Week 1 Videos:

0-0 Course Info

Midterm & Final are open CLRS book.

0-1 Course History

0-2 HW Submission

- single PDF
- readable
- can resubmit as many times before deadline → only last submission is marked

0-3 Exam Submission

- join Bb Collab during exam / Zoom
  ↳ can ask TAs questions
  ↳ corrections/exam typos announced here
- 7 days for remarking
- broken down into intervals
- every new problem must start on a new side of a page

1-1 Asymptotics Intro

Asymptotics

Performance or complexity:

- Time
- Space

Algorithm
Computer
Turing Machine

To classify algos → using concept of "n"

Let's say someone found algo that has:
$n^4 + 3n$ (in terms of n)
↑ input n

Alg 1: $n^4 + 3n$ } (in terms of n)
time to run → $O(n^4)$

Alg 2: $700^5 n^3 + 10^6 n^2 + 10^6 n$ } → $O(n^3)$ ✓ faster

Which algo. do you use?
- Alg 2 seems like it takes more time
- need a way to classify algos w/ notation to be able to compare them

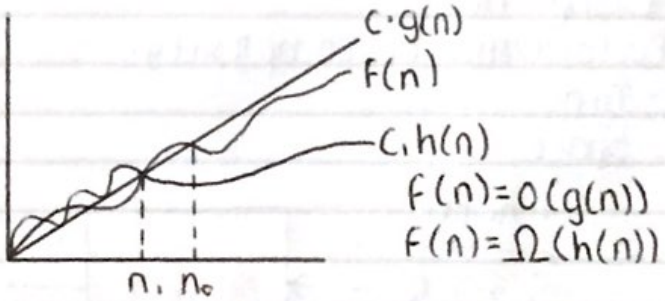
⇒ Alg 2 is much faster

1-2 Big Oh

Big-Oh

We say that $f(n) = O(g(n))$ iff (= if & only if)

$O(g(n)) = \{f(n)$: ∃ exist a +ve constants c & n_0 s.t. $0 \leq f(n) \leq cg(n)$ ∀ $n \geq n_0\}$



Asymptotically this is an "upper bound"

→ value of f(n) will not be slower than value of g(n)

Examples:
$13n + 7 \in O(n)$
$13n + 7 \leq 14n \quad n_0 = 7$

$\frac{1}{2}n^2 - 3n = O(n^2)$
We need prove $\frac{1}{2}n^2 - 3n \leq c \cdot n^2 \leftrightarrow \frac{1}{2} - \frac{3}{n} \leq c$
true for $c = \frac{1}{2}$ (or larger)
$\forall\ n \geq 1 = n_0$

$n! = 1 \cdot 2 \cdot 3 \dots \cdot n$
$\leq \underbrace{n \cdot n \cdot \dots \cdot n}_{n \text{ times}} = n^n$
implies $n! = O(n^n)$

what about $\log n!$
$n! = O(n^n) \leftrightarrow \log n! = O(n \log n)$

1-3 Big Omega

Big-Omega
We say $f(n) = \Omega(h(n))$ iff
$\Omega(h(n)) = \{f(n) : \exists$ +ve constants $c_1 \& n_1$ s.t.
$0 \leq c_1 h(n) \leq f(n) \quad \forall\ n \geq n_1\}$

Called a "lower bound" for $f(n)$

→ $f(n)$ will not be faster than $h(n)$

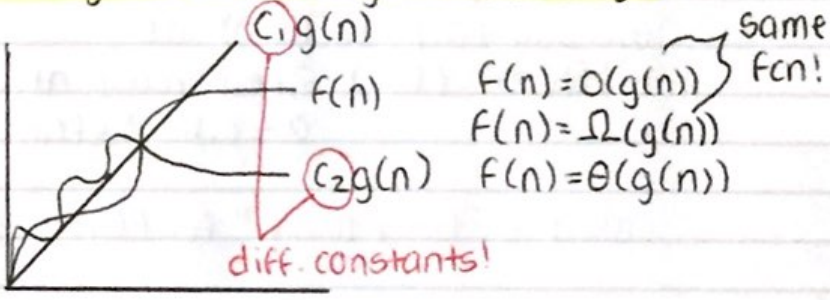
Examples:
$f(n) = 1 + 2 + 3 + \dots + n = \Omega(?)$
$f(n) = 1 + 2 + 3 + \dots + n$
$\geq [\frac{n}{2}] + ([\frac{n}{2}] + 1) + ([\frac{n}{2}] + 2) + \dots + n$
$\geq \underbrace{[\frac{n}{2}] + [\frac{n}{2}] + \dots + [\frac{n}{2}]}_{\frac{n}{2} \text{ times}}$
$\geq \frac{n}{2} \cdot \frac{n}{2} = \frac{n^2}{4}$
Therefore for $c_1 = \frac{1}{4}\ n_1 = 1$
$f(n) = \Omega(n^2)$

$f(n) = \frac{1}{2}n^2 - 3n = O(n^2)$
$= \Omega(?)$

If suspect "?" is "n^2"
→ need to prove: $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leftrightarrow$
$c_1 \leq \frac{1}{2} - \frac{3}{n}$ → where did 7 come from?
Allow $n \geq 7$, then:
$c_1 \leq \frac{1}{2} - \frac{3}{7} \leftrightarrow$
$c_1 \leq \frac{7}{14} - \frac{6}{14} \leftrightarrow$ so that c_1 is +ve
$c_1 \leq \frac{1}{14}$
↑ needs to be +ve (cannot be 0, can be decimal 0.1 for example, >0)
∴ for $c_1 \leq \frac{1}{14}$ $f(n) = \Omega(n^2)$ } $\Theta(n^2)$
$= O(n^2)$ }

1-4 Theta Asymptotics

Big-Theta
we say that $f(n) = \Theta(g(n))$
$\Theta(g(n)) = \{ f(n) : \exists$ +ve constants $c_1, c_2,$ & n_0 s.t
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall n \geq n_0 \}$



Called an asymptotic "tight bound"
↳ desirable → single fcn is sufficient to characterize how fcn behaves

Examples:

Arithmetic series:

$1 + 2 + 3 + \ldots + n = \Omega(n^2)$

$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

$= \frac{1}{2}n^2 + \frac{1}{2}n = O(n^2)$

$\downarrow$

$1 + 2 + 3 + \ldots + n = \Theta(n^2)$

Prove that

$\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$

Need to show: $O(n^{k+1})$ & $\Omega(n^{k+1})$

① $O(n^{k+1})$: $\sum_{i=1}^{n} i^k \leq \sum_{i=1}^{n} n^k = n \cdot n^k = n^{k+1}$ QED

② $\Omega(n^{k+1})$: $f(n) = \sum_{i=1}^{n} i^k$?

$2f(n) = \sum_{i=1}^{n} i^k + \sum_{i=1}^{n} (n-i+1)^k$

descending

$1^k + 2^k + 3^k \ldots + n^k + n^k + (n-1)^k + \ldots + 2^k + 1^k$

$= \sum_{i=1}^{n} i^k + (n-i+1)^k \geq$

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

$\times 2$

$= \Omega(n^{k+1})$

1-5 Asymptotics Properties

Properties

Transitivity

$f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$

Symmetry
$f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$

Transpose
$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$

Cookbook:

$n^a \in O(n^b)$ iff $a \leq b$
$\log_a n \in O(\log_b n)$ $\forall a, b$
$c^n \in O(d^n)$ iff $c \leq d$
if $f(n) \in O(f'(n))$ & $g(n) \in O(g'(n))$, then
↕ NOT deriv! (Just another fcn)
$f(n) \cdot g(n) = O(f'(n) \cdot g'(n))$
$f(n) + g(n) = O(\max\{f'(n), g'(n)\})$

Analysis:
Best case → minimum am't of time → Ω
Worst Case → O
Average Case ↔ similar
Expected Case ↔ similar
Amortized Case ↔ similar

Types of algo: Monte Carlo ("lies"), Las Vegas ("no lie")

Randomized Algorithm
Probabilistic
=!!= ↗ symbol same as " ,copy abv"
↓
Probabilistic Analysis
e.g. quick sort

2-1 Logarithms

* all $\log(f(n))$ in lecture videos are log-base-2 of $f(n)$

Logarithms

Def: $a = b^c \leftrightarrow \log_b a = c$

Properties

$a = b^{\log_b a}$

$\log_c(ab) = \log_c a + \log_c b$

$\log_b a^n = n \log_b a$

$\log_b a = \frac{1}{\log_a b}$

$\log_b \frac{1}{a} = -\log_b a$

$\log_b \frac{a}{c} = \log_b a - \log_b c$

$a^{\log_b n} = n^{\log_b a}$

Note: $\log n \equiv \log_2 n \rightarrow$ when base missing $\equiv$ base = 2

$$\log^{(i)} n = \begin{cases} n & \text{iff } i = 0 \\ \log(\log^{(i-1)} n) & \text{if } i > 0 \end{cases}$$

$\log^{(3)} n = \log\log\log n$ → 3 times

what is min # times that we need to take log of a quantity to get ≤ 1

$\log^* n = \min\{ i : \log^{(i)} n \leq 1 \}$

↳ "such that"

$\log^* 2 = 1$

$\log^* 4 = 1 + \log^* 2 = 2$

$\log^* 2^{2^{2^2}} = 1 + \log^* 2^{2^{2^2}}$

$= 1 + 1 + \log^* 2^2 = 4$

$\log^* 2^{512} = 5$

atoms in universe $\approx 10^{18}$

→ Log-star grows very slow!

$O(\log^* n) \neq O(1)$

↳ for practical purpose if algo takes $\log^* n$ time, equivalent to constant time

Fibonacci #s

Recursive quantity defined as follows:

$F_i = F_{i-1} + F_{i-2}$ where $F_0 = 0$ & $F_1 = 1$

Can be proven that:

$$F_i = \frac{\Phi^i - \hat{\Phi}^i}{\sqrt{5}}$$

Golden Conjugate (GC) $\Phi = \frac{1+\sqrt{5}}{2}$

GC Inverse $\hat{\Phi} = \frac{1-\sqrt{5}}{2}$

2-2 Summations

Arithmetic Series

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \Theta(n^2)$$

Geometric Series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \ldots + x^n = \frac{x^{n+1} - 1}{x - 1}$$

Infinite Series

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{if } |x| < 1$$

Ex:

Show that $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ when $|x|<1$

Proof:

$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ $\leftrightarrow$ Differentiate both sides over x

$\leftrightarrow \sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$

$\leftrightarrow$ multiply both sides by x

$\leftrightarrow \sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$

Telescoping

Assume a_i = something

It holds:

- $\sum_{i=1}^{n} a_i - a_{i-1} = a_n - a_0$
- $\sum_{k=0}^{n-1} a_k - a_{k+1} = a_0 - a_n$

Proof

$$\sum_{i=1}^{n} a_i - a_{i-1} = (a_1 - a_0) + (a_2 - a_1) + (a_3 - a_2) + \vdots + (a_n - a_{n-1})$$

$$a_n - a_0$$

Ex:

Prove $\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = 1 - \frac{1}{n}$

$= \sum_{k=1}^{n-1} \frac{1}{k} - \frac{1}{k+1} = 1 - \frac{1}{n}$

a_1 a_n

Binomial Coefficient

$$(x+y)^r = \sum_{i=0}^{r} \binom{r}{i} x^i y^{r-i}$$

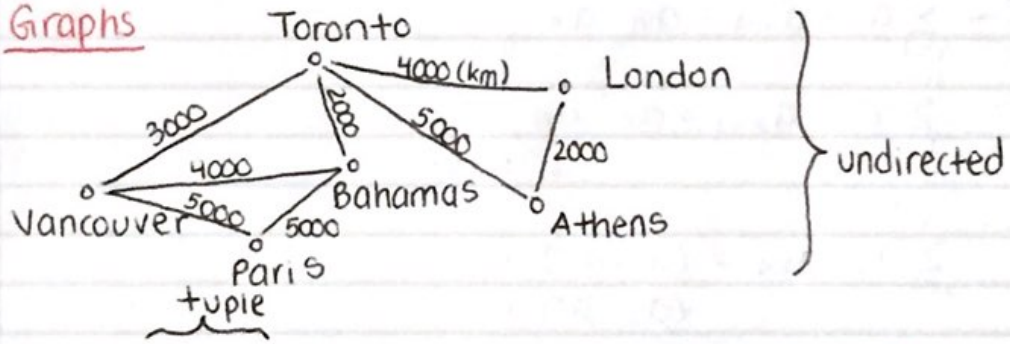
↓

"r choose i" or Binomial Coefficient

$$\frac{r!}{i!(r-i)!}$$

Week 2 Videos: 2-3 Intro to Graphs

Graphs



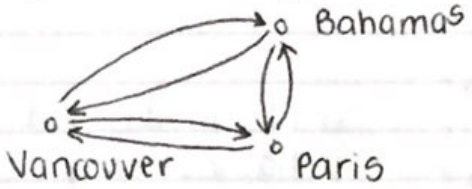
tuple

Graph G(V, E) defined over set of vertices V (cities) & set of edges E. (direct flights)

Properties:

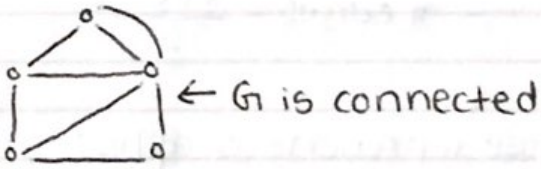
- directed or undirected (edges)
  - ↳ edges only go one way (specific direction)
  - ↳ edges don't have direction (can go both ways)

- weighted or unweighted
  ↳ cost, distances, profits, penalties ...
- path : sequence of edges, b/w adjacent vertices

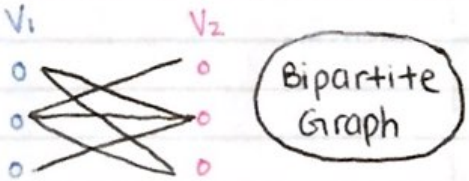


ex: path from Paris to Bahamas
  ↳ Paris - Bahamas
  ↳ Paris - Vancouver - Bahamas
  ↳ Paris - Vancouver - Paris - Bahamas → not simple path!

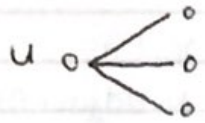
- simple path : no vertex is repeated (in the path)
- cycle = simple path w/ same start/end vertex
- connected G : ∃ path b/w every 2 vertices (otherwise G is disconnected)



- Bipartite graphs : V can be divided into 2 sets V_1 & V_2 s.t. $V_1 \cap V_2 = \emptyset$ and $V_1 \cup V_2 = V$ and adjacencies exist only b/w elements of V_1 & V_2.



- Vertex Degree = # edges adjacent to vertex
  undirected G: only degree
  directed G: in-degree & out-degree

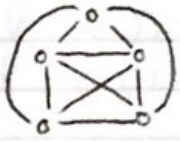


u has degree 3

u has in-degree 1
u has out-degree 2

- clique : every 2 vertices have an edge (undirected graph) (strongly connected graph)

K_4 → 4 vertices



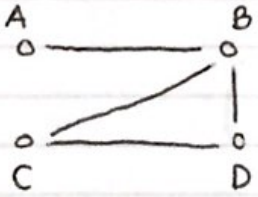
K_5

↳ clique can be contained in a graph, doesn't need to be entire graph

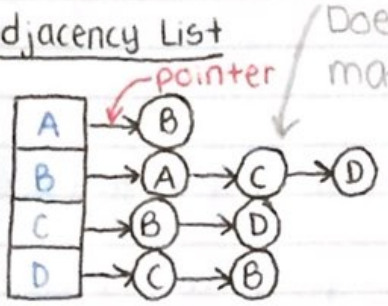
$$\#\text{edges} = \frac{V(V-1)}{2}$$

How do we represent graphs?

Representations:
① Adjacency List
② Adjacency Matrix



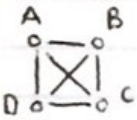
Adjacency List



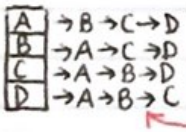
Adjacency Matrix

	A	B	C	D
A		1		
B	1		1	1
C		1		1
D		1	1	

unweighted (for weighted, put weight + magnitude)



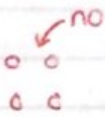
$|V| = n$ (# vertices)



	List	Matrix
Time	$O(n)$	$O(1)$
Memory	~~$O(E)$~~ worst case → $O(n^2)$	$O(n^2)$

↳ #edges

→ $O(V+E)$



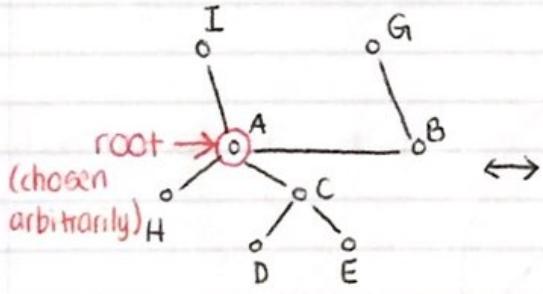
Corollary: If your graph is sparse use a list
↳ $|E| \ll O(V^2)$
otherwise wasted space in matrix

Time → how much time to query/determine if an edge in graph exists

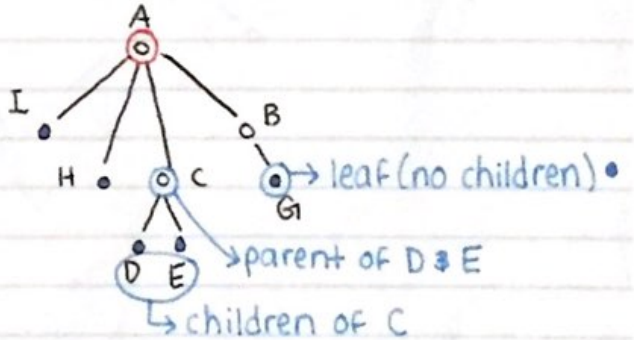
2-4 Intro to Trees

Trees

A tree is a graph that is:
- connected
- acyclic
- undirected



↔

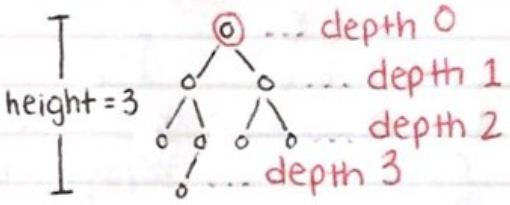


If given a graph where root is not already specified → can pick any node & name it the root.

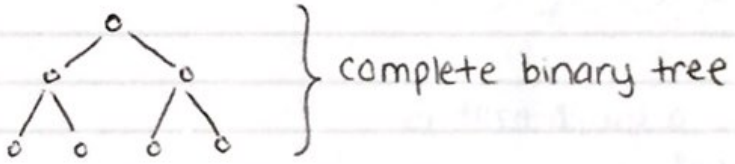
Binary tree : every node has up to 2 children (generalize to k children → k-ary tree)

Depth of a node: length of path frm root to that node.

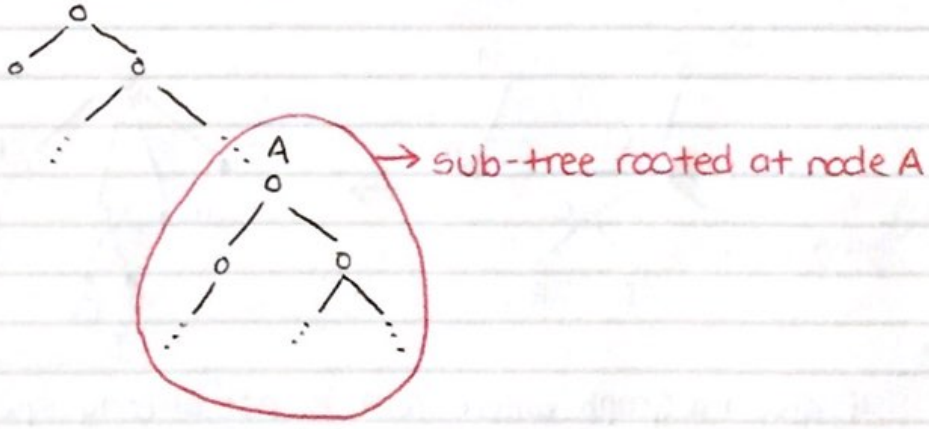
Height: # of edgest of longest path frm the node to a leaf.



Complete k-ary tree: every internal node (every node except leaves) has exactly k children & all leaves have same depth



Sub-tree rooted at a node:



Theorem: Every 2 of following statements are equivalent.

1. G is a tree.
2. Every 2 vertices in G are connected by a unique simple path.
3. G is connected but if any edge is removed, resulting graph is disconnected.
4. G is connected $\Leftrightarrow$ $|E| = |V| - 1$ (#edges = #vertices - 1)
5. G is acyclic $\Leftrightarrow$ $|E| = |V| - 1$
6. G is acyclic but if edge is added then new graph has a cycle.

3-1 Induction

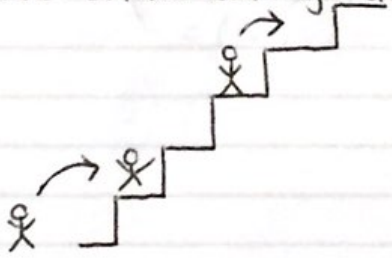
Induction (proof technique):
Predicate P ie., $P(n)$: "$n \leq 2^n$, $\forall n \geq 1$"

3 Steps: Basis, Hypothesis, & Inductive Step

$$[P(1) \wedge \forall n\ (P(n) \rightarrow P(n+1))] \rightarrow \forall n\ P(n)$$

$P(1)$: Basis
$P(n)$: Hypothesis
$P(n+1)$: Inductive Step
↳ use Hypothesis!

Graphically: child & stair case
- basis: first step
- hypothesis: child can start at any step & move up 1
- inductive: child can go all the way up



Ex1: Prove $1+2+3+\ldots+n = \frac{n(n+1)}{2}$ (Arithmetic series)

Basis: Show it holds for $n=1$:

$1 = \frac{1(1+1)}{2}$ true

Hypothesis: Assume $1+2+\ldots+n = \frac{n(n+1)}{2}$

Step: Need to prove: $\underbrace{1+2+\ldots+n}_{\frac{n(n+1)}{2}} + (n+1) = \frac{(n+1)((n+1)+1)}{2}$

$\frac{n(n+1)}{2} + (n+1) =$ true

(use hypothesis)

Ex2: Prove that the sum of the first n odd positive integers is n^2.

Basis: $n=1$ or "the first odd int"

$1 = 1^2$

Hypothesis: $1+3+5+\ldots+(2n-1) = n^2$

assume it is true

Step: Need prove it works for the first $(n+1)$-positive odd integers.

$\underbrace{1+3+5+\ldots+(2n-1)}_{n^2 \text{ (hypothesis)}} + (2n+1) = (n+1)^2$

$n^2 + (2n+1) =$

3-2 Induction (cont.)

Prove that $n < 2^n$, $\forall\, n \geq 2$.

Basis: It holds $n = 2$ $\quad 2 < 2^2$

Hypothesis: Assume $n < 2^n$ for up to n

Step: Need show for $n+1$. We got

$$n+1 < 2^n + 1 \leq 2^n + 2^n = 2^{n+1}$$

Prove that for S its powerset 2^S has 2^n subsets.

Powerset

$S = \{A, B, C\}$

Powerset S or 2^S : $2^S = \{0, A, B, C, AB, AC, BC, ABC\}$

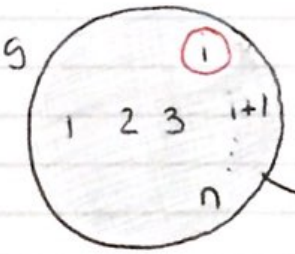
It has 2^3 elements (or subsets)

Powerset is combination of every element in set including empty set.

Basis: $n = 1$ (obvious)

Hypothesis : Assume it's true for a set w/ n-elements.

Step: Need to show it works for set S where $|S| = n+1$ elements.



Hypothesis applies to n elements
→ look at smaller set

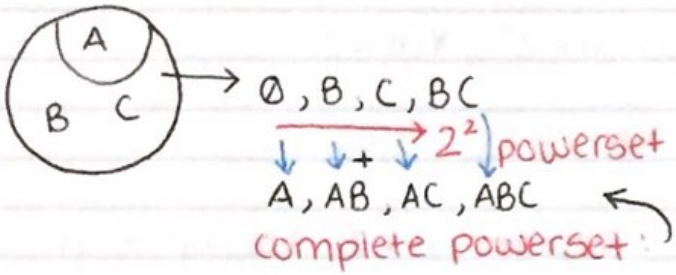
2^n (hypothesis)

$2^n + 2^n = 2^{n+1}$

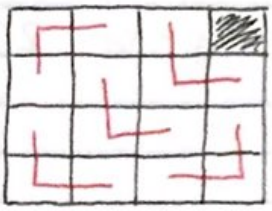
↓

if we attach element i in each element of the yellow powerset

Back to $S=\{A,B,C\}$ ...



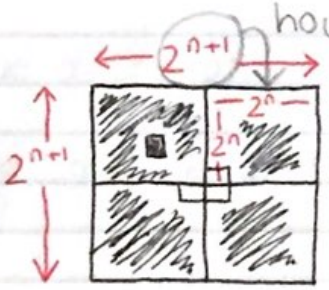
Show that every $2^n \times 2^n$ chessboard w/ any one square blank it can be tiled w/ ⊞-shape tiles.



Basis: $n=2$ (example before)

Hypothesis: we assume that any $2^n \times 2^n$ chess board w/ any square left blank can be tiled by ⊞-tiles.

Step:



how? page before says

$2^n + 2^n = 2^{n+1}$

3-3 Contradiction

Proposition $P(n)$ = T or F, assume opposite.

Assume $\neg P$ (NOT P) holds

$\neg P \rightarrow \ldots \rightarrow \ldots \rightarrow$ False

Eg. You cannot grow cherries in February outside in Ontario.
Assume you can grow cherries. → False!

Ex:
Show that if $(3n+2)$ = odd then n = odd.

Assume that $(3n+2)$ = odd BUT n = even.
If n = even: $3n+2 = 3(2k)+2$
$= 6k+2$
$= 2(3k+1)$
= even — a contradiction.

Ex:
If $x^2-5x+4 < 0$ then $x > 0$

Assume toward a contradiction (ATaC) that $x^2-5x+4<0$
BUT $x \leq 0$.

Then, $x^2 \leq 5x-4$ but if $x \leq 0$
then $x^2 < 0$
which is a contradiction.
(power of 2 cannot be <0 → can be 0 at worst case)

Ex:
Show that $\sqrt{2}$ = irrational
ATaC that $\sqrt{2}$ = rational
→ $\sqrt{2} = a/b$ where a & b no common factors
→ $2 = a^2/b^2 \rightarrow a^2 = 2b^2$
→ a^2 = even # → $a = 2c$ (*)
→ $2^2c^2 = 2b^2 \leftrightarrow b^2 = 2c^2$ = even

this is a contradiction b/c we assume that a & b have no common factors but now they are both even!

(*) we will prove that a^2 = even → a = even

ATaC that a^2 = even → a = odd

$a = \underbrace{2k+1}_{\text{odd}} \rightarrow a^2 = (2k+1)^2 = 4k^2+1+4k$
$= 2(2k^2+2k)+1$
= odd

a contradiction as a^2 = even.

* common problem: hypothesis vs. assumption.
↳ use hypothesis in induction step!

4-1 Recurrences Mergesort

Recurrence: a quantity that is defined recursively in terms of smaller values of itself.

$T(n) = 3T(\frac{n}{2}) + 4T(\frac{n}{5})$
= closed form?
$O(n)$?
$O(n \log n)$?
$O(n^2)$...?

Mergesort

Mergesort(A, p, r) ← A: array containing #s; p: left index; r: right index

if p < r

$q = \lfloor \frac{p+r}{2} \rfloor$

Mergesort(A, q+1, r) → $T(\frac{n}{2})$ ← ½ inputs

Mergesort(A, p, q) → $T(\frac{n}{2})$

Merge(A, p, q, r) → $\Theta(n)$

↳ takes 2 already sorted array & puts them together in new sorted array

↳ 2 arrays → one indexed frm p to q, other q to r

$T(n)$ = total time of Mergesort

$T(n) = T(\lceil \frac{n}{2} \rceil) + T(\lceil \frac{n}{2} \rceil) + \Theta(n)$

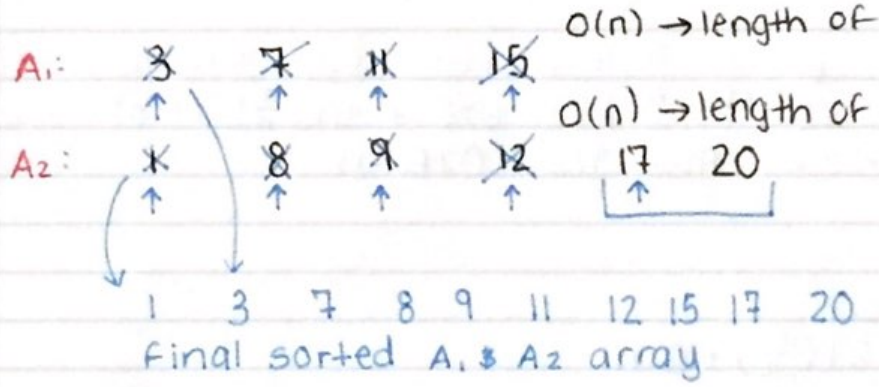
$T(n) = 2T(\lceil \frac{n}{2} \rceil) + \Theta(n)$

↳ what is closed form of this?

↳ $\lceil \ \rceil$: ceiling

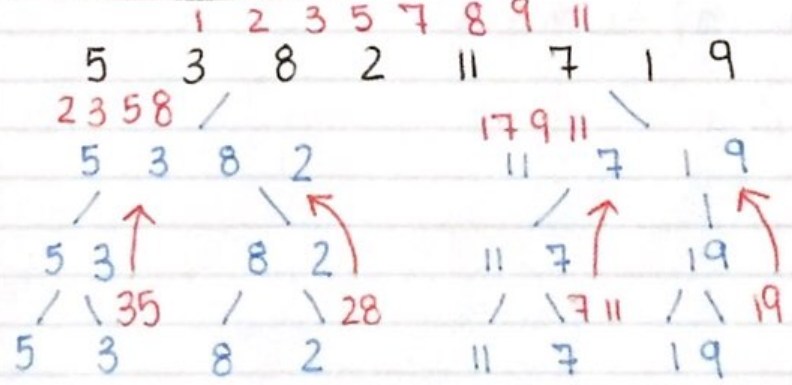
Merge:

- pointers advance & compare values



Total Time: O(n)

Mergesort Example



what is run time?

4-2 Recurrences Master Theorem

Master Method (cookbook)

Let $a \geq 1$ & $b > 1$ & $f(n)$ some fcn. Recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$ ← use to solve recurrences w/ this format

has soln:

Final asymptotic value of $T(n)$ depends on how these compare ↳ which is bigger

CASE 1:
If $f(n) = O(n^{\log_b a - \epsilon})$ for $\epsilon > 0$
then $T(n) = \Theta(n^{\log_b a})$

* need to find a specific (positive, NOT $=0$) epsilon

CASE 2:
If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

CASE 3: ↙ lower bounded → $f(n)$ grows very fast
If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for $\epsilon > 0$ & $af\left(\frac{n}{b}\right) \leq cf(n)$
For $0 < c < 1$ then $T(n) = \Theta(f(n))$

Ex:
Mergesort
$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$
$a = 2$
$b = 2$
Application of case 2.

$T(n) = \Theta(n \log n) \rightarrow \Theta(n \log n)$

$\log_b a = \log_2 2 = 1$

Ex:
(Trying to find closed form)

$T(n) = 9T(\frac{n}{3}) + n$
$a = 9 \quad b = 3 \quad f(n) = n$
$n = O(n^{2-1})$ for $\epsilon = 1$

Case 1:
$T(n) = \Theta(n^{2})$ → $\log_3 9 = 2$

Ex:

$T(n) = 3T(\frac{n}{4}) + n\log n$
$a = 3 \quad b = 4 \quad f(n) = n\log n$

We got $f(n) = \Omega(n^{\log_4 3 + 0.2})$
B/c $\log_4 3 = O(n^{0.793})$

We got $3f(\frac{n}{4}) = 3\frac{n}{4}\log\frac{n}{4} \leq \frac{3}{4}n\log n$

condition holds, hence: $T(n) = \Theta(n\log n)$

Week 3 Videos: 4-3 Recurrences Substitution

Substitution (Induction)
Mergesort
$T(n) = 2T(\lceil \frac{n}{2} \rceil) + n$

we "guess" the answer, & then use induction to prove it.

We guess $T(n) = O(n\log n)$ & we will prove it.

Base: (later...)

Hypothesis:
Assume it holds $\forall$ strictly smaller than n
(& we will prove n)

$$T(\tfrac{n}{2}) \leq c \lfloor \tfrac{n}{2} \rfloor \log \lfloor \tfrac{n}{2} \rfloor$$

Step:
we need prove that
$T(n) \leq cn\log n$ ← apply hypothesis
$$T(n) \leq 2c \lfloor \tfrac{n}{2} \rfloor \log \lfloor \tfrac{n}{2} \rfloor + n$$
$$= cn\log n - cn\log 2 + n$$
$$= cn\log n - cn + n$$
$$= cn\log n + (1-c)n$$
$$\leq cn\log n \quad \text{for } c \geq 1$$

Base:
We try prove that $T(n) \leq cn\log n$
$n=1$
$T(1) \leq c1\log 1 = 0$
"Arbitrarily" set $T(2)=4$
$T(3)=5$ ... & see if it works!

$T(2) \leq c2\log 2 = 2c$
$T(3) \leq c3\log 3 = c \cdot 3 \cdot 1.584$
$= c \cdot 4.755$
they work for $c \geq 3$
start base from $n=2$ (double base)

Erroneous Method
we guess $T(n)=O(n)$ ... we build the proof (step)

$$T(n) \leq 2c \lfloor \tfrac{n}{2} \rfloor + n$$
$$\leftrightarrow T(n) \leq cn + n = (c+1)n \quad \text{QED!}$$

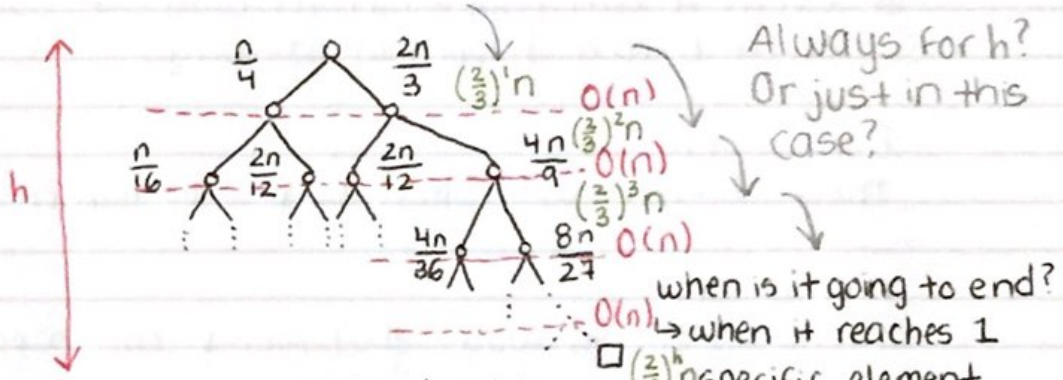
Error: different constant! $c+1$ not c.

4-4 Recurrences Tree

Recursion Tree

$T(n) = T(\frac{n}{4}) + T(\frac{2n}{3}) + n$

Draw recursion tree of this recurrence.



Total amount of work = $T(n) = h \cdot O(n)$

How much work does this recurrence do at every level?
→ does work that is proportional to $O(n)$

→ h? what is height of tree?

$(\frac{2}{3})^h \cdot n = 1 \leftrightarrow$
$n = (\frac{3}{2})^h \leftrightarrow$
$h = \log_{3/2} n = O(\log n)$

$h = \log n$
$\Rightarrow T(n) = O(n \log n)$

5-1 Rules of Sum & Product

Permutations & Combinations

→ rule of sum & product form basis for these

Suppose we have 2 events...
IF event A can happen in m-ways &
event B can happen in n-ways

Rule of Product

There $(n \times m)$-ways that A & B can happen.

Rule of Sum

There are $(n+m)$-ways that A or B can happen.

Ex:
You have 5 Latin
7 Greek
10 French books.
In how many ways one can select 2 different books?

$5 \times 7 + 5 \times 10 + 7 \times 10 = 155$ ways
↑
or

In how many ways one can select any 2 books?

22×21 ways

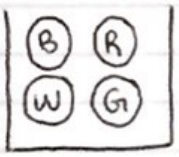
22 ways to select first book
21 books to select 2nd book

5-2 Permutations

Permutations

$P(n, r)$ = the # ways to arrange r out of n distinct objects where the order is important

Suppose have bucket... w/ colour balls...



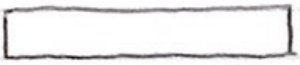
B R G ≠ R B G

$$P(n,r) = n \cdot (n-1) \cdot (n-2) \dots (n-r+1)$$
$$= \frac{n!}{(n-r)!}$$

→ n ways to select 1st item
→ (n-1) ways to select 2nd item ...
⋮

Ex:
In how many ways n-ppl can sit to form a ring?

Suppose put ppl. on table.



Sitting on table $P(n,n)$

Can rotate round table → result stays same.

$$\frac{P(n,n)}{n} = (n-1)!$$

↳ # ways n-ppl can sit in a ring

If not all objects are distinct but...
q_1 1st kind
q_2 2nd kind
⋮
q_t t^{th} kind

ways "objects can be arranged" / to permute objects:

$$\frac{n!}{q_1!\,q_2!\,\ldots\,q_t!}$$

Ex: 5 dashes & 8 dots can be arranged in $\frac{13!}{5!\,8!}$ ways

Ex:
Show that $(k!)!$ is divisible by $(k!)^{(k-1)!}$

Want to show.
$\frac{(k!)!}{(k!)^{(k-1)!}}$ = some integer

may show up in exam
Use a combinatorial argument
Consider $k!$ objects as follows:

k of 1st kind
k of 2nd kind
⋮
k of $(k-1)!$ kind
total of $k!$ objects → In how many ways can we permute them (these objects)?

$$\frac{n!}{q_1!\,q_2!\,\ldots\,q_t!}$$

→ b/c k for 1st kind, k for 2nd...

$$\frac{k!}{\underbrace{k!\,k!\ldots k!}_{(k-1)!}} = \frac{(k!)!}{(k!)^{(k-1)!}}$$

QED

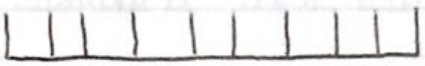
Ex:
Among 10 billion #s b/w [1 ... 10,000,000,000] how many contain the digit "1"?
(11 digits)

look at range [0 ... 9,999,999,999]
(10 digits)

How many do not contain 1?

Imagine have 9 buckets & throwing #s into buckets, digits 0-9, cannot throw 1

eliminate 0



→ In 9^{10} ways → why to power of 10?
9: 0, 2-9 ; 10: # digits in number

9^{10} (-1) do not contain "1" frm original range, which means that $10^{10} - (9^{10} - 1)$ contain "1"
→ in range [0 ... 9,999,999,999] but original range does not include 0 so (-1)

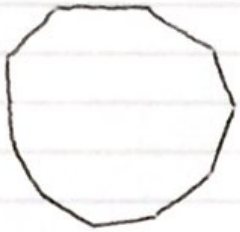
5-3 Combinations

Combinations
C(n,r) = # of combinations of r out of n-objects where order does not matter.

$$C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!(n-r)!} = C(n,n-r) = \binom{n}{r}$$

↳ "n choose r"

Ex:
How many diagonals a decagon has?



$C(10,2) - 10 = 35$

↳ the actual decagon sides!

Ex:
In how many ways can 3 #s be selected frm $1, 2, \ldots 300$ so that their unique sum is divisible by 3?

There are 100 #s that leave remainder 1
=11= =11= 2
=11= =11= 0

$C(100,3) + C(100,3) + C(100,3) + 100^3$ ← select 1 # frm each of abv. groups of 100 #s

$= 1,485,100$

Ex:
11 scientists work on a secret project. They lock the project documents into a cabinet st. the cabinet can open iff at least 6 scientists are present w/ their keys.

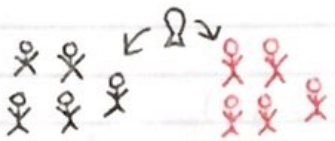
(A) What is the smallest # locks needed?

(B) what is the smallest # keys each scientist needs to have?

bijection: 1-1 correspondence

Ⓐ ∀ group of 5 scientist ⟷ 1 lock they cannot open

$C(11,5) = \frac{11!}{5!6!}$



$C(11,6) = \frac{11!}{6!5!}$

For every 2 diff groups of 5 scientists, there should be at least 1 lock that these 5 cannot open.

This lock should be diff. for ∀ group of 5 (or you will have 6 or more scientist that cannot open)

∀ group of 5 ⟷ ∃ 1 lock

Result: $C(11,5) = 462$ locks

Ⓑ Everytime an individual scientist goes to a group of 5 he needs to have key open the lock those 5 cannot.

$C(10,5) = 252$ keys

6-1 Probability Intro

Probability

Experiment over sample space S w/ outcomes that we call "events"

Ex: Fair coin $S = \{H, T\}$

2 fair coins $S = \{HH, HT, TH, TT\}$

what we believe in math → truths

Probability Axioms

- $Pr(a \in S) \geq 0$
- $Pr(S) = 1$
- $Pr(A \overset{OR}{\cup} B) = Pr(A) + Pr(B) - Pr(A \cap B)$
- $Pr(A \overset{AND}{\cap} B) = Pr(A) \cdot Pr(B)$ if events indy (independent)
- if $\forall$ events $a \in S$ we have $Pr(a) = \frac{1}{|S|}$
  then Uniform Probability Distribution

Ex:
Flip 2 fair coins $Pr\{\text{at least one H}\}$?

$$Pr\{\text{at least one H}\} = Pr\{HH, HT, TH\}$$
$$= \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = \frac{3}{4}$$

Ex:
Flip a fair coin n times.

$$P\{\text{exactly k heads}\} = \frac{\binom{n}{k}}{2^n}$$

↳ 2 possibilities $\{H, T\}$, n flips

2^n: uniformly distributed outcomes

$\binom{n}{k}$: # sequences w/ k heads

Bayes' Theorem (on conditional probability)

$$Pr(A|B) = \frac{Pr(A \cap B)}{Pr(B)} = \text{"Probability of A happening, if B happens?"}$$

Equiv: $Pr(A \cap B) = Pr(A|B)\,Pr(B)$
$= Pr(B|A) \cdot Pr(A)$

$$Pr(A|B) = \frac{Pr(B|A)\cdot Pr(A)}{Pr(B)}$$

$$= \frac{Pr(B|A)\,Pr(A)}{\underbrace{Pr(A)Pr(B|A) + Pr(\bar{A})Pr(B|\bar{A})}_{Pr(B)}}$$

$Pr(\bar{A})Pr(B|\bar{A})$ → A does not happen

Ex: A = both coins H
B = at least one H

(Flip 2 fair coins)

$$Pr(A|B) = \frac{Pr(A\cap B)}{Pr(B)} = \frac{1/4}{3/4} = \frac{1}{3}$$

Ex: we have 2 coins → one is fair, one is biased & brings H always

experiment: pick one coin at random & flip twice

$$Pr\left(\underbrace{\text{Picked biased}}_{A} \,\middle|\, \underbrace{\text{we observe 2 Hs}}_{B}\right) = Pr(A|B)$$

$$= \frac{^{1}Pr(B|A)\cdot Pr(A)^{1/2}}{Pr(A)\,Pr(B|A) + Pr(\bar{A})\,Pr(B|\bar{A})}$$

$Pr(A)$: $1/2$; $Pr(B|A)$: 1; $Pr(\bar{A})$: $1/2$; $Pr(B|\bar{A})$: $1/4$

$$= \frac{1\cdot 1/2}{1/2 + 1/8} = \frac{4}{5}$$

6-2 Discrete Random Variable

Discrete Random Variables

A function: maps events in sample space S to real #s

r.v. (random var) X $\Pr(X=x) = \sum_{\{s \in S : X(s)=x\}} \Pr(s)$

Ex: pair of dice
X = max of 2 rolls

$\Pr(X=3) = \sum [s \in \underbrace{1\text{-}3, 2\text{-}3, 3\text{-}3, 3\text{-}2, 3\text{-}1}_{\text{in these cases, max is 3}}] = \frac{5}{36}$

Expected Value → average

$$E[X] = \sum_{x \in X} x \cdot \Pr(X=x)$$

Ex: you flip 2 coins
win \$3 for H, lose \$2 for T
r.v. X = earnings

$E[X] = \underset{\$3+\$3}{6}P(HH) + \underset{\$3-\$2}{1}P(HT) + 1P(TH) + (-4)P(TT)$
$= 6 \cdot \frac{1}{4} + \frac{1}{4} + \frac{1}{4} - 4 \cdot \frac{1}{4}$
$= \$1$
↑ expected earning if we play

Properties:

$E[x+y] = E[x] + E[y]$

$E[ax] = aE[X]$

If x & y are independent: (this must be true)
$E[x \cdot y] = E[x] \cdot E[y]$

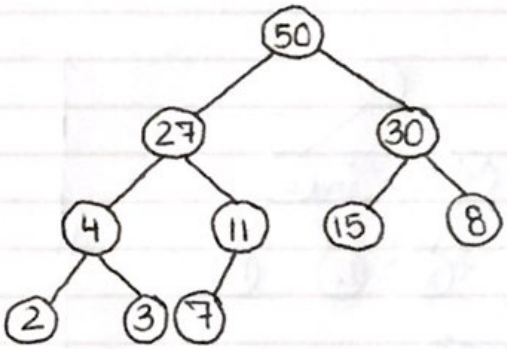
} Proof is good hwk!

Week 4 Videos: 7-1 Intro to Heapsort

Heaps & Heapsort

In place sorting algorithm: Does not require more memory other than the array that contains the #s
↳ does not need more space than array provided to sort

Properties of a Heap:
It is a binary tree with
ⓐ Heap Shape Property 
ⓑ Heap Ordering: key parent > key child
↳ do NOT confuse w/ BST!
↳ no relative ordering b/w siblings

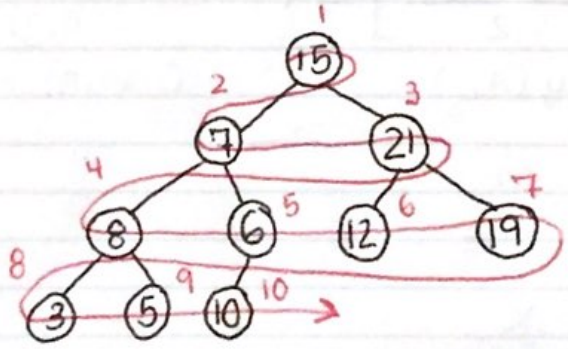


Array A

1	2	3	4	5	6	7	8	9	10
15	7	21	8	6	12	19	3	5	10

← indices start at 1!

algo operates in terms of array → tree diagram is just to help visualize



} not heap b/c does not have heap order

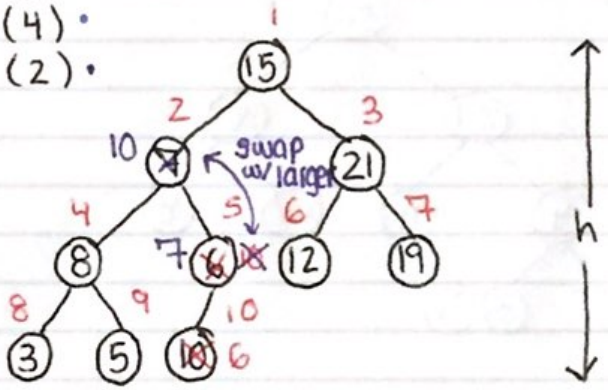
For node i
- parent is $\lfloor \frac{1}{2} \rfloor$ ← floor
- left child is $2i$
- right child is $2i+1$

7-2 Heap Operations

Heapify (or "bubble down")

Heapify (A, i) → O(log n)
  compare A(i) w/ A(2i), A(2i+1)
  swap w/ larger
  repeat until ✗ swap

Heapify (5) •
Heapify (4) •
Heapify (2) •



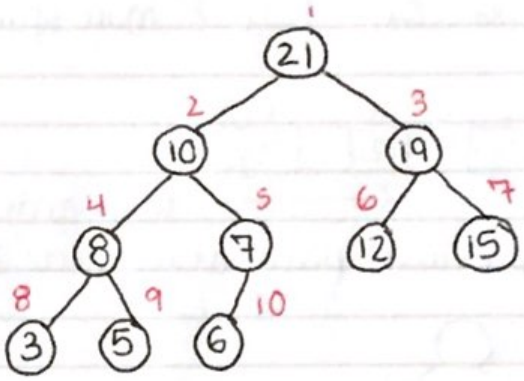
Build Heap

Build Heap (A)
  for $i = \lfloor \frac{\text{length}(A)}{2} \rfloor \rightarrow 1$ — O(n) ↳ call n times
    Heapify (A, i) — O(n) × O(log n)

time: O(n log n)

	1	2	3	4	5	6	7	8	9	10
A	21	10	19	8	7	12	15	3	5	6

Previous heap tree example sorted:

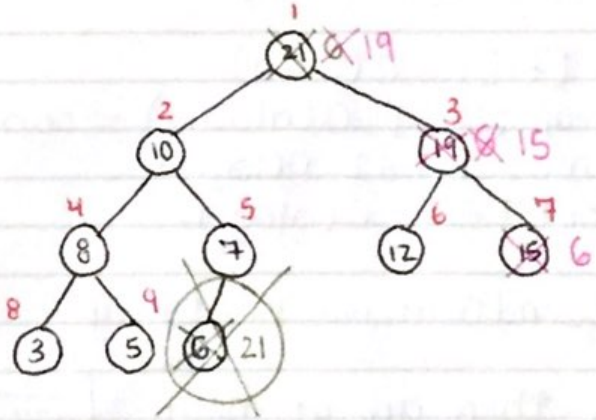


7-3 Heapsort

Where is the max element in a heap?
Max Heap: max element is at the root)
(min heap: smallest at the root)

Delete Max(A) O(logn)

swap(A(1) ↔ A(length(A))
length(A) = length(A) - 1
Heapify(A,1)



Heapsort(A)
  Build Heap(A) → O(n)
    for i=1 ... length(A) O(n)
      Delete Max(A) O(log n)

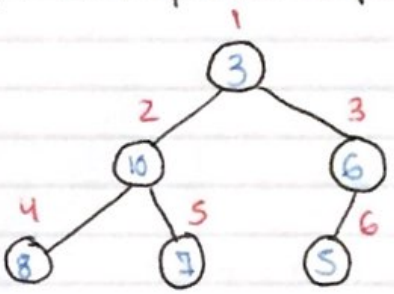
} in place algo, does not require more space

total O(n log n)

	1	2	3	4	5	6	7	8	9	10
A	3	10	6	8	7	5	12	15	19	21

← #s in order

→ graph from previous page after a few iterations of Heapsort(A)



7-4 Heap Runtime & Priority Queue

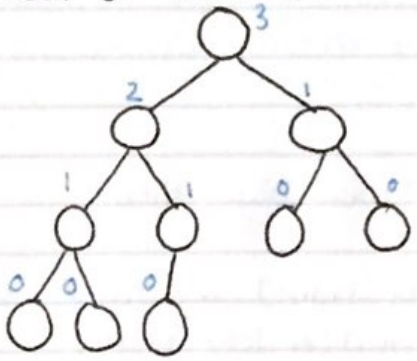
What is the runtime of heapsort?
→ can prove by induction that height of tree w/ n nodes is O(log n)
  ↳ ∴ heapify is O(log n)

Heapify [7-2] : O(log n)
Build Heap [7-2] : O(n log n) → can prove tighter bound
Delete max [7-3] : O(log n)
Heapsort [7-3] : O(n log n)

Tight bound O(n) for Build Heap

Lemma: There are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes of height h in a heap

Intuition for proof of Lemma:



height

$n = 10$

$\left\lceil \frac{10}{2^{1+1}} \right\rceil = \left\lceil \frac{10}{4} \right\rceil$

$= 3$

↳ there are at most 3 elements of height one

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) \quad (1)$$

↑ height of heap; ↑ lemma; ↳ # elements at particular level

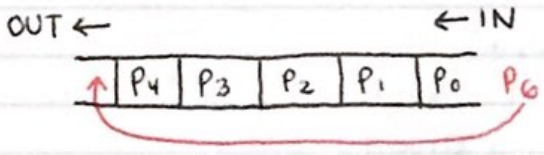
$$= O\left(n \Sigma \left(\frac{1}{2}\right)^h \cdot h\right) = O(2n) = O(n)$$

similarity!

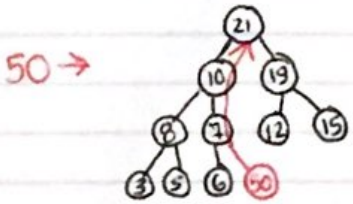
$$\text{But } \sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2} \overset{x=1/2}{=} \frac{1/2}{(1-1/2)^2} = 2$$

∴ Tight bound $O(n)$ for Build Heap $\rightarrow O(n)$

Priorities Queues



Priority queues can implemented w/ heaps!
- insert → (similar to bubble down, but bubble up)
- extract_max

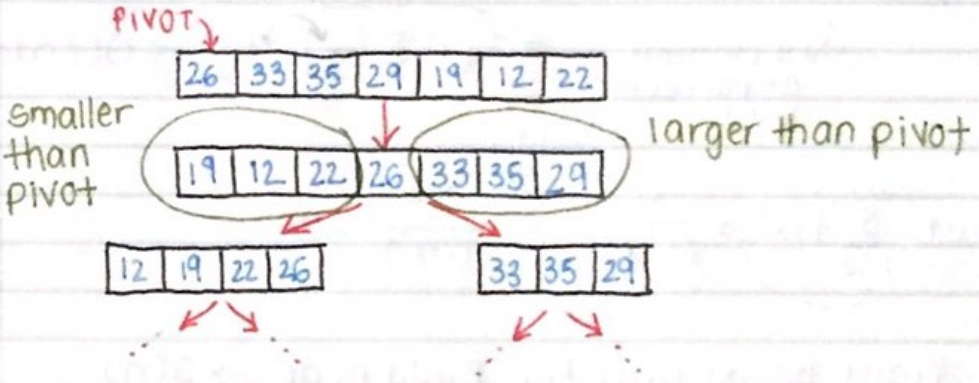


8-1 Quicksort Introduction

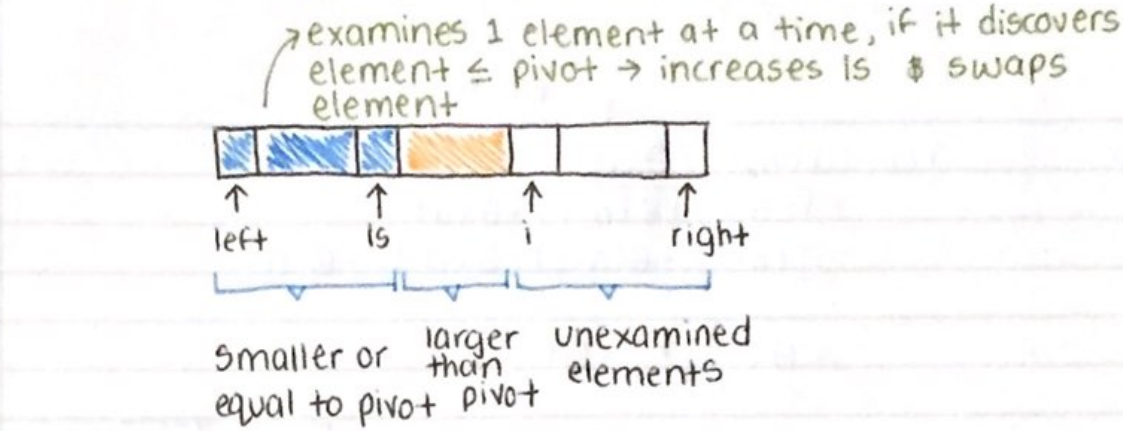
Quicksort

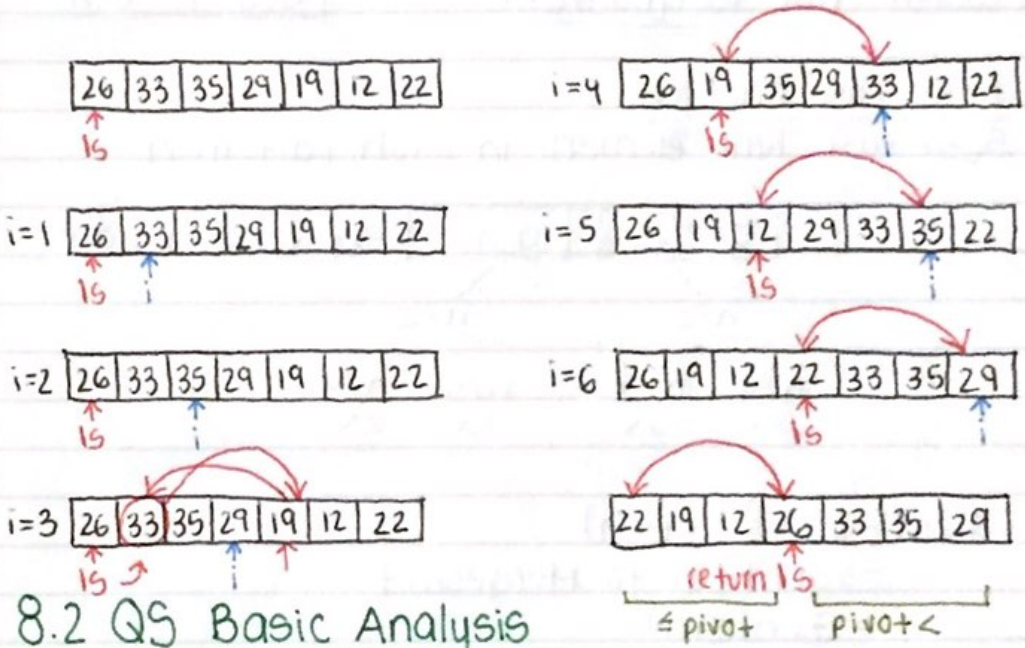
A "divide & conquer" algorithm

```
Quicksort (A, left, right)  // Array A, left & right indices of array
    pivot = Partition (A, left, right)
    if (pivot > left)
        Quicksort (A, left, pivot)
    else if (pivot ≤ right)   if (pivot+1 < right)
        Quicksort (A, pivot+1, right)
```




```
int Partition (A, left, right)
    ls = left
    pivot = A(left)
    for i = left+1 ... right
        if (A(i) ≤ pivot)
            ls = ls + 1
            swap (A(i), A(ls))
    swap (A(left), A(ls))
    return ls
```





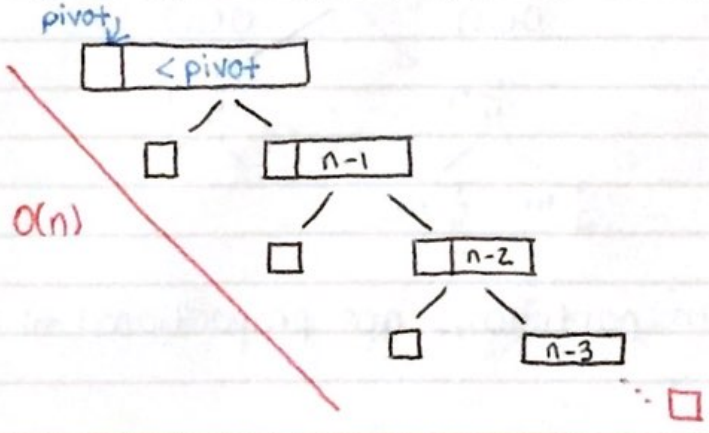
8.2 QS Basic Analysis

Worst Case Analysis (descending)

Worst case is when the array is sorted (or reverse sorted) (ascending order)

partition

$T(n) = T(n-1) + \Theta(n) = T(n-2) + \Theta(n-1) + \Theta(n) \ldots$



partition

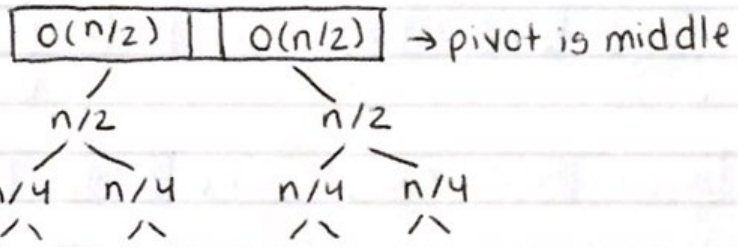
$$\ldots T(n) = T(n-1) + \Theta(n)$$
$$= T(n-2) + \Theta(n-1) + \Theta(n)$$
$$= T(n-3) + \Theta(n-2) + \Theta(n-1) + \Theta(n)$$
$$= \ldots$$
$$= \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

worst case is quadratic!

O(1)	O(n)

Best Case Analysis

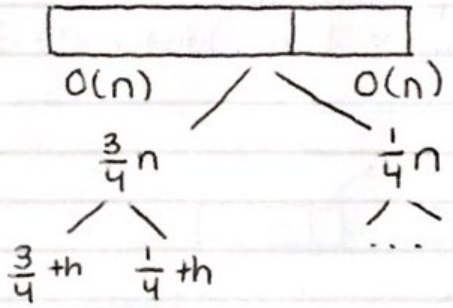
Best case: half element in each partition



$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
$$= \text{identical to Mergesort}$$
$$= \Theta(n \log n)$$

Expected or "Balanced" Partition

- pivot is not left most or middle element

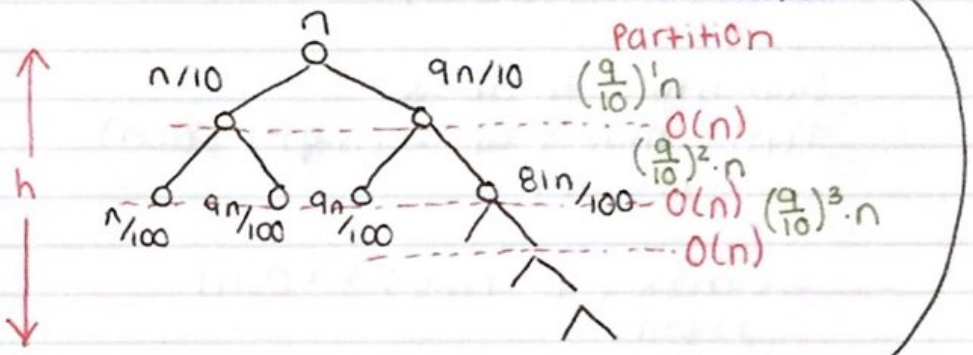


Both partitions are proportional to O(n).

Illustrative example:

$9/10^{ths}$ in one partition &
$1/10^{th}$ on the other

$T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + \Theta(n) = \Theta(n \log n)$ ← time to partition
↑ smaller constant



total amount of work
$= h\,\Theta(n) = \Theta(n \log n)$ ← substitute h
↳ work at every level

h?
$(\frac{9}{10})^h \cdot n = 1 \leftrightarrow (\frac{10}{9})^h = n \leftrightarrow h = \log_{10/9} n = O(\log n)$

suppose quicksort giving much worse partition (than $\frac{9n}{10}$):

$T(n) = T(\frac{9999n}{10000}) + T(\frac{n}{10000}) + \Theta(n)$
$= \Theta(n \log n)$ → still $\Theta(n \log n)$!
↳ bigger constant

8-3 QS Worst Case

(official proof for worst case)
→ previous was not official b/c no one told us sorted sequence exhibits worst behaviour

Worst Case of Deterministic Quicksort (official)

$$T(n) = \max_{1 \le q \le n-1} \{T(q) + T(n-q)\} + \Theta(n)$$

Use substitution:
We guess $T(n) \le cn^2$

Substitute to above:

$$T(n) \le \max_{1 \le q \le n-1} \{cq^2 + c(n-q)^2\} + \Theta(n)$$

$$= c \max_{1 \le q \le n-1} \{q^2 + (n-q)^2\} + \Theta(n)$$

→ for what value of q results in maximized value

Achieves max at endpoints 1 & n-1
b/c 2nd derivative wrt q is +ve.

Set $q = 1$, then:

$$T(n) \le cn^2 - 2c(n-1) + \Theta(n)$$
$$\le cn^2$$

for large value of c st. it dominates the constant in $\Theta(n)$ & $-2c(n-1) + \Theta(n)$ becomes negative.

Therefore, (officially):

$T(n) = \Theta(n^2)$ for worst case

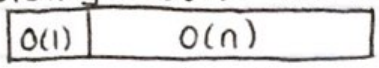
8-4 Randomized QS

Randomized Quicksort

```
Randomized_Partition(A, left, right)
    i = random(left, right)
    swap(A[left], A[i])
    return Partition(A, left, right)
```

Average case
= O(cnlogn)
↓
constant
(hides)
↳very small
compared to
other sorting
techniques

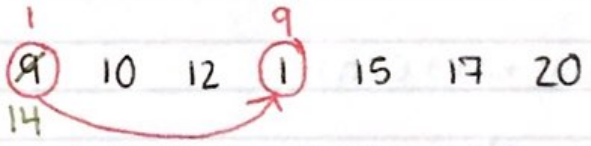
Avoiding: (worst case behaviour)



→ picks random element & swaps w/ left-most element

Is it possible for Randomized QS to demonstrate worst behaviour? yes

Suppose you have...



(typo in lecture → 9 should be 14)

Sometimes may get worst case, but over sequence of this procedure → behaviour of algo O(nlogn)

Average Case for Randomized QS

$$T(n) = \frac{1}{n}(T(1) + T(n-1))$$

$\times \frac{1}{n} = \Theta(n)$ → absorbed

$$\sum_{q=1}^{n-1} T(q) + T(n-q) + \Theta(n)$$

partition

worst case is n^2

$$\frac{1}{n}(T(1) + T(n-1)) \leq \frac{1}{n}(\Theta(1) + \Theta(n^2))$$

$$= \Theta(n)$$

Therefore,

$$T(n) = \frac{1}{n} \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n)$$

$$= \frac{2}{n} \sum_{k=1}^{n-1} T(k) + \Theta(n) \quad (2)$$

k b/c move from 1 expression to another

We will use substitution (induction) to show that this is: $an\log n + b$, for large a & b

Lemma (to prove later)

$$\sum_{k=1}^{n-1} k\log k \leq \frac{1}{2}n^2\log n - \frac{1}{8}n^2$$

} assume true for now

$$(2) \leq \frac{2}{n} \sum_{k=1}^{n-1} (ak\log k + b) + \Theta(n)$$

$$= \frac{2a}{n} \sum_{k=1}^{n-1} k\log k + \frac{2b}{n}(n-1) + \Theta(n)$$

→ (using the Lemma)

$$\leq an\log n - \frac{a}{4}n + 2b + \Theta(n)$$

$$= an\log n + b + (\Theta(n) + b - \frac{a}{4}n)$$

$$\leq an\log n + b$$

for large enough a to make this negative

... back to Lemma...:

$$\sum_{k=1}^{n-1} k\log k = \sum_{k=1}^{\lceil \frac{n}{2}-1 \rceil} k\log k + \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k\log k$$

$$\leq \underbrace{(\log \tfrac{n}{2})}_{(\log n - 1)} \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k$$

$$= \log n \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k + \log n \sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} k$$

$$= \log n \sum_{k=1}^{n} k - \sum_{k=1}^{\lceil \frac{n}{2} \rceil - 1} k \leq$$

$$\leq \frac{1}{2} n(n-1)\log n - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2}$$

$$\leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

→ randomized cannot eliminate worst case, presents opportunity for expected case to be guaranteed of $O(n\log n)$

Week 5 Videos: 9-1 Lower Bound on Sorting & Counting Sort

Lower bound in comparison-based sorting

Any sorting algorithm on n-elements that uses comparisons to sort elements of unlimited range (ie., $-\infty$ to ∞) makes no better than $\Omega(n\log n)$

Will learn about algo. w/ run time of $O(n)$
↳ counting sort & radix sort
↳ Do they violate the lower bound? No! B/c they sort numbers w/in a range.

Counting Sort

A = array with the numbers #s are in range [0...k]
Not in place as it uses auxilliary arrays but it is a stable sorting algorithm.

Stable sorting

$5_a\ 4\ 3_a\ 12\ 5_b\ 3_b\ 7\ 5_c$

$3_a\ 3_b\ 4\ 5_a\ 5_b\ 5_c\ 7\ 12$

```
C[i] = 0   ∀ i ∈ [0...k]

for j = 1 ... length(A) do
    C[A(j)] = C[A(j)] + 1
```
counts how many #s exist from every # in original range in array A — O(n)

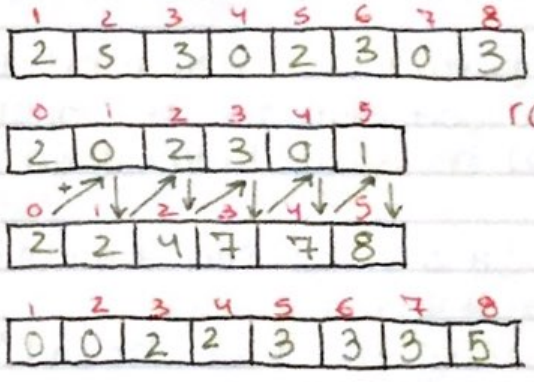
```
for i = 1 ... k
    C[i] = C[i] + C[i-1]
```
prefix sums → smaller or = to # — O(k)

```
for j = length(A) ... 1
    B[C[A[j]] = A[j]
    C[A[j]] = C[A[j]] - 1
```
places #s in appropriate position — O(n)

total time O(n+k)
k = range of #s
if k = O(n) then O(n+k) = O(n) linear time

Example:

range [0...5]



$B(C(A(j))) = A(j)$ for $j = A \ldots 1$

$C(A(j)) = C(A(j)) - 1$

$B(C(A(8))) = A(8)$

$B(C(3)) = A(8)$

$B(7) = A(8)$

9-2 Radix Sort

Radix Sort

Stable sorting algorithm

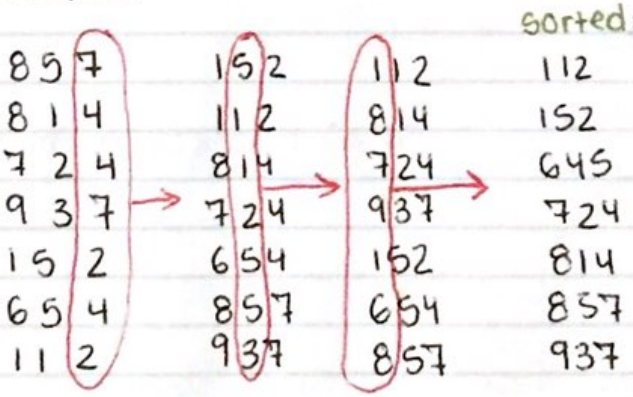
Key idea: sort from LSB → MSB (or digit)

How algo operates:

→ # digits

```
Radix Sort(A, d)
    for i = 1...d
        stable sort A on digit-i
            ↳ use counting sort
```

Example



Runtime

n = # numbers
k = range of the digits
d = # digits $\forall$ number

one pass: $\theta(n+k)$ counting sort
all passes: $d \cdot \theta(n+k) = \theta(dn+dk)$

If we assume that d, k = constants
then total time = $\theta(n)$

Motivational Example

One thousand (1000) #s that are 64-bit each.

Radix sort time?

what do you mean by passes?

Every "digit" is 16-bits
Radix takes "(4) passes/number"
why 4?

Quicksort time?

$$O(n \log n) = \frac{O(1000 \cdot 10)}{1000} \approx 10 \text{ passes/number}$$

10 Selection Sort

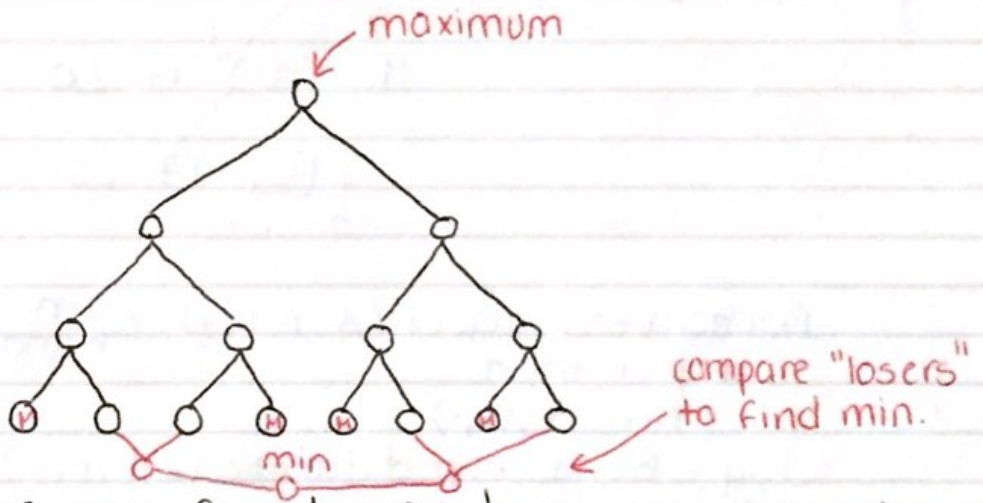
Selection

set unsorted #s → want to find 2nd smallest, 3rd biggest, etc?

can we do w/out sorting #s which would take O(nlogn)?

Select max? (how long?)
n-1 comparisons

Tournament Tree



Assuming you found max how many comparisons to find the min?

an additional $\frac{n}{2}-1$ comparisons

If you got the max how many additional comps. (comparisons) for the 2nd max?

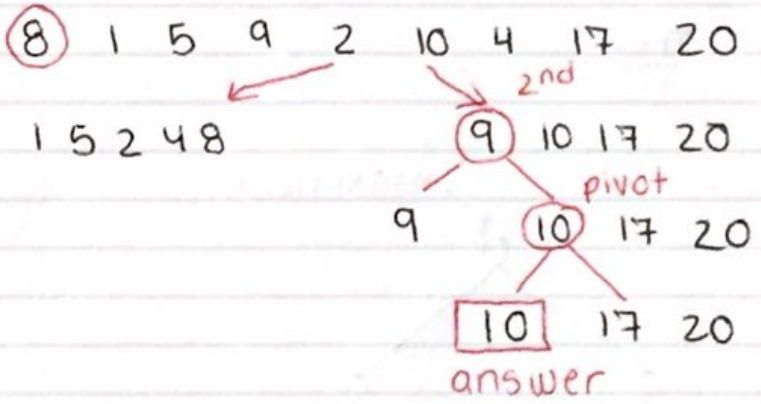
$O(\log n)$

Selecting the k^{th} max (time it will take is:)

$$n + k\log n$$

Example

Find the 7^{th} minimum element.



Randomized Select (A, p, r, i): → similar to Randomized quicksort

```
if p=r then
    return A(p)
q = Randomized Partition (A, p, r)
k = q-p+1
if i ≤ k then
    Randomized Select (A, p, q, i)
else
    Randomized Select (A, p, q, i-k)
```

$$T(n) \leq \frac{1}{n}\left(T(\max(1,n-1)) + \sum_{k=1}^{n-1} T(\max(k,n-k))\right) + \Theta(n)$$

$= O(n)$ run time (expected)

11-1 Binary Search Trees Intro

Binary Search Trees (BST)

Data structures for dynamic set operations ⟷ INSERT, DELETE, SEARCH

Time of these ops is proportional to height $\underline{h}$ of tree
$O(h)$

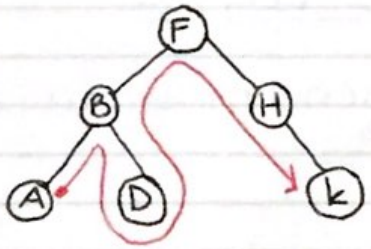
Each node contains these fields:
- key: value stored
- left: pointer to left child (if no left child, pointer is NULL)
- right: pointer to right child
- p: points to parent (p[root[T]] = NIL)

We also have root[T] pointing to root of BST

BST Property

① If y is in left subtree of x then key[y] ≤ key[x]
② If y is in right subtree of x then key[y] ≥ key[x]

Example:



Inorder: (inorder walk)
A B D F H K

Inorder tree walk (recursive)

L Root Right

ⓐ make sure x is not NIL
ⓑ recursively print keys in x's left subtree
ⓒ print x's key
ⓓ recursively print x's right subtree

Ways to Traverse Tree

a) In-order walk (left, root, right)
b) pre-order walk (root, left, right)
c) post-order (left, right, root)

Time $\Theta(n)$ ↓ visit/print every node once

Minimum $O(h)$ ← height of tree
go to leftmost node

Maximum $O(h)$
go to rightmost node

Successor of key x:

↳ next element after x in in-order walk

Predecessor of key x:

↳ previous element before x in in-order walk

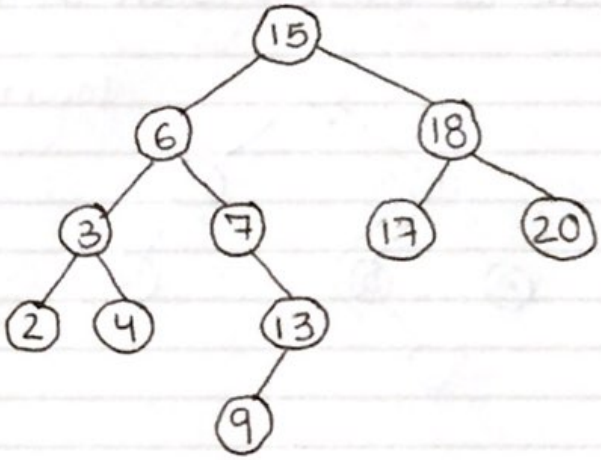
Successor Time O(h) → traverse down

① If node x has a non-empty right subtree
then return the minimum in x's right subtree
② If node x has an empty right subtree.
x's successor is y, where y is the predecessor of x (x is the maximum in y's left subtree]

Predecessor Time O(h) → traverse up

Symmetric to successor (check book)

Example:



a) successor of 15: (17) (leftmost node in 15's right subtree)
b) successor of 6: (7)
c) successor of 4: (6)
↳ " for which node in tree is 4 max in left subtree?"
d) predecessor of 6: (4)
↳ max in left subtree

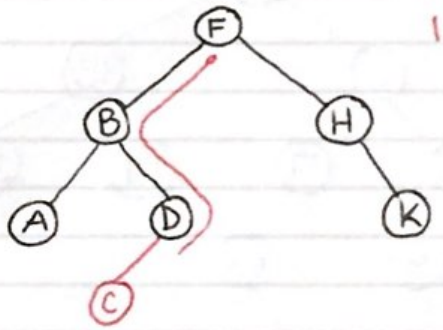
11-2 Binary Search Tree Operations

SEARCH/INSERT/DELETE

SEARCH key k — Time O(h) → worst case root to leaf of longest path
start at root
go left or right by comparing k w/ nodes
if find k then return node
else if reach leaf & not found
then return NIL

INSERT key k — Time O(h) → SEARCH + constant step
search for k (ignore duplicate)
add new node w/ value k where search is done

Ex:



Insert ©

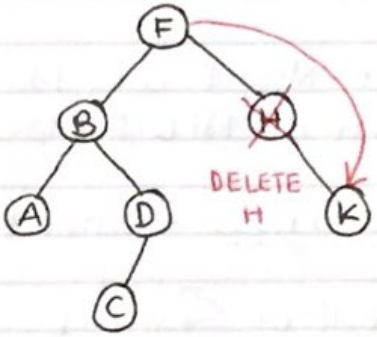
DELETE node z → called w/ node instead of key
Time O(h)
↳ worst case find successor that's far away

Case 1: z has no children
just delete z, parent of z points to NIL

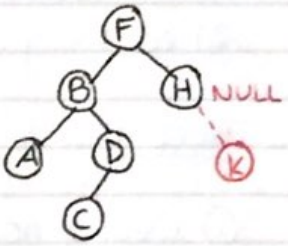
Case 2: z has one child
delete z, parent of z points to z's child

Case 3: z has two children
a) find successor of z (say y)
b) delete y from tree (case 1 or 2)
c) replace z's key w/ y's key

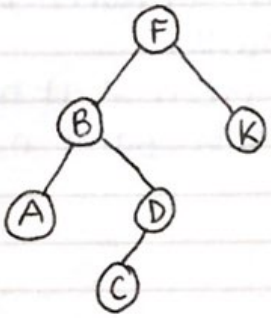
Ex:



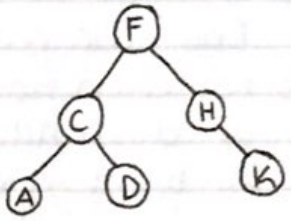
DELETE K → Case 1



DELETE H → Case 2



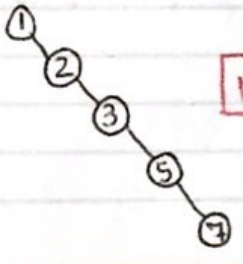
DELETE B → Case 3



12-1 Red Black Trees Properties

Red-Black Trees (also AVL trees, B-trees, ... that solve imbalance)
Solve the imbalance issue of BSTs.

Example: INSERT 1, 2, 3, 5, 7 ...



IMBALANCE !!

O(h) → O(n)
↳ want log n → standard BST do not guarantee this

Red Black Tree is a BST.

$ each node has a color ↔ black / red

Conventions

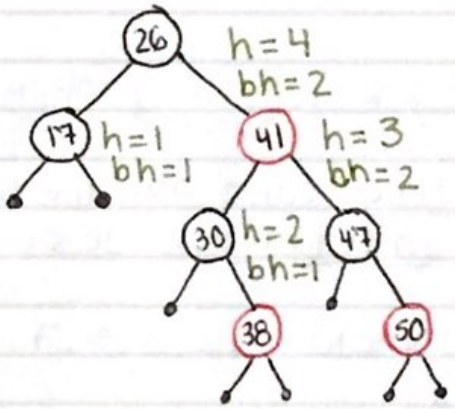
1) all leaves are NIL $ color black
2) root's parent is NIL $ black

Red-Black Properties → inherit BST properties as well

① Every node either ↔ red / black
② The root is black
③ Every leaf is black
④ * If node is red then both children black (no two red nodes in row!!)
⑤ * For each node, all paths from that node to descendant leaves have same number of black nodes.

give structure / balance to BST

Example:



Definitions

① Height of node is # edges in longest path to leaf.
② Black-height of node x: bh(x) is # black nodes on path from x to a leaf (not counting x)

12-2 Red Black Trees Balance Proof

Claim 1: Any node w/ height h has black-height $\geq h/2$

Proof: by property 4, $\leq h/2$ nodes on path from node to leaf are red.
$\Rightarrow \geq h/2$ are black □

Claim 2: the subtree rooted at any node x contains $\geq 2^{bh(x)} - 1$ internal nodes (not leaf)

Proof: (by induction) on height of x

I.B (Inductive Basis):

height of x = 0 → x is leaf → bh(x) = 0
The subtree at x has 0 internal nodes
$2^0 - 1 = 0$ ✓

hypothesis similar

I.S. let height of x be h & black-height bh(x) = b

Any child of x has height h-1 & black-height ↗ b (same as parent x) / ↘ b-1

black-height ↗ b (if child red) / ↘ b-1 (if child black)

... by I.H. (hypothesis) each child has

$\geq 2^{bh(x)} - 1$ internal nodes

Thus, subtree rooted at x has $\geq 2 \cdot (2^{bh(x)-1} - 1) + 1$

$= 2^{bh(x)} - 1$ internal nodes

□

Lemma: a red-black tree w/ n internal nodes has height $\leq 2\log(n+1)$

Proof: Let h & b be the height & black-height of root. By claims 1 & 2:

$$n \geq 2^b - 1 \geq 2^{h/2} - 1$$

add 1 (to both sides) & take logs:

$\Rightarrow \log(n+1) \geq h/2$

$\Rightarrow h \leq 2\log(n+1)$ □

$h = O(\log n)$ → tree is balanced if red-black properties satisfied

12-3 Red Black Trees Operations

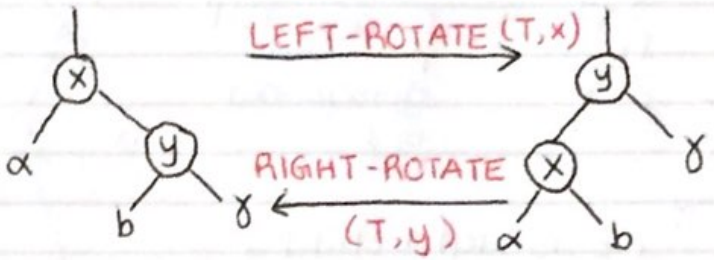
Operations

MIN, MAX, SUCC, PRED, SEARCH are same as in BST
Time: $O(h) \Rightarrow O(\log n)$

INSERT & DELETE?
May violate properties!

Rotation helps fix violations

- tree-restructuring
- in order walk & BST property are preserved



Time: O(1) → rearranging pointers (local change)

Insert (T, z)

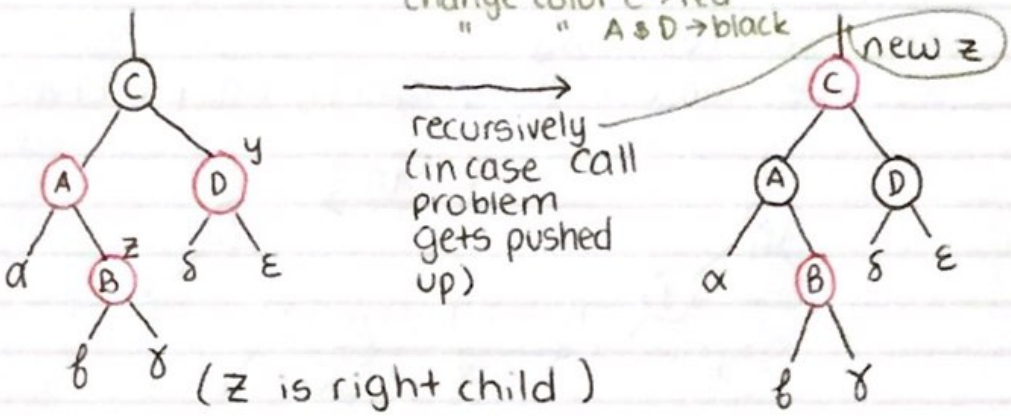
Search & insert & color node red
z z z

5 properties need to be satisfied:
① ✓
② violation possible
③ ✓
④ violation possible
⑤ ✓

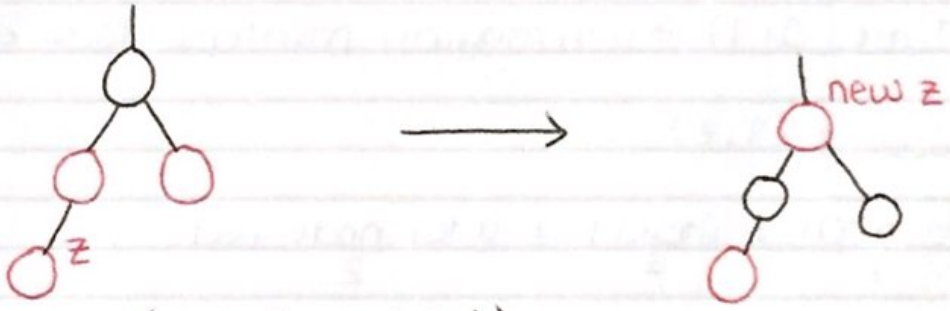
RB-INSERT-FIXUP (T, z)
↑ inserted node
→ pay attention to z's uncle (sibling of z's parent)

Note: let y be parent[z]'s sibling

Case 1: y is RED re-coloring:
change color C → red
" " A & D → black



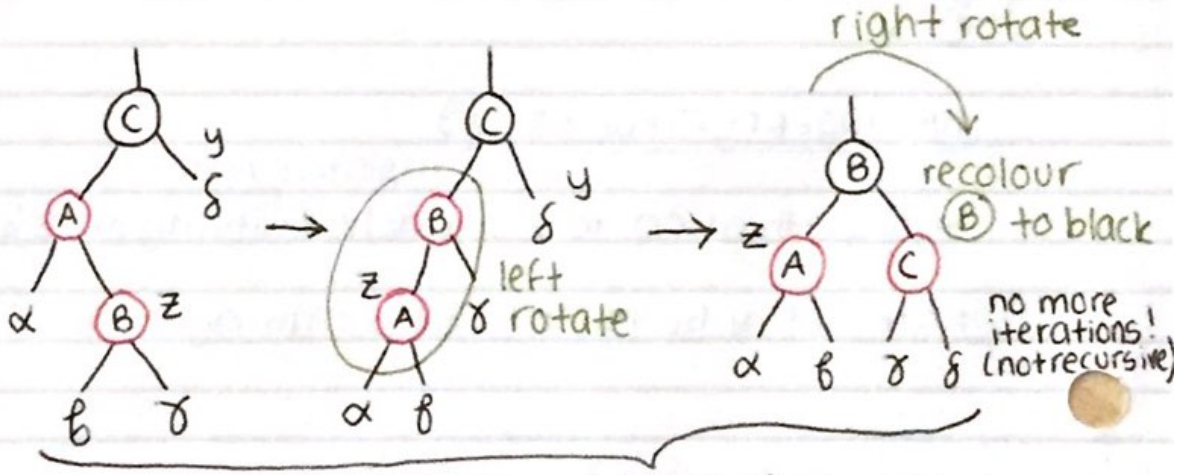
(z is right child)



(z is left child)

Case 2: y is black, z is right child

Case 3: y is black, z is left child



balance the trees
& re-color the new root

Really → there are 6 cases!!

Cases 1, 2, 3 (parent[z] is left child)
Cases 4, 5, 6 (symmetric)
↳ in 1, 2, 3 parent of z is a left child, 4, 5, 6 parent is right (parent[z] is right child)

Time: O(logn) → bottom to top in worst case!

Insert is also O(logn)

↳ search: O(logn)
↳ color node red: O(1)
↳ call this fcn: also O(logn)

what fcn?

Delete

8 cases!!! (look in textbook)

Time: O(logn)

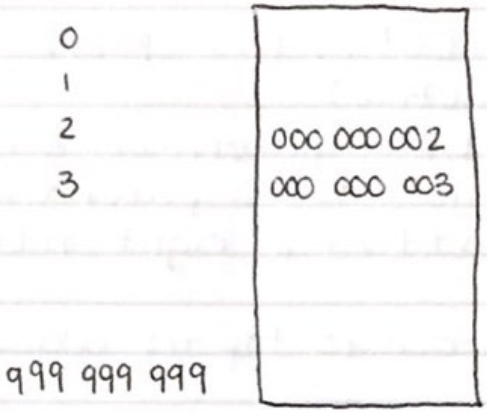
Week 6 Videos: 13-1 Motivation for Hash Tables

Hash Tables

Motivation:
XXX-XXX-XXX
want to store SIN's & do the following operations:
- insert
- delete
- search

Using: (1) an array
(2) a linked list

... using an array:

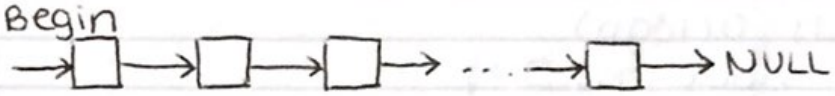


Insert, delete, search
O(1) time

Canada has ≈ 35,000,000

1 billion entries array!
More 97% array space goes wasted! space

... w/ a linked list:



Memory: O(n)
as many element we store

Time: insert/delete/search
O(n)

Hash tables provide "nearly" O(1) time for the operations w/ ≈ O(n) space

13-2 Intro to Hash Tables

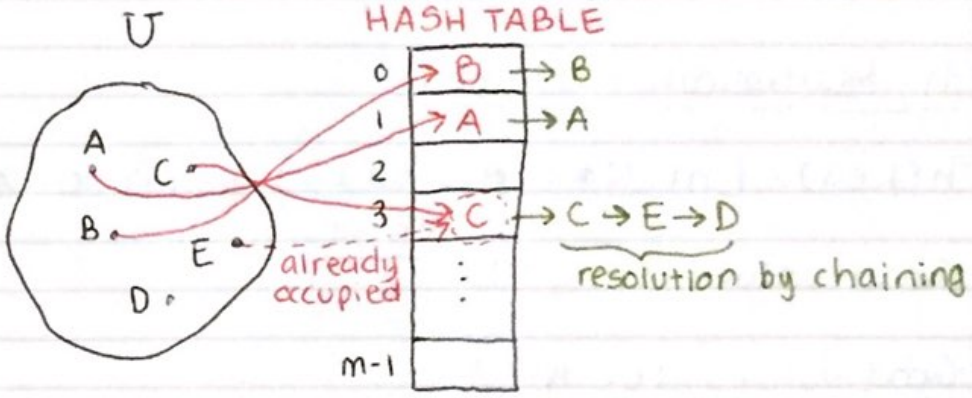
Hash Tables

U = universe of n-keys
m = size of hash table
we want $m \geq n$ (or they cannot fit)
$\alpha = \frac{n}{m} \leq 1$ is load factor

Hash Function h(key):

$h: U \to m$
↑ integer [0.. m-1]

How hash fcn functions:



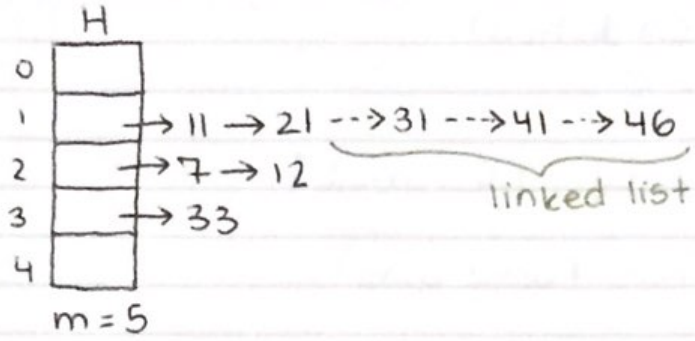
h(A) = 1
h(C) = 3
h(B) = 0
h(E) = 3 → COLLISION (hash table "bucket"/spot already taken)
h(D) = 3
↳ Resolution by chaining •
↳ " " open addressing

13-3 Resolution by Chaining

A good hash fcn → avoid as many collisions as possible.

h: Division Method

h(key) = k mod m
Bad values for $m = 2^p$ ← any power of 2
Good values for m = prime #s → experimentally



h: Multiplication Method

$h(key) = \lfloor m \cdot ((key\ A) \bmod 1) \rfloor$ where $0 < A < 1$

Good values for $m = \frac{\sqrt{5}-1}{2}$? so specific?

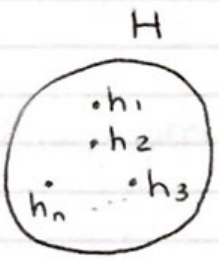
Good values for $m = 2^p$

Compute h(key) by using bitwise AND, OR, etc CPU operations (CLRS)

[grad. content]

h: Universal Hashing

Introduce randomization (like Quicksort) to perturb (introduce "noise") in the input



⇒ family of hash fcns

How to build H family of hash fcns?
Number theory!

13-4 Resolution by Open Addressing

Upon a collision on probe i_0 we examine other cells i_1, i_2, i_3 etc.

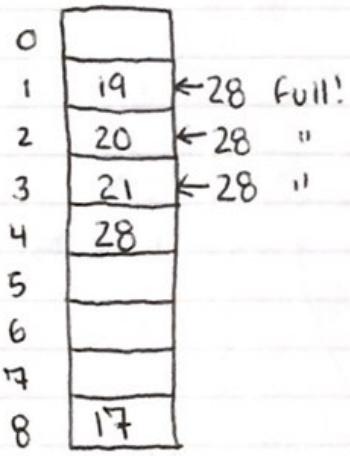
We do not build linked lists like in chaining but we only use the hash table entries.

Open Addressing:

Linear probing:

$i_0 = h(k)$

$i_{j+1} = (i_j + c) \bmod m$



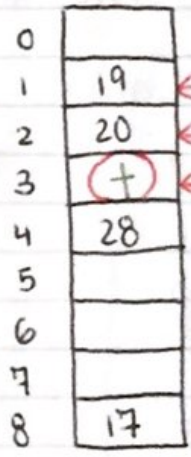
$m = 9$

$c = 1$

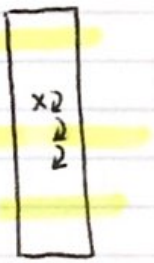
→

delete 21

looking for 28



Clustering



Double Hashing
Don't use a predetermined step "c" but a secondary hash fcn as a "step".

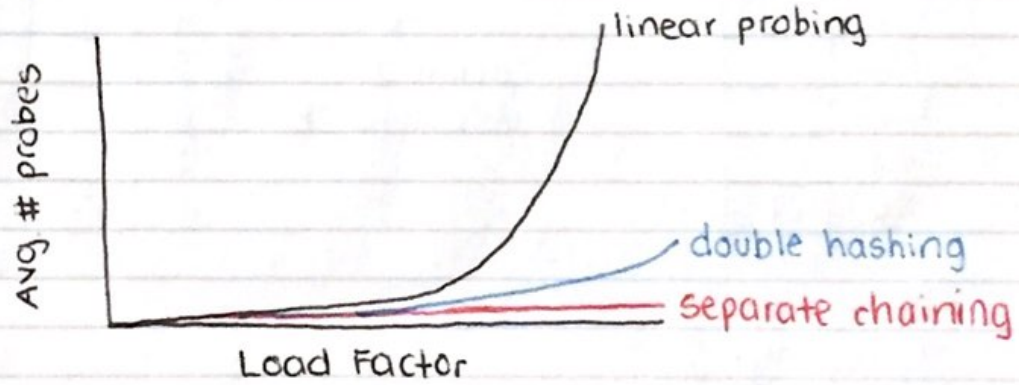
$$i_0 = h_1(k)$$
$$i_{j+1} = (i_j + h_2(k)) \bmod m$$

h_2 = secondary hash fcn
h_1 = primary ≠

Runtime

For successful search ($\alpha = \frac{n}{m} \leq 1$) ← load factor
Expected # collisions

Chaining	Linear Probe	Double Hashing
$1 + \frac{\alpha}{2}$	$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$	$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$



*guaranteed on exam (dynamic on midterm, greedy on final or vice versa)

14-1 Intro to Dynamic Programming

Dynamic Programming

"Divide & conquer"
→good for maximization/minimization problems

Two characteristics:

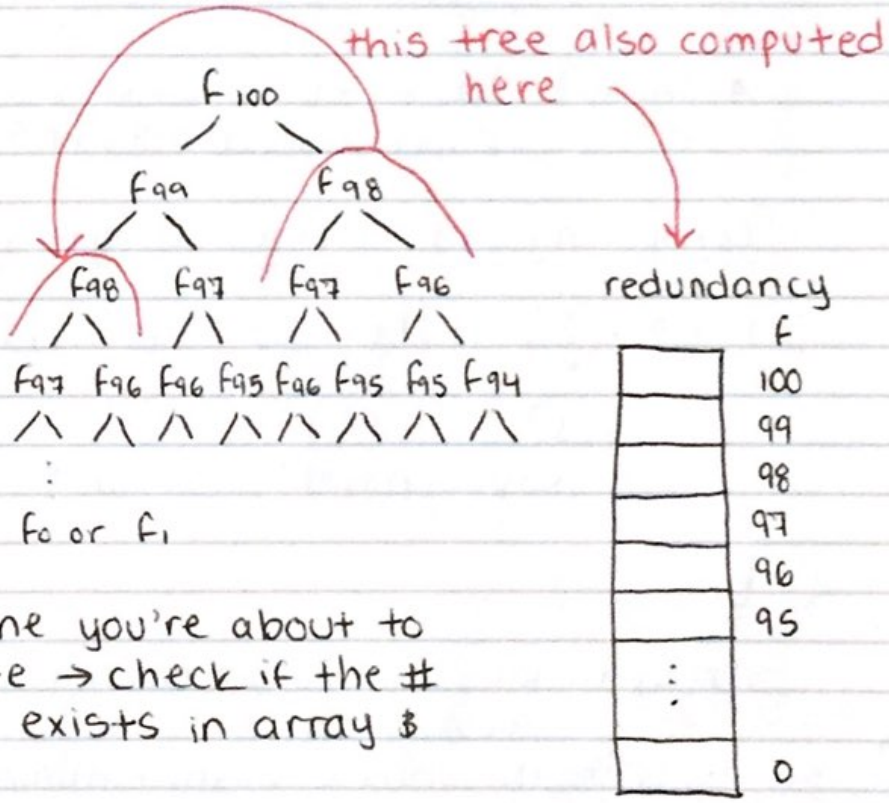
① **Optimal substructure:**
Optimal solution to the "big problem" entails optimal solutions to similar-type subproblems (like greedy)

② **Overlapping subproblems:**
As you break the large problem into smaller, often you need to re-calculate same thing. Store intermediate solutions & reuse them (memoization)

Ex: Fibonacci #s

$f_n = f_{n-1} + f_{n-2}$, where $f_0 = 0$ & $f_1 = 1$

$f_{100} = ?$



→ everytime you're about to calculate → check if the # already exists in array & use it

14-2 DP Matrix Multiplication

Need to multiply n-matrices

$A_1 A_2 A_3 \ldots A_n$

in this order.

How to parenthesize them to minimize the # of scalar multiplications?

Assume matrix A_i has dimension $P_{i-1} \times P_i$.

Motivational Example

A_1	A_2	A_3
10×100	100×5	5×50

$(A_1 A_2) A_3 = 10 \times 100 \times 5 + 10 \times 5 \times 50$

10×5

$= 5000 + 2500 = 7500$ multiplications

$A_1 (A_2 A_3) = 100 \times 5 \times 50 + 10 \times 100 \times 50$

100×50

$= 25000 + 50000 = 75000$ multiplications

order of magnitude difference

Computing all parenthesizations exhaustively:

$$P(n) = \sum_{k=1}^{n-1} P(k) P(n-k)$$ recurrence

$= \Omega(4^n / n^{3/2})$ exponential!

$n = 100$ ← will not provide soln in reasonable time

Key idea:

An optimal parenthesization for

$A_1 A_2 \ldots A_n$

involves the index k that minimizes # of multiplications.

$\underbrace{(A_1 A_2 \dots A_k)}_{\#\text{ mult}}\underbrace{(A_{k+1} \dots A_n)}_{\#\text{ mult}}$

$(A_1 A_2 \dots A_{k-1})(A_k \dots A_n)$

$A_1 A_2 A_3 (A_4((A_5 A_6)(A_7 A_8)))$

Let $m(i,j) = \{$ min # of scalar multiplications to calculate $A_i \dots A_j \}$

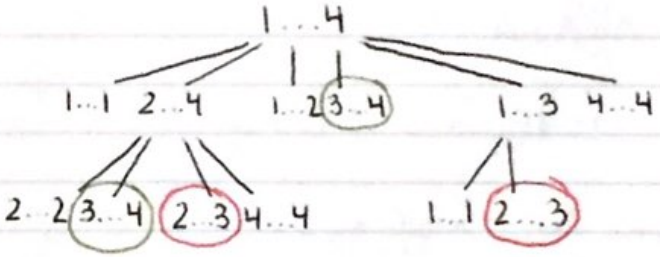
$$m(i,j) = \begin{cases} 0 & \text{if } i=j \\ \min\{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i<j \\ & i \le k < j \end{cases}$$

$m(i,j)$: 2 indices (unlike fibo #s w/ 1 index → array) → use table!

Optimal substructure principle: optimal solns to subproblems

$p_{i-1} p_k p_j$: # multiplication to find product $A_{i,k} \times A_{k+1,j}$

Naive recursive implementation of previous recurrence:



We recalculate results!
Solution?
Memoization (store intermediate results)

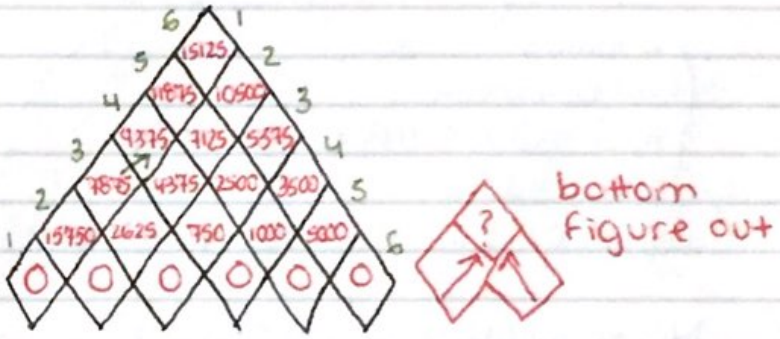
Use a 2-dimensional table to memorize intermediate results.

Ex: $A_1 : 30 \times 35$
$A_2 : 35 \times 15$
$A_3 : 15 \times 5$
$A_4 : 5 \times 10$
$A_5 : 10 \times 20$
$A_6 : 20 \times 25$

} dimensions of the table

$A_1 A_2 A_3 A_4 A_5 A_6$ →

cell A[i,j] contains min multiplication for $A_i \ldots A_j$
m(i,j) (product)



$m(1,3) = A_1 A_2 A_3$?

$(A_1 A_2) A_3 = 15750 + 30 \times 15 \times 5$

$A_1 (A_2 A_3) = 2625 + 30 \times 35 \times 5$

$A_1 | A_2 | A_3 | A_4$
→ → →
run index to calculate the following:
(# of multiplications req'd → want to minimize)

$A_1(A_2A_3A_4) = 4375 + 30 \times 35 \times 10$

$(A_1A_2)(A_3A_4) = 15750 + 750 + 30 \times 15 \times 10$

$(A_1A_2A_3)A_4 = 7875 + 30 \times 5 \times 10$

Run time?

$O(n^2)$ squares $\times$ $O(n)$ to run the index
$= O(n^3)$ time

Naive $\rightarrow \Omega(4^n/n^{3/2})$

14-3 DP Longest Common Subsequence

Longest Common Subsequence (LCS)

Problem Statement:
Given 2 strings
$X = x_1 x_2 \ldots x_m$ ← sequence of characters
$Y = y_1 y_2 \ldots y_n$

Find a subsequence (not necessarily consecutive but in the original order) which is common to both strings.

springtime
pioneer

hello
eloquent

Naive

$\Theta(n2^m)$ if $m \leq m$

Theorem:

If $Z_i = z_1 \ldots z_k$ is a LCS of X & Y

case 1: - if $x_m = y_n$ then $x_m = y_n = z_k$ & $Z_{i\,k-1}$ is a LCS of X_{m-1} & Y_{n-1}

case 2: - if $x_m \neq y_n$ then $z_k \neq x_m$ implies that Z_i is a LCS of X_{m-1} & Y_n

case 3: - if $x_m \neq y_n$ then $z_k \neq y_n$ implies Z_i is a LCS of X_m & Y_{n-1}
↳ $y_1 y_2 \ldots y_{n-1}$

Proof: (contradiction)

Assume there's a contradiction (ATaC) that $x_m = y_n$ but $x_m = y_n \neq z_k$ or $Z_{i\,k-1}$ is not a LCS of X_{m-1} & Y_{n-1}

if $x_m = y_n \neq z_k$ then add $x_m (= y_n)$ to Z_i & create a "longer" LCS, a contradiction as Z_i is a LCS.

If $Z_{i\,k-1}$ is not a LCS of X_{m-1} & Y_{n-1}, let W be one

$$|W| \overset{+1}{\cup} x_m > |Z_{i\,k-1}| \overset{+1}{\cup} x_m$$

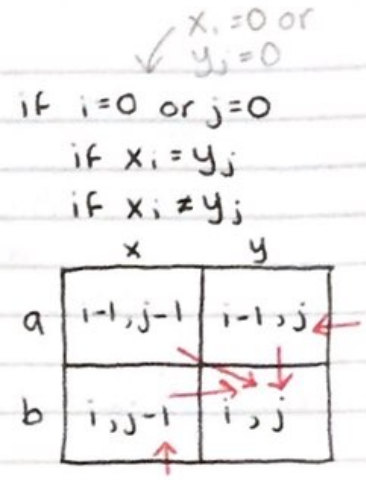
a "longer" LCS > Z_i

a contradiction

Theorem builds the recursive soln:

$c[i,j]$ = length of LCS of X_i & Y_j

$$c[i,j] = \begin{cases} 0 & \text{if } x_i = 0 \text{ or } y_j = 0 \\ c[i-1,j-1]+1 & \text{if } x_i = y_j \rightarrow \text{case 1} \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } x_i \neq y_j \rightarrow \text{case 2 \& 3} \end{cases}$$

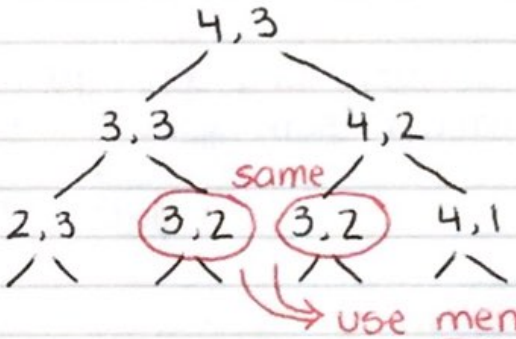


(to explain run time)

Naive implementation

Consider strings "bozo" & "bat"

c[4,3]



∴ can use 2D array!:

		a	m	p	u	t	a	t	i	o	n
	0	0	0	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	1	2	2	2	2	2
n	0	1	1	1	1	1	2	2	2	2	3
k	0	1	1	1	1	1	2	2	2	2	3
i	0	1	1	1	1	1	2	2	3	3	3
n	0	1	1	1	1	1	2	2	3	3	4
g	0	1	1	1	1	1	2	2	3	3	4

What is the actual common subsequence?
↳ use array to indicate where values comes from
↳ everytime you have diagonal arrow ⇒ "pain"

Runtime

Naive:
$\Theta(nZ^m)$

$O(n \cdot m)$

↳ length of each string

(diagram on other side of page)

Week 7 videos: 15-1 Greedy Algorithms Intro & Class Scheduling

Live Q+A (10/19/2020):

Greedy Principle Optimal Substructure:

ATaC that greed soln G is not optimal but
∃ "some" optimal O soln where n > m

n classes m greedy

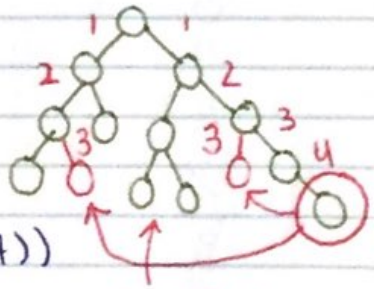
$g_1\ g_2\ g_3 \cdots g_m$

~~O_1~~ $O_2\ O_3 \ldots O_m \ldots O_n$

g_1 finishes before or at same time as O_1

Greedy Algorithms

" Greedy is good,
Greedy is right
Greedy works
- Oliver Stone ("Wall Street" (1987))



cannot be optimal b/c height is larger than it needs to be

Principles / Characteristics:

*expect to see greedy algo problem on midterm/ final

Greedy Principle: A global optimal is reached by doing local greedy choices.

Optimal Sub-structure (like dynamic)

Notes:
- Theory of Matroids
- They usually provide good approx. solns to NP-Complete problems
- good to max/min solns

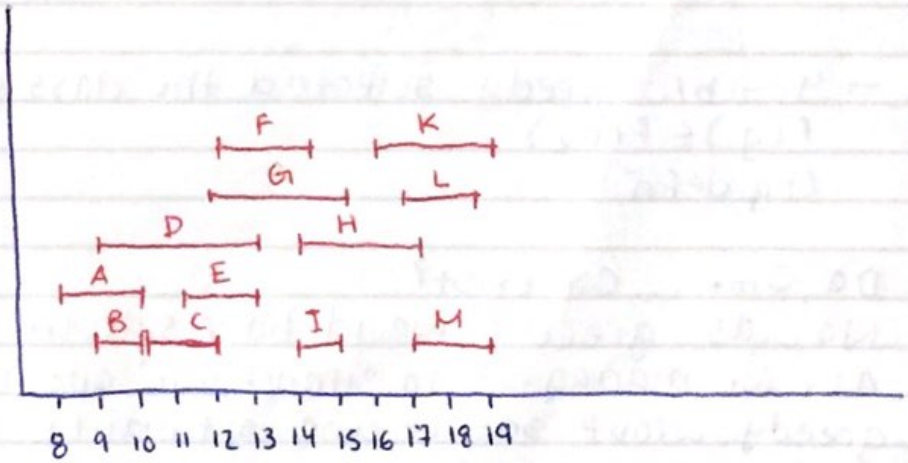
Activity Selection Problem

There's a set of (possibly) overlapping classes but only 1 classroom.

How to maximize # of classes to fit in classroom?

Algo:
- sort by finish time
- schedule w/ earliest finish time possible

O(nlogn) - time
↳ just need to sort



Sort by finish time:

(B) → disqualifies A & D b/c overlap

~~A~~

(C) ⇒ I can fix a max of 5 classes B C F I L

~~E~~ ~~D~~

(F) ⇒ but I could A C F I L (soln may not be unique)

(I)

~~G~~

~~H~~

(L)

~~M~~

~~K~~

Proof of correctness:

Why greedy G is optimal?

"assume towards contradiction"

ATaC that greedy soln G is not optimal but some other soln O is.

Do those classes exist?

$g_1\ g_2\ g_3 \ldots g_m$

$\cancel{O_1}\ \cancel{O_2}\ \cancel{O_3} \ldots \cancel{O_m} \ldots (O_n)$ ← more cases b/c optimal (more # cases fit)

$g_1\ g_2\ g_3 \quad g_m$

Can I replace O_1 w/ g_1?

→ yes b/c greedy selected the class g_1

$f(g_1) \leq f(O_2)$

(by defn)

Do $O_{m+1} \ldots O_n$ exist?

No, as greedy would have selected them.

Abv. we managed to "transform" optimal to greedy w/out sacrificing optimality. Hence " is optimal.

15-2 Knapsack Problem (Dynamic vs. Greedy)

A thief enters a store that has n items. Item-i values v_i $ weighs w_i. Thief can only carry W weight.

Fractional knapsack (greedy)
can take a portion of an item (like rice, beans, etc...)

0-1 knapsack (dynamic)
Thief can only take/leave the whole item (jewelry store).

Frame the solns:

Fractional version - Greedy

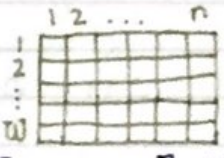
Sort items per $\frac{v_i}{w_i}$ and will keep on taking in descending order maximizing the value for the W he can deliver.

$O(n \log n)$
↳ sort!

0-1 version - Dynamic

Sort the items in some (any) order to consider:

c[i, w] = the max value taking frm items 1...i w/ "leftover" weight w



$$c[i,w] = \begin{cases} 0 & \text{if } w = 0 \\ c[i-1, w] & \text{if } w_i > w \\ \max(c[i-1, w], c[i, w-w_i] + v_i) & \text{if } w_i \le w \end{cases}$$

Runtime $O(nW)$ ← b/c n rows $ W columns

15-3 Intro to Huffman Encoding

Huffman Codes for Data Compression

You are given a file w/ text characters appearing in some frequency:

chars	type	a	b	c	d	e	f
f(c)	frequency	45	13	12	16	9	5
d(c1)	fixed size	000	001	010	011	100	101
d(c2)	var size	0	101	100	111	1101	1100

Let B(T) = size of encoded file

$$= \sum_{char} f(c)d(c)$$

= # bits needed

Problem:
You want to minimize B(T) ie., # bits to encode file

Is there fixed size reasonable/best way to do encoding? No.

Is there a better encoding that will minimize # bits needed?

Example:
Suppose we want to write:

d a d
011 000 011 → 9 bits

→ Not best encoding b/c "a" appears frequently in file, allocated "a" 3 bits which is same for "f" which only appears 5% of time
→ want to have variable encoding
↳ allocate less bits that have higher frequency
↳ save more bits

* how does it know 111 is "d" if there is "1110" for "g" for example? → are encodings all unique?
↳ ie., size of each encoding? how does it determine this to determine letter?

Using var size:

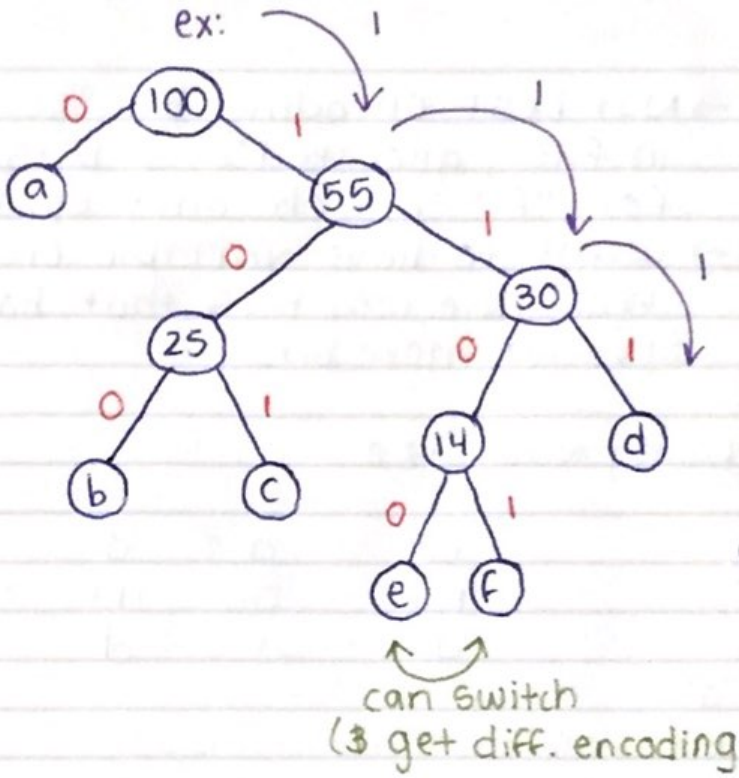
d	a	d	
111	0	111	→ 7 bits
d	a	d	

↳ by construction using tree (check next page)

chars	type	a	b	c	d	e	f
f(c)	frequency	45	1	12	16	9	5
d(c)	variable size	0	100	101	111	1100	1101

d	a	d
111	0	111

Greedy Algorithm:
Repeat:
Unite 2 smallest frequency keys building a binary tree by summing their frequencies.



what data structure to implement this algo?

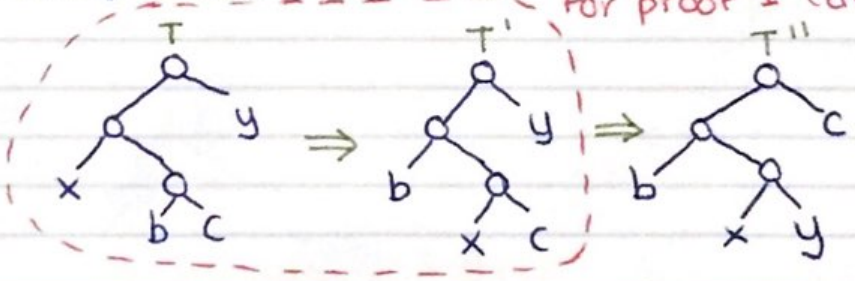
Use a min-heap to extract the min!

$O(n \log n)$ time

15-4 Huffman Proofs

Prove GREEDY PRINCIPLE

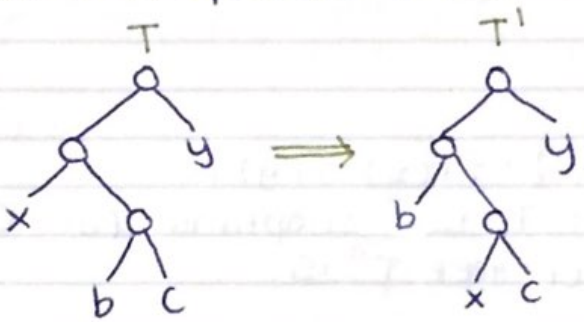
Every optimal tree can have the two lowest frequency keys being siblings that differ in only 1 bit



ATaC that keys x & y are the lowest frequency but they don't differ by a bit. We will replace x ↔ b & show that

$$B(T) \geq B(T')$$

(Assume T is optimal tree)



After we swap x ↔ b we have the following tree weights

$$B(T) - B(T') = [f(b) - f(x)][d_T(b) - d_T(x)]$$

(f(b) − f(x): frequency; $d_T(b) - d_T(x)$: # bits)

$$\geq 0 \iff B(T) \geq B(T') \rightarrow T' \text{ remains optimal}$$

review this proof (a little confusing)

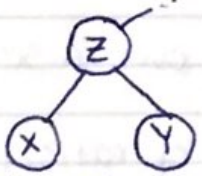
why?

$f(x) \leq f(b)$

$d_T(x) \leq d_T(b)$

↑ b/c in original tree, b was "deeper" down than x

Prove OPTIMAL SUBSTRUCTURE

Let 

be in the optimal T for character set C. Then $T' = T - \{x, y\}$ is an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$ where $f(z) = f(x) + f(y)$

We replace in the tree x & y w/ z where
$f(x) + f(y) = f(z)$

First we got the following:

$f(x)d_T(x) + f(y)d_T(y) = f(z)d_{T'}(z) + \{f(x)+f(y)\}$

weight bit difference b/w T & T'

$B(T) = B(T') + \{f(x)+f(y)\}$

ATaC that T' is not optimal for G' but some other tree T'' is.

$$B(T'') < B(T')$$
$$B(T'') + \{f(x)+f(y)\} < B(T') + \{f(x)+f(y)\}$$

Lecture says B(T'') → typo? Should be B(T') right?

$$B(T'') < B(T)$$

contradicting B(T) was optimal (which was our original assumption)

Week 8 Videos: 16-1 Amortized Analysis Introduction

Amortized Analysis

We saw best/worst/expected/average case analysis.

Amortized analysis:
- we analyze a data structure over a sequence of operations
- we do not analyze a single operation
- it provides a guarantee of the average performance on worst case
- deterministic (no probability involved)

Ex: $220/7 days → cost/day for food
(not necessarily how much you pay every day, sometimes could be more/less, but it is how much expected over a period of time, kind of similar to average)

2 Methods to analyze:
- Aggregate → brute force
- Accounting → more structure/intuitive
(- Potential → not covered in class)

Examples:

STACK

We have a stack w/ PUSH, POP, MULTIPOP ↱ pop out everyone frm stack

BINARY COUNTER

```
Increment(CTR)
    i = 0
    while i<length(CTR)
          and A[i] = 1  do
          CTR[i] = 0  /* turn off 1-bits */
          i = i+1
    if i<length(CTR) then
          CTR[i] = 1; /* turn on highest bit */
```

0001 → 0010 → 0011

0011 → 0100 → etc

16-2 Aggregate Method

AGGREGATE: Stack

		cost	
Naive:	POP	O(1)	
	PUSH	O(1)	
	MULTIPOP	O(n)	← full list

Therefore, n-ops can cost up to $n \cdot O(n) = O(n^2)$

Aggregate:

You cannot pop more than what you push...
∴ you can push at most n-items.

Total am't of cost:
is $O(n)$ or $\frac{O(n)}{n} = O(1)$ / operation

AGGREGATE: Binary Counter

Naive: cost comes from flipping a bit. If we call INCREMENT $O(n)$ times on a counter that has n-bits this may cost

$$O(n) \cdot n = O(n^2) \text{ total}$$

or

$$\frac{O(n^2)}{n} = O(n) \text{ / increment}$$

Aggregate:

→ Flips every other time or $\frac{n}{2}$ times → $\frac{n}{2^1}$

→ Flips all the time n times → $n/2^0$

0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
⋮ → flips every $\frac{n}{4}$ times → $\frac{n}{2^2}$

→ flips every $\frac{n}{8}$ times → $\frac{n}{2^3}$

For n-increments, the total # flips:

$$\sum_{i=0}^{n} \frac{n}{2^i} \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n = O(n)$$

or $\frac{O(n)}{n} = O(1)$ amortized time

16-3 Accounting Method

Accounting Method

Introduction:

- In accounting method the data structure behaves like a bank
- we charge $ for each op. This is the amortized cost of the op

- when
  (1) amortized cost < actual cost
      ⇒ deposit left over $s as credit on data structure
  (2) amortized cost > actual cost
      ⇒ use the credit already saved (deposited) on the data structure

what if no credit previously saved?

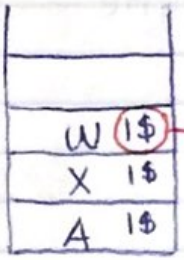
- your data structure cannot run on -ve credit! If you end up w/ -ve balance then your analysis is wrong

ACCOUNTING : STACK

	Actual	Amortized
PUSH	O(1)	2 $
POP	O(1)	0
MULTIPOP	O(n)	0

→ can be $4, etc (2$ for push/pop) need to maintain

EX:



use money that is deposited by PUSH

1 $ pays for the push
1 $ gets deposited to pay the pop

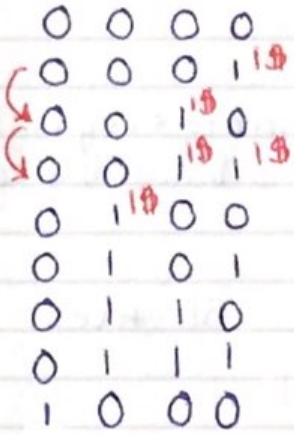
1$ to push, other 1$ goes on data structure as credit

For n operations, I use
$2n \$ = O(n)$ in total or,
$\frac{O(n)}{n} = O(1)$ amortized time

ACCOUNTING: Binary Counter

Charge 2 $ for every 0→1 flip:

1 $ pays flip, and
1 $ stays as credit to pay for the subsequent 1→0 flip



for n-increments, we need 2n $ or
$O(n)$ $ or $\frac{O(n)}{n} = O(1)$/increment in an amortized sense

17-1 Splay Trees Defns: * Sleator & Tarjan (1984)

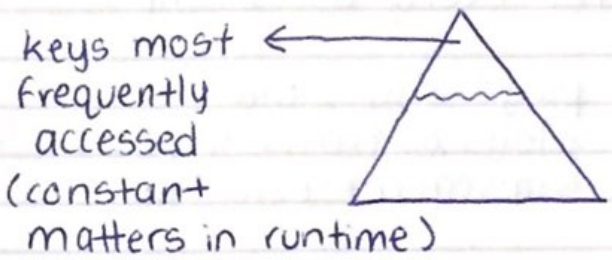
Splay Trees

Weighted Dictionary Problem: ← splay trees provide optimal soln to this

We want a data structure to insert, delete, search, sort.

If keys & the frequency they appear are known in advance, then we can build an optimal BST using dynamic programming.

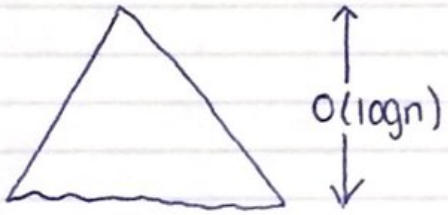
If not, Splay Trees achieve the theoretical optimal bound (in an amortized sense)

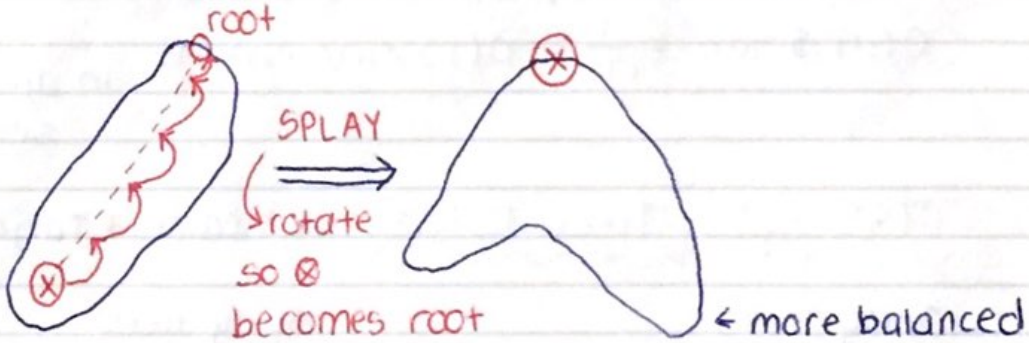


Splay Trees: are BSTs

key(left) ≤ key(parent) < key right

but they have no balancing condition like R-B trees



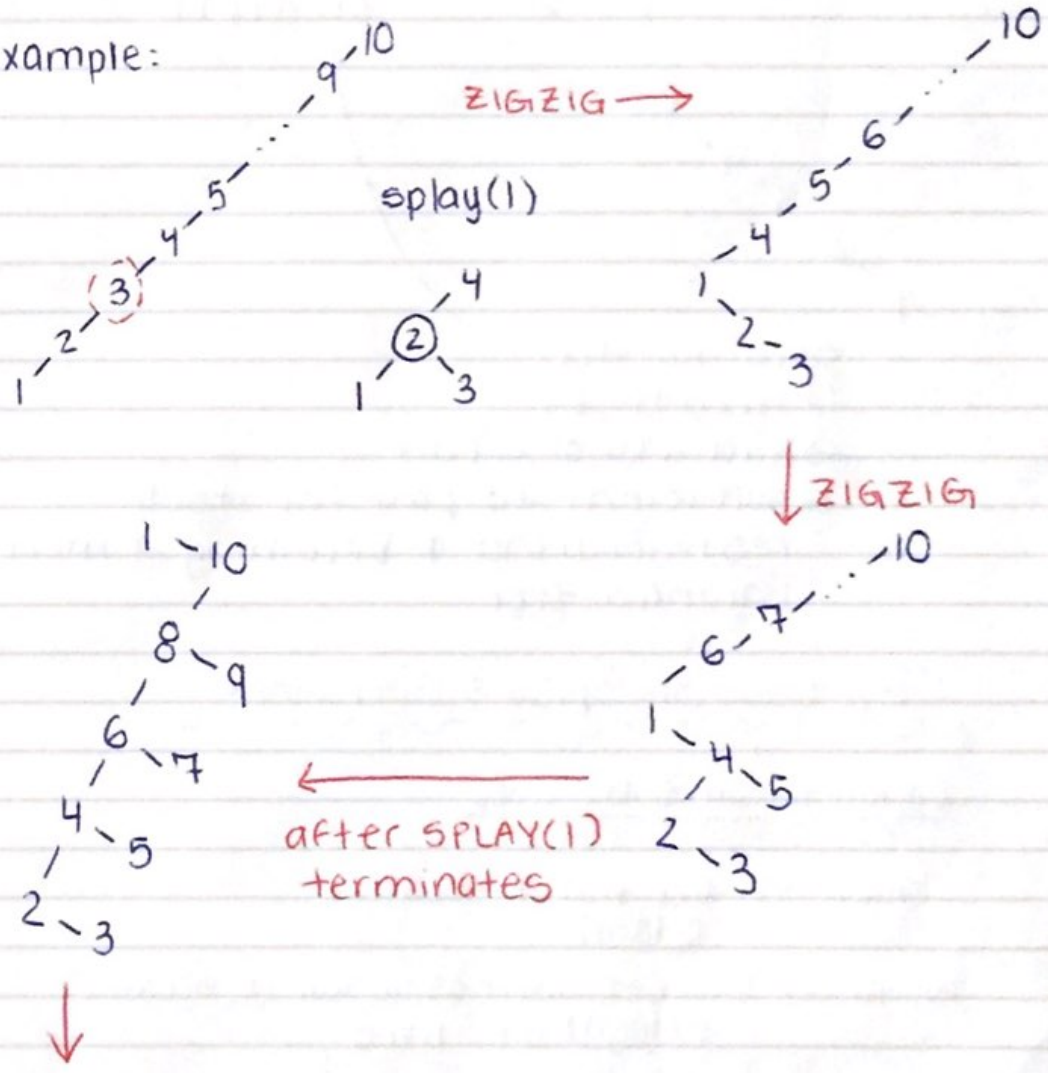


unbalanced tree is "rich" on credit

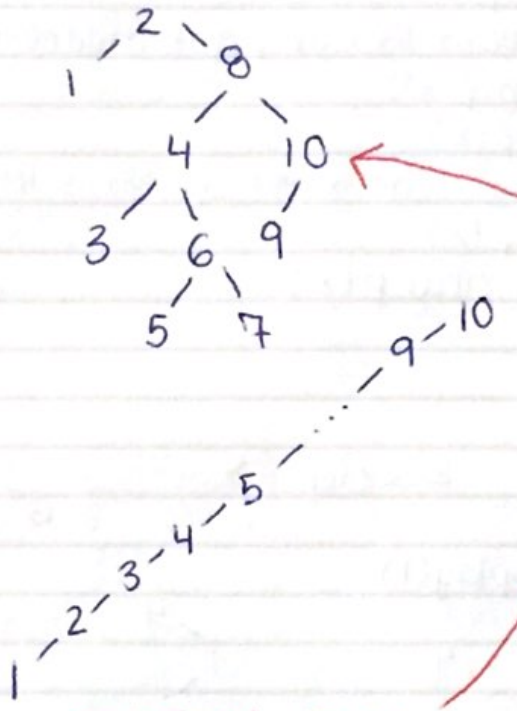
a more "balanced" has less credit.

```
SPLAY(x)
    while x ≠ root
zig [   if p(x) = root
            rotate p(x)
    if p(x) ≠ x both left or right children
zig [       rotate p(p(x))
zig [       rotate p(x)
    else (parent is right, x left; parent is left, x right)
zig [       rotate p(x)
zag [       rotate new p(x)
```

Example:



↓ SPLAY(2)



SPLAY(2) → pays logn $
SPLAY(1)

original tree
(linked list)
↳ had a lot of credit
↳ sufficient to pay for itself restructuring & becoming a more balanced tree

17-2 Cost of Splay & Operations

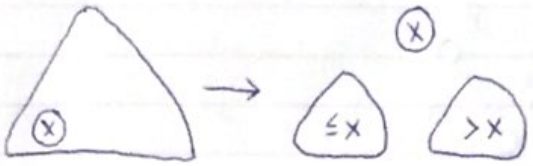
Operations on Splay Trees

SEARCH(x): like BST & SPLAY(x)
O(logn)

INSERT(x): like BST insert as a leaf & SPLAY(x)
O(logn) amortized

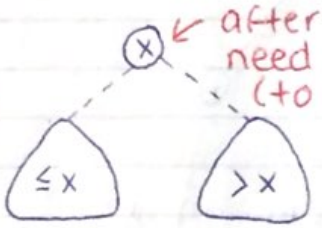
DELETE(x): like a BST & SPLAY(P(x))
O(logn)

SPLIT(x):
O(logn)



SEARCH(x) & then split into 3 components

JOIN
O(log(n))



after create new tree, need to deposit $ over here (to maintain credit invariant)

x takes logn $s

O(log(n))

COST OF SPLAY

Let weight $w_T(x)$ be the # of nodes in subtree of x including x.

Define $rank(x) = \lceil \log w_T(x) \rceil$

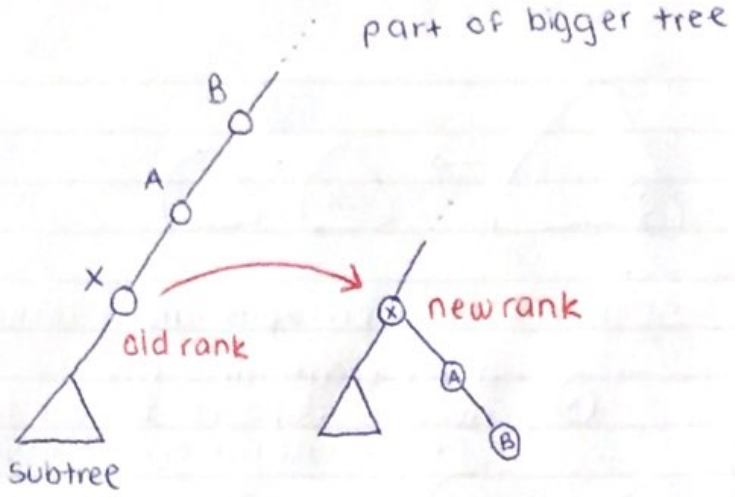
CREDIT INVARIANT
Every node x has rank(x) $s on it

Credit Invariant
Stack: ∀ element has 1 $
Counter: ∀ 1-bit has 1 $

CLAIM (to be proved later)
Every zig, zig-zig, or zig-zag costs:

$$3[\text{newrank}(x) - \text{oldrank}(x)]$$

except zig that may require sometimes an extra $1

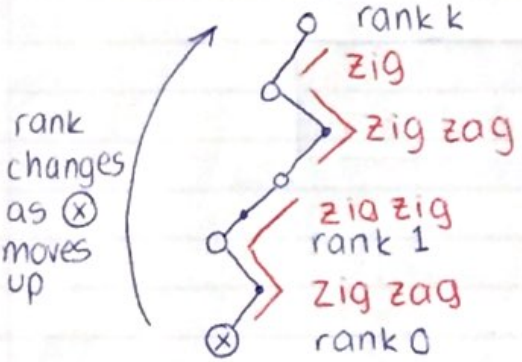


How much does this zig-zig cost?

$3(nr(x) - or(x))$

↓ newrank ↓ oldrank

Assuming this is true... How much SPLAY costs? ⇒ Amortized cost is: It costs $O(\log n)$



Total am't of dollars:

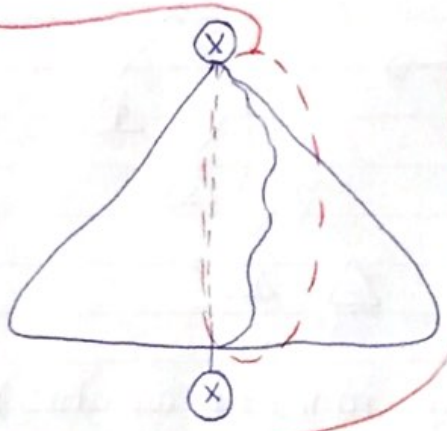
$3(\text{rank}_1 - \text{rank}_0) +$
$3(\text{rank}_2 - \text{rank}_1) +$
$3(\text{rank}_3 - \text{rank}_2) +$
$\vdots$
$3(\text{rank}_k - \text{rank}_{k-1}) + 1$ ← zig

$= 3(\text{rank}_k - \text{rank}_0) + 1$
$\leq 3(\log n - 0) + 1 = O(\log n)$

Cost of Insert:

INSERT(x): like BST insert as a leaf & SPLAY(x)

- search for right position of the node to be inserted
- insert it as a leaf
- inserted node, after a sequence of splays, becomes new root of the tree
- all nodes in path where inserted node was added change rank
  ↳ need $ to pay for change in rank (credit invariant)



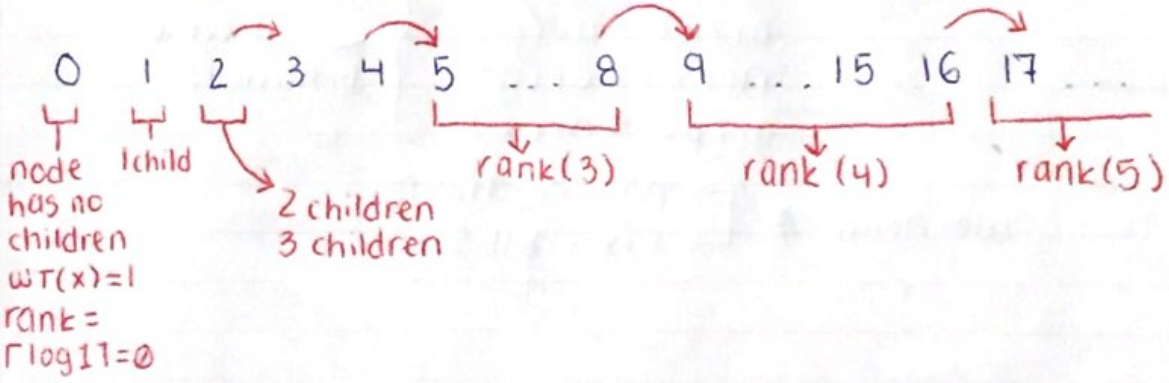
At most O(log n) nodes may change in rank. By how much? By 1 unit.

So we need at most O(log n) $s to fix those ranks.

recall: $rank(x) = \lceil \log wT(x) \rceil$

A node changes in rank iff it had weight $2^k \to 2^{k+1}$

why? (explained below)



17-3 Proof of Splay Claim

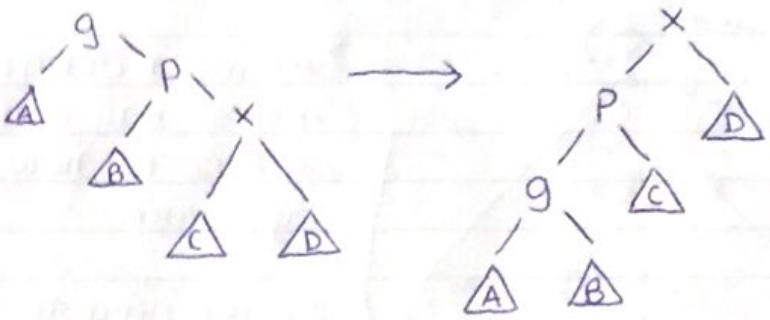
Recall Claim

Every zig, zig-zig, or zig-zag costs:

$$3[\underset{nr}{newrank(x)} - \underset{or}{oldrank(x)}]$$

except zig that may require sometimes an extra $1.

Zig-zig



only g, p, & x can change ranks (& weights)

We have $or(x) + or(p) + or(g)$

vs.

$nr(x) + nr(p) + nr(g)$

But:
- $nr(p) \leq nr(x)$
- $nr(g) \leq nr(p)$
- $nr(x) = or(g)$
- $or(p) \geq or(x)$

} worst case behaviour

We need $ → pay rotations
→ fix ranks

How much $ does new tree require?

Tree requires:

maximize $ consider worst case (play safe)

$nr(x) + nr(p) + nr(g) - or(x) - or(g) - or(p)$
$\leq 2nr(x) - or(x)$ fix ranks

↳ (solved using inequalities on previous page)

which is fine as claim gives x3 this am't. Allocate x2 times to fix ranks & leave the x1 (third one) to pay rotations (O(1) costs)

Problem

what happens if $nr(x) = or(x)$?

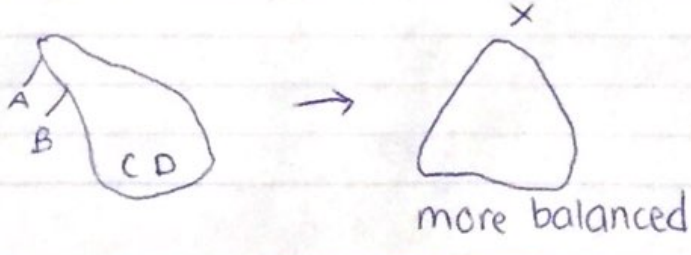
In this case, there's no need to fix ranks, but how do we pay for rotations? (if $nr(x) = or(x)$, claim gives 0 costs)

How do we pay for the rotations?

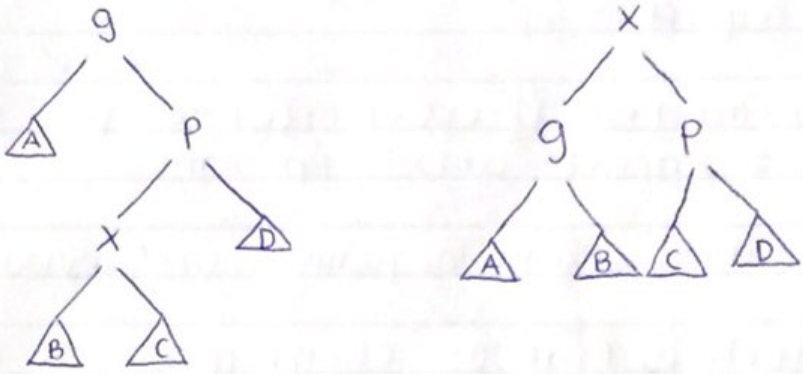
b/c log

we deduce that more than half of the tree nodes were under x. But now subtrees C & D split. And further that less than half of nodes were under A & B.

So g in new tree drops in rank to pay for the rotations.



Zig-zag



Analysis is identical (to zig-zig) unless $nr(x)=or(x)$

In that case we don't need to fix ranks but we need to pay for rotations:

In both cases...
$B+C > 1/2$

2 cases:

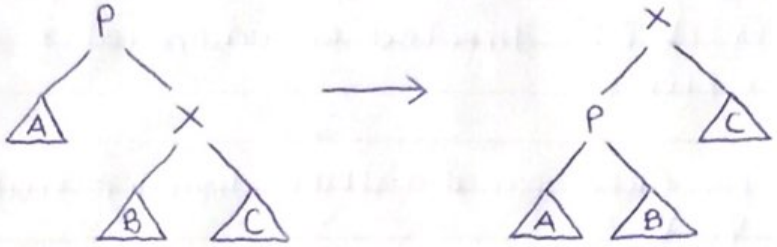
Case 1 — Case 1 symm $C \gg B$

$B \gg C$ (B had way more nodes than C)
$ in the new tree they split p drops in rank to pay for the rotation

Case 2

$B \approx C$ then both g $ d drop in rank to pay for the rotations.

Zig



we got that $or(x) \leq nr(x)$
$or(p) \geq nr(p)$

$nr(x) + nr(p) - or(x) - or(p)$
$\leq nr(x) - or(x)$

We got x3 this am't from CLAIM but again the problem is how we pay for rotations if $nr(x) = or(x)$

We don't know if p drops in rank, so we use the extra 1 $ the CLAIM allocates to pay for the zig.

Week 9 Videos: 22-1 Graph Algorithms Intro & Breadth-First Search

Elementary Graph Algorithms

Recap on Graphs

- $G = (V, E)$ vertices/edges
- directed, undirected, weighted, unweighted, path (simple), cycle
- adjacency list/matrix

ex: adj[u] nodes adjacent to u

Breadth-First Search (BFS)

Input: $G = (V, E)$, directed or undirected & source vertex s

Output: For each node $u \rightarrow d[u] =$ distance from s to u
$\forall u \in V$
$\pi[u]$: u's predecessor on path $s \rightarrow u$

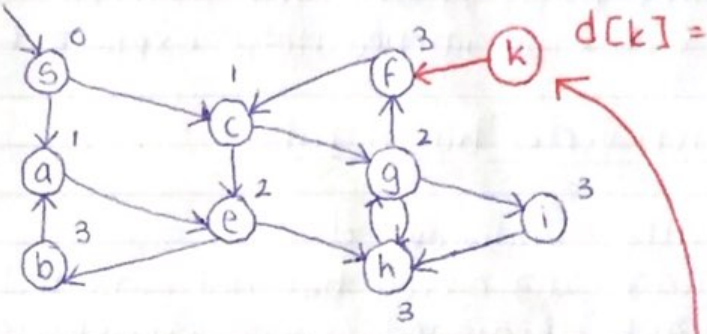
Idea: Sending a "wave" out from s

- 1st (iteration) hit all vertices 1 edge away
- 2nd hit ⟶ 2 edges away
- etc

Note: Use FIFO Q to maintain "wavefront"
- $v \in Q$ iff wave has hit v but has not come out of v yet

```
BFS (V, E, s)        ← where wavefront is
   for each u∈V - {s}
        do d[u] ← ∞
   d[s] ← 0
   Q ← 0
   ENQUEUE(Q, s)
   while Q ≠ 0
        do u ← DEQUEUE(Q)
           for each v∈Adj[u]
                do if d[v] = ∞
                   then d[v] ← d[u] + 1
                        ENQUEUE(Q, v)
```

Ex:



iter 0: s
iter 1: a, c
iter 2: e, g
iter 3: f, i, h, b
iter 4: done (Q will be empty)

Note: BFS may not reach all vertices

Time: $O(V+E)$
- every vertex is enqueued at most once
- every edge is examined at most once (if you have directed graph) or twice (undirected graph)

22-2 Depth First Search

Depth-First Search DFS

Input: $G = (V, E)$, directed or undirected
NO source vertex !! ⚠

Output: $d[v]$ = discovery time
$f[v]$ = finishing time
$\pi[v]$ as well

Idea: unlike BFS, in DFS the moment we discover a vertex we immediately explore from it.

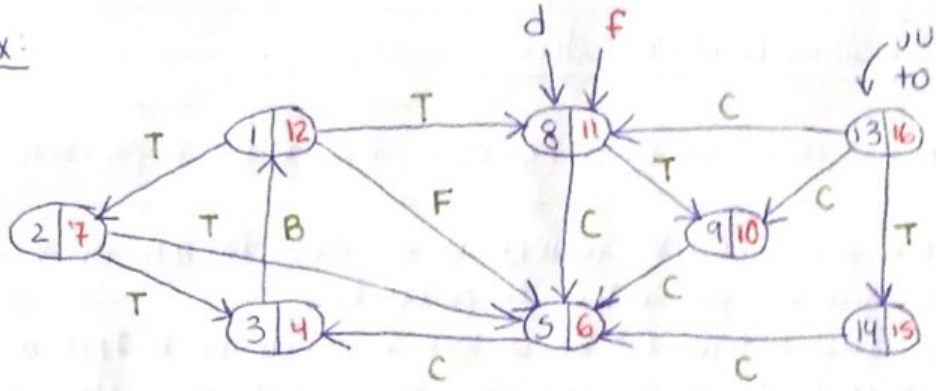
Each vertex has a color:

WHITE = undiscovered
GRAY = discovered but not finished
BLACK = finished (found everything reachable from it)

```
DFS (V, E)
    for each u∈V
        do color[u] ← WHITE
    time ← 0
    for each u∈V
        do if color[u] == WHITE
            then DFS-VISIT(u)
```

```
DFS-VISIT(u)
    color[u] ← GRAY
    time = time + 1
    d[u] ← time
    for each v∈Adj[u]
        do if color[v] == WHITE
            then DFS-VISIT(v)
    color[u] ← BLACK
    time ← time + 1
    f[u] ← time
```

Ex:



Time: $\Theta(V+E)$

↳ b/c you will see every node/edge in DFS

Definitions & Theorems

Thm (White-Path Theorem)

v is a descendant of u iff at time d[u], ∃ a path $u \to v$ consisting of only WHITE vertices

Thm (Parenthesis Theorem)

∀ u, v exactly one of the following is true:

① $d[u] < f[u] < d[v] < f[v]$ OR
$d[v] < f[v] < d[u] < f[u]$
∄ neither of u & v are descendant of the other

② $d[u] < d[v] < f[v] < f[u]$
∄ v is descendant of u

③ $d[v] < d[u] < f[u] < f[v]$
∄ u is descendant of v

⚠ $d[u] < d[v] < f[u] < f[v]$ cannot happen!

Classification of Edges

(T) **Tree Edge**: in DFS forest. Found by exploring (u,v).

(B) **Back Edge**: (u,v) where u is descendant of v
↳ signifies presence of a cycle

(F) **Forward Edge**: (u,v) where v is descendant of u but not tree edge (ie, did not use the edge to discover v, but it is edge in graph)

(C) **Cross Edge**: any other edge

Thm (no proof given → try at home!)

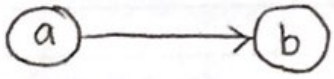
In DFS of undirected graph we get only Tree & Back edges.

22-3 Topological Sort

Topological Sort

Partial order ⇒ Total order → either $a>b$ or $b>a$ $\forall a\,b$
(DAG)
↳ Directed Acyclic graph

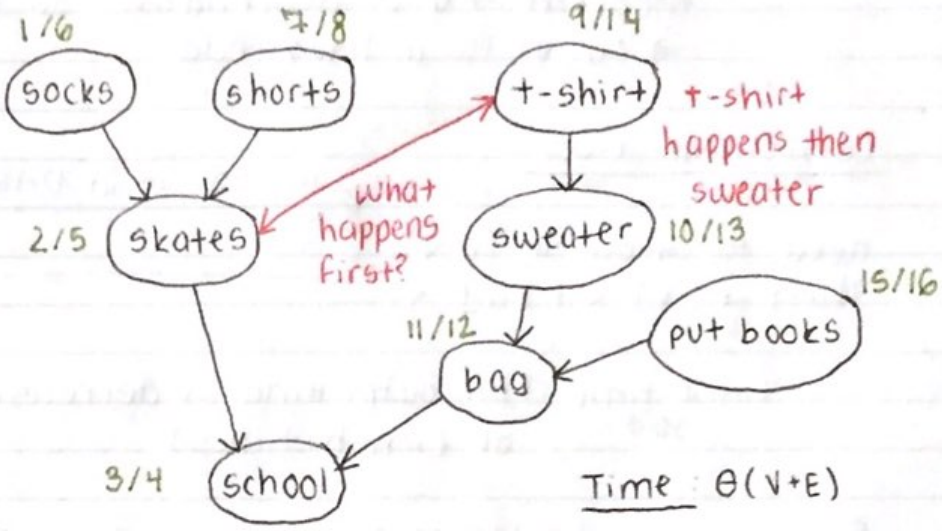
- $a>b$ & $b>c \Rightarrow a>c$
- may have a & b st neither $a>b$ or $b>c$



⇒ a precedes b

"a must happen before b"

Ex: DAG (dressing up for school)



Time: $\Theta(V+E)$

Total order cannot violate partial order.

Topo-Sort (V, E)
call DFS(V, E)
outputs vertices in decreasing order of finishing times

Total order: books → shirt → sweater → bag
→ shorts → socks → skates → school

Lemma: A directed graph G is acyclic iff DFS yields no back edges.

Proof: ① cycle ← if back edge
(easy → prove on own)

② cycle → back edge

proof: suppose G contains a cycle C. Let v be first vertex in C that is discovered. Let (u,v) be preceding edge in cycle C. A time d[v] vertices in C form a →

(to)
white path $v \rightsquigarrow u$. By white-path theorem $\Rightarrow u$ is descendant of v
$\Rightarrow (u,v)$ is a back edge.

Correctness Proof

need to show if $(u,v) \in E$ ← if edge is in DAG
then $f[v] < f[u]$
↑
finish time (b/c outputting in decreasing order of finish times)

Proof: when exploring (u,v) what are colors of u & v? (can tell us about finish times)

- u is GRAY
- is v also GRAY?
  NO! otherwise v would be an ancestor of $u \Rightarrow (u,v)$ is a back edge $\Rightarrow$ contradiction to Lemma
- is v WHITE?
  yes, possible $\Rightarrow v$ becomes descendant of $u \Rightarrow$ by parenthesis theorem
  $d[u] < d[v] < \boxed{f[v] < f[u]}$

This is what we want to show

- is v BLACK?
  yes, possible $\Rightarrow v$ is already finished! since we are currently exploring (u,v)
  $\Rightarrow u$ is not finished
  $\Rightarrow \boxed{f[v] < f[u]}$

□
↳ ends proof

22-4 Strongly Connected Components

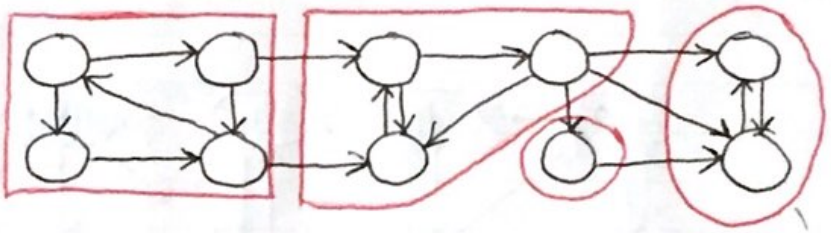
Strongly Connected Components (SCCs)

→ where connectivity is very dense → every node is reachable from every node w/in subgraph

Input: directed $G=(V,E)$
Output: all SCCs of G
↳ moment you add another node, not a SCC anymore

SCC = maximal set of vertices $C \subseteq V$ st $\forall\ u,v \in C$ both $u \rightsquigarrow v$ & $v \rightsquigarrow u$ exist.

Ex:



4 SCCs

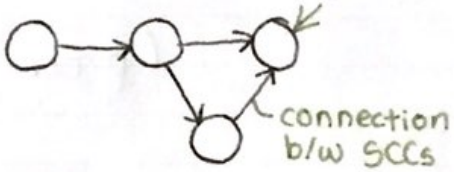
Algorithm uses G^T = transpose of G

- $G^T = (V, E^T)$ → E^T all edges of G reversed
  $E^T = \{(u,v) : (v,u) \in E\}$
  ↑ original

"G^T is G w/ edges reversed"

observe: G^T & G have same SCCs

Component Graph



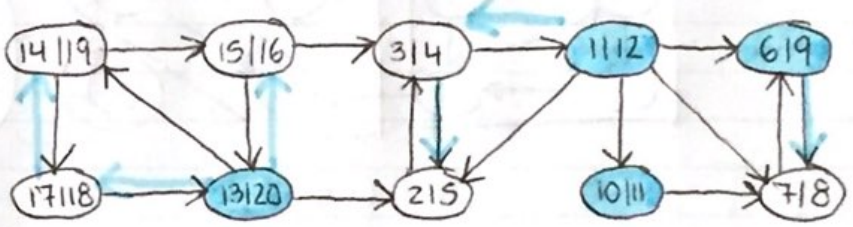
- $G^{SCC} = (V^{SCC}, E^{SCC})$
- V^{SCC} has one vertex per SCC in G
- E^{SCC} has edge if $\exists$ edge b/w SCCs in G

Algo / pseudocode:

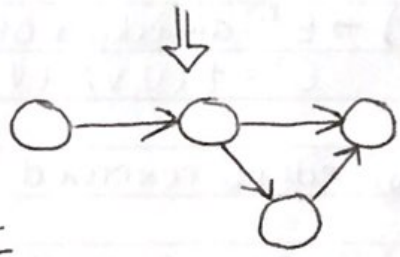
SCC(G)

① call DFS(G) → compute finish times f[u]
② compute G^T
③ call DFS(G^T) in main loop of DFS take nodes in decreasing order of f[u]'s (as computed in first DFS)
④ output vertices in each tree of DFS forest created at step ③ as separate SCC

Time $\Theta(V+E)$



(compute transpose by reversing edges)



output

(proof in textbook)

23-1 Minimum Spanning Trees Intro

Minimum Spanning Trees

→ graph optimization problem

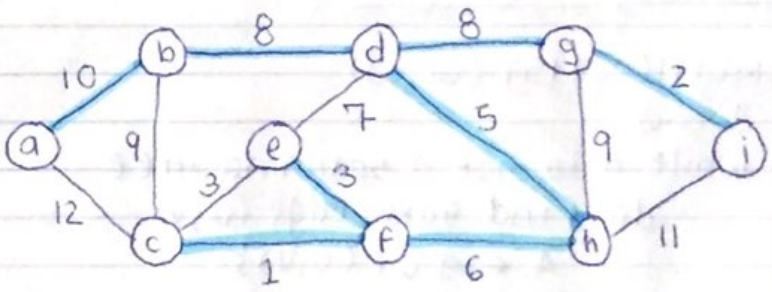
Real-world example: (nodes, edges)
- Town w/ set of houses & a set of roads b/w houses
- Road connects 2 & only 2 houses
- Road connecting houses u & v has repair cost $c(u,v)$ (edge weight)
- Assume all roads broken

Goal: repair enough roads to guarantee
1) everyone stays connected
2) total repair cost is min

Model: $G = (V, E)$ undirected weight $w(u,v)$ on each $(u,v) \in E$ → roads are bi-directional

Goal: Find $T \subseteq E$ such that
① T connects all vertices (T is spanning tree)
② $w(T) = \sum_{(u,v) \in T} w(u,v)$ is minimized
} minimum spanning tree

Ex:



This is an MST (—) w/ total weight 43.

Properties:

① MST has |V|-1 edges
② No cycles (b/c it's a tree)
③ Might not be unique

eg if chosen edge (c,e) instead of (f,e) then still MST

23-2 Generic Algorithm & Correctness Proof

Building up the solution

- we will build a set of edges A
- initially, A is empty
- as we add edges to A we ensure
  ↳ A is a subset of some MST
- safe edge: if A is subset of some MST, an edge (u,v) is safe iff $A \cup \{(u,v)\}$ is also subset of some MST

Note: we will be adding only safe edges

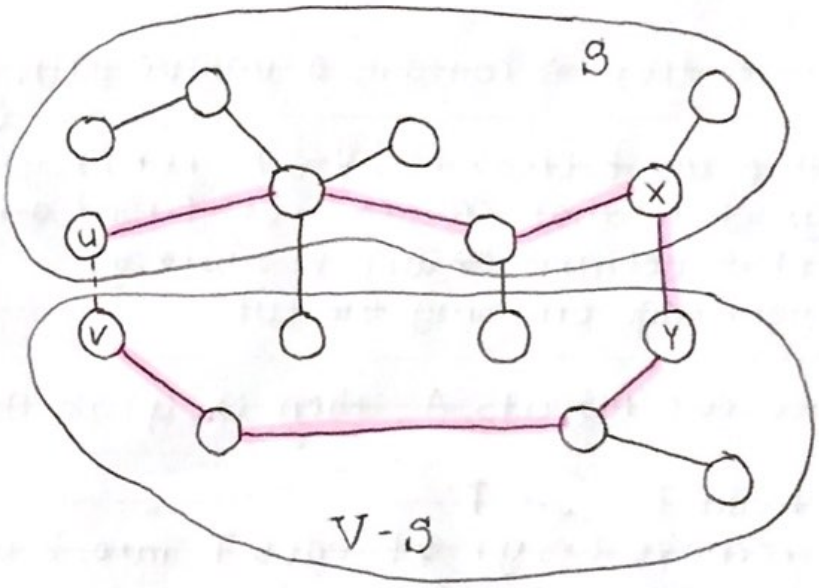
Generic MST algorithm

```
GENERIC-MST(G,w)
    A ← ∅
    while A is not a spanning tree
        do find safe edge (u,v)
            A ← A ∪ {(u,v)}
    return A
```

How to find safe edge?

Let $S \subset V$ & $A \subseteq E$

- A cut $(S, V-S)$ is a partition of V into disjoint sets V & $V-S$
- $(u,v) \in E$ crosses $(S, V-S)$ if one endpoint in S & the other in $V-S$, or vice versa
- cut respects A if no edge in A crosses the cut



An edge is a light edge crossing cut iff its weight is minimum over all edges crossing the cut

Need to prove light edge is safe!

Theorem: Let A be subset of (some) MST. Let $(S, V-S)$ be cut respecting A, & (u,v) be light edge crossing $(S, V-S)$. Then (u,v) is safe for A.

Proof: Let T be an MST that includes A. If T contains (u,v) we are done (trivial case). Assume instead T does not contain (u,v). We will construct a different MST T' that includes

$$A \cup \{(u,v)\}$$

↑ light edge

T is a tree $\Rightarrow$ contains a unique path p b/w u & v.

Path p must cross $(S, V-S)$. Let (x,y) be an edge of p that crosses cut. Based on how (u,v) is defined $\Rightarrow w(u,v) \leq w(x,y)$

↳ light edge crossing the cut

Since cut respects A, then (x,y) is not in A.

To build T' from T:

1) remove (x,y) → breaks T into 2 components
2) Add (u,v) → reconnects

$$T' = T - \{(x,y)\} \cup \{(u,v)\}$$

T' is a spanning tree

$$w(T') = w(T) - w(x,y) + w(u,v)$$
$$\leq w(T)$$

since $w(u,v) \leq w(x,y)$

T' is spanning tree &

$w(T') \leq w(T) \Rightarrow$ T' is also a MST

↑ equality holds, not strictly <

Now $A \subseteq T$ & $(x,y) \notin A \Rightarrow A \subseteq T'$

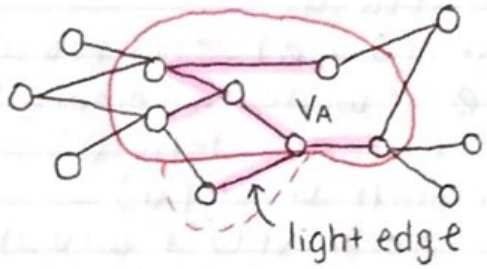
$$A \cup \{(u,v)\} \subseteq T'$$

Finally, since T' is MST, then (u,v) is safe for A. □

→greedy b/c always take edge of smallest weight

23-3 Prim's Algorithm

Prim's Algo



- builds 1 tree, therefore A is always a tree
- At each step, find light edge crossing $(V_A, V-V_A)$, where V_A = vertices that A is incident on
- Implementation: how to find light edge quickly

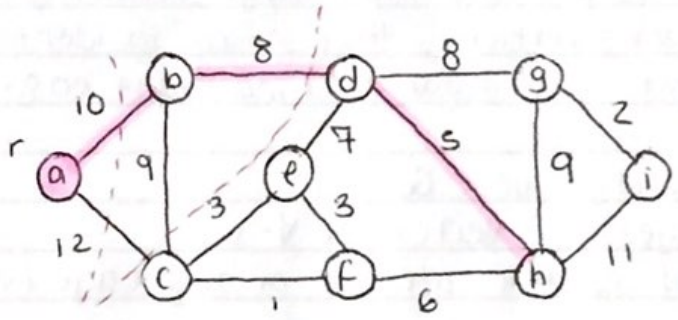
Uses priority queue Q
- each object is vertex in $V-V_A$
- key of v is min weight of any edge (u,v), where $u \in V_A$
- EXTRACT-MIN → v s.t. $\exists\ u \in V_A$ & (u,v) is light edge crossing $(V_A, V-V_A)$
- key of v is ∞ if v not adjacent to any nodes in V_A

Assume: r root of tree is passed as input (can be any vertex)

weight of edges
some root r

```
PRIM(V, E, w, r)
    Q ← ∅
    for each u ∈ V
        do key[u] ← ∞
           T[u] ← NIL
           INSERT(Q, u)
    DECREASE-KEY(Q, r, ∅) /* set key of root to ∅ */
    while Q ≠ ∅ /* while not empty */
        do u ← EXTRACT-MIN(Q)
           for each v ∈ Adj[u]
               do if v ∈ Q & w(u,v) < key[v]
                  then π[v] ← u
                       DECREASE-KEY (Q, v, w(u,r))
```

Ex:



Time:

Q is binary heap

- init Q : $O(V \log V)$
- decrease key of r : $O(\log V)$
- while loop : $|V|$ EXTRACT-MIN
  $\Rightarrow O(V \log V)$
  $\leq |E|$ DECREASE-KEY calls
  $\Rightarrow O(E \log V)$

Total: $O(E \log V)$ → when use standard heaps

Note: can use Fibonacci heaps w/ amortized $O(1)$ Decrease-Key
$\Rightarrow O(V \log V + E)$ total

PS: check Kruskal's algo in CLRS

24-1 Shortest Paths Intro & Properties

Shortest Paths

Motivation: how to find the shortest route b/w 2 points on a map?

Input: (1) directed $G=(V,E)$
(2) weight function $w: E \to \mathbb{R}$
↑ weight of edge

weights on edges could represent distance, cost, penalty, delay, etc.

weight of a path $p = \langle v_0, v_1, \ldots, v_k \rangle$ (sequence of vertices)

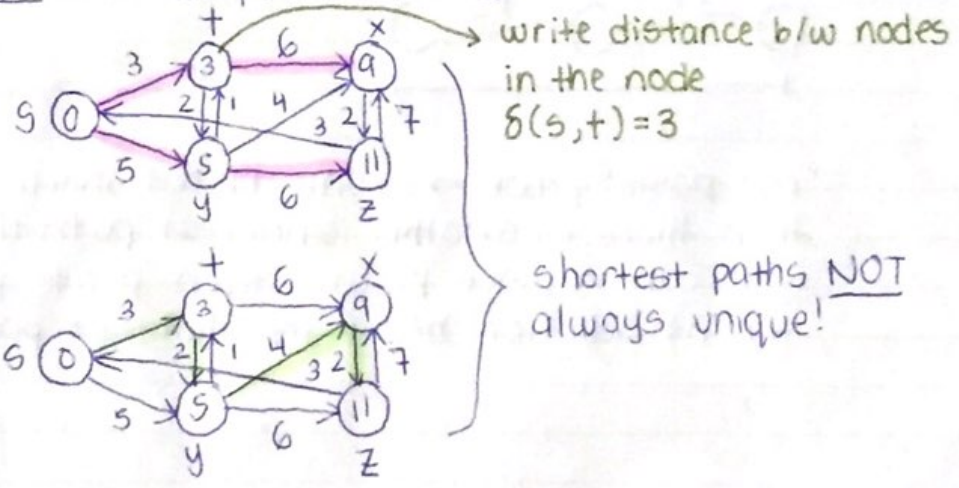
$$\sum_{i=1}^{k} w(v_{i-1}, v_i)$$

↳ weight of all edges b/w v_{i-1} & v_i

Shortest path weight u to v:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\}, & \text{if } \exists p \\ \infty, & \text{if } \nexists p \end{cases}$$

Ex: shortest paths frm s



Variants

shortest path

1) Single source (SSSP)
2) Single destination
3) Single pair
4) all-pairs

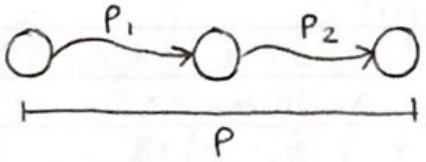
For SSSP we will learn
a) Dijkstra (greedy, no negative weights)
b) Bellman-Ford (negative weights ok)
c) Application: difference constraints

Basic Properties & Assumptions

1) Negative-weight edges?
Ok to have, as long as no negative-weight cycles in G that are reachable from source.

2) Optimal substructure

Lemma: any sub-path of a shortest path is a shortest path



"cut-paste" proof → assume p_1 not shortest, then there is another shortest path that you can replace p_1 w/, then prove p → contradiction b/c p is shortest path

3) Cycles??
shortest paths cannot contain cycles!
(ruled out -ve weight cycles, +ve weight cycles make path worst, 0-weight cycles redundant)

Output:

For each vertex $v \in V$
a) $d[v] = \delta(s,v)$
- initially: $d[v] = \infty$, $d[s] = 0$
- progressively $d[v]$ gets smaller but always $d[v] \geq \delta(s,v)$

b) $\pi[v]$ = predecessor of v on shortest path from s

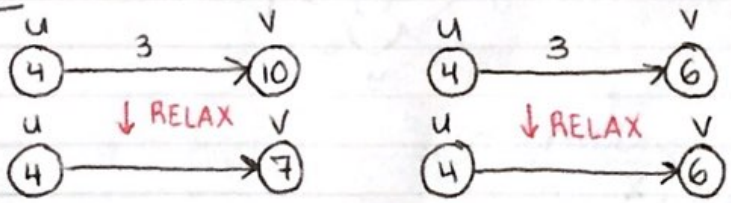
INITIALIZATION

```
INIT-SINGLE-SOURCE(V,s)
    for each v∈V
        do d[v] ← ∞
           π[v] ← NIL
    d[s] ← 0
```

RELAXATION of edge (u,v)
(improving d[v] estimate)

```
RELAX(u,v,w)
    if d[v] > d[u] + w(u,v)
       then d[v] ← d[u] + w(u,v)   /* use edge (u,v) to
            π[v] ← u                 improve path */
```

Example:



24-2 Dijkstra's Algorithm

Dijkstra's Algorithm (1956)

⚠ NO negative-weight edges
↳ (o/w greedy property does not hold)

Maintain 2 sets of vertices:

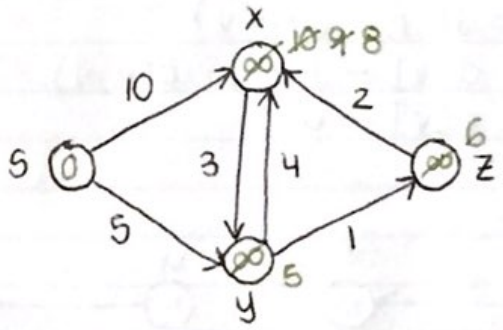
S = vertices whose final shortest path weight is determined

Q = priority queue = $V - S$
→ keys = $d[v]$

Pseudocode:

```
Dijkstra(V, E, w, s)
    INIT-SINGLE-SOURCE(V, S)
    S ← ∅
    Q ← V
    while Q ≠ ∅
        do u ← EXTRACT-MIN(Q)
           S ← S ∪ {u}
           for each v ∈ Adj[u]
               do RELAX(u, v, w)
```

Example:



Time: like Prim's algo
↳ if binary heap
→ $O(E \log V)$

Order of adding to S:
s, y, z, x

Week 10 Videos: 24-3 Bellman-Ford

Bellman-Ford
(aka Bellman-Ford-Moore)

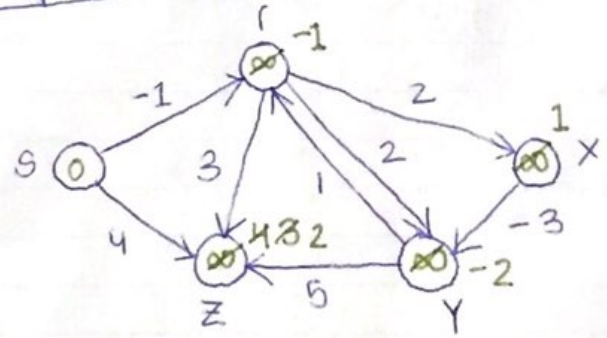
- allows negative-weight edges (diff. frm Dijkstra's)
- computes $d[v], \pi[v] \ \forall v \in V$ (↳ predecessor)
- returns TRUE if no negative weight cycles reachable from s & returns FALSE otherwise

Pseudocode:

```
BELLMAN-FORD(V, E, w, s)
    INIT-SINGLE-SOURCE(V, s)
    for i ← 1 to |V|-1                    } compute shortest
        do for each (u,v) ∈ E             } path
            do RELAX(u, v, w)             }
    for each (u,v) ∈ E                    } determines if
        do if d[v] > d[u] + w(u,v)        } there is a -ve
            return FALSE                  } weight cycle
    return TRUE
```

Time: $\Theta(V \cdot E)$ → slower than Dijkstra's but also permits -ve-weight edges

Example:



order (s,r), (s,z), (r,x), (x,y), (y,z), (r,z), (r,y), (y,r)

iteration 1: $(s,r), (s,z), (r,x), (x,y), (y,z), (r,z)$

iteration 2: $\emptyset$

⋮

4 ✓

Proof that TRUE/FALSE are returned correctly (in pseudocode on previous page)

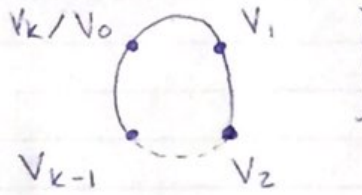
① If $\nexists$ negative-weight cycle
→ Bellman-Ford returns TRUE

Proof: "easy"
↳ triangle inequality property
$d[v] = d[u] + w(u,v)$

② If $\exists$ negative-weight cycle then Bellman-Ford returns FALSE

Proof: assume $\exists$ neg. weight cycle

$C = \langle v_0, v_1, \ldots, v_k \rangle$, w/ $v_0 = v_k$



} cycle (-ve weight cycle)

$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$

ATAC Bellman-Ford returns TRUE

$\Rightarrow d[v_i] \leq d[v_i - 1] + w(v_{i-1}, v_i)$

true for $1 \leq i \leq k$

sum around C (cycle c):

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i))$$

$$\leftrightarrow \boxed{\sum_{i=1}^{k} d[v_i]} \leq \boxed{\sum_{i=1}^{k} d[v_{i-1}]} + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$v_0 = v_k$ (diagram on previous page)

$$\leftrightarrow 0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

↳ but we said the cycle is a -ve (weight) cycle

↳ should be $0 <$

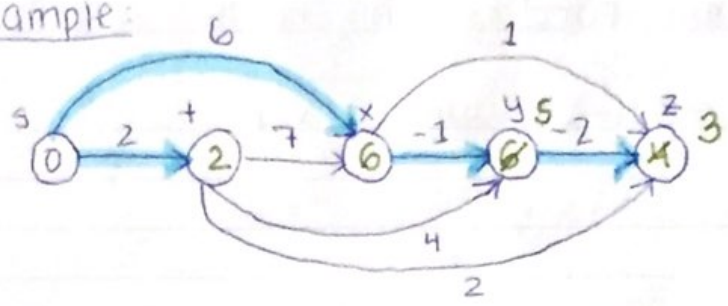
contradiction! since C is neg. weight cycle!

SSSPs in DAGs

↳ single source shortest paths

```
DAG-SHORTEST-PATHS (V, E, w, s)
    topologically sort vertices
    INIT-SINGLE-SOURCE (V, s)
    for each vertex u in topo order
        do for each v ∈ Adj[u]
            do RELAX (u, v, w)
```

Example:



1 2 3 4 5 in topological order

Time: $\Theta(V+E)$

correctness: based on path relaxation property (in textbook)

24-4 Difference Constraints

Difference Constraints

Special case of Integer Programming

→ satisfy a set of constraints while maximizing/minimizing some objective fcn

Difference constraints is a simplified version (no objective fcn).

- given set of inequalities
  $x_j - x_i \leq b_k$
- x's are variables $1 \leq i, j \leq n$
- b's are constants $1 \leq k \leq m$

Goal: find set of values for variables x that satisfy all of the m inequalities or show that no such values exist. →

→ If set of values exists, we call it a **feasible solution**

Example:

$x_1 - x_2 \leq 5$
$x_1 - x_3 \leq 6$
$x_2 - x_4 \leq -1$
$x_3 - x_4 \leq -2$
$x_4 - x_1 \leq -3$

Solution:

$\underline{x}$ ← vector
$\underline{x} = (0, -4, -5, -3)$
also:
$\hat{\underline{x}} = (5, 1, 0, 2) = \underline{x} + 5$
↳ alternative soln

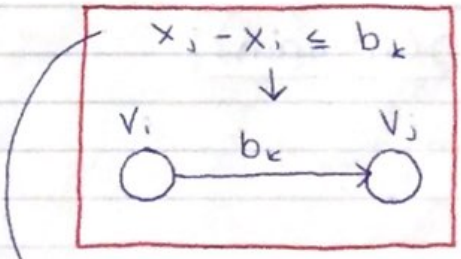
Lemma: if x is feasible soln then $x+d$ is also feasible for any constant d.
(proof in book)

Algorithm

① Build constraint graph $G=(V,E)$
(weighted, directed)

$V = \{v_0, v_1, \ldots, v_n\}$: one vertex per x variable plus v_0 as a pseudo-start vertex

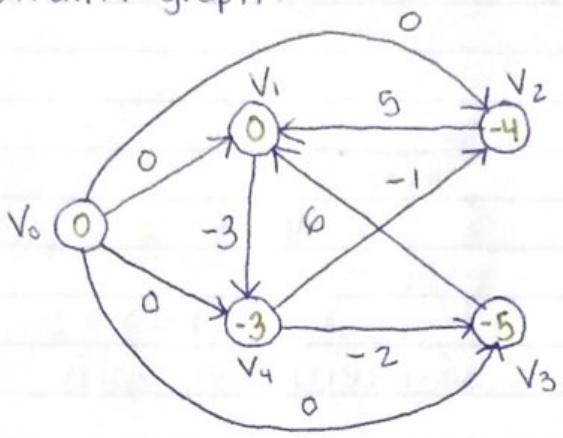
$E = \{(v_i, v_j) : x_j - x_i \leq b_k \text{ is a constraint}\}$
$\cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\}$



$w(v_0, v_j) = 0, \forall j$
$w(v_i, v_j) = b_k$
if $x_j - x_i \leq b_k$

$x_j \leq x_i + b_k$
$d[v_j] \leq d[v_i] + w(v_i, v_j)$

constraint graph:



Theorem: (1) If G has no neg. weight cycle
then $\underline{x} = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n))$
is a feasible soln
(2) If G has neg. weight cycle
then no solution.

Algorithm (cont'd)

(2) Run Bellman-Ford → if neg weight cycle
⇒ no soln o/w use theorem to output
feasible soln

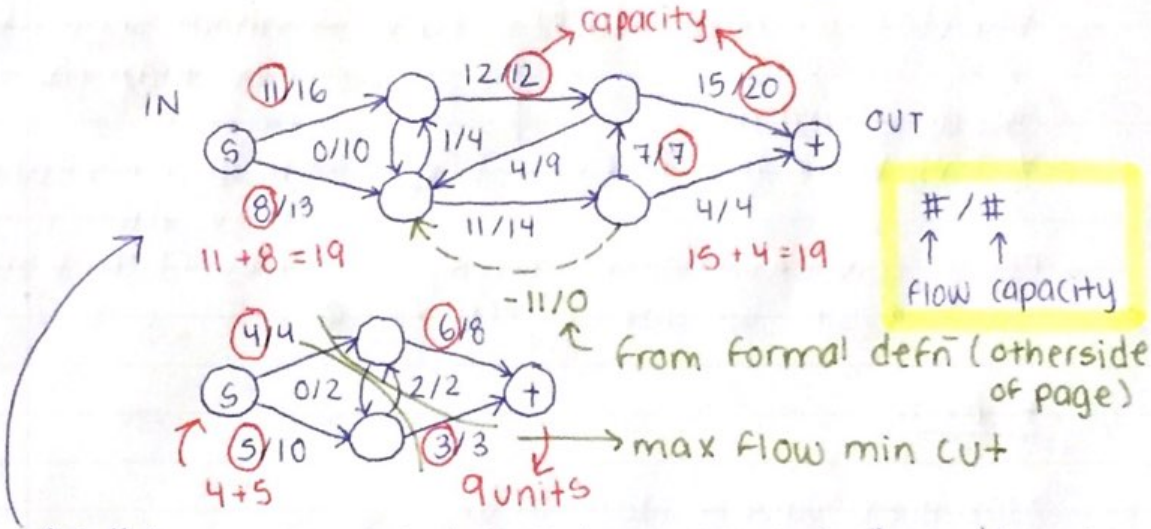
Time: $O(n+m)$ [build graph]
$O(V \cdot E) = O((n+1)(m+n))$

$$\boxed{\text{Time} = O(n^2 + nm)}$$

18-1 Max Flow Definitions

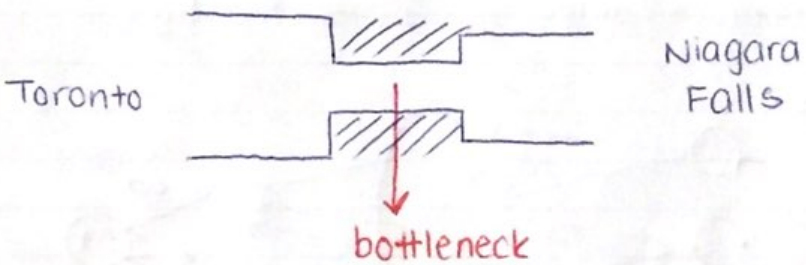
Maximum Flow

Defn: (informal)
We are given a directed positively weighted graph w/ a source S & a sink t. Think of it like water pipes w/ capacity we want to maximize water quantity frm source s to sink t. (no leakage)



Another ex: you are trying to maximize # people you can fly b/w source & sink.

MAX FLOW/MIN CUT Theorem



Defn: (formal)

Given positively weighted, connected, directed graph G where for every edge $u \to v$ there is a capacity $c(u,v) \geq 0$. If $(u,v) \notin E$ then $c(u,v) = 0$. We distinguish two vertices as source s & sink t. A flow is a fcn $f: V^2 \to \mathbb{R}$ w/ properties:

CAPACITY CONSTRAINT
$\forall (u,v) \in V$ we have $f(u,v) \leq c(u,v)$ → cannot send more flow than capacity of edge

↑ respects

SKEW SYMMETRY
$\forall (u,v) \in V$ we got $f(u,v) = -f(v,u)$ → if there is flow frm $u \to v$, there is -ve flow from $v \to u$

FLOW CONSERVATION (no leakage)
$\forall u \in V - \{s,t\}$ we got $\sum_{v \in V} f(u,v) = 0$

Problem:

Maximize value of flow in G.

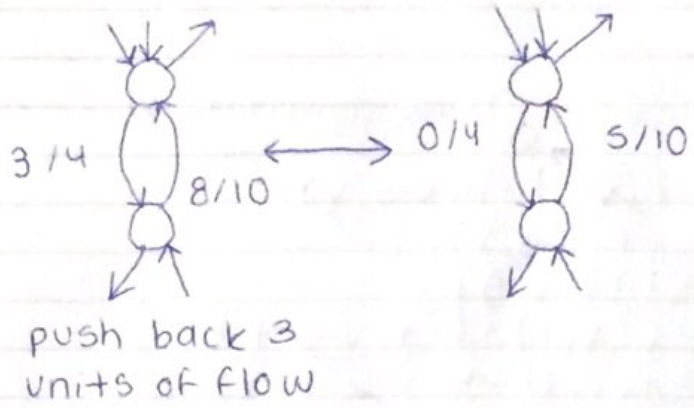
$|f| = \sum_{v \in V} f(s,v)$

Observations:

Multiple sources / sinks can be solved by...

add super source (ss) & super sink (st)

The below are equivalent in terms of flow mathematics:



18.2 Max Flow Math

Notation: $f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y)$ (set)

(try some proofs on own time)

Lemma (Properties):

(1) $f(X,X) = \emptyset$

(2) $f(X,Y) = -f(Y,X)$

(3) $f(X \cup Y, W) = f(X,W) + f(Y,W)$ if $X \cap Y = \emptyset$

(4) $f(W, X \cup Y) = f(W,X) + f(W,Y)$ if $X \cap Y = \emptyset$

Prove (2):

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y) = \sum_{x \in X} \sum_{y \in Y} -f(y,x) \quad \text{(from skew symm property)}$$

$$= -\sum_{x \in X} \sum_{y \in Y} f(y,x)$$

$$= -f(Y,X)$$

Ex

Prove that all units of flow leaving s they enter +

Proof

(flow conservation)

$$
\begin{aligned}
|f| = f(s,V) &= f(V,V) - f(V-s,V) \\
&= -f(V-s,V) \\
&= f(V,V-s) \\
&= f(V,+) + f(V,V-s-+) \\
&= f(V,+) - f(V-s-+,V) \\
&= f(V,+)
\end{aligned}
$$

($f(V,V)$ and $f(V-s-+,V)$ are $\emptyset$)

Defn:

A cut (S,T) of a flow network is a partition of the vertices in a disjoint sets (S,T) s.t. $S \cup T = V$, $s \in S$ & $+ \in T$

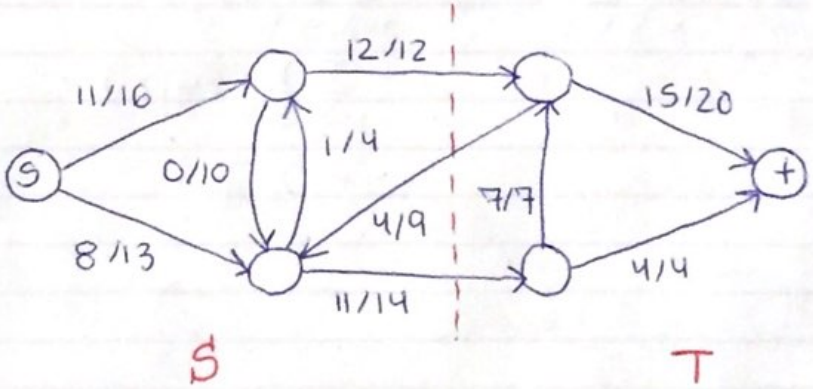
cut capacity $c(S,T) = \sum_{S \to T}^{\text{edges}}$

flow of cut it is $f(S,T)$

$c(S,T) = 26$

$f(S,T) = 19$

$12 + 11 - 4 = 19$

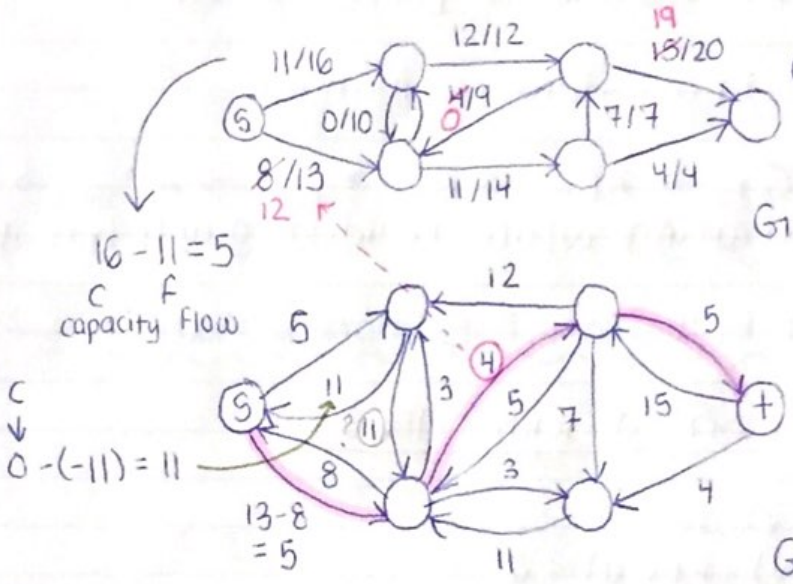


18-3 More Defns & Ford Fulkerson Introduction

Residual Capacity of edge (u,v) → $c_f(u,v)$:
It is the am't of additional flow that can pass t/o edge (u,v)

$$c_f(u,v) = c(u,v) - f(u,v)$$
(f(u,v): existing flow)

Residual Network: (depends on flow)
A graph (network) $G_f = (V, E_f)$ where $E_f = \{(u,v) \in V^2 : c_f(u,v) > 0\}$



→ this new network G_f provides ways to make flow in network G_1 bigger

Augmenting Path (AP)
It is a simple path $s \overset{AP}{\rightsquigarrow} t$ in G_f

Residual Capacity of AP p:
$c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$
↳ minimum residual capacity for all edges that belong to path (in pink on diagram)

Intuition: AP is a sequence of edges where capacity exceeds existing flow & more flow can be pushed in.

Ford Fulkerson (Generic)

- Initialize $f(u,v) = \emptyset \ \forall \ (u,v) \in E$
- while $\exists$ Augmenting Path $s \overset{AP}{\rightsquigarrow} t$ in residual network G_f → Increase Flow in G by adding the residual capacity of AP

→operates in an iterative manner:

$G_0 \rightarrow G_{f_0} \rightarrow AP_{G_{f_0}} \rightarrow G_1 \rightarrow G_{f_1} \rightarrow AP_{G_{f_1}} \rightarrow G_2$
keep iterating while residual graph has no AP

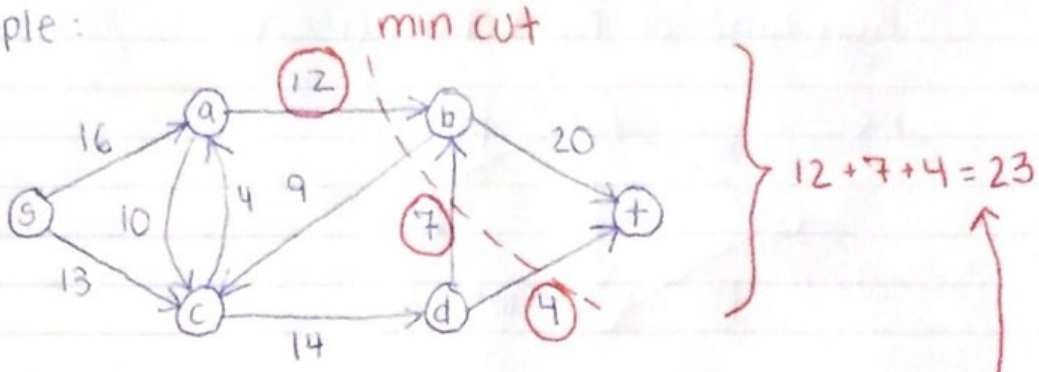
18-4 Ford Fulkerson & Edmonds Karp

Ford Fulkerson (Official, 1956)

$\forall$ edge (u,v) do
$f(u,v) = f(v,u) = \emptyset$

while $\exists$ augmenting path AP $p: s \overset{AP}{\rightsquigarrow} t$ in G_f do
$CF(AP) = \min\{ cf(u,v) \in AP \}$
For every $(u,v) \in AP$ do
$f(u,v) = f(u,v) + CF(u,v)$ ← increase flow w/ residual capacity
$f(v,u) = -f(u,v)$

Example:



Augmenting Paths	Flow Added
$s \to a \to b \to c \to d \to t$	4
$s \to a \to c \to d \to b \to t$	7
$s \to c \to a \to b \to t$	8
$s \to c \to b \to t$	4
max flow is	23

Why Ford-Fulkerson terminates?
B/c of max flow/min cut theorem!

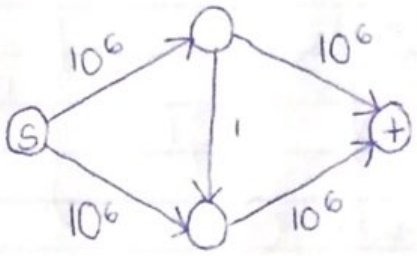
Max Flow - Min Cut Theorem

∀ Flow f on graph G, the following statements are equivalent:
- f is a max-flow for G
- G_f has no augmenting path
- $|f| = c(S,T)$ for some cut (*)

(*) Another theorem proves that this is the minimum cut

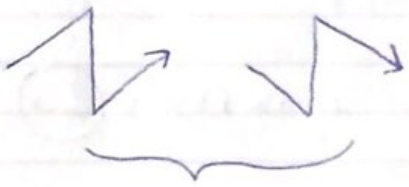
Run time of Ford Fulkerson?

Ex:



Suppose we use BFS to look for an AP → O(E)

Max Flow = 2×10^6



does these iterations
2×10^6 times

Runtime = $O(|f_{max}| \cdot E)$

↑ size of flow

↖ BFS takes O(E) to find path

→ undesirable solution?

Edmond-Karp's algorithm: (1972)

Always choose the augmenting path w/ minimum # of edges (shortest path)

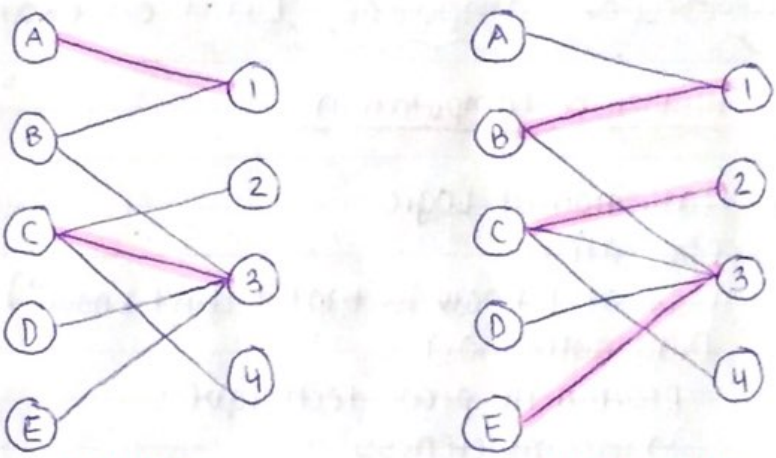
↳ put unit on every edge & run shortest path algo

Runtime: $O(VE^2)$

18-5 Maximum Bipartite Matching

Maximum Bipartite Matching

Matching:
A subset $M \subseteq E$ in G s.t. $\forall v \in V$ at most one edge from M is incident on v.

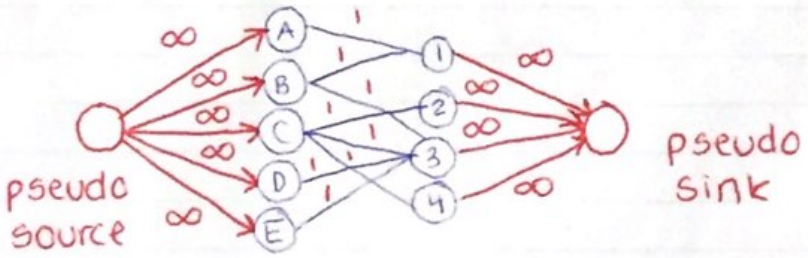


MAXIMAL — no other edges o/w a vertex will have >1

MAXIMUM

We are interested in a maximum matching M (ie, how many # of edges) in the bipartite graph.

We reduce the problem to Max Flow.



- Introduce pseudo source/sink connected to the two partitions of the bipartite graph
- Put ∞ capacity on new edges, capacity 1 on original edges
- Run Edmond-Karp $|F| = |M_{max}|$

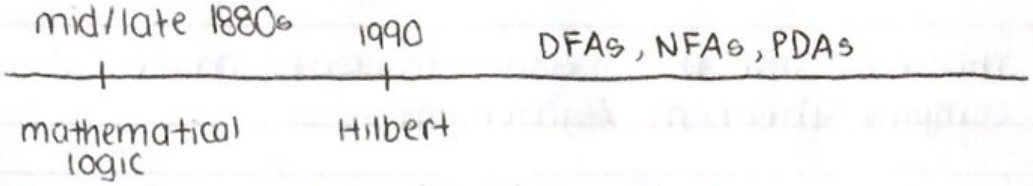
Week 11 Videos: 19 History of Computing [not on exam]

A Brief History of Computation

1800's Mathematical Logic
(Socrates)
("what I know is that I don't know")
Diagonalization
- prominent proof technique
  - ↳ incompleteness
  - ↳ halting problem

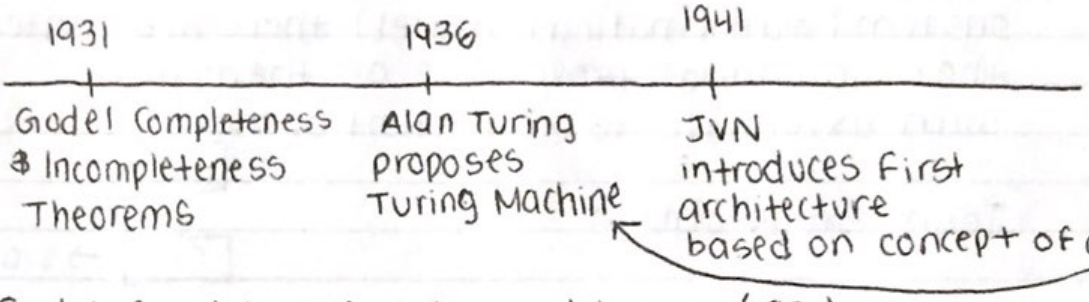
1900's Computability
(1950-1980)

1972 Complexity
NP-Completeness



(Hilbert) → machine / algorithm

Is there a way that we can compute (answer) any mathematical problem?.



Godel Completeness & Incompleteness (1931)

Axiomatic System

Epitome of first order (predicate) logic

Every house is a physical object
$\forall x (house(x) \rightarrow object(x))$
"some objects are houses"
$\exists x (object(x) \cap house(x))$
"Every house has an owner"
$\forall x (house(x) \rightarrow \exists y\ owns(y,x))$

Axiomatic system:
Is a finite set of first order axioms that can form the basis to create more complex math statements.

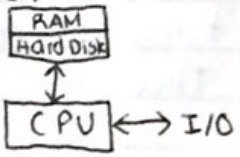
Alphabet + Grammar = complex statements

Theorems use the axioms to create more complex theorems/statements.

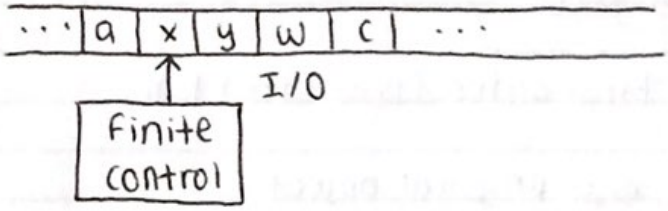
completeness: ∀ statement ∃ axiomatic that can prove it or disprove it.

incompleteness: Given a fixed axiomatic system (mathematical model) there are statements that we cannot prove. One of them is if w/in axiomatic is consistent or not.

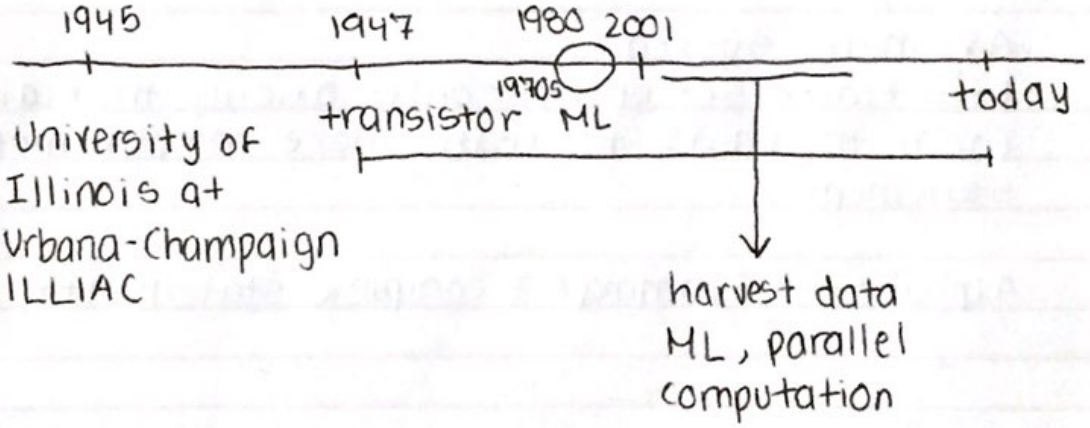
John Von Neumann



Turing machine



Basis of modern computation Church-Turing Thesis:
(conjecture → cannot be proven, assumed true)
No model of computation can compute something the TM cannot compute.



Quantum Computing
q-bit ≠ bit
Sorting $O(\sqrt{n})$ time

computer $\stackrel{?}{=}$ mind

25-1 Theory of Automata

Theory of Computation / Automata

Preliminaries / Defns

Alphabet: finite, non-empty, ordered (which characters come first) set of symbols (or characters)

String over alphabet: sequence of symbols from that alphabet

Example: $\Sigma_1 = \{a, \ldots, z\}$ and $\Sigma_2 = \{0, \ldots, 9\}$ (1 - unary, 2 - binary)

abb is a string over Σ_1
123 is a string over Σ_2
ab3 is not a string over Σ_1 or Σ_2

Note: string w/ no symbols = empty strings = ε

Concatenation: concatenation of strings a & b is written as ab. The notation a^i means the string obtained by concatenating i copies of a.

Example: concat 01 w/ 100 $\Rightarrow$ 01100

- $a = 01$ then $a^0 = \varepsilon$, $a^1 = 01$, $a^2 = 0101$

$\varepsilon\varepsilon = \varepsilon$	
$a\varepsilon = a$	
$\varepsilon a = a$	

Σ^* : the set of all strings over Σ ex: $\{0,1\}^*$ = set of all binary strings

Σ^+ : $\Sigma^* - \{\varepsilon\}$

Reverse of string:
if $a = a_1 a_2 \ldots a_n$ then $a_n a_{n-1} \ldots a_1$ is called the reverse of a & is denoted a^R

Languages: ("problems")
sets of strings frm a particular alphabet

If Σ is alphabet & L (↗ language) is a subset of Σ^*, then L is called a language over Σ. Each element of L is a string over the language.

Note: $\{\varepsilon\}$ and $\emptyset$ are languages over every alphabet.
↑ empty string ↑ language that contains nothing

Operations: $L_1 \cup L_2$
$L_1 \cap L_2$
complement of L is $\bar{L}$
$L_1 - L_2$

Examples: $L_1 = \{\varepsilon, 0, 1\}$ & $L_2 = \{\varepsilon, 01, 11\}$

- $L_1 \cup L_2 = \{\varepsilon, 0, 1, 01, 11\}$
- $L_1 \cap L_2 = \{\varepsilon\}$
- $\bar{L_1} = \{00, 01, 10, 11 \ldots\}$ → everything that does not include $\varepsilon, 0, 1$

L^i: concat i copies of L
($L^0 = \{\varepsilon\}$)

$L^0 \cup L^1 \cup L^2 \cup \ldots$ = Kleene Closure
$= L^*$

Example: $L_1 = \{\varepsilon, 0, 1\}$ & $L_2 = \{01, 11\}$

- $L_1^2 = \{\varepsilon, 0, 1, 00, 01, 10, 11\}$
- $L_2^3 = \{010101, 010111, 011111 \ldots\}$

Can use properties of strings to describe language (in which strings belong)

Example: $L = \{0^p \mid p \text{ is a prime } \#\}$
$= \{00, 000, 00000, \ldots\}$

25-2 Regular Languages & DFAs-1

A regular expression over Σ is defined as follows:

① ε is a regular expression (RE)
② $\forall a \in \Sigma$, a is a RE
③ If R, S are RE, then $R+S$ is a RE ("R or S")
("or")
④ If R, S are RE, then RS is also a RE
⑤ If R is a RE, then R^* is also a RE
⑥ If R is RE, then (R) is also RE

- A regular language frm a regular expression R is denoted L(R) & is set of all strings that R denotes.

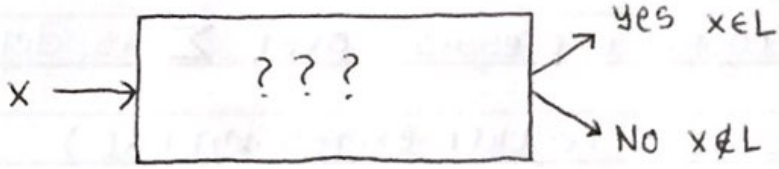
Examples:

- $L(0) = \{0\}$
- $L((0+1)(0+1)) = \{00, 01, 10, 11\}$
- $L(0^*) = \{\varepsilon, 0, 00, 000, \ldots\}$
  ↑ ∞ length
- regular expression RE $(0+1)^*1$ denotes all strings of 0s & 1s that end w/ 1
- ~~$L((1^*01^*01^*)^*)$~~ denotes set of all binary strings w/ an even # of 0s

correction: $1^*1(1^*01^*01^*)^*$ or $(1^*1(01^*0))^*$

25-2 Regular Languages & DFAs-2

Language L is $x \in L$? ⇒ $L = \{$all dir graphs G that are strongly connected$\}$
is input G strongly connected?
↳ input string x (encoding of G)
& we want to answer if $x \in L$



Finite Automata are language recognition devices.

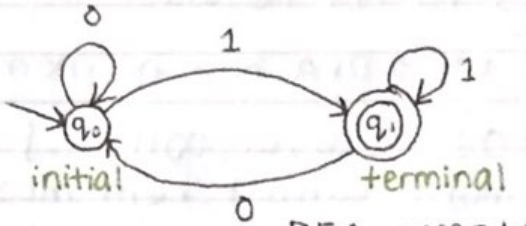
- we say a finite automaton accepts language L if ↗ yes (accepts), $\forall x \in L$ ↘ No (rejects), $\forall x \notin L$

*may be on final

Deterministic Finite Automata

(check handout)

restricted computation models:
- states
- state transitions
- special terminal state



→ DFA accepts

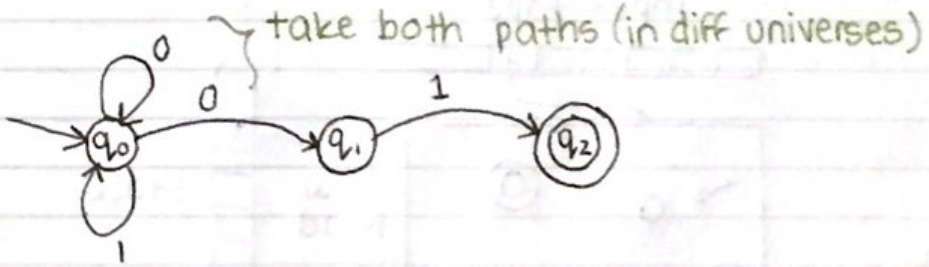
Ex: $w = 1101$ (strings read from left to right)
$w' = 1100$ → DFA rejects

$L = \{ w \in \{0,1\}^* \mid w \text{ ends w/ } 1 \}$

Theorem: A language is regular iff it is accepted by some DFA.

Nondeterministic Finite Automata (NFA)

$L = \{ w \in \{0,1\}^* \mid w \text{ ends w/ } 01 \}$
(reg. exp. $(0+1)^*01$)
↑ expression



$w = 1001$ is accepted!

Theorem: For each NFA $\exists$ an equivalent DFA accepting same language.

Languages that cannot be accepted by DFA/NFA:

- $\{0^p \mid p \text{ is prime}\}$ is not regular
  - ∞ prime #s → DFA has no memory
- $\{0^n 1^n \mid \forall \text{ int } n \geq 0\} = \{\varepsilon, 01, 0011, \ldots\} \rightarrow \infty$ set
  - ↳ no memory: cannot remember # of 0's seen to ensure #0s = #1s are equal

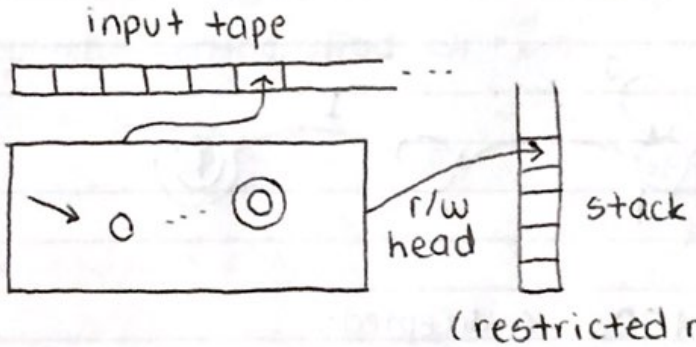
Context-Free Languages

$L = \{0^n 1^n \mid \forall \text{ int } n \geq 0\}$ ← pop same # of times as push ↓ stack empty ↓ accept!

if $S \in L$ then $0S1 \in L$ (0 ↙ zero), $00S11 \in L$ ...

set of rules that allow you to do more than just regular expressions (see handout)

Nondeterministic Pushdown Automata (NPDA)



Theorem: languages accepted by NPDA is exactly set of context-free languages.

Example where NPDA cannot accept

$L = \{0^n 1^n 2^n \mid \forall n \geq 0\}$
↳ cannot detect middle in string
↳ need more memory to determine when to push/pop

$L = \{www \mid \forall w \in \{0,1\}^*\}$

... need something more powerful
→ enter Turing Machines
abstract model for computers programs/algo

25-3 Turing Machines

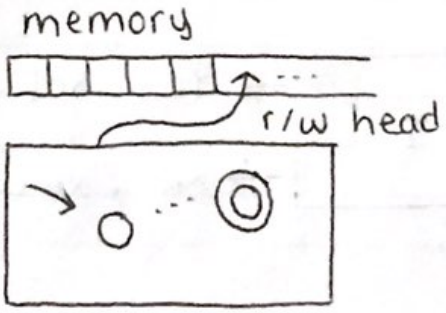
Turing Machines (TM)

End goal: take any decision problem & figure out how difficult it is to solve it (put it into categories of complexity)

A TM is finite state machine connected to an infinite length memory tape.

Memory tape: initially holds the input string & can be written to & read from using r/w head (can move in both directions)

TM:



Basic operations:

① Put the finite state machine in a new state
② Either:
  (a) write symbol in tape square where r/w head currently resides
  (b) move r/w head one square L or R

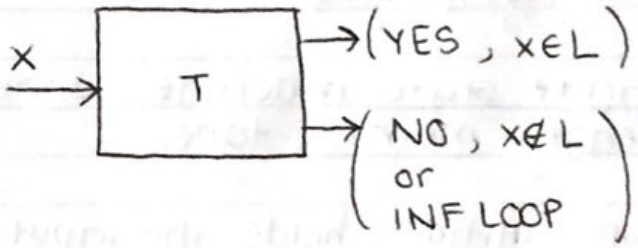
Halt State: signals the end of computation

⚠ TM does not have to halt unlike DFA/NFA/NPDA (end when done parsing input?)

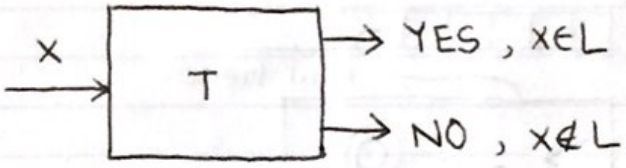
→ if reach halt state, computation ends

⚠ TM can go into an ∞ loop!!

TM T accepts language L → weaker result



TM T decides language L



The Church-Turing Thesis

if an algo exists then there exists an equivalent TM for it.

TM ↔ algo / program

Universal TM (UTM) ↔ computer

↓

On input ⟨M, w⟩ it simulates TM M on input string w.

M specifies TM ie. algo / program
w " input

Notation: ⟨x⟩ encoding of input x in binary
⟨M⟩ " of implementation of TM M in binary

(read handout for more details)

There exist undecidable languages ↔ there exist unsolvable decision problems. (result does not give YES/NO)

Example: The canonical example:

HALTING PROBLEM (HP)
⟨does input program terminate on a given input string?⟩

→ to be a program that determines if all programs terminate, need to determine if you terminate, then you need to be passed into another program to determine this, etc... (no conclusion)

Diagonalization proves that HP is undecidable (unsolvable). (in handout)

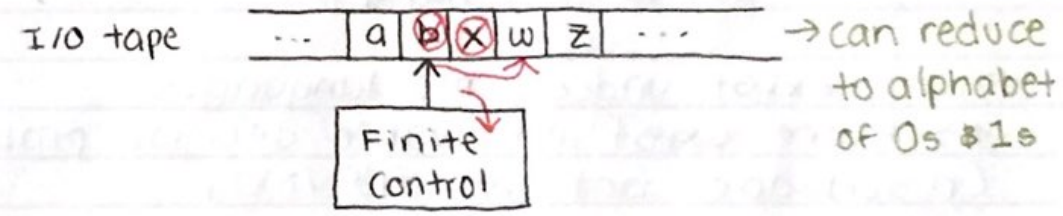
⚠ there are ∞ more undecidable problems than decidable problems.
- undecidable problems → uncountably ∞ many → map to $\mathbb{R}$
- decidable problems → countably ∞ many → map to $\mathbb{N}$

20-1 Intro to NPC

NP - Completeness

Problems vs. problem instances
- Problems → defn
- problem instances → an actual problem

Recall TM:



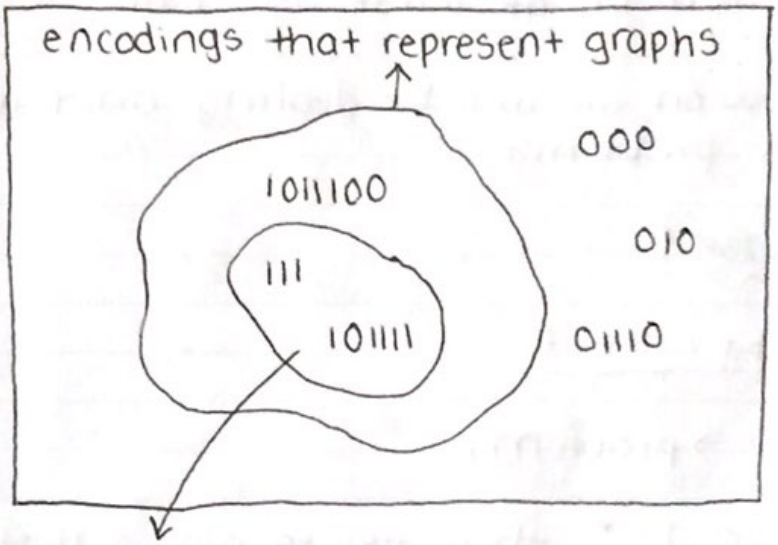
Gödelization

$\Sigma = \{0, 1\}$
$L = \{0, 1, 01, 10, 00, 11, 000, \ldots\}$

Every problem can be imagined as a subset of strings $\{0,1\}^*$

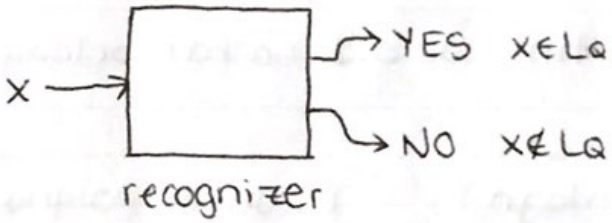
Does a graph have a path of at most k-edges?

$\{0,1\}^*$



Q = Does a graph have a path of length k b/w 2 vertices?

$L_Q = \{ x \in \Sigma^* : Q(x) = 1 \}$



Q is a decision problem → requires yes/no answer

Shortest paths: what is the SP b/w those 2 vertices?
↳ not a decision problem

Observation:
If general problem is "easy", then the decision version is also "easy".

If the decision is "hard" then the general problem will be at least as "hard".

From now on we will be dealing ONLY with decision problems.

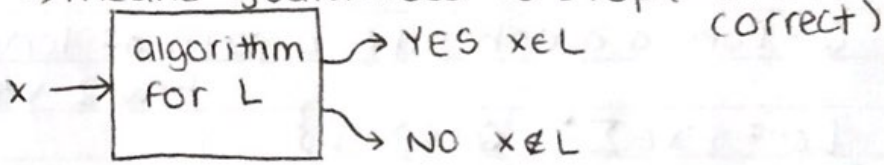
20-2 Class P

Complexity class P

(Language → problem)

$P = \{ L \in \{0,1\}^* :$ there exists algo A that decides L in polynomial time$\}$

↳ means guaranteed to stop (= answer correct)



G, k=5, U/V

Every algorithm we did so far belongs to class P!

$O(n^2)$ $O(n \log n)$ $O(VE^2)$ ... } run in polynomial time

THM:

P = { languages that can be accepted in poly-time}

Proof:

If accepted in poly time, that means machine A if it stops it takes cn^k steps. →

→
Simply create machine A' that simulates A for cn^k steps & halts w/ acceptance if A halts, otherwise rejects the strings.

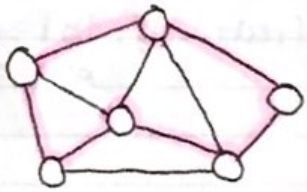
A' decides the language in poly-time.

20-3 Polynomial Time Verification

HAMILTONIAN CYCLE

HAM-CYCLE:
In an undirected graph, find a simple (not repeating a vertex) cycle that traverses all vertices.

Commentary:
Elegant & simple problem that if we can solve we can address ... computational biology problems.



Today the only exact solns we have are exponential (ie. enumerate every cycle).

Verification

Algo A verifies a problem if given instance $x \in L$ of the problem there exists a certificate (or witness, basically a "candidate soln") st $A(x,y) = 1$. The language verified is:
$L = \{ x \in \{0,1\}^* : \exists\, y \in \{0,1\}^* \text{ st } A(x,y) = 1 \}$

Ham Cycle: hard to solve but what abt. if you are to verify a candidate soln?. To verify it this takes poly-time!.

20-4 NP Reducibility

NP → nondeterministic polynomial time

Complexity Class NP

Informally:
It is the class of problems that have a poly-time verification algorithm.

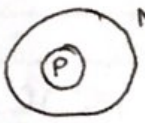
Formally:
Language (problem) L belongs to complexity class NP if ∃ poly-time algo A ∄ a constant c st. L contains the string:

$$L = \{ x \in \{0,1\}^* : \exists \text{ certificate } y, |y| = O(|x|^c) \text{ st } A(x,y) = 1 \}$$

(y → candidate soln)

Thm:

$P \subseteq NP$ (P is subset of NP)



why?
If $L \in P$ that means ∃ algo to decide (solve) L in poly time. Hence, ignore the certificate ∄ solve it in poly-time.

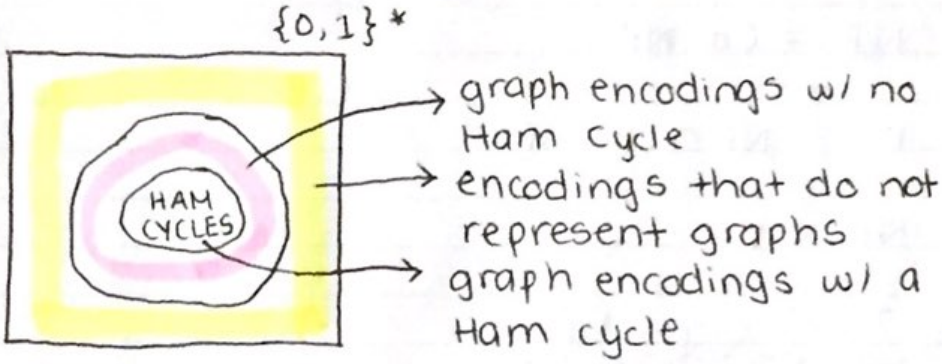
Complexity Class co-NP

$L \in NP \Rightarrow \bar{L} \in co\text{-}NP$

co-NP = complement NP

Ex: co-Ham-Cycle

"$x \in \bar{L}$ is not a graph, or if it is, it doesn't have a Ham Cycle" (spaces in diagram below)

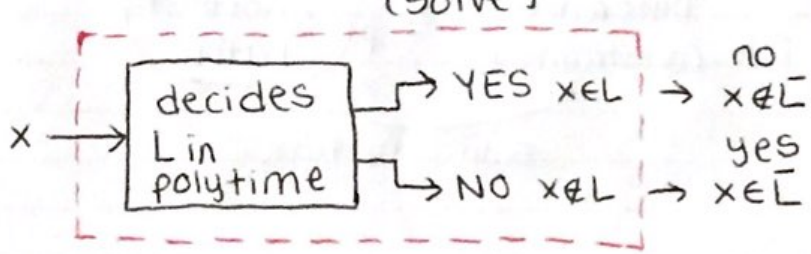


Thm

Class P is closed under complement.
That is
$L \in P \Rightarrow \bar{L} \in P$

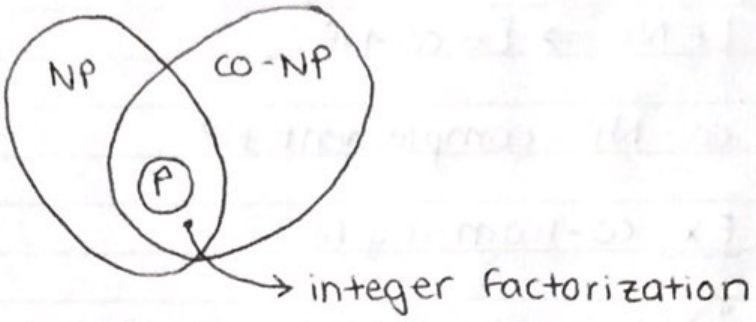
Proof:

IF $L \in P$ then $\exists$ TM A deciding L in poly time



deciding L in polytime

Relation b/w complexity classes:



Open Questions:

$NP \stackrel{?}{=} Co\text{-}NP$

$P \stackrel{?}{=} NP \cap Co\text{-}NP$

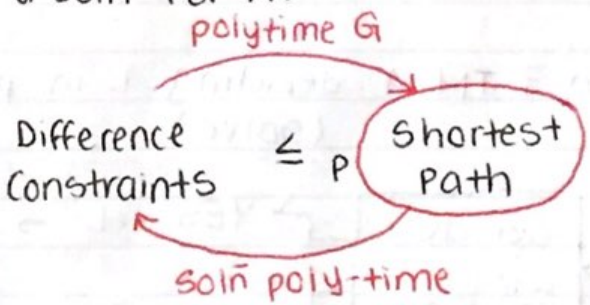
$NP \stackrel{?}{=} P$

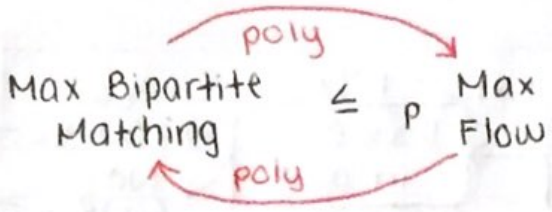
Week 12 Videos: 20-5 Class NP

Reducibility (informal)

we say that $A \leq_P B$ ("problem A is polynomially reducible to B") iff ∃ poly-time algo that given an instance of A we can transform it to some instance of B, solve B & then in poly-time obtain a soln for A.

Ex:





Reducibility (formal)

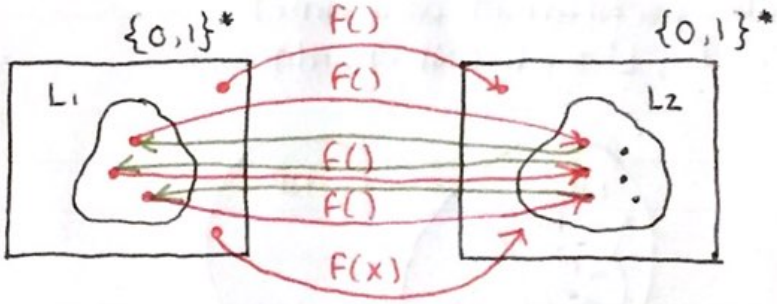
Language L_1 is polynomially reducible to L_2 denoted as

$$L_1 \leq_p L_2$$

iff $\exists$ poly-time algo $f()$:

$$x \in L_1 \text{ iff } f(x) \in L_2$$

That is, if we can answer if $f(x) \in L_2$, we can answer if $x \in L_1$



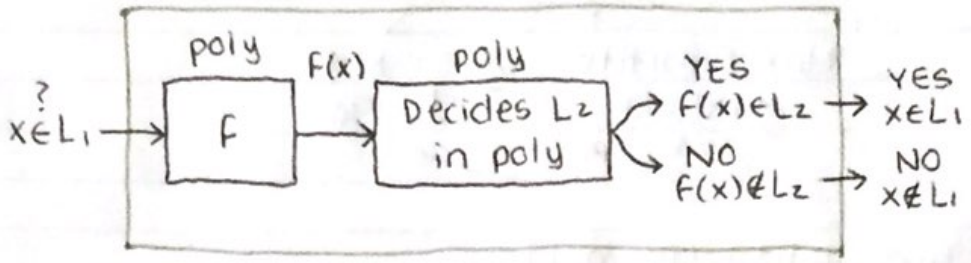
→ every instance of L_1 can map into L_2
(every instance not in L_1 can map not into L_2)

→ L_2 may contain other instances

THM:

if $L_1 \leq_p L_2$ & $L_2 \in P$ (ie. L_2 can be solved in poly-time) then $L_1 \in P$

if we can solve L_2 in polytime & $L_1 \leq_p L_2$ then we can solve L_1 in polytime.

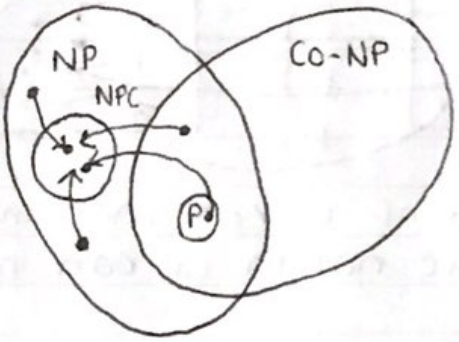


Machine deciding L_1 in poly-time

20-6 NPC Definition

Complexity Class NP-Complete

Language $L \in NPC$ iff
- $L \in NP$ (verified in poly-time)
- $\forall L' \in NP$, $L' \leq_p L$ (NP-hardness)



Practice Theorems

THM: if $\exists L \in NPC$ & $L \in P \Rightarrow P = NP$

THM: $NP = Co\text{-}NP$ iff $\exists L \in NPC$ st $\bar{L} \in NP$

$\Rightarrow$ straightforward

⇐ Let $L \in NPC$ & $\bar{L} \in NP$
Pick any $L' \in NP$ (where $\bar{L}' \in co\text{-}NP$)
Since $L' \leq_p L$ this implies that $\bar{L}' \leq_p \bar{L}$
which implies that $\bar{L}' \in NP$.

→ back to:
Language $L \in NPC$ iff
- $L \in NP$
- $\forall L' \in NP, L' \leq_p L$

Tough problem!
→ somehow need to show it holds <u>for every</u> $L' \in NP$ → goodluck!

*on exams/ hwk <u>Methodology to prove NPC:</u>

Given language L to show $\in NPC$:

(a) Prove that a soln can be verified in poly time.

(b) Select known $L' \in NPC$

(b1) Describe algo $f()$ that given instance $x \in L'$ it "translates" it to $f(x)$ st
$x \in L' \leq_p f(x) \in L$

(b2) Show $f()$ takes poly time to compute.

→ works b/c:

$NP_1 \leq_p L'$
$NP_2 \leq_p L'$
$NP_3 \leq_p L'$
} $L' \leq_p L$ ↓ NPC

logic:
all NP complete languages can be reduced to L' in poly-time → L' can be reduced to L in poly-time (so all NP complete languages can be reduced to L in poly-time)

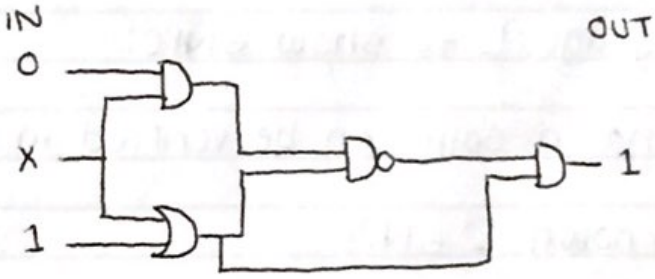
chicken & egg problem!

Methodology abv solves the problem as we just need reduce an existing $L' \in NPC$ problem to L

But what is the first NPC problem?

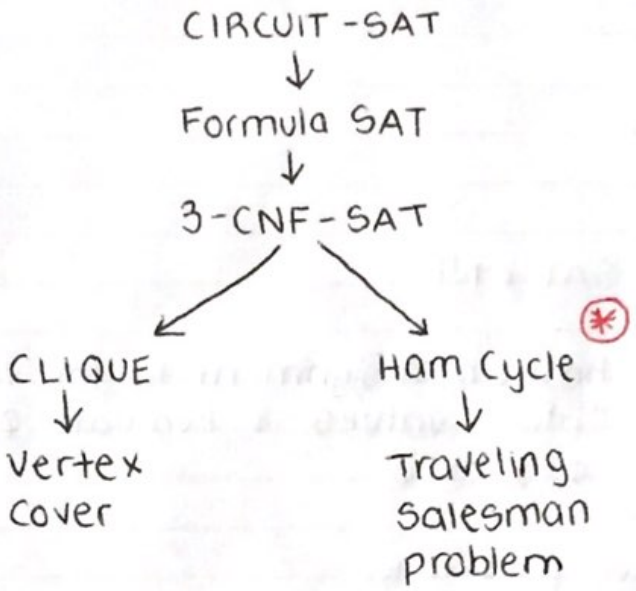
Answer:
Cook's Theorem (1971):
Circuit SAT is NPC
↑ satisfiability

Circuit - SAT (Cook 1971)
Given a AND, OR, NOT digital circuit w/ a <u>single</u> output find an input vector that makes the output 1.



$\forall$ NP problem $\leq_p$ circuit SAT

First NPC problem:



21-1 NPC Formula Sat

Formula SATisfiability

A formula Φ w/ n-Boolean variables & connectives

$$\wedge \text{ (AND)}, \quad \vee \text{ (OR)}, \quad \neg \text{ (NOT)}, \quad \rightarrow \text{ (IMPLICATION)}, \quad \leftrightarrow \text{ (EQUIVALENCE)}, \quad (,)$$

Decision Problem: $FSAT = \{\Phi : \text{Formula } \Phi \text{ has a satisfying Boolean var assignment st } \Phi = 1\}$

$$\Phi = (x_1 \vee \bar{x}_2) \leftrightarrow ((x_3 \rightarrow \bar{x}_4) \vee (\bar{x}_2 \vee \bar{x}_3))$$

Is there a Boolean assignment to variables x_1-x_n st $\Phi = 1$?

$a \to b$

a	b	$a \to b$
0	0	1
0	1	1
1	0	0
1	1	1

$a \leftrightarrow b$
↳ both sides need to be same

(a) Formula SAT $\in$ NP

Given a Boolean assignment to the variables $x_1 \ldots x_4$ "plug" values & evaluate Φ to check if $\Phi = 1$ or 0

This takes poly-time.

(b) CIRCUIT-SAT $\leq_p$ FORMULA-SAT

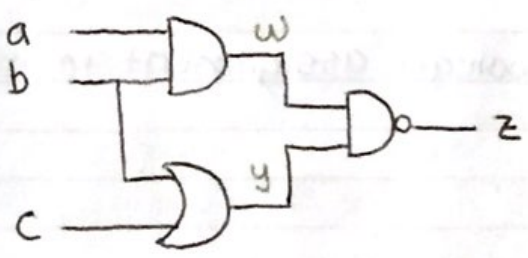
that is, given a Boolean circuit we will construct a formula Φ s.t.
circuit = satisfiable iff Φ is satisfiable

⚠ WRONG! (common mistake below) ⚠

Formula SAT $\leq_p$ Circuit SAT

↳ trying to prove opposite direction
↳ ERRONEOUS!

ex:



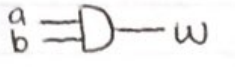
① Name internal lines

needs to be 1 for $\Phi = 1$

② $\Phi = [(a \wedge b) \leftrightarrow w] \wedge [(b \vee c) \leftrightarrow y] \wedge [(w \wedge y) \leftrightarrow \bar{z}] \wedge z$

need in order to have $\Phi = 1$

part of the graph:



$\Leftrightarrow (a \wedge b) \leftrightarrow w$

characteristic function

circuit is satisfiable $z = 1$

$\updownarrow$

Φ is satisfiable ($\Phi = 1$)

③ $\forall$ gate we introduce a constant # of literals. Therefore the construction takes poly-time.

21-2 & 21-3 CNF SAT

3-CNF-SATISFIABILITY

You are given a formula Φ in Conjuctive Normal Form (CNF):

$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_4 \vee x_5) \wedge \dots$

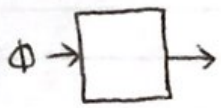
clauses

Every clause has 3 literals

↳ a variable that can be complemented or not

$\Phi = (\ ) \wedge (\ ) \wedge (\ ) \wedge (\ ) \dots$

SAT solvers



$\Phi = 1$
or
Φ cannot be satisfied

2-CNF: polynomial time

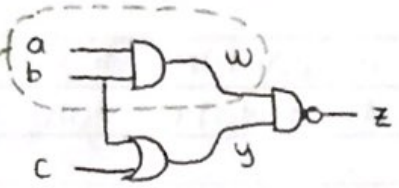
2.73-CNF $\in$ NPC

DECISION PROBLEM: is there a Boolean assignment to variables $x_1 \ldots x_n$ s.t. $\Phi = 1$

(a) show 3-CNF $\in$ NP
Similar to formula-SAT

CLRS: Formula SAT $\leq_p$ 3-CNF-SAT

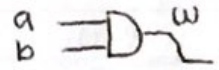
(b) we will prove the following CIRCUIT-SAT $\leq_p$ 3-CNF-SAT



Given a circuit we will create a 3-CNF formula Φ s.t.:

CIRCUIT is SAT
iff
Φ is SAT

Label internal lines



$w \leftrightarrow a \wedge b$

characteristic fcn:

a	b	w	$w \leftrightarrow a \wedge b$	
0	0	0	1	
0	0	1	0	←
0	1	0	1	
0	1	1	0	←
1	0	0	1	
1	0	1	0	←
1	1	0	0	←
1	1	1	1	

Maxterms above:

$\Phi_{IN} = (\bar{a} \wedge \bar{b} \wedge w) \vee (\bar{a} \wedge b \wedge w) \vee (a \wedge \bar{b} \wedge w) \vee (a \wedge b \wedge \bar{w})$

complement the above:

$\Phi_{AND} = (a \vee b \vee \bar{w}) \wedge (a \vee \bar{b} \vee \bar{w}) \wedge (\bar{a} \vee b \vee \bar{w}) \wedge (\bar{a} \vee \bar{b} \vee w)$

CNF!

$\Phi = \Phi_{AND} \wedge \Phi_{NAND} \wedge \Phi_{OR} \wedge (z = 1)$

need 3 literals for CNF!

$z = 1 \equiv (z \vee k \vee l) \wedge (z \vee \bar{k} \vee l) \wedge (z \vee k \vee \bar{l}) \wedge (z \vee \bar{k} \vee \bar{l})$

(Introduce pseudo variables k, l)

$\Rightarrow \Phi = \Phi_{AND} \wedge \Phi_{NAND} \wedge \Phi_{OR} \wedge (\quad) \wedge (\quad) \wedge (\quad) \wedge (\quad)$

We claim:

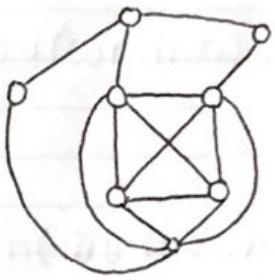
circuit is satisfiable $\longleftrightarrow$ Φ has a satisfying assignment

True by construction!

The reduction was polynomial in time: $\forall$ circuit gate, I introduced a constant # of clauses.

21-3 Clique

clique: a complete graph on V vertices



k=5?

Decision Problem: Does G have a clique of size k?

Application: social network, chemistry, networks, computational biology.

Greedy algos/heuristics: good to approximate sub-optimal solns to NPC problems in poly-time.

1992: Approximating clique is NPC!

Proof:

(a) Clique $\in$ NP

Just check the vertices if they are pairwise connected (takes poly-time).

(b) 3-CNF-SAT $\leq_p$ CLIQUE

assign 1 1

$\Phi = (\bar{x}_1 \cup x_2 \cup x_3) \wedge (x_1 \cup \bar{x}_2 \cup \bar{x}_3) \wedge (x_1 \cup x_2 \cup x_3)$

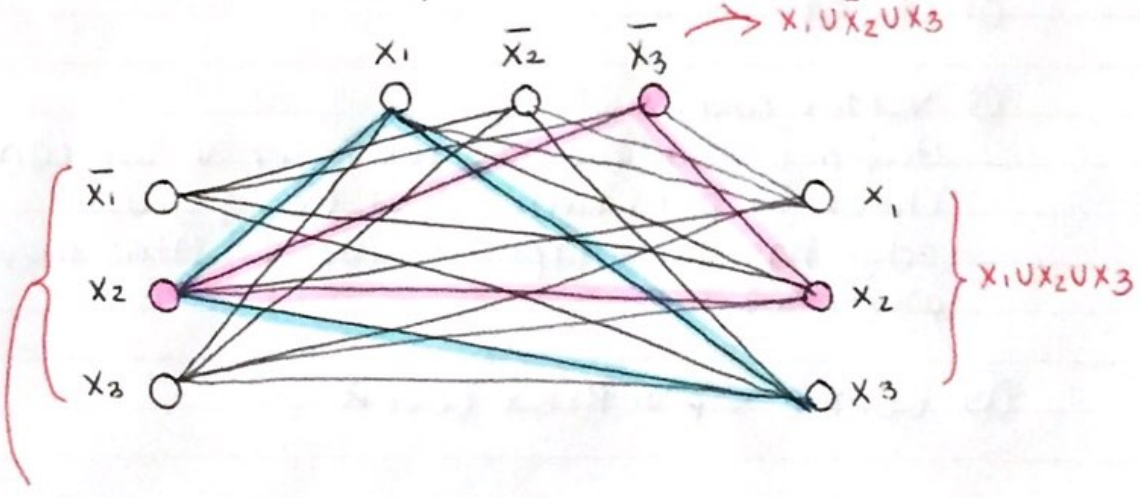
$x_3 = 0 \rightarrow \bar{x}_3 = 1$

We need to build a graph that

Φ is satisfiable iff G has a clique of a certain size

→ introduce triplets of vertices
→ every triplet corresponds to a different claus of formula Φ
→ connect vertices that are not complements of each other wl a bar (e.g. cannot connect $\bar{x}_1$ & x_1 but can connect x_1 & x_1)

complements



$\bar{x}_1 \cup x_2 \cup x_3$

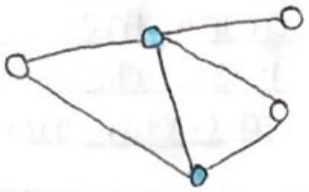
claim: Φ has a satisfying assignment $\leftrightarrow$ G we made has a clique equal to # of Φ clauses

$x_1 = 1 = x_2 = x_3$ satisfies Φ!

For every clause we introduced a polynomial # of vertices & edges, hence the reduction is of polynomial time.

21-4 Vertex Cover

Vertex Cover: a subset $V' \subseteq V$ of G that if $(u,v) \in E$ then at least one vertex exists in cover V'



• → example of vertices that make a vertex cover

Optimization Problem:
Find the minimum vertex cover

Decision Problem: Does G have a vertex cover of size k?

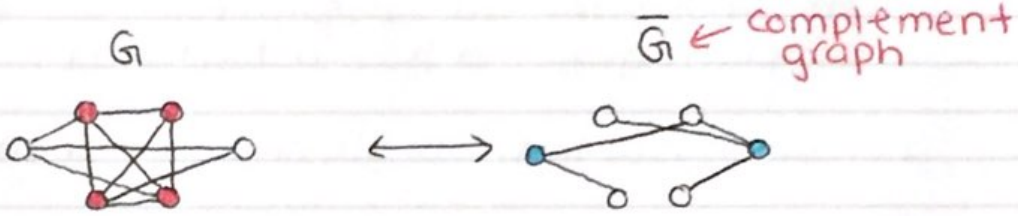
(a) Vertex cover $\in$ NP
Given a candidate vertex cover V' we can check in poly time that every graph edge has an endpoint into V' (this takes polytime)

(b) CLIQUE $\leq_p$ VERTEX COVER

Given a graph G to find a clique of size k

$\bar{G}$

(G ANOTHER) that has a vertex cover of size ?



→ clique of size 4
→ vertices that are not in clique are in vertex cover

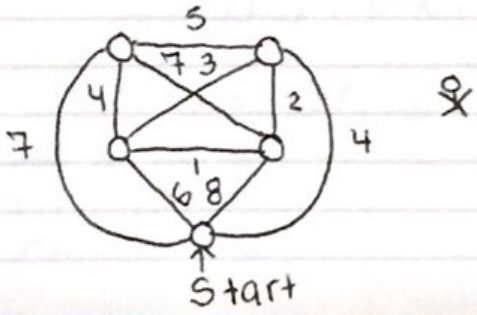
→: If red vertices are a clique in G then the white G vertices are a vertex cover in the complement graph

←: claim that if blue vertices form a vertex cover then remaining vertices are a clique

Transformation takes poly-time.

21-5 Travelling Salesman Problem

TSP: Given a complete positively weighted G find a HAMCYCLE of minimum weight.



→ vertices are cities
→ edge weights are mileage
→ salesman needs to start from beginning & traverse all cities only once by doing minimum mileage possible
→ complete graph → has a HAMCYCLE

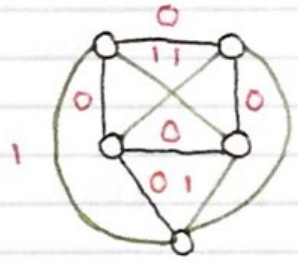
Decision problem: Is there a TSP of weight k?

ⓐ TSP ∈ NP

ⓑ HAMCYCLE $\leq_p$ TSP

(assume HAMCYCLE is NPC)

Given a graph G to find a Ham Cycle, add the remaining edges to create a complete graph



Then assign weight $w(u,v)$ ∀ edge (u,v)

$$w=(u,v)=\begin{cases}0 & \text{if } (u,v)\in G \\ 1 & \text{if } (u,v)\notin G\end{cases}$$

G has a HAM CYCLE
↕
G_{NEW} has a TSP = ∅
(value ∅ b/c uses edges in HAM CYCLE of original graph)