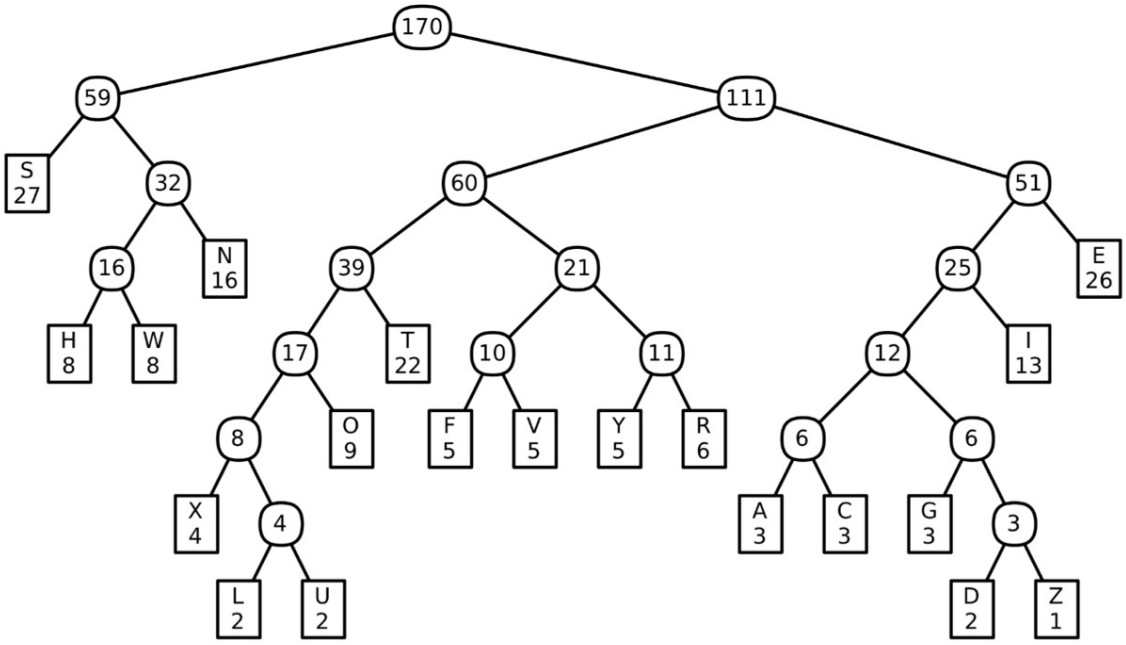
A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

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

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	

Another:

0100111100001010000101010001

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

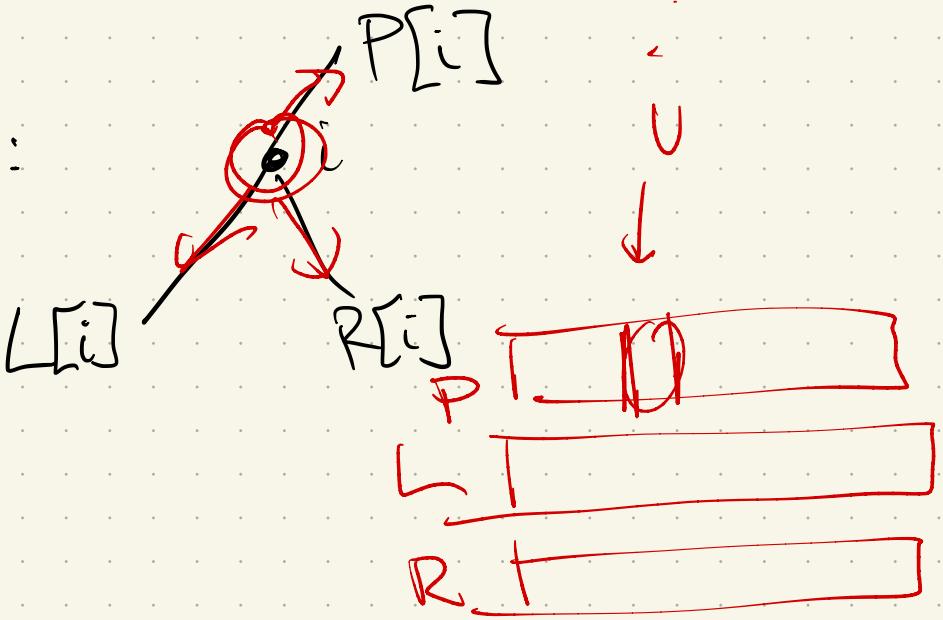
Implementation: use priority queue

```
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

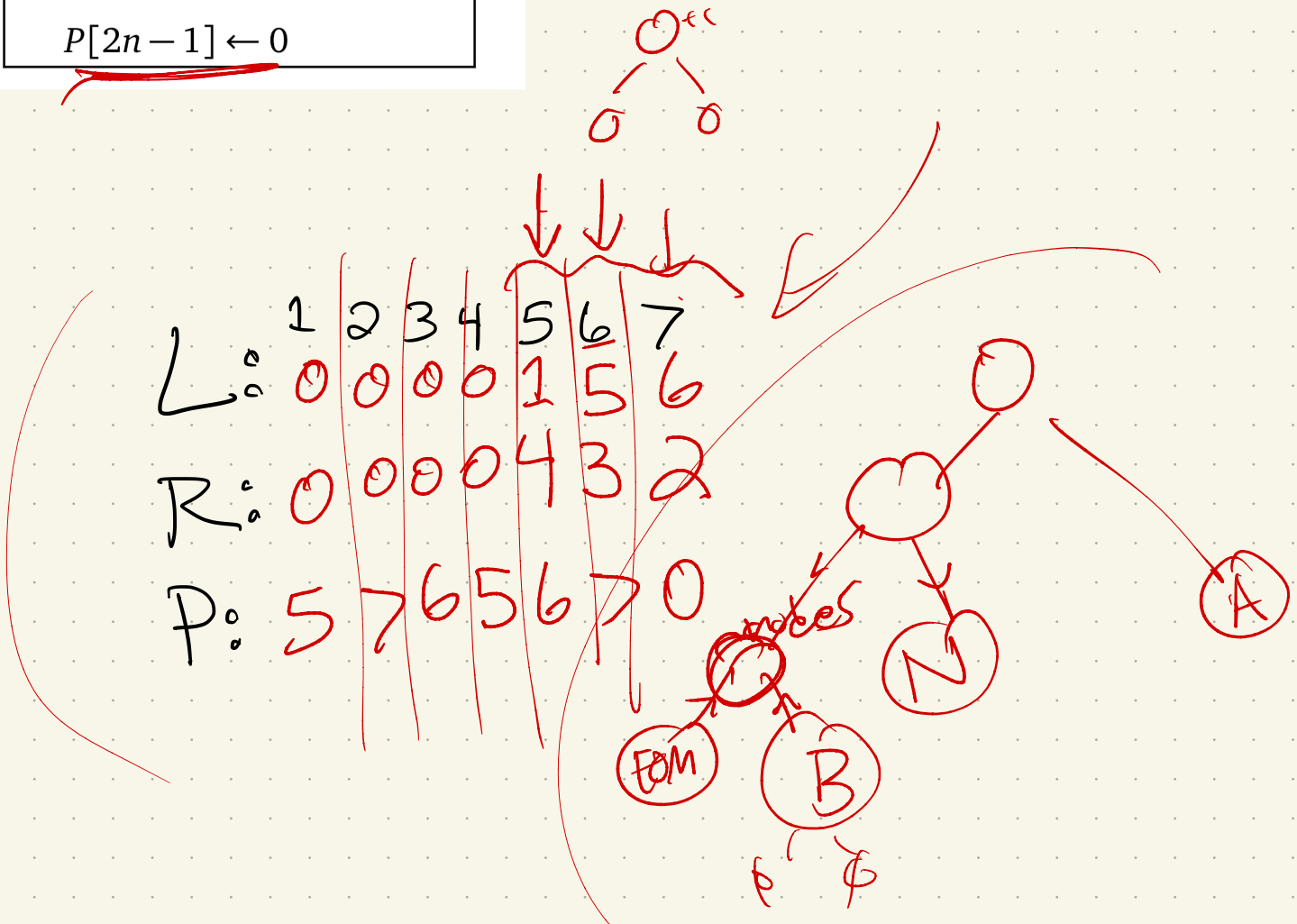
```
BUILDHUFFMAN(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      INSERT(i, f[i])
  for i ← n to 2n − 1
      x ← EXTRACTMIN( )
      y ← EXTRACTMIN( )
      f[i] ← f[x] + f[y]
      L[i] ← x; R[i] ← y
      P[x] ← i; P[y] ← i
      INSERT(i, f[i])
  P[2n − 1] ← 0
```

n log n

n times

n log n

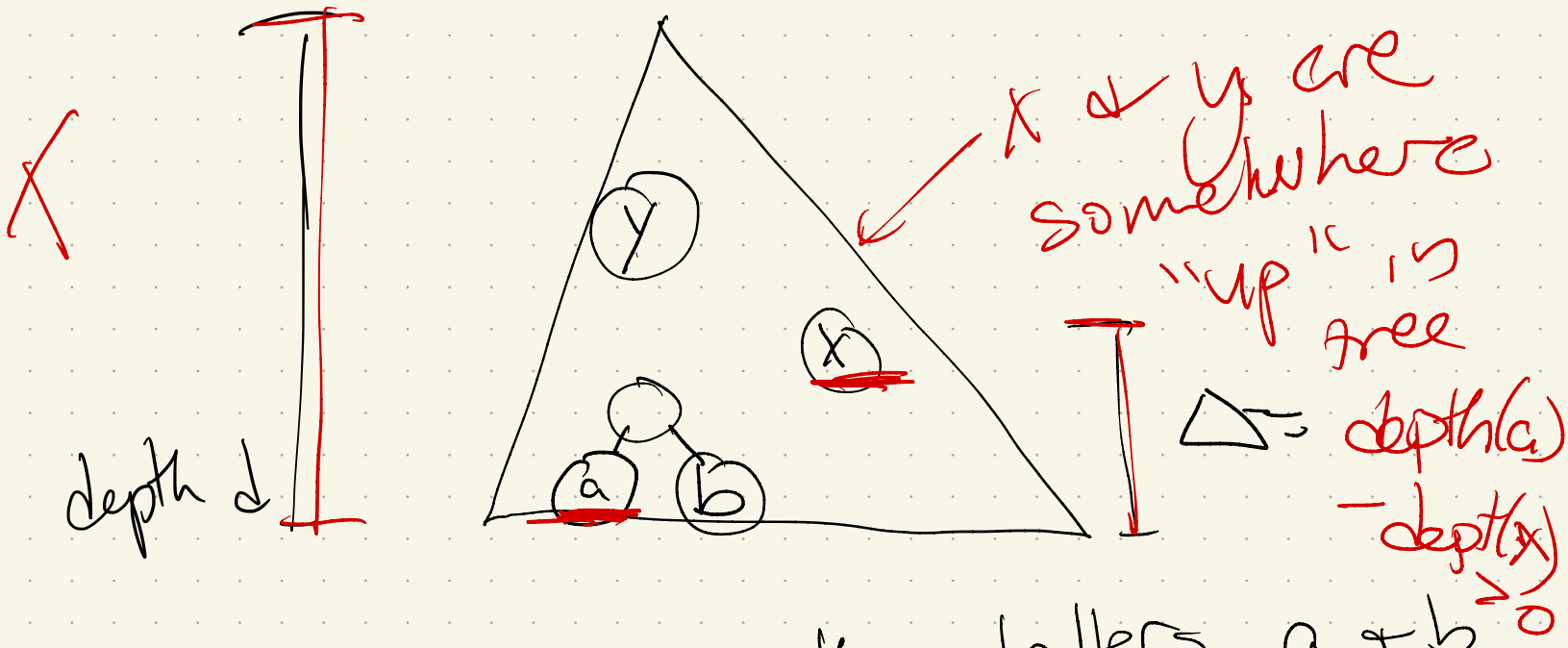
	1	2	3	4	5	6	7
L:	0	0	0	0	1	5	6
R:	0	0	0	0	4	3	2
P:	5	7	6	5	6	7	0



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 frequen-) $f[x] - f[a] \leq 0$

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':

Swap a and x in tree

(All other nodes stay same.)

$$\text{Cost}(T') = \sum_{i=1}^{n} f[i] \cdot \text{depth}(T')$$

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

$$- \Delta \cdot f[a]$$

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

and $f[a] \geq f[x]$, so ≤ 0 ∎

Thm: Huffman trees are optimal.

pf:

Use induction (+ swap).

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

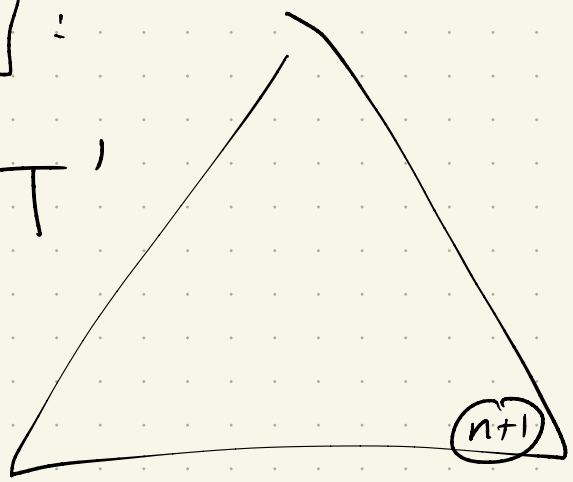
Why?

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!)

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

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

↳

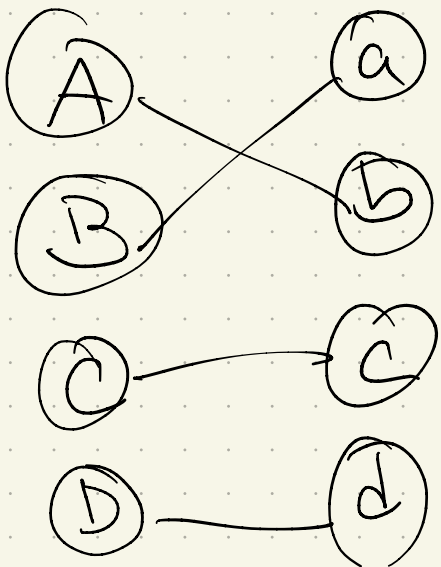
Stable matching

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"

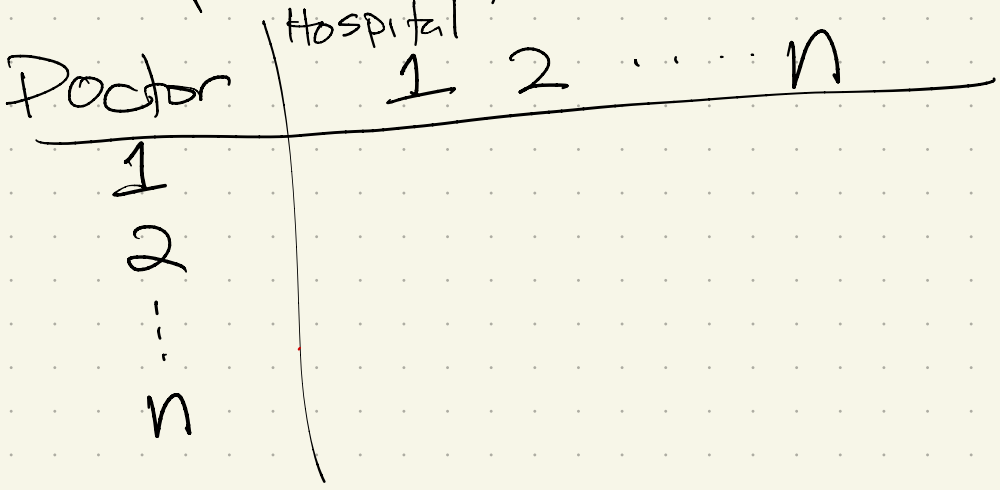
(long history of strange papers & variants.)

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"?

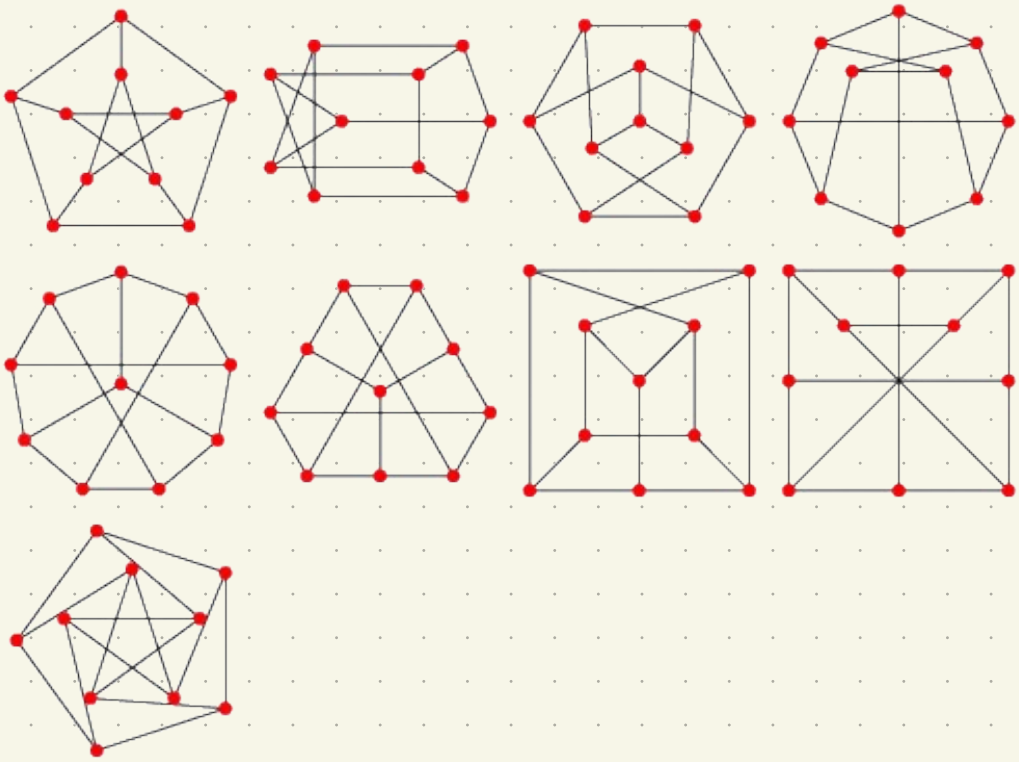
Graphs

A graph $G = (V, E)$ is an ordered pair of 2 sets:

V = vertices =

E = edges =

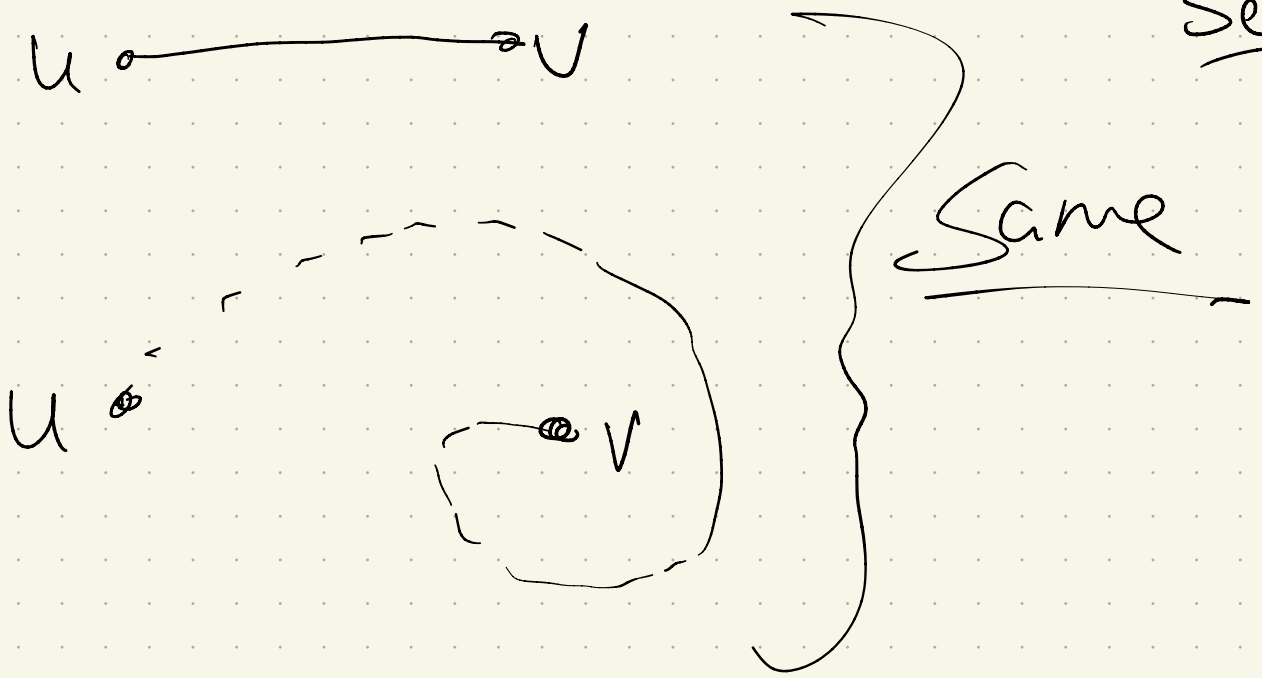
We often draw them, but they do <u>not</u> come with coordinates.



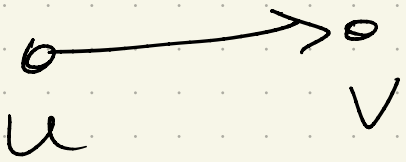
"Edges" → not straight!

An edge is a pair $\{u, v\}$:

Set

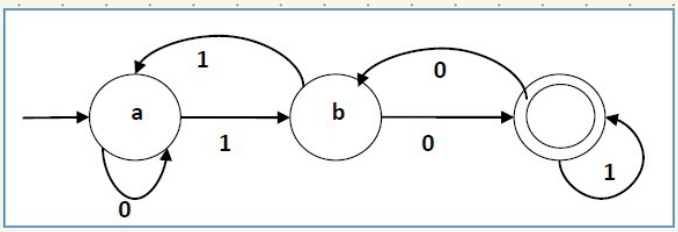


Directed edges $e = (u, v)$

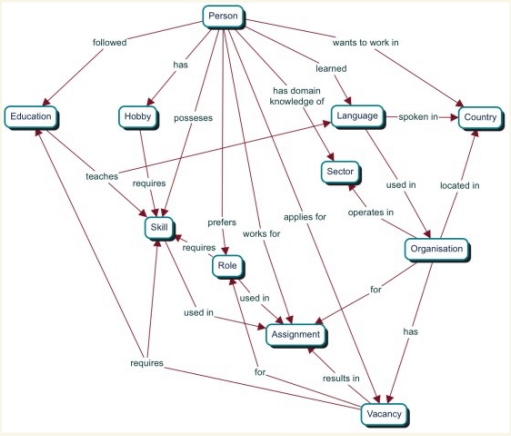


Why study them?

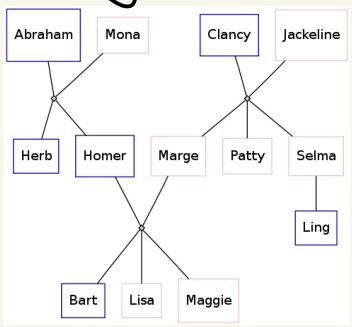
DFA:



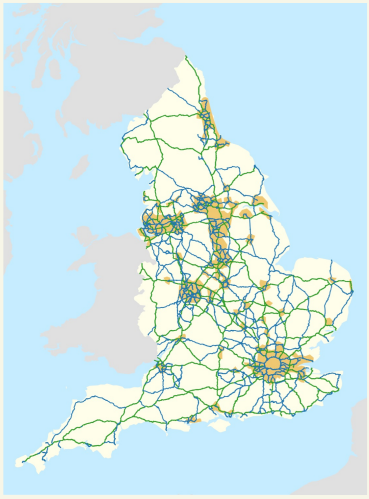
concept map:



lineages:



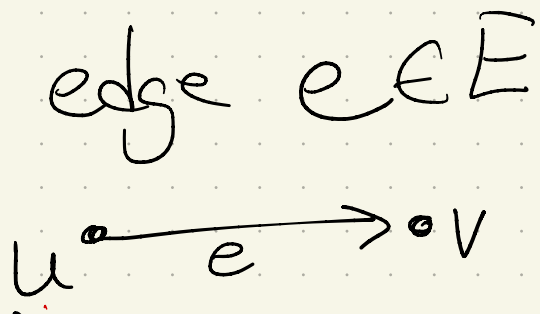
road network:



Why so much history??

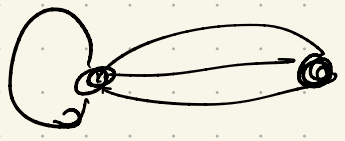
Definitions: See book!

- Vertices (nodes), V
- Edges, E
- endpoints of an edge $e \in E$
- head & tail

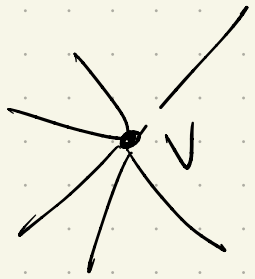


↳ $u \to v$ or (u, v)

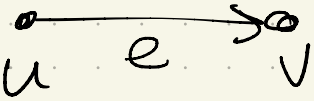
- simple: no parallel edges or 1-edge loops



- adjacent
- degree(v)



- predecessor & successor



indegree & outdegree

More!

- Subgraph
- Walk
- Path

  Note: If you have a walk $u \rightsquigarrow v$, can make a path.

  How?

- Connected
- Closed

- cycle
- tree

First: some "easy" bounds.

Lemma: $E \leq \frac{V(V-1)}{2}$

pf:

Lemma: $\sum_{v} d(v) = 2E$

pf:

First question

Computers don't do well with images! So pictures won't help them.

We need to store this info (somehow).

Ideas from data structures:

Adjacency (or vertex) lists:

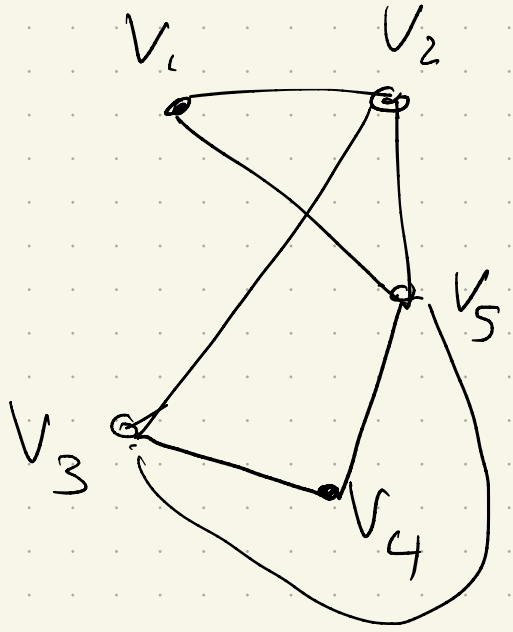
V_1 :

V_2 :

V_3 :

V_4 :

V_5 :



Size:

lookup: time to check if u & v are neighbors:

Implementation:

More buried data structures!

Could use:

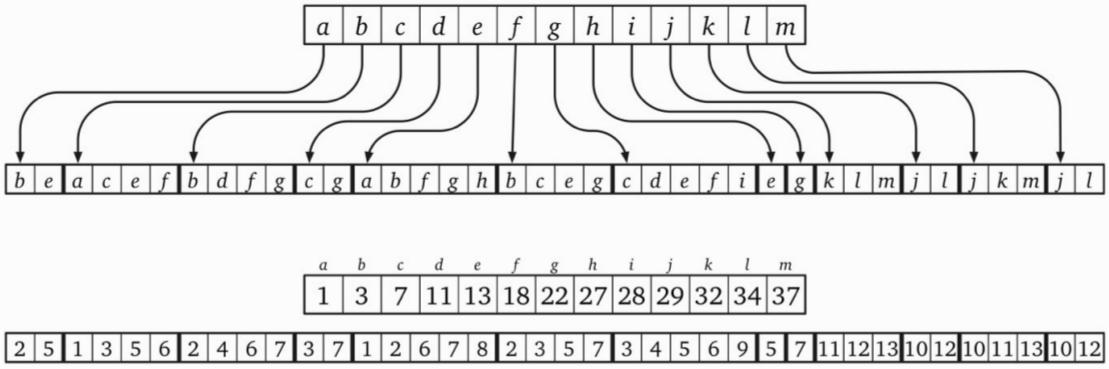


Figure 5.10. An abstract adjacency array for our example graph, and its actual implementation as a pair of integer arrays.

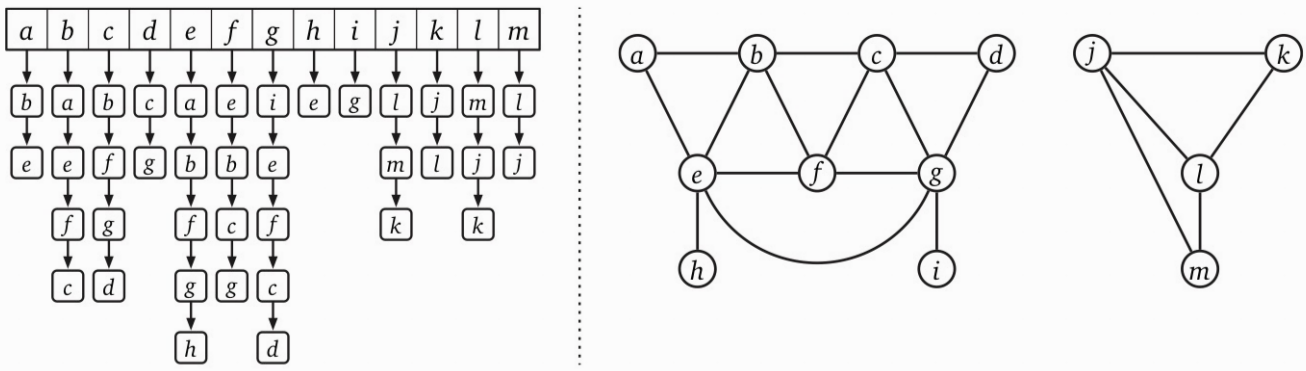
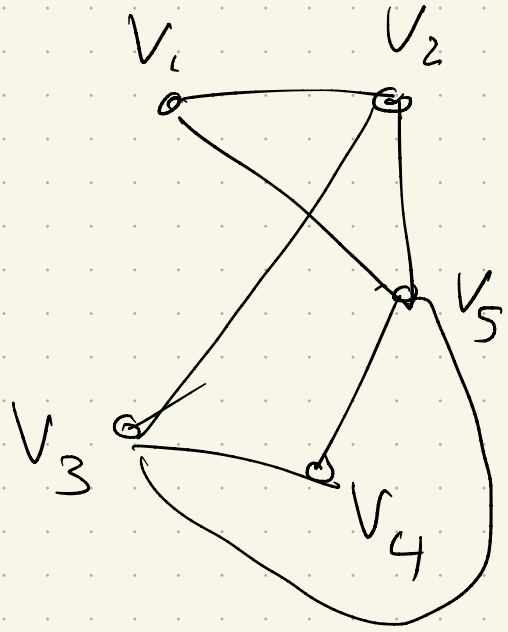


Figure 5.9. An adjacency list for our example graph.

Adjacency Matrix

	V_1	V_2	V_3	V_4	V_5
V_1					
V_2					
V_3					
V_4					
V_5					



Space:

check nbr:

Implementation:
More data structures!

	a	b	c	d	e	f	g	h	i	j	k	l	m
a	0	1	0	0	1	0	0	0	0	0	0	0	0
b	1	0	1	0	1	1	0	0	0	0	0	0	0
c	0	1	0	1	0	1	1	0	0	0	0	0	0
d	0	0	1	0	0	0	1	0	0	0	0	0	0
e	1	1	0	0	0	1	1	1	0	0	0	0	0
f	0	1	1	0	1	0	1	0	0	0	0	0	0
g	0	0	1	1	1	1	0	0	1	0	0	0	0
h	0	0	0	0	1	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	1	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0	1	1	1
k	0	0	0	0	0	0	0	0	0	1	0	1	0
l	0	0	0	0	0	0	0	0	0	1	1	0	1
m	0	0	0	0	0	0	0	0	0	1	0	1	0

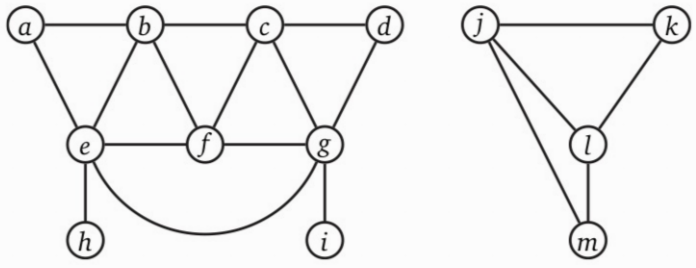


Figure 5.11. An adjacency matrix for our example graph.

Which is better?

Depends!

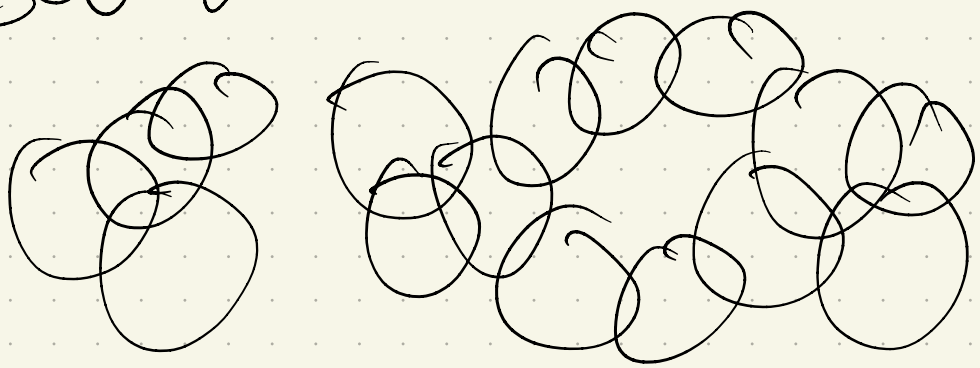
	Adjacency matrix	Standard adjacency list (linked lists)	Adjacency list (hash tables)
Space	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to test if $uv \in E$	$O(1)$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$
Time to test if $u \to v \in E$	$O(1)$	$O(1+\deg(u)) = O(V)$	$O(1)$
Time to list the neighbors of v	$O(V)$	$O(1+\deg(v))$	$O(1+\deg(v))$
Time to list all edges	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to add edge uv	$O(1)$	$O(1)$	$O(1)^*$
Time to delete edge uv	$O(1)$	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$

In the rest of this book, unless explicitly stated otherwise, all time bounds for graph algorithms assume that the input graph is represented by a standard adjacency list. Similarly, unless explicitly stated otherwise, when an exercise asks you to design and analyze a graph algorithm, you should assume that the input graph is represented in a standard adjacency list.

Really - might depend on input!

- Size of graph
- Freq. of changes
- representation: usually, some "word problem" is handed to you! You'll have to build the graph.

Ex: Given a set of overlapping circles, find the largest set where no 2 intersect:



Even more:

- Space available
- language used
- previous "legacy" code
- other developers

⋮

To repeat - too keep it simple here:

In the rest of this book, unless explicitly stated otherwise, all time bounds for graph algorithms assume that the input graph is represented by a standard adjacency list. Similarly, unless explicitly stated otherwise, when an exercise asks you to design and analyze a graph algorithm, you should assume that the input graph is represented in a standard adjacency list.