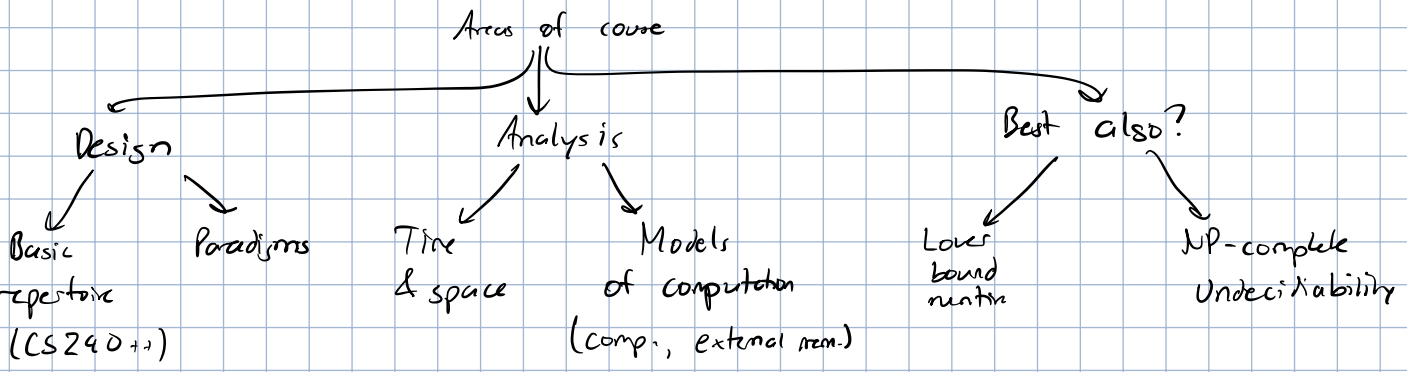
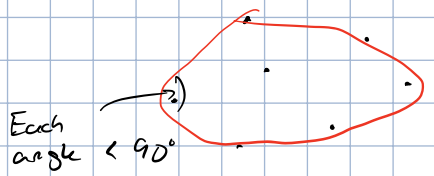
Intro



Convex Hull

Problem: we have n pts $\Rightarrow$ Subset of n pts. s.t. it contains all points in set in minimum.



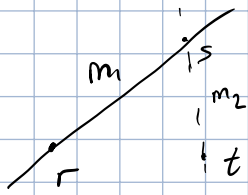
$\Rightarrow$ Rubber band to contain

- Mathematically: Polygon with line ℓ. Each ℓ goes through 2 pts. + only 1 side has pts. to its sides.

Method #1: Brute Force

for all pairs of pts r, s $\Rightarrow O(n^2)$
  draw a line from r to s
  for all other pts t: $\Rightarrow O(n)$
    $\rightarrow$ Check if on both sides
    if not $\rightarrow$ convex hull

$O(n^3)$



$$m_1 = \frac{x_2 - x_1}{y_2 - y_1}$$

$$m_2 = \frac{x_3 - x_1}{y_3 - y_1}$$

If $m_1 = m_2 \Rightarrow$ On same line
O.w. $\Rightarrow$ on one side:

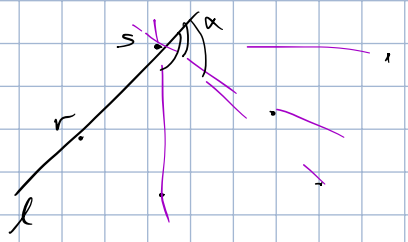
$$\frac{x_2 - x_1}{y_2 - y_1} \lessgtr \frac{x_3 - x_1}{y_3 - y_1}$$

$$S = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$$

S { $0 \Rightarrow$ same line; $<0 \Rightarrow$ one side; $>0 \Rightarrow$ other side }

Check if S changes sign!

Method #2: Jarvis' March



What is next line to draw?
↳ Line to draw will minimize α

In 1 iter ⇒ finding min. α is $O(n)$
How many times to iter? ⇒ # of convex hull pts

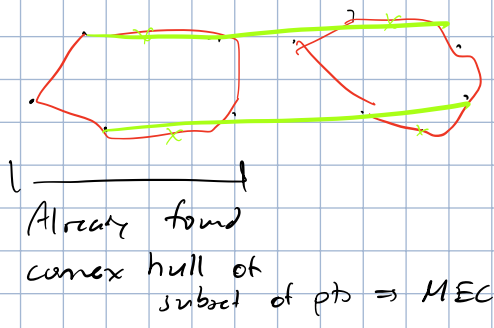
} $O(nh)$ → output dependent

Method #3: Reduction

1. Sort pts by x-coordinate ⇒ $O(n \log n)$
2. Find upper & lower convex ⇒ $O(n)$
3. Go through all possible ℓ + check if $\alpha < 90°$ ⇒ $O(n)$

} $O(n \log n)$

Method #4: Divide & Conquer



⇒ Combine?

1. Divide pts into halves
2. Create convex hulls
3. Find + upper + lower hulls
4. Join

Step 1: median finding ⇒ $O(n)$

Recursion:

$$T(n) = 2T(n/2) + O(n)$$

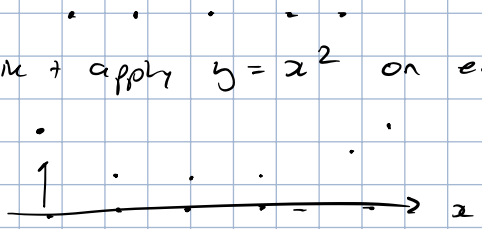
↳ $T(n) \in O(n \log n)$

Best also?

"Proof" that we can't do better

Assume algo exists $< O(n \log n)$ ⇒ Do smth. impossible. Sorting!

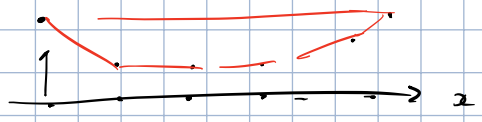
1. n pts.



2. Number line + apply $y = x^2$ on each

3. Convex hull

⇒ $< O(n \log n)$



4. Output hull $O(n)$

Entire algo $< O(n \log n)$

Why "proof"

↳ Sorting cannot do better than $O(n \log n)$ ⇔ only using comparisons.

↳ We did computation + comparison ⇒ not same comp. model ⇒ not defn. proof

Best: $O(n \log h)$

ANALYZING ALGORITHMS

Models of comp.

- Defn: defines fundamental op. of an algo ⇒ time & space compl.

1. Pseudo-code

- Issue: integer size is not taken in acct

```
fun fib(n):
    i, j = 0, 1
    for k 1 to n do
        i, j := j, i+j
    return j
```

} Looks like $O(n)$
∴ If $i + j$ big ⇒ long time.

2. Bit cost

Improve by looking at bit cost:

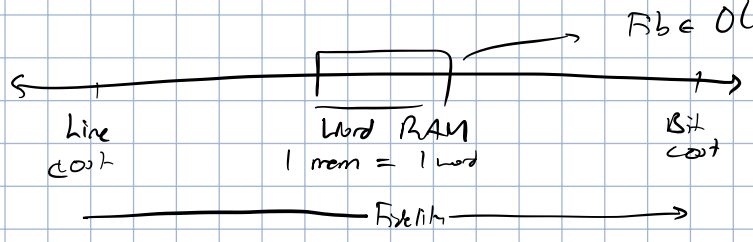
Size of $a = \lfloor \log a \rfloor + 1$

$\in \Theta(\log a)$

∴ Multiplying #s: $O(\log(a) \log(b))$

3. RAM model

Defn: access mem. takes 1 time unit



} Uni. Word RAM

Note: worst-case time is most common

Only compare $\Theta(f_1)$ vs. $\Theta(f_2)$, never do $O(f_1)$ vs. $O(f_2)$

In terms of multiple variables

ALGORITHM PARADIGMS

Reduction

Use a known algo to solve a new problem.

Ex:// 2 sum

① Brute force:

Look at all pairs $\Rightarrow O(n^2)$

② Reduce to sort & binary search

1) Sort $\Rightarrow O(n \log n)$
2) For each $i \Rightarrow$ binary search for $m - A[i] \Rightarrow O(n \log n)$

$O(n \log n)$

③ Improve 2nd part via 2 pointer approach

2) $\Rightarrow O(n)$

Ex:// 3 sum

Reduce to 2 sum problem:

1) Sort $\Rightarrow O(n \log n)$
2) For each $i \Rightarrow$ call 2 sum to find $A[j] + A[k] = m - A[i]$ (change) } $O(n^2)$
   - only do 2nd part of 2 sum

$\therefore O(n^2)$

Modified 2 sum a bit to reduce work of sorting (do it once)

You can change underlying algo calling → faster.

Divide & Conquer

1) Divide into smaller problems.
2) Recurse: solve smaller //
3) Conquer: combine smaller problems.

Analysis: recurrence relations:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + cn^k$$

- a: # of divide
- $T\left(\frac{n}{b}\right)$: Amount of recursive work (size of small. prob)
- cn^k: work of conquer

Solving recurrence:

Method #1: Recurrence tree

1) First level → divide into children → conquer
2) Sum up the work for each level
3) For the # of levels:

$$\left(\frac{n}{b}\right)^i = \text{base case.}$$

(i = # of level)

Solve for i

4) Find total work of last level: (opt.)

Work of last level = # of nodes × $\frac{\text{work}}{\text{node}}$ (Base case)

$$= a^i \times \frac{\text{work}}{\text{node}}$$

5) Sum of each level:

3) $\sum_{i=0}^{i}$ (level i work) = (path)

For ease: assume n < power of a ⇒ do analysis on a

∴ $n < n'$ where n' is smallest pow of 2 above n & $n' < 2n$

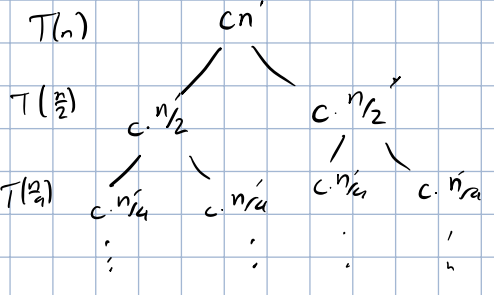
↳ Works: $\underline{\underline{O}}(\ldots)$

Ex:

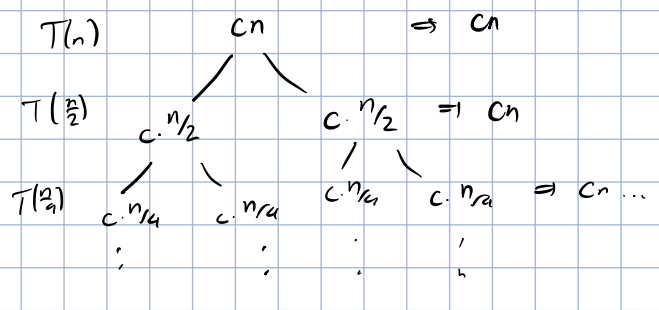
$$T(n) = \begin{cases} 2T(n/2) + cn, & n > 1 \\ 1, & n = 1 \end{cases}$$

Assume $n < n'$, n' is the smallest pow. of 2 greater than n

① Tree:



② Sum each level



$\therefore T(n) = \#$ of levels $\cdot\ cn$

③ Find # of levels:

$$\frac{n}{2^i} = 1$$

$$n = 2^i$$

$$i = \log n$$

④ Find total:

$$T(n) = \log n \ cn$$

$$T(n) \in O(n \log n)$$

⑤ Count to n:

$$T(n) \leq n' \log n'$$
$$\leq 2n \log 2n$$
$$\in T(n \log n)$$

Method # 2: Induction:

Given a runtime ⇒ prove

Ex:// $$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil \frac{n}{2} \rceil) + n - 1, & n > 1 \\ 0, & n = 1 \end{cases}$$

Guess that $T(n) \in O(n \log n)$.

① Claim:

$$T(n) \in O(n \log n) \iff \exists c, \forall n > 1, \ T(n) \leq cn \log n$$

② Base case:

$n = 1$: $T(1) = 0 \leq c \cdot \log(1)$

$0 \leq 0$ ✓

③ Ind. hyp:

Assume $T(n') \leq cn' \log n' \quad \forall n' < n, \ n \geq 2$

④ Ind. step:

Since ceil + floor ⇒ even + odd analysis

Even: $T(n) = 2T\left(\frac{n}{2}\right) + n - 1$

$\leq c\frac{n}{2}\log\left(\frac{n}{2}\right) + n - 1$ ✓ By ind. hyp.

$c\frac{n}{2}\log(n/2) = \frac{cn}{2}\cdot\log n - \frac{c}{2}\log 2$
$= \frac{cn}{2}\cdot\log n - \frac{cn}{2}$
$= cn\log n - cn$

$\Rightarrow \leq cn\log n - cn + n - 1$

we want $-cn + n - 1 \leq 0$, find c

$\therefore c = 1$

$\therefore T(n) \leq cn\log n \quad \forall n$ if $c \geq 1$

Odd:

$$T(n) = T\left(\frac{n-1}{2}\right) + T\left(\frac{n+1}{2}\right) + n - 1 \Leftarrow \left.\begin{array}{l}\lfloor\frac{n}{2}\rfloor = \frac{n-1}{2}\\ \lceil\frac{n}{2}\rceil = \frac{n+1}{2}\end{array}\right\} \text{If } n \text{ is odd}$$

$$\vdots$$

$$\leq cn\log n + \underbrace{\left(1 - \frac{c}{2}\right)(n-1)}_{<0}$$

True if $c \geq 2$

Note: ind. hyp must look exactly to the claim, inc. constant!

↳ What if we can't? ⇒ Prove something stronger

Method #3: Master's Theorem

Let $T(n) = aT\left(\frac{n}{b}\right) + cn^k$, $a \geq 1, b > 1, c > 0, k \geq 0$

$$\therefore T(n) \in \begin{cases} \Theta(n^k) & \text{if } a < b^k \Rightarrow \text{Non-recursive work dominates } (cn^k) \\ \Theta(n^k \log n) & \text{if } a = b^k \Rightarrow \text{Each level has same amt. of work} \\ \Theta(n^{\log_b a}) & \text{if } a > b^k \Rightarrow \text{Work} \uparrow \text{ as \# of levels} \uparrow \end{cases}$$

Intuition:

① Recurrence tree

$T(n)$ — cn^k $\Rightarrow cn^k = cn^k\left(\frac{a}{b^k}\right)^0$

$c\left(\frac{n}{b}\right)^k$ $c\left(\frac{n}{b}\right)^k$ ... $\Rightarrow ca\left(\frac{n}{b}\right)^k = cn^k\left(\frac{a}{b^k}\right)^1$

— — — — —

② # of levels:

$$\frac{n}{b^i} = 1$$

$$b^i = n$$

$$i = \log_b n$$

③ Work on bottom:

$a^{\log_b n} T(1)$ $\Rightarrow$ # of nodes $= \left(\frac{\text{\# of children}}{\text{node}}\right)^{\text{\# of levels}}$

④ Sum of work on all levels

$$\sum_{i=0}^{\log_b n - 1} cn^k \left(\frac{a}{b^k}\right)^i$$

$$\therefore T(n) = a^{\log_b n} T(1) + \sum_{i=0}^{\log_b n - 1} cn^k \left(\frac{a}{b^k}\right)^i$$

Case 1: $a < b^k \Leftrightarrow \log_b a < k$

$\frac{a}{b^k} < 1 \Rightarrow cn^k \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^k}\right)^i$ is a geom. series w/ $r < 1 \Rightarrow O(1)$

$$\therefore T(n) = n^{\log_b a} T(1)$$
$$\leq n^k$$
$$\therefore T(n) \in \Theta(n^k)$$

Case 2: $a = b^k \Leftrightarrow \log_b a = k$

$$cn^k \sum_{i=0}^{\log_b n - 1} 1^i \in \Theta(\log_b n \cdot n^k) \quad \log_b n \in O(\log n)$$
$$\in \Theta(n^k \cdot \log n)$$

$$T(n) = n^{\log_b a} T(1) + \Theta(n^k \cdot \log n)$$
$$\in \Theta(n^k \cdot \log n)$$

Case 3: $a > b^k$

$$\sum_{i=0}^{u-1} x^i \in \Theta(x^u) \quad \text{if } x > 1$$

$$\therefore n^k \sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^k}\right)^i \in \Theta\left(\left[\frac{a}{b^k}\right]^{\log_b n}\right)$$
$$\in \Theta\left(n^k \cdot n^{\log_b a} \cdot \frac{1}{(b^{\log_b n})^k}\right)$$
$$\in \Theta(n^{\log_b a})$$

$$T(n) = n^{\log_b a} \cdot T(1) + \Theta(n^{\log_b a})$$
$$\in \Theta(n^{\log_b a})$$

Ex:// Counting inversions:

	1	2	3	4
I like	B	D	C	A
You like	A	D	B	C

# of inversions = # of pairs in reverse order

= # of crossings

= 4

#1: Brute force

Check all pairs ⇒ crossings. $O\binom{n}{2} = O(n^2)$

#2: Divide and conquer, more

Input: $A = [1 \dots n]$, permutation

↳ Someone else chose

$[1, 2, 3, 4] \Rightarrow [3, 2, 1, 4]$

1. Divide into 2 ($A + B$)
2. Get inversions
3. Combine:

Must fire. ⇒ simplify

For each element in B, # of greater elements in A

$O(n^2)$

$T(n) = 2T(n/2) + O(n^2) \in O(n^2)$ no gain

#3: D&C, optimize

We don't care about order in A or B.

$[3, 2, 1, 4]$

$[3, 2]$ $[1, 4]$

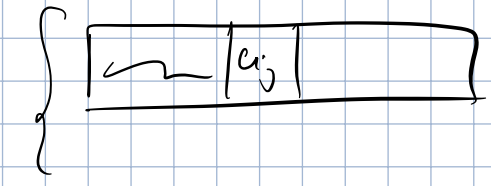
A: +1 +1

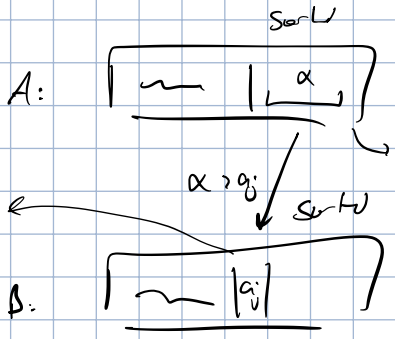
Very useful

Maybe sorting can help.

Merge sort:

$O(n)$





# of elem > a_{ij}

⇓

Every time elem in B is moved ⇒ count # of α

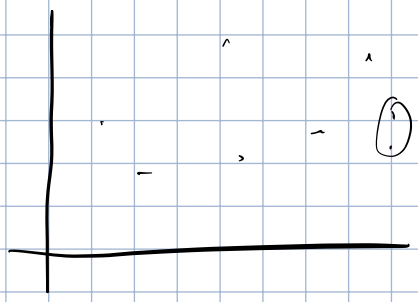
$\therefore T(n) = 2T\left(\frac{n}{2}\right) + O(n) \in O(n \log n)$

Also:

```
Wrapper(A):
    sort(A)
    recur(A)
```

```
recur:
    base case: ____
    divide A
    divide B
    merge(A, B)
    cont →
```

Ex.// Closest pair of points



$\arg\min_{p,q} d(p,q)$

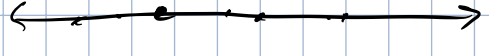
$(x_p - x_q)^2 + (y_p - y_q)^2$

#1: Brute force

Calculate dist for each pair ⇒ take min

$O(n^2)$

#2: Simplification to 1D

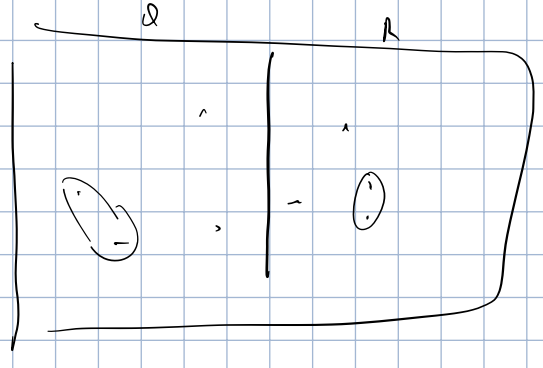


Check consec. pairs ⇒ take min

Why? It's sorted!

#3: D & C

1. Divide: put in left Q, right R
2. Recursive: find closest pairs in Q, R
3. Combine:

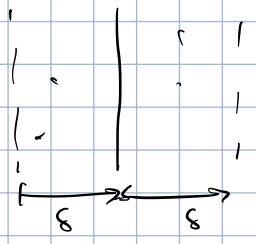


⇒ Only way to get better pair is

1 point in Q, 1 point in R

AND $d_{q,r} < \min d(Q,R)$

a) Find all points in $\min(d(Q), d(R))$ from middle

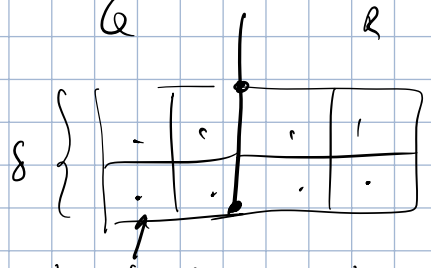


To remain $O(n \log n)$ → presort in combine

↳ b) Reduce to 1D case ⇒ sort by y + get close pairs.

↳ c: Time complexity

Claim: we only need to compare 7 points ahead.



# of pts in each square ≤ 1

If we have 2 ⇒ pair $< \delta$ (contr.)

Compare 8 points

1 point ⇒ $O(1)$ comp. ⇒ $O(n)$ for entire comb.

Ex:// Multiplying multi-precision nts.

Large num mult ⇒ bits must be in account

# of bits of $a \in O(\log a)$

#1: School method

$O(nm)$ (each digit needs mult.)

#2: Naive D & C

1. Pad each number

$$\begin{array}{r} 0034 \\ \times\ 2192 \end{array}$$

2. Divide into smaller problem

$$\begin{array}{r} 00\,|\,34 \\ \times\ 21\,|\,92 \end{array}$$

a b

3. Shift soln by $10^{2 \cdots}$, add

$$T(n) = 4T(n/2) + O(n) \in O(n^2)$$

#3: Karatsuba's Algo

$$\underset{w}{9}|\underset{x}{81} \times \underset{y}{12}|\underset{z}{34}$$

$$(10^2 w + x)\cdot(10^2 y + z) = 10^4 wy + 10^2(wz + xy) + xz$$

4 sub problem

Really 800 start.

Q: reduce # of subproblems?

$$(w+x)(y+z) = wy + (wz + xy) + xz$$

can form via oth

3 sub problems, one sub problem uses 2 others.

Also:

$p = wy$

$q = xz$

$r = (w+x)(y+z)$

return: $10^4 p + 10^2(r - p - q) + q$

$$T(n) = 3T(n/2) + O(n) \in O(n^{\log_2 3})$$

Extensions

1. Numbers of diff length

a has n digits, b has m digits

a) Break a into $O(n/m)$ blocks of m digits

b) Multiply each chunk w/ b => via Karatsuba

c) Add up all products

$$O((n/m)\, m^{\log_2 3}) = O(n m^{\log_2 3 - 1})$$

Ex:// Multiplying matrices

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} c & e \\ f & g \end{bmatrix}$$

#1: Naive brute force

$O(n^3)$

#2: Divide & conquer

$A_{11}B_{11} + A_{12}B_{21}$ ← 1 subproblem

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

# of mult ⇒ 8 mat matrix multiplies.

$$T(n) = 8T(n/2) + O(n^2) \in O(n^3)$$

#3: Repeated subproblems

$$T(n) = 7T(n/2) + O(n^2) \in O(n^{\log_2 7})$$

Very common algo ⇒ many problems reduce to matrix mult.

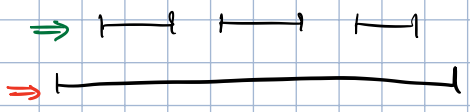
Greedy Algorithms

- Situation: minimize / optimize ⇒ greedy choice (simple rule)
- Ex:// Interval sched.

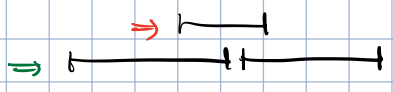
P: Set of activities ⇒ 1 activity / time. Max set of disjoint activity.

① Greedy rule ⇒ try on small example

1. Select earliest activity



2. Select shortest activity



3. Select activity w/ fewest conflicts

4. Select activity that ends earliest

② Pseudo code

```
Sort array by end time
A := ∅
for i from 1 → n:
    if Arr[i] doesn't overlap w/ last activity in A:
        A.append(Arr[i])
return A
```

} O(n log n)

③ Proof

Proof techniques:

① "Stays ahead"

Prem: short-term greedy → extend to optimal.

a) Set up optimal & greedy solns:

A(greedy): $a_1, \ldots, a_k$

B(optimal): $b_1, \ldots, b_k, b_{k+1}, \ldots, b_\ell$ } Sorted by end time.

b) Claim that subset $a_i \Leftrightarrow b_i$ is optimal (not sure).

$a_1, b_2, \ldots, b_k \ldots b_\ell$

This is valid:

$\text{end}(a_1) \leq \text{end}(b_2) \Rightarrow$ greedy algo would have chose b_2

↳ Disjoint

This is an optimal soln ⇒ ℓ has not changed.

↳ By induct

$a_1, \ldots a_k, b_{k+1}, \ldots b_\ell$

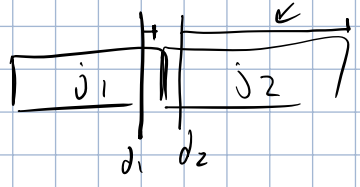
c) Optimal soln length = greedy length

Contr.: If $\ell > k \Rightarrow$ more intervals. Greedy should have chosen.

$\therefore \underline{\ell = k}$

② Exchange proof

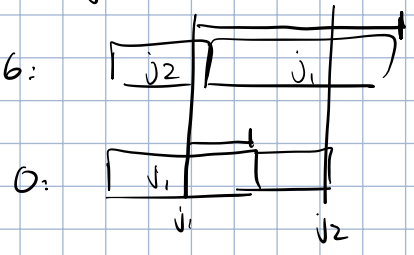
Ex:// Jobs w/ deadlines & time to complete. Schedule s.t. minimizing max late



① Greedy approaches:

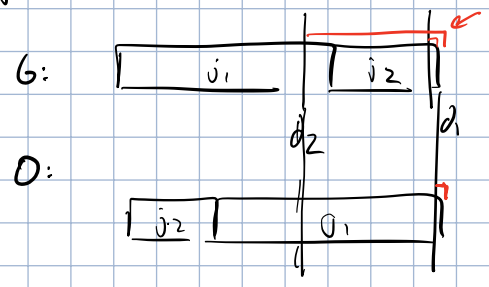
No waste of time ⇒ continuous.

1. Shortest job first



✗

2. Do jobs w/ less slack first $(d_i - t_i)$



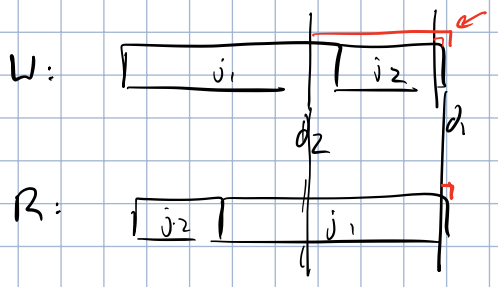
✗

3. Jobs in order of deadline. ✓

① Pseudocode

② Proof:

a) Think of small case:



$\ell_R(2) \leq \ell_W(2)$

↳ Why: 2 is schedule earlier

$\ell_R(1) \leq \ell_W(2)$

↳ Why: $j_1 - d_1 \leq j_2 - d_2$

$\therefore \quad \ell_W \geq \ell_R$

b) Generalize

Greedy ordering: $1 \dots n, \Rightarrow d_1 \leq d_2 \leq \dots$

Optimal ordering: Assume not matching greedy.

$\exists\ i, j$ s.t. $d_i \geq d_j$, $i \leq j$

Exchange proof

Claim: switching i w/ j will lead to a better soln.

Optimal → greedy

↳ "Sorting" array ⇒ finite

↳ Proof: Use the same proof as above

$\therefore$ Contradiction ⇒ it's not an optimal soln ⇒ $\boxed{\ell_G = \ell_O}$

Dynamic Programming

Overlapping subproblems ⟶ combine into an optimal solution

Solve these

Ex:// Text segmentation:

Input: $A[1 \ldots n] = [a, c, \ldots]$

$W[i, j] = \begin{cases} T \\ F \end{cases}$ ⇒ if word

Prob.: is it possible to split A into words?

DP approach:

① Identify sub-problems

$split(k) = \begin{cases} T & \text{if } A[1 \ldots k] \text{ is splittable} \\ F & \text{o.w.} \end{cases}$

We want $split(n)$

② Combine subproblems:

a) $k \to 0, n$

b) Set $split[k] := F$

c) Try to split all indices in $F \Rightarrow T$

③ Base case:

$split[0] := T$

0..0

④ Also:

```
split[0] = T
for k from 1 → n:
    split[k] = F
    if j from 1 → k-1:
        if split[j] & word[j+1, k]:
            split[k] = T
```

$O(n^2)$

Ex:// Longest increasing subseq.

Input: A of nums

Prblm: longest subseq of numbs that is increasing.

① Subproblems:

Fnd longest subsequence of all elemts prior i

Take max of it OR + 1 if elem < i

② Also:

$dp[n] = [1, \ldots, \overrightarrow{\ }]$ subseq. of 1 ∞ BC

```
fr i frm 0 → n:
    fr j from 0 → i:
        if A[j] < A[i]:        (building on)
            dp[i] = max(dp[i], 1 + dp[j])
        elx:                   (isnt / taking subseq.)
            dp[i] = max(dp[i], dp[j])
retn dp[n]
```

$O(n^2)$

Ex:// Longest common subseq.

Gvn 2 strs ⇒ find longest substring common subseq.

① Subproblems:

$dp[n][m] \Rightarrow$ final

$$dp[i][j] = \max \begin{cases} dp[i-1][j-1] + 1 & \text{if } A[i] = B[j] \\ dp[i][j-1] \\ dp[i-1][j] \end{cases} \text{ if } A[i] \neq B[j]$$

(i ↑, j →)

② Base cases:

$dp[i][0] = 0$

$dp[0][j] = 0$

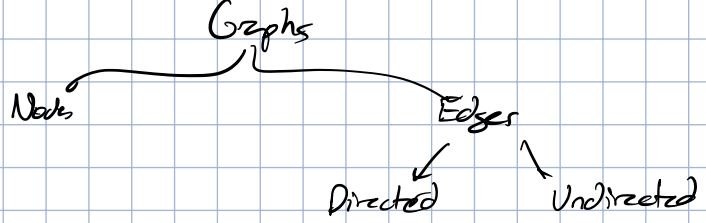
Tips:

1. Think about subproblems
2. Inclusion / exclusion

3. Reframe problem dep. on conditions.

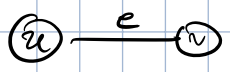
GRAPH ALGORITHMS

Intro



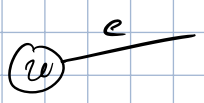
Terminology:

- Adjacency:



$\Rightarrow u$ is adj. to v

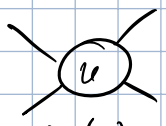
- Incidence:



$\Rightarrow e$ is incident to u

- Degree:

Undirected:

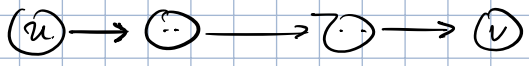


$deg(u) = 4$

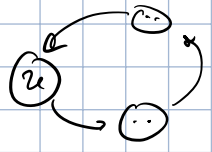
Directed:

indegree = 3
outdegree = 2

- Path:

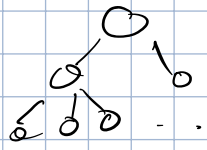


- Cycle:



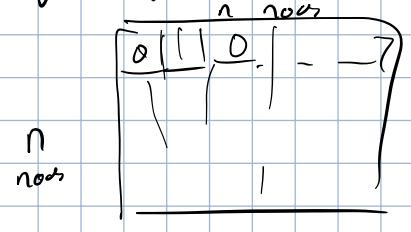
  - Simple cycle: cycle w/ no cycles in it

- Tree: graph w/ no cycle:



Programmatic Representation:

Adjacency Matrix:



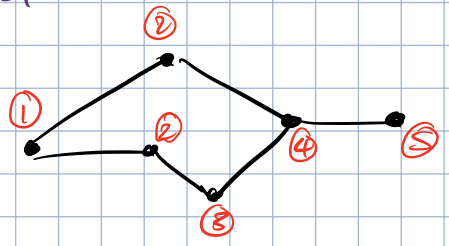
$\Rightarrow$ matrix $[i][j] = \begin{cases} 0 \Rightarrow i \text{ \& } j \text{ not connected} \\ 1 \Rightarrow i \text{ \& } j \text{ are connected.} \end{cases}$

Adjacency List:

{
0: [A, B, C, D] → ∅ ⇒ connected to A-D
⋮
n:
}

⇒ Primary method of Graph rep.

BFS Traversal



"Level" traversal

Implementation: Queue

↳ Also:

```
explore(v):
    for each rei u in v:
        if u is undiscovered:
            u is discovered
            add u to queue.
```

```
bfs(G):
    all vertices → undiscovered   ← prevent cycles
    pick v0 → queue.
    ↳ v0 is discovered.
    while que is not empty:
        v := remove from que.
        explore(v)
```

Time complexity:

Go through all nodes
↳ For each node, process all edges } $O(n+m)$ (n: nodes, m: edges)

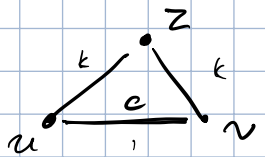
Lemma: length of shortest path from $v_0 \to v$ is level of vertex v

Ex:// P: Check if graph is bipartite

Main fact: Graph has an odd cycle ⇔ not bipartite.

Also:
1. BFS & put vertices in odd or even levels.
2. Not bipartite if edge between already discovered nodes:

Proof:
① Draw:



② Logic

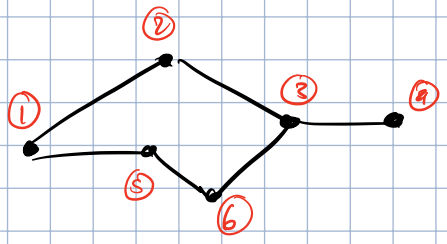
If u & v are in different levels (one is odd, other even), this is fine.

$\therefore u$ & v are in same level

Let length for $z \to u = z \to v = k$

$\therefore$ Cycle length $2k+1 \Rightarrow$ odd cycle $\to$ not bipartite.

DFS Traversal



Implementation: Stack

↳ Also:

```
explore(v):
  mark v to be discovered
  for nei u in v:
    if u is undiscovered:
      explore(u)
  mark v to be finished.
```

```
DFS(G):
  all verti. undiscovered.
  for v ∈ V:
    if v is undiscovered
      explore(v)
```

Runtime: $O(n+m)$

Lemma:

① DFS(v_0) will explore all vertices connected to v_0

② All exploration done w.r.t. ancestor

Addition: discover & finish times

```
explore(v):
  discover(v) = time
  time += 1
  for . . .
    . .
  finish(v) = time
  time += 1
```

Parenthesis system:

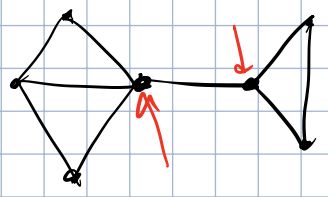
$$[\ [\ ]\ ]$$

$$d(v) < d(u) < f(v) < f(u)$$

Ex:// Find cut vertices in a graph

① Graph theory

What is a cut vertex: u is a cut vertex $\Leftrightarrow$ removing u would cause G to be disconnected.



Terminology:

Root: where you start exploring.

Non-root: $\neg$ (")

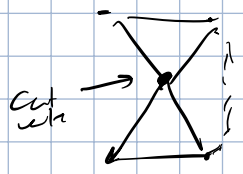
Claim #1: Root is a cut vertex $\Leftrightarrow$ > 1 child

↳ Proof:

$\Rightarrow$ Root is cut vertex. Remove $\Rightarrow$ disconnect graph $\Rightarrow$ 2 components $\therefore$ Root has at least 2 children.

$\Leftarrow$: Disconnects root $\Rightarrow$ root in 1 comp., children in another.

Claim #2: Non-root vertex v is cut vertex $\Leftrightarrow$ some subtrees of v have edge going to ancestor of v.



Proof:

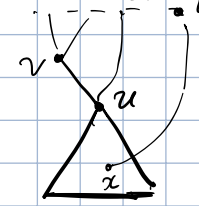
$\Rightarrow$: Assume v is cut vertex. By defn., subtree disconnects if v removed $\Rightarrow$ no edge b/w subtree & ancestor.

$\Leftarrow$: No subtree connected to ancestor. v can remove $\Rightarrow$ 2 comp.

② Also

Q: How to find if subtree is connected to ancestor?

Tip: Use discovery & finish times!! Helpful in cross-level analysis.



Analyze v $\Rightarrow$ looks at child u

Look at subtree at u & all descendants x

We know x connects w/ ancestor of v if if connects w/ a node w w/ $d(w) < d(v)$.

∴ Def:

$$low(u) = \begin{cases} \min\{d(w) : x \text{ is desc. of } u, (x,w) \text{ is edge}\} \\ \min\{d(w) : (u,w) \text{ is an edge}\} \end{cases}$$

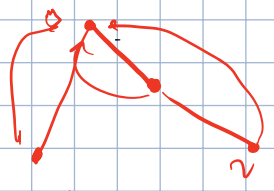
If $low(u) \leq d(v) \Rightarrow$ not a cut vertex.

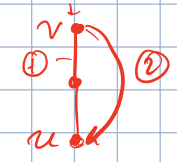
Modify DFS to calculate $low(u)$ at each level

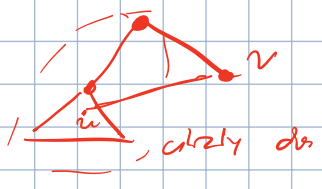
DFS on Directed Graphs

Obj: Use DFS to characterize edges

```
explore (v):
  mark (v) := discovered
  d(v) := time , time += 1
  for u ∈ AdjList (v):
    if mark(u) := undiscovered:
      explore (u)
    else:
      if mark(u) not finished
        (v, u) is a back edge
      else if d(u) > d(v)
        (v, u) is a forward edge
      else
        (v, u) is a cross edge
```







No change in DFS also

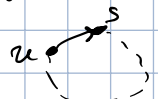
Ex:// Detect directed cycle

① Graph theory:

Claim: Graph has back edge $\Leftrightarrow$ Graph has cycle

Proof:

$\Leftarrow$: Assume graph has cycle.



$\Rightarrow$ Start at $s \to u \to s$, not finished!

$\therefore$ Back edge

⇒ Assume has back edge:

Name: $v_1 - \cdots - v_k$. v_k has back edge to v_1.

By char. of back edge ⇒ v_k will explore v_1 before finish → cycle.

② Also:

DFS → output true if back edge detected.

Ex:// Topological sort of acyclic graph

Obj: give sorting of vertices in order of edges

① Graph Theory

Claim: if edge (u, v), $\text{finish}(u) > \text{finish}(v) \Rightarrow u \to v$

↳ Case 1: u discover b/f v:

① u will explore v

② By DFS, u will finish only if v is finish.

↳ Case 2: v discover b/f u:

⇒ No cycle → v cannot link to u.

② Also:

① Get finish times of vertices

② Give vertices in order of finish time (hi → lo)

Minimum Spanning Trees

Obj: spanning tree (all vertices in G) where $\min\left(\sum_{i=1}^{m} w_i\right)$, $n-1$ edges

① Kruskal's Algorithm

Idea:

order edges by weight

$T = \emptyset$

for i from $1 \to m$: ← Check if $\text{find}(u) \neq \text{find}(v)$

if e_i does not create cycle:

$T = T \cup \{e_i\}$ → union (set of u, set of v)

Proof: exchange arg.

MST T, w/ edge $e_1 \cdots e_{n-1}$. Induction on e_i, show that MST matching T on first $e_1 \ldots e_i$ edges.

BC: $i = 0 \Rightarrow$ trivial

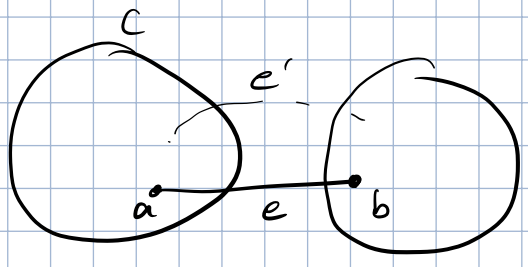
IH: Assume MST M matching T on $i-1$ edges.

IS:

als T: $t_1 \ldots t_{i-1}$ t_i

"optimal" M: $m_1 \ldots m_{i-1}$ m_i (=)?

Let $t_i = e = (a, b)$. Let C be component that contains a



Assume $m_i \neq t_i$. $m_i = e'$. Under our algo, $w(e) \leq w(e')$

Create a better MST: $M' = (M - \{e'\}) \cup \{e\}$

↳ Spanning: still connects all vertices

Minimum: $W(M') = W(M) - W(e') + w(e) \leq W(M) \Rightarrow M'$ is MST

Implementation:

① Sorting: $O(m \log m) \in O(m \log n)$

↳ $m \leq n^2 \Rightarrow \log m \leq 2 \log n \Rightarrow \log m \in O(\log n)$

② To prevent cycle, every edge added in algo must be connects diff. components.

Use union-find also: disjoint sets w/ following ops:

- Find $(x) \Rightarrow$ which set contains x
- Union $\Rightarrow$ unite two sets

} set will be components in our algo!

Implementation:

(A) Array $S \Rightarrow$ $S[i]$ gives component of i^{th} vertex

(B) Linked list of sets

Ex:

$S = [1, 2, 1, 2, 1, 1, 3]$ (indices 1 2 3 4 5 6 7)

$L = \begin{cases} C_1: 1, 3, 5, 6 \\ C_2: 2, 4 \\ C_3: 3 \end{cases}$

Operation impl:

Find(x): go to $S[x]$ $\Rightarrow$ $O(1)$ op.

Union():

1. Link two sets $\Rightarrow O(1)$
2. Go into S and rename nodes $\Rightarrow O(n)$
   - Only do this for smaller set

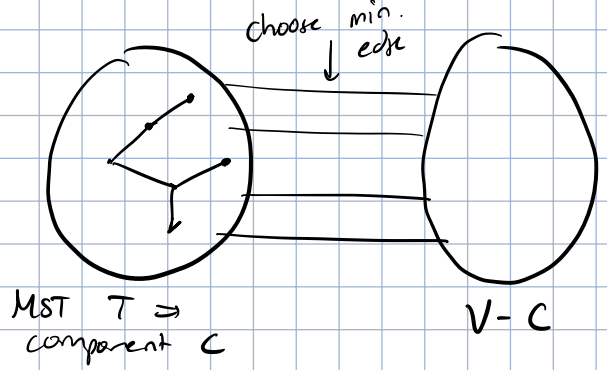
In Context:

Renaming happens $O(\log n)$ times $\Rightarrow$ union takes $O(n \log n)$ total

Total runtime: $O(m \log n) + O(m) + O(n \log n) \in O(m \log n)$

(sort) (finds) (union)

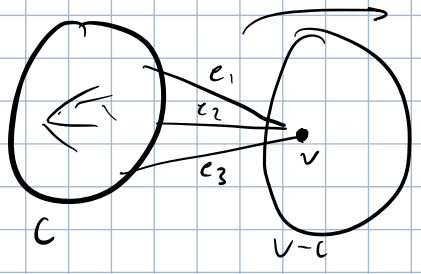
② Prim's Algorithm



Implementation:

① Create heap of vertices V-C.

(vertex, min weight of edge from $C \to v$)



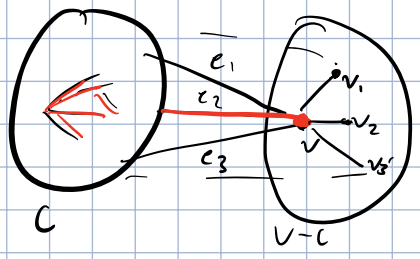
If v not connected $c \longrightarrow (v, \infty)$

② Implement ops:

1. ExtractMin (heap)
2. Insert (heap, v, weight)
3. Delete (heap, v) $\Rightarrow$ keep an array as record of which vertex is deleted.

} $O(\log n)$

② On selection of v in $V-C \Rightarrow$ change of weights of neighbors.



Why? v is now part of $C \Rightarrow$ consider direct edges from $V \to v_1 .. v_3$
& change heap
↳ Delete & insert

Also:

$C = \{s\}$

$T = \emptyset$

while $C \neq V$:

find min edge from $T \to V-C$ (u,v) s.t. $u \in C$, $v \in V-C$:

$C = C \cup \{v\}$

$T = T \cup \{e\}$

Time complexity:

$n-1$ times to find min edge ($n-1$ edges in tree)

Also need to recompute edge weight $\Rightarrow O(m)$

↳ Worst case: v connects to all vertices $\Rightarrow m$ computations!

Total cost: $O(m \log n)$

Shortest Path in Weighted Graph

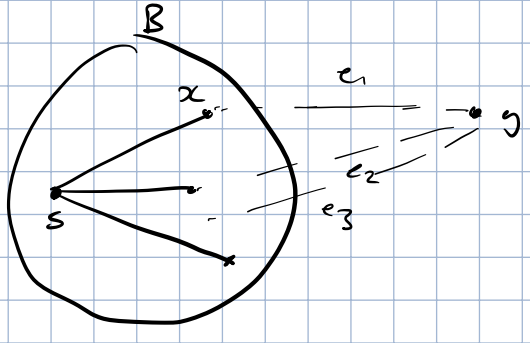
Shortest path from $s \to t$

Unweighted → BFS

Weighted → If non-neg. weights → Dijkstra's Also

Dijkstra's:

Idea:



To connect y w/ path:

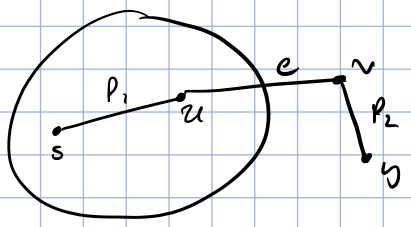
① For $x \in B$, $y \notin B$

② Minimize $d(s,x) + w(x,y)$ ($d(s,y)$)

③ Add into tree

Rules on claim: $d(s,y)$ is min distance from $s \to y$

↳ Proof:



Total path P:

$w(P) \geq w(P_1) + w(e)$ ⇒ Assume $w(P_2) \geq 0$

$\geq d(s,u) + w(u,v)$

$\geq d$

Implementation:

```
def djikstra (s, t):
    d(v) = ∞ ∀ v        ⇒ min path from s → v
    d(s) = 0
    B = ∅
    while |B| < n:
        y = vertex from V − B w/ min d from s (store in heap)
        B = B ∪ {y}
        for each edge (y, z)
            if d(y) + w(y, z) < d(z)
                d(z) = d(y) + w(y, z)
                Parent(z) = y
```

} Updates neighbors in V−B w/ smaller weights

Cost: $O(m \log n)$

Dynamic Programming for Shortest Paths

Shortest path in weighted graphs

- Given u & v ⇒ find shortest uv path ⊆ Given u, find shortest uv path ∀ v (single-source shortest paths)
  - No neg. weights in graph ⇒ Djikstras
  - No cycles ↓ $O(n+m)$ also
  - No neg. cycle ↓ Bellman-Ford $O(n \ln + m)$
- Find shortest uv path ∀ u, v — Floyd-Warshall $(O(n^3))$

Single-source shortest path — no cycles:

① Use topological sort

$u_1, u_2 \ldots u_n$ (if edge (directed) $(u_i, u_j) \Rightarrow u_i < u_j$)

Note: if v comes s ⇒ no path from $s \to v$

② Discard all nodes b/f s in topological sort (no path)

③ Also:

topological sort of G — ①

start at node $s \rightarrow v_1$ — ②

$d_i = \infty \ \forall i$

$d_1 = 0$

for i from $1 \rightarrow n$:

  for each edge (v_i, v_j):

    if $d_i + w(v_i, v_j) < d_j$

      $d_j = d_i + w(v_i, v_j)$

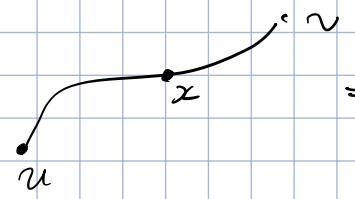
} Takes the min distance from $s \rightarrow d_j$

Runtime: $O(n + m)$

Single-source shortest path — no neg. weight cycle:

Also: Bellman-Ford

Core idea:



$\Rightarrow$ ux, xv is short then $uv \Rightarrow$ there can be subproblem.

We can use dynamic prog.!

Also:

① Let $d_i(v) \Rightarrow$ weight of shortest path from $s \rightarrow v$ using $\leq i$ edges

② Init:

$$d_1(v) = \begin{cases} 0 & \text{if } v = s \\ w(s,v) & \text{if } v \text{ connect to } s \\ \infty & \text{o.w.} \end{cases} \Rightarrow \text{do for all } v$$

③ Goal:

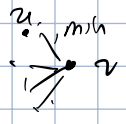
Find $d_{n-1}(v) \Rightarrow$ if n edges, we have cycle $\rightarrow$ drop 1 to create a path

④ Subproblems:

$$d_i(v) = \min \begin{cases} d_{i-1}(v) & \text{, no extra edge } i \text{ needed} \\ \min_u (d_{i-1}(u) + w(u,v)) & \end{cases}$$

↳ you need to know edges going to v

↳ u min



Final algo:

$d_1(v)$ $\forall v$ init

for i from $2 \to n-1$:

  for $v \in$ vertex set:

    $d_i(v) = d_{i-1}(v)$

    for each edge (u, v):

      $d_i(v) = \min(d_i, d_{i-1}(u) + w(u, v))$

$\to O(n(n+m))$

Simplified also:

$d(v) = \infty \ \forall v$ (upper bound of path length)

$d(s) = 0$ ← only need $n-1$ edges!

~~for~~ i ~~from~~ $1 \to n-1$:

  for $(u, v) \in E$:

    $d(v) = \min(d(v), d(u) + w(u, v))$

→ no use of i

Can detect negative weight cycles $\Rightarrow$ $d(v)$ will not change!

All pairs shortest path

Algo: Floyd-Warshall algo

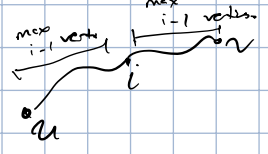
Restriction: no negative weight cycle

Define $D_i[u, v]$: length of shortest uv path using vertices $\{1, 2, \ldots i\}$

Goal: $D_n[u, v]$

Init:

$$D_0[u, v] = \begin{cases} 0 & \text{if } u = v \\ w(v, v) & \text{if } (u, v) \in E \\ \infty & \text{o.w.} \end{cases}$$



Recursion

$$D_i[u, v] = \min \begin{cases} D_{i-1}[u, i] + D_{i-1}[i, v], & \text{uses } i \\ D_{i-1}[u, v] & \text{don't use } i \end{cases}$$

Algo:

Init $D_0[u, v]$:

for i from $1 \to n$:

  for u from $1 \to n$

    for v from $1 \to n$:

      $D_i[u, v]$ as calc.

$\to O(n^3)$ time

$O(n^3) \longrightarrow O(n^2)$ space

P vs. NP

Exhaustive Search

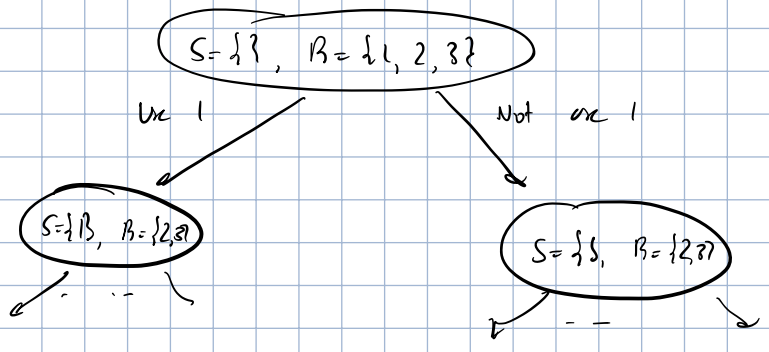
To solve hard problem

- Heuristics
  - +: quick
  - -: not accurate
- Approximate:
  - +: good enough
  - -: fine
- Exact soln:
  - + good start
  - - exponential

Back tracking: try all possibilities, if soln. exists

① Start w $S = \{\}$ (stole), $R = \{1, \dots\}$ (all elem.)

② For each R, put in S or not put in S & explore



Runtime: $O(2^n)$

↳ 2 choices per move

↳ up to n moves in height of tree

Ex// Subset sum problem

Q: Given $R = \{1, \dots, n\}$ (items) w/ weight, find item to include to get W

Also:

```
def backtrack(S, R, w, W):        # w: sum so far, W: limit
    if w == W: ret. success
    if no more elem to add (R = ∅), or w > W or sum_{i=0}^{|R|} w[i] < W
        ↳ ret. fail
    // Include i:
    if backtrack(S ∪ {i}, R / {i}, w + w[i], W) ⇒ ret. succ.
    // Not include i
    if backtrack(S, R / {i}, w + w[i], W) ⇒ ret. succ.
    retn fail
```

Better than DP if $W >> n$

Branch & bound: exhaustive optimization

↳ Also: $A :=$ set of configs.

$opt := \infty$ (minimizing)

while $A \neq \emptyset$:

C = remove most promising config

for all child configs of C (C'): → branch

if lower bound(C') < opt → add C' to A

↳ bound

Intro to P vs. NP

Polynomial time: $O(n^k)$, k is some constant

Reduction:

2 problems, X & Y. $X \leq Y$ ⇒ an algo to solve Y can be used to solve X

↳ X is "easier" than Y

Polynomial reduction: $X \leq_p Y$ (poly-time algo for Y can solve X)

Implications:

① If Y can be solved in P, $X \in P$ if reduction is poly-

② If X cannot be solved in P ⇒ Y cannot be solved in P

∴ $Y \leq_p X$ & $X \leq_p Y$ ⇒ X & Y are equiv. difficult

Ex:// Hamiltonian path vs Hamiltonian cycle (visits every vertex once)

Obj: Show HAMPATH $\leq_p$ HAMCYCLE

① Reduction algo

```
A_HAMPATH(G):
    ∀ s, t ∈ V(G):
        Create G' by creating edge (s, t)
        if A_HAMCYCLE(G') == T:
            return True
    return False
```

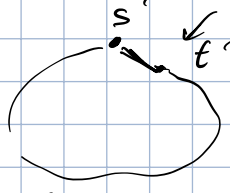
② Polynomial reduction?

Yup! $O(n^2)$ calls to $A_{HAMCYCLE}$

③ Correctness: $A_{HAMPATH}(G) = T \Leftrightarrow G$ has Hamiltonian path.

$\Leftarrow$ If G already had Hamiltonian path from $s \to t$, adding edge creates Hamiltonian cycle.

$\Rightarrow$ Assume $A_{HAMPATH}(G) = T$. This means $A_{HAMCYCLE}(G') = T$



Case 1: $(s', t') \notin$ Hamiltonian cycle

Remove any edge in Hamiltonian cycle → HPath in G!

Case 2: $(s', t') \in$ H cycle

Remove $(s', t') \to$ path in G

NP-completeness is about decision problems

↳ Optimization problems → decision probs.

Formally:

Defn. of P: decision problem that can be solved in poly-time.

Defn. of NP: non-deterministic poly-time.

↳ Not ($\neg$ P)

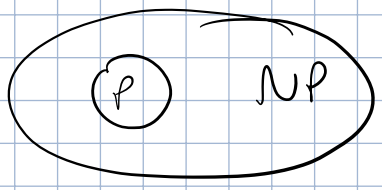
NP & NP-Complete

Better defn. of NP: Whenever algo returns T → we can check a certificate in polynomial time.

Certificate:

- HAMPATH: list of vertices.
- INDEPSET: list of vertices & edges

} → Can check in poly-time if it is a HAMPATH / INDEPSET...



Verification algo:

Decision problem $X(x)$. Verification algo $A(x, y)$, x → input, y → cert.

$\therefore X(x)$ = there exists some certificate y when $A(x, y) = T$

Run in poly-time if <u>A is P</u> & <u>length(y) ≤ poly(length(x))</u>

Ex:// Travelling salesperson

Obj: traverse graph w/ tour of cost ≤ k.

Verification algo $A(G, k, p)$: (x = G, k; y = p)

p: permutation of vertices that is visited in TSP

check if P is a valid perm.

check if edges b/w $P[i]$ & $P[i+1]$ exist

Sum edges ≤ k

Runtime: check all edges $m \leq n^2 \Rightarrow$ poly-time

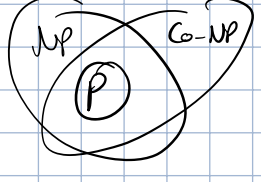
$\therefore$ TSP ∈ NP

co-NP: whenever algo returns F → we can check a certificate in polynomial time

Ex:// PRIME(x) ∈ co-NP.

Certificate if PRIME(x) = F ⇒ a, b s.t. $a \times b = x$

Check certificate in poly-time.



NP-complete:

① Problem ∈ NP

② For every $Y \in NP$, $Y \leq_p X$

How to show ②? Use transistive property of $\leq_p$

∴ Show that $Z \leq_p X$ if Z is known NP-complete problem!

NP-Complete Problems

Many-to-one reduction:

Show $X \leq_p$ ⇒ assume A_Y

① Create algo for input to $X \longrightarrow$ input to Y

② Run A_Y once

③ Correctness: $A_X(x)$ is Yes $\Longleftrightarrow$ $A_Y(y)$ is Yes.

④ Poly-tx: step 1 takes poly time.

① 3-SAT

Input: Boolean formula, we want to find assignment of variables s.t. formula $\longrightarrow$ T

Note: each clause is made of at most 3 variables.

$$(\_ \vee \_ \vee \_) \wedge (\_ \vee \_ \vee \_)$$

clause clause

② INDEP-SET

Input: Graph G & value k

Problem: Does G have indep. set of size k

⇓

Set of vertices not adj. to each other.

Theorem: INDEP-SET is NP-complete

Proof:

① Indp-SET $\in$ NP

② 3-SAT $\leq_p$ INDEP-SET

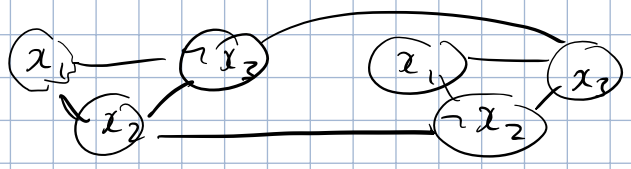
Idea: build a graph G w/ indep. size k $\Leftrightarrow$ formula is satisfiable.

Create vertex for each variable & connect if

a. In clause

b. Same variable negation

Ex: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$



Set k = # of clauses. ⇒ Call INDEP-SET

Correctness:

Graph will have indep. set of size $\geq k$ ⟺ formula is satisfiable

⟸ Assume satisfying assign. Set 1 literal in each clause to be true → indep. set of size m

⇒ Assume indep. set of size m. 1 literal / triple clause in graph. Make that true → all else false

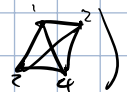
Polynomial time? Yup! 1 call to INDEP-SET

② Clique

Input: undirected graph $G(V, E)$, $k \in \mathbb{Z}$

Output: does G have a clique of size $\geq k$

clique: set of vertices that are pairwise join



Theorem: Indep set $\leq_p$ Clique.

↳ Idea: complement of indep-set → clique.

③ Vertex cover

Input: undirected graph $G(V, E)$, $k \in \mathbb{Z}$

Output: Does G have vertex cover of size $\leq k$

vertex cover: set of vertices where all edges in G have endpoint in set

Theorem: Indep-set $\leq_p$ vertex-cover ⇒ NP complete

Idea: If S is vertex cover → V/S is indep set.

④ Directed Hamiltonian cycle

Theorem: NP complete ($3\text{-SAT} \leq_p DHC$)

⑤ Undirected Hamiltonian cycle

Theorem: NP complete ($DHC \leq_p UHC$)

⑥ Subset sum:

Input: n items w/ weight w_i & threshold W

Output: is there subset of items s.t. $\sum_{i \in S} w_i = W$

Theorem: NP-complete ⇒ $3\text{-SAT} \leq_p$ Subset sum

⑦ Circuit SAT

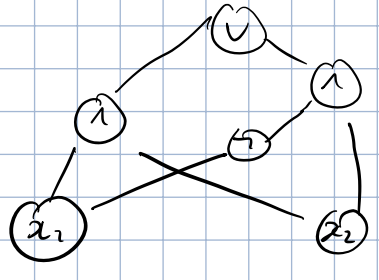
Input: circuit

↳ DAG w/ 3 types of vertices

1. Inputs
2. Operators
3. Sink (output)

} Create a Boolean.

Ex: $(x_1 \wedge x_2) \vee (\neg x_1 \wedge x_1)$



Output: circuit is satisfiable

Theorem: all problems in NP $\leq_p$ circuit SAT

↳ Proof: any verifier also → circuit

Approximation Algorithms

Goal: create a "good-enough" soln. in poly-time for problem.

Ex:// Find vertex cover of min. size.

Also:

while uncovered edge (u, v):
put u, v into cov
make all edges incident to u & v as covered.

Next, show that algo ans $\leq$ (2) $\times$ |optimal ans.| ← approx factor.

↳ Note that all edges covered form matchings

Undecidability

Problem w/ no algo $\longrightarrow$ undecidable problem.

Ex:// Halting problem

Input: program & some input

Output: does the program ever halt?

Usually due to self-contradiction.

Can show problem X is undecidable if X $\leq$ undecidable problem

Ex:// Halt-no input is undecidable

Show halting problem $\leq$ halt-no input

Also: take program & hardcode input $\rightarrow$ halt-no input

Correctness: halt no input w/ modded prog is T $\Leftrightarrow$ prog halts on input