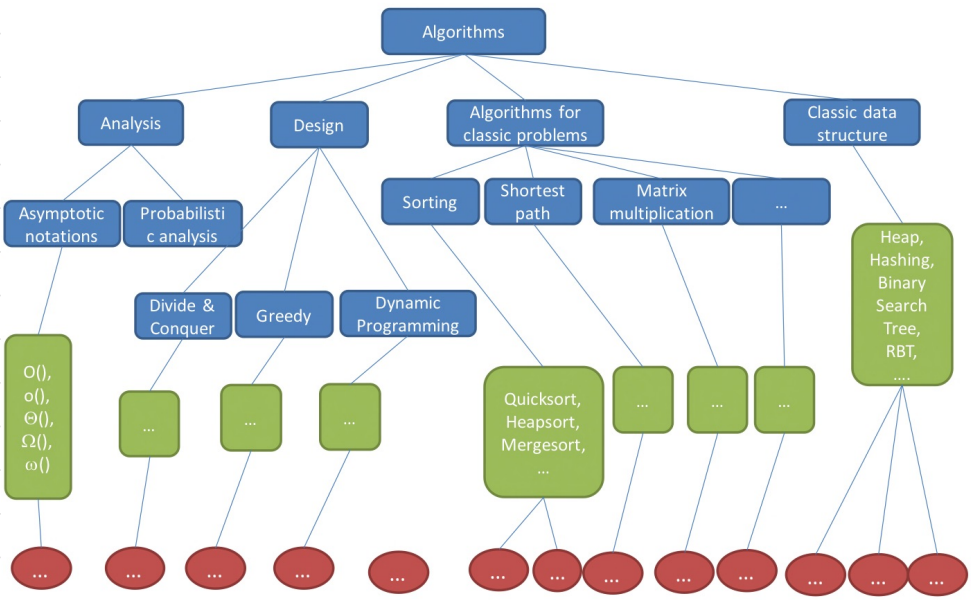
Design and Analysis of Algorithms

1. Asymmetric Notation (5 notations)
2. Time and Space complexity
3. Divide and conquer (all sorting algos)
4. Greedy methods (Job sequencing, knap sack, optimal memory pattern, Huffman encoding, Dijkstra's, MST)
5. Graph Traversal (DFS, BFS, connected components)
6. Dynamic Programming (shortest path, Multistage Graph, BST, TSP, 0/1 knap sack, LCS, Matrix chain, Multi, SOS)
7. Hashing
8. P, NP, NP4, NPC



Algorimm

↳ finite set of steps to solve a problem

↳ not ambigious

Analysis

process of comparing two algos w.r.t time, space

PRIORY

↳ before execution

↳ independent of hardware

↳ counts iterations

↳ approx value

POSTERIOR

↳ after execution

↳ dependent on hardware hence fixed time

↳ exact value

ASYMPTOTIC NOTATIONS

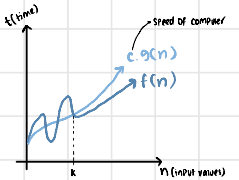
mathematical way to represent time complexity

no of operations in terms of (n)

13

1. Big-Oh (O)

tells max time



↳ worst case
↳ upper bound (at most)

$n \geq 1$

$f(n) = O\,g(n)$

$f(n) \leq c \cdot g(n)$

$c > 0 \quad n \geq k$

$k \geq 0$

eg// $f(n) = 2n^2 + n$

$f(n) = O(\ )$

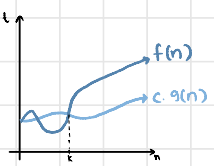
$2n^2 + n \leq c \cdot g(n^2)$

biggest so use it

$2n^2 + n \leq 2n^2$ $\quad n \leq 0$ ✗

$2n^2 + n \leq 3n^2$ $\quad n \geq 1$ ✓ ← to this is c

2. Big Omega (Ω)



↳ Best case
↳ lower bound (at least)

$n \leq 1$

$f(n) = \Omega\, g(n)$

$f(n) \geq c \cdot g(n)$

$n \geq 0$

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

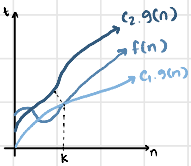
$f(n) = \Omega\, g(\ )$

$2n^2 + n \geq c \cdot g(n^2)$

$2n^2 + n \geq 2n^2$

$n \geq 0$

3. Theta (Θ)



↳ Average case
↳ exact time

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

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

$f(n) = \Theta\, g(\ )$

$c_1 g(n^2) \leq 2n^2 + n \leq c_2 g(n^2)$

$2n^2 \leq 2n^2 + n \leq 3n^2$

Q) $n^3 + 4n^2 = \Omega(n^2)$

$n^3 + 4n^2 \geq c\,n^2$

$n^3 + 4n^2 \geq 2n^2$ $\qquad$ $n^3 + 4n^2 \geq 3n^2$

4 Little Ω or O

make it $>, <$
rather than $\geq, \leq$

little omega (ω)

Q1) $n^2 + 42n + 7 = O(n^2)$

$n^2 + 42n + 7 \leq c(n^2)$

$n^2 + 42n + 7 \leq n^2 + 42n^2 + 7n^2$ → make bigger

$\leq 50n^2$

↓

$c \quad n \geq 1$

$\frac{n^2}{n^2} + \frac{42n}{n^2} + \frac{7}{n^2}$

$1 + \frac{42}{n} + \frac{7}{n^2} \leq c$

↙ ↘

1 ∞

$1 + 42 + 7$ $1 \leq c$

$= 50$ True

Q10)

10. For each of the following questions, indicate whether it is T (True) or F (False) and justify using some examples e.g. assuming a function? [15 Points]

- For all positive $f(n)$; $\omega(f(n))+O(f(n))=\Theta(f(n))$
- For all positive $f(n)$; $f(n)+o(f(n))=\Theta(f(n))$
- For all positive $f(n),g(n)$,*and* $h(n)$: if $f(n)=O(g(n))$ *and* $f(n)=\Omega(h(n))$ *then* $g(n)+h(n)=\Omega(f(n))$

i) $f(n)$; $\omega(f(n)) + O(f(n)) = \Theta(f(n))$

let $f(n)=n$

$\omega(f(n))=\Omega(n)$ as lower bound

$O(f(n))=O(n)$ as upper bound

$\rightarrow \omega(f(n)) + O(f(n))$

$=\Omega(n)+O(n)\neq\Theta(n)$

Hence false

$\omega \rightarrow \Omega$

let $f(n)=n^2$

$\Omega(n^2)+O(n^2)=\Theta(n^2)$

LB UB

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

neither $= n^2$

False

ii) $f(n)$; $f(n)+o(f(n)) = \Theta(f(n))$

let $f(n)=n^2$

$O(f(n))=O(n^3)$ as upper bound

$\rightarrow f(n)+o(f(n))$

$= n^2 + O(n^3)$

$=\Theta(n^2) = \Theta(n^2)$

Hence True

let $f(n)=n^2$

$n^2+O(n^2)=\Theta(n^2)$

UB

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

$n^2=\Theta(n^2)$

True

iii) $f(n)$, $g(n)$, $h(n)$

if $f(n)=O(g(n))$

$f(n)=\Omega(h(n))$

then

$g(n)+h(n)=\Omega(f(n))$

let $f(n)=n^2$

$g(n)=n$

$h(n)=1$

$\hookrightarrow f(n)=O(g(n))$ as n^2 grows as fast as n

$f(n)=\Omega(h(n))$ as n^2 grows as fast as 1

$\hookrightarrow g(n)+h(n)$

$= n+1 = \Omega(f(n))$

Hence true

let $f(n)=n$

$g(n)=n\log n$

$h(n)=\log n$

$\therefore\ g(n)+h(n)=\Omega(f(n))$

$n\log n+\log n=\Omega(n)$

14

ASYMPTOTIC NOTATIONS

			Reflexive ($a=a$)		Symmetric ($a>b \rightarrow b>a$)		Transitive ($a \le b \le c \rightarrow a \le c$)	
Big (O)	$f(n) \le c.g(n)$	$a \le b$	$n^2 = n^2$	✓	$n^2 \le n^3$, $n^3 \ge n^2$	×	$n^2 \le n^3 \le n^4 \rightarrow n^2 \le n^4$	✓
Big (Ω)	$f(n) \ge c.g(n)$	$a \ge b$	$n^2 = n^2$	✓	$n^3 \ge n^2$, $n^3 \le n^2$	×	$n^4 \ge n^3 \ge n^2 \rightarrow n^4 \ge n^2$	✓
Theta (Θ)	$c_1.g(n) \le f(n) \le c_2.g(n)$	$a=b$	$n^2 = n^2$	✓	$n^2 = n^2$, $n^2 = n^2$	✓	$n^2 = n^2 = n^2 \rightarrow n^2 = n^2$	✓
Small (o)	$f(n) < c.g(n)$	$a < b$	$n^2 < n^2$	×	$n^2 < n^3$, $n^2 > n^3$	×	$a < b < c \rightarrow a < c$	✓
Small (ω)	$f(n) > c.g(n)$	$a > b$	$n^2 > n^2$	×	$n^3 > n^2$, $n^3 < n^2$	×	$a > b > c \rightarrow a > c$	✓

15

Comparisions of various time complexities

order of Big O → as it tells max time; constant

hashing, median in sorted arrays

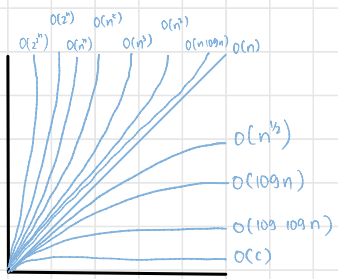
1. $O(c)$ $\quad O(10^3) = O(10^9)$ — even though this is bigger BUT in terms of algo both are same $= O(1)$
2. $O(\log(\log n))$ $\quad \log_2(\log_2 1000) = \log_2 10 = 4$
3. $O(\log n)$ $\quad \log_2 1000 = 10$
4. $O(n^{1/2})$ $\quad 1000^{1/2} = 33$
5. $O(n)$ (linear) $\quad 1000$
6. $O(n \log n)$ $\quad 1000(\log_2 1000) = 1000(10) = 10{,}000$
7. $O(n^2)$ (quadratic) $\quad (1000)^2 = 1{,}000{,}000$
8. $O(n^3)$ $\quad (1000)^3 = 1{,}000{,}000{,}000$
9. $O(n^k)$ (Polynomial) $\quad (1000)^k = 1{,}000$ k times

Dynamic Problems, Travelling salesman, matrix chain multiplication

10. $O(2^n)$ (exponential) $\quad 2^{1000}$ — has 302 digits so imagine
11. $O(n^n)$ (factorial) $\quad 1000^{1000}$
12. $O(2^{2^n})$ (double exponential) $\quad 2^{2^{1000}} = 2^{\text{302 digits}} =$ very big

slower ↓

always use big values like '1000' to check as small values have negligible difference



$\Theta(n)$
↳ is linear
↳

$O(\log n)$
$b^n = n$
$n \log_b = \log_n$
$n = \log_b n$

1.5

Time complexities of all searchin and sorting Algorimms

Algos	Best case	Avg case	Worst case
Binary search	$O(c)$ first element	$O(\log n)$ no of elements	$O(\log n)$
* Quick sort	$O(n \log n)$	$O(n \log n)$ *	$O(n^2)$ 1, 2, 3, 4, (5) pivot element
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
* Insersion sort *	$O(n)$ sorted array	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Heap sort	$O(n \log n)$	$O(n \log n)$ for 1 element	$O(n \log n)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Radix	$O(nk)$	$O(nk)$	$O(nk)$
Counting	$O(n+k)$	$O(n+k)$ range	$O(n+k)$
Bucket	$O(n+k)$	$O(n+k)$	$O(n^2)$
Insersion/Deletion in Heap	$O(1)$	$O(n \log n)/O(n)$ heapify insersion	$O(n \log n) / O(n)$ heapify insersion
Height of CBT	$O(\log n)$	$O(\log n)$	$O(\log n)$

Huffman	$n \log n$	$n \log n$	$n \log n$
Prims	$O(n^2)$ for matrix	$O(V+E) \log V$ vertices, edges; for heap	→ no worst, avg, best case
Kruskal	$O(E \log E)$	$O(E \log E)$	$O(E \log E)$
DFS, BFS	$O(V+E)$	$O(V+E)$	$O(V+E)$
Dijkstras	$O(V^2)$	$O(V^2)$	$O(V^2)$

log rules

1. $\log(a \times b) = \log a + \log b$
2. $\log a/b = \log a - \log b$
3. $\log a^b = b \log a$
4. $\log_a 1 = 0$
5. $\log_a a = 1$
6. $\log_2 2 = 1$
7. $b^n = n \Rightarrow n = \log_b n$
   $\log_2 8 \Rightarrow 2^3 = 8$
8. $a^{\log_a b} = b$
   $2^{\log_2 n} = n$

$O(n \log n)$
↳ when input is recursively broken up

$O(n^2)$
↳ algos that deal with pairs of data
↳ nested loop

1.7

comparision of various time complexities

Q) $f_1(n) = n^2 \log_{10} n$ $\quad f_2(n) = n(\log_{10} n)^{10}$

1. Substitute → Put big value

$n = 10^9$

$f_1(16) = (10^9)^2 \log_{10}(10^9)$
$= 10^{9} \times 10^9 \times 9$
$= 10^9 \times 9$
$= 10^9$ → bigger

$f_2(16) = 10^9 (\log_{10} 10^9)^{10}$
$= 10^{9} \times 9^{10}$
$= 9^9 \times 9$
$= 9^9$

2. Simplify

$n = 10^9$

$f_1(n) = n \times n \log_{10} n$
$= n$
$= \log n$
$= \log n$
bigger

$f_2(n) = n \log_{10} n (\log n)^9$
$= (\log n)^9$
$= \log(\log n)^9$
$= 9 \log(\log n)$ ← constant doesn't matter

$f_2(n) = f_1(n)$ → ?

$f_2(n) \leq c\, f_1(n)$

1.8

Q) $f_1(n) = 2^n$ $\quad f_2(n) = n^{3/2}$ $\quad f_3(n) = n \log_2 n$ $\quad f_4(n) = n^{\log_2 n}$

$n = 16$

$f_1 = 2^{16}$

$f_2 = 16^{3/2}$

$f_3 = 16 \log_2 16$
$= 16 \times 4$

$f_4 = 16^{\log_2 16}$
16^4

RECURRENCE RELATION

↳ a function that calls itself

2.)

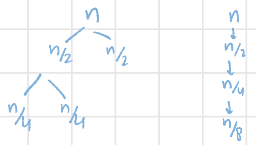
Binary Search

↳ sorted array

```
void BinarySearch(int arr[],int lo,int hi,int x)
{
 int mid;

 if(hi > lo)
        {printf("Number is not found"); }
else
        { mid = (hi + lo)/2;

                if ( arr[ mid ] == x)
                        { printf("Element is found at index %d ",mid);
                         exit(0);
                        }

                else if (arr[mid] > x)
                        { BinarySearch(arr, x, hi, mid-1);}

                else
                { BinarySearch(arr, x, mid+1, low); }
        }
}
```

n/2

O(c)



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

Solving Recurrence Relation

1. Substitution method
   ↳ aka Back Substitution
2. Master Theorem
3. Recursion Tree
4. Guess and Test

1. Back Substitution Method

PRO → always gives correct ans
CON → too many mathematical calculations → slow

22/ Q) $T(n) = \begin{cases} T(n/2) + C & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$ → Binary Search

$T(n) = T(n/2) + C$ — ①
$T(n/2) = T(n/4) + C$ — ②
$T(n/4) = T(n/8) + C$ — ③

Substitute 2 into 1
$T(n) = T(n/4) + 2C$
$= T(n/2^2) + 2C$

→ Substitute 3
$T(n) = T(n/8) + 3C$
$T(n) = T(n/2^3) + 3C$
⋮ Predicting iterations
$T(n) = T(n/2^4) + 4C$

for k iterations
$T(n) = T(n/2^k) + kC$
↳ to remove T, let $2^k = n$
$k \log 2 = \log n$ ∴ log 2 = 1
$k = \log n$
$= T(n/n) + \log n\, C$
$= 1 + \log n \cdot C$ (constants)
$= O(\log n)$

23/ Q) $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ n * T(n-1) & \text{if } n > 1 \end{cases}$ → base/termination condition

$T(n) = n \times T(n-1)$ — ①
$T(n-1) = (n-1) \times T(n-2)$ — ②
$T(n-2) = (n-2) \times T(n-3)$ — ③

Substitute 2 in 1
$T(n) = n \times (n-1) \times T(n-2)$

Substitute in 3
$T(n) = n \times (n-1) \times (n-2) \times T(n-3)$ → eliminate using base condition
⋮ goes upto (n-1) steps
$T(n - (n-1))$
$T(1)$

$T(n) = n \times (n-1) \times (n-2) \times (n-3) \ldots \times 1$
$= n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$
$= n \times n(1 - \frac{1}{n}) \times n(1 - \frac{1}{n}) \times \ldots \times n(3/n) \times n(2/n) \times n(1/n)$ (take n common)
$= n^n (1 - \frac{1}{n})(1 - \frac{2}{n}) \ldots (3/n)(2/n)(1/n)$ → negligible difference
$= O(n^n)$

2. Master Method

pro → faster
con → can only solve specific ones

↳ $T(n) = aT(n/b) + f(n)$

$a \geq 1$, $b > 1$

↳ solution is

$T(n) = n^{\log_b a}\,[u(n)]$

↳ $R(n) = \dfrac{f(n)}{n^{\log_b a}}$

↳ if conversion

$T(n) = aT(n/b) + n^k (\log n)^p$

if R(n)	u(n)
n^r , $r > 0$	$O(n^r)$
n^r , $r < 0$	$O(1)$
$(\log n)^i$, $i \geq 0$	$\dfrac{O(\log_2 n)^{i+1}}{i+1}$

2.6/

Q) $T(n) = 8T(n/2) + n^2$

$a = 8$, $b = 2$, $f(n) = n^2$ → format correct

solution

$= n^{\log_2 8}\,[u(n)]$

$= n^3\,[u(n)]$

↳ $\dfrac{n^2}{n^{\log_2 8}} = \dfrac{n^2}{n^3} = n^{-1}$

$\therefore n^r$, $r < 0 \rightarrow O(1)$

$= n^3\, O(1)$

$= O(n^3)$

2.7/

Q) $T(n) = T(n/2) + c$

$a = 1$, $b = 2$, $f(n) = c$

solution

$T(n) = n^{\log_2 1}\,[u(n)]$

$= n^0\,[u(n)]$

$= 1\,[u(n)]$

↳ $\dfrac{c}{1} = c$ convert to 3rd case $(\log_2 n^0)\,c$

$\therefore (\log n)^i$, $i \geq 0 \rightarrow \dfrac{O(\log_2 n)^{i+1}}{i+1}$

$= O(\log_2 n)^1$

2.8/

Q) $T(n) = \begin{cases} T(\sqrt{n}) + \log n & \text{, if } n \geq 2 \\ O(1) & \text{else} \end{cases}$

$T(n) = T(\sqrt{n}) + \log n$

convert to format

→ let $n = 2^m$

$T(2^m) = T(2^{m/2}) + m$ ($m \log_2 2 = m$)

→ let $T(2^m) = S(m)$

$S(m) = S(m/2) + m$ → converted

$a = 1$, $b = 2$, $f(n) = m$

→ solution

$T(n) = n^{\log_2 1}\,[u(n)]$

$= 1\,[u(n)]$

↳ m

$\therefore n^r$, $r > 0 \rightarrow O(n^r)$

$= O(m)$

$= O(\log n)$

$n = 2^m$

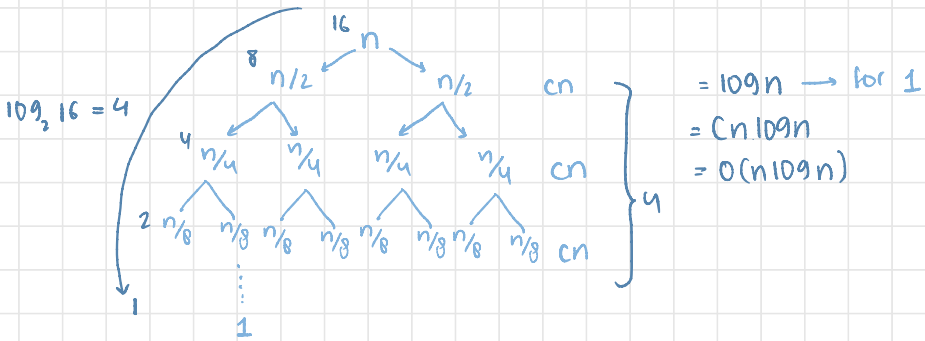
$\log n = m \log 2$ $\therefore \log_2 2 = 1$

$m = \log n$

3. Recursive Tree method

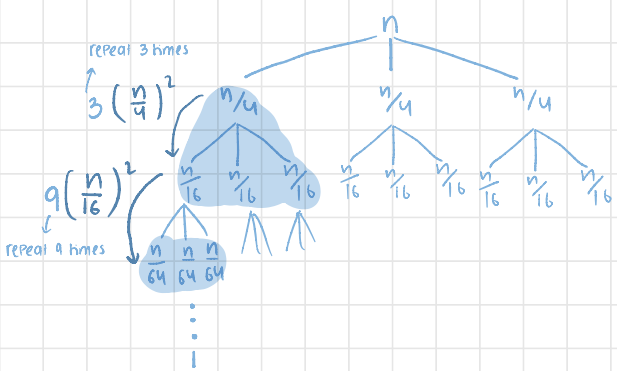
2nd Q) $T(n) = 2T(n/2) + cn$ → merge sort

(split in 2)



2nd Q) $T(n) = 3T(n/4) + cn^2$

(split in 3)



cn^2 $\left(\frac{n}{4}\right)^2 = \frac{n^2}{16}$ → for 1 part

$\frac{3}{16}n^2$ → for 1 level

$\left(\frac{3}{16}\right)^2 n^2$ → for 2 level

$$cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \left(\frac{3}{16}\right)^3 cn^2 \ldots 1$$

$$cn^2\left[1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \left(\frac{3}{16}\right)^3 + \ldots\right]$$

Geometric Progression: $1 + r + r^2 + r^3 + \ldots \rightarrow \frac{1}{1-r}$, $r < 1$

$$= cn^2\left[\frac{1}{1 - 3/16}\right]$$

$$= cn^2 \frac{16}{13}$$ ← constant so ignore

$$\therefore O(n^2)$$

Guess and Test method

↳ make gusses

↳ use mamamehcal induchon to solve

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

Guess 1 = $O(n)$

need $T(n) \leq cn \quad \therefore c > 0$

assume $T(n/2) \leq c\frac{n}{2}$ $\qquad cn + 2c\frac{n}{2}$

Thus $T(n) \leq 2c\frac{n}{2} + n$

$= (c+1)n$

32

Divide and conquer

↳ to design algorithm

↳ if problem to big

break it into smaller sub problems

then combine

1. How it works
2. Time complexity
3. Reccurence Relation

Stable Radix sort
Stable Bucket sort
Stable Counting sort
Stable Merge sort
Stable Bubble sort inplace
Stable Insersion sort inplace
Selection sort inplace
Quick sort inplace
Heap sort inplace
Insersion/Deletion in Heap

$$\frac{\text{no of comparisions}}{\text{no of swaps}}$$

Space complexity

↳ total amount of memory space used by an algorithm

inplace: not take extra space

Stable: order of same element same

eg 5_A 2 1 5_B

1 2 5_A 5_B ✓

1 2 5_B 5_A ✗

Randomized algorithms

- We used randomness to minimize the possibility of worst case running time
- Other uses
  - approximation algorithms, i.e. random walks
  - initialization, e.g. K-means clustering
  - contention resolution
  - reinforcement learning/interacting with an ill-defined world

3.) faster than merge sort

QUICK SORT

inplace

↳ partitions with pivot

CONS

↳ requires extra storage

↳ Time complexity

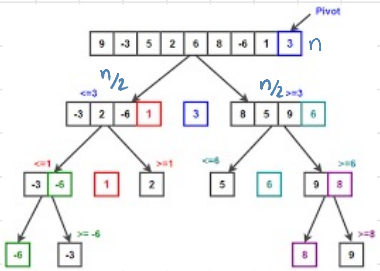
$O(n \log n)$

Avg/best case → pivot already mid num

$T(n) = n/2 + n/2 + n$ → scan entire array

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

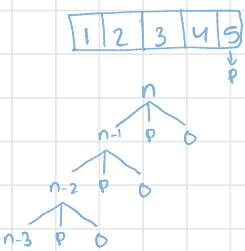
solve using master theorem



* worst case → already sorted array / reverse sorted array $O(n^2)$

$T(n) = T(n-1) + n$ → scan entire array

solve using substitution method



↳ to avoid worst case

inject randomness into data

Is Quicksort correct?

- Assuming Partition is correct
- Proof by induction
  - Base case: Quicksort works on a list of 1 element
  - Inductive case:
    - Assume Quicksort sorts arrays for arrays of smaller < n elements, show that it works to sort n elements
    - If partition works correctly then we have:
    - and, by our inductive assumption, we have:

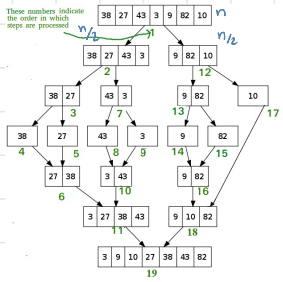
3/3

merge sort

↳ divides into subarrays till 1 element is left

↳ merge array sortedly

- Pro's:
  - O(*n* lg *n*) worst case - *asymptotically optimal for comparison sorts*
- Con's:
  - Doesn't sort in place



↳ Time complexity

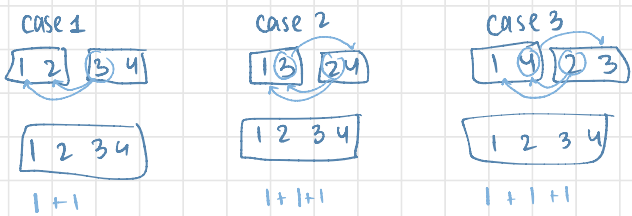
Avg/best/worst case $n(\log n)$

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

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

Q) The no of comparisons by merge sort also, in merging 2 sorted lists of length 2

Time complexity: $n \log n$



$= \frac{8}{3} = 2.67$ (3 ← no of cases)

Inductive Proof of Correctness

- **Initialization:** (the invariant is true at beginning)

  Prior to the first iteration of the loop, we have $k = 1$, so that $A[1,k-1]$ is empty. This empty subarray contains $k-1 = 0$ smallest elements of L and R and since $i = j = 1$, $L[i]$ and $R[j]$ are the smallest element of their arrays that have not been copied back to A.

- **Maintenance:** (the invariant is true after each iteration)

  assume $L[i] <= R[j]$, the $L[i]$ is the smallest element not yet copied back to A. Hence after copy $L[i]$ to $A[k]$, the subarray $A[1..k]$ contains the k smallest elements. Increasing k and i by 1 reestablishes the loop invariant for the next iteration.

- **Termination:** (loop invariant implies correctness)

  At termination we have k = s+t + 1, by the loop invariant, we have A contains the k-1 (s+t) smallest elements of L and R in sorted order.

Correctness of MergeArray

- Loop-invariant
  - At the start of each iteration of the **for** loop, the subarray $A[1:k-1]$ contains the $k-1$ smallest elements of $L[1:s+1]$ and $R[1:t+1]$ in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back to A

3.4/

Bubble Sort

- no of swapping
- time complexity
- array looks like after 2 pass

↳ swaps each element

↳ Time complexity

Avg / worst case $O(n^2)$

S: 1 2 3 4 ... n-2 n-1

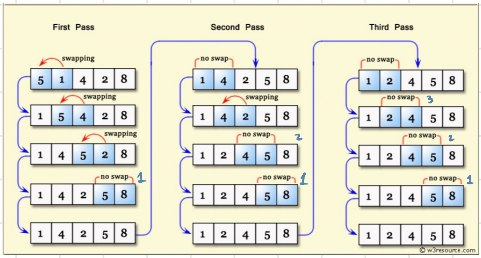
S: n-1 n-2 n-3 ..... 2 1

2S: n n n ..... n n

$S = \frac{(n-1)}{2}$

$= \frac{n^2-n}{2}$ n

Solve using Asymptotic notation



n-1 n-2 ..... 2 1

best case

$= O(n)$ → ordered

Radix Sort

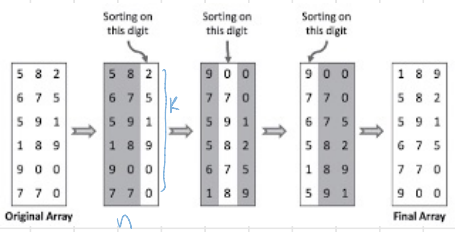
stable

↳ Time complexity

Avg/best/worst case

$T(n) = O(nk)$ → range

- Pro's:
  - Fast
  - Asymptotically fast (i.e., O(*n*) when *d* is constant and *k*=O(*n*))
  - Simple to code
  - A good choice
- Con's:
  - Doesn't sort in place
  - Not a good choice for floating point numbers or arbitrary strings.



35/ Insertion Sort

aka online algo → as closest want for entire array

majority used

→ stable

→ in place

↳ moving elements to sorted side (left)

↳ compare adjacent values

↳ move to left 1 by 1

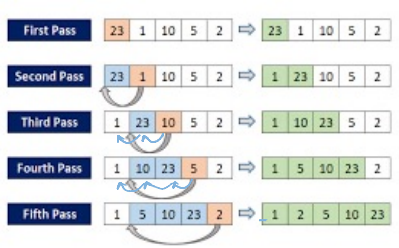
- Insertion sort:
  - Pro's:
    - Easy to code
    - Fast on small inputs (less than ~50 elements)
    - Fast on nearly-sorted inputs

↳ Time complexity

Avg/worst case $O(n^2)$

$T(n) = \frac{n(n-1)}{2}$

$= \frac{n^2 - n}{2}$



C	S
0	0
1	1
2	2
3	3
⋮	⋮
(n-1)	(n-1)

* best case $O(n)$

$T(n) = n - 1$ → only comparisions, no swaps

10	20	30
10	20	30
10	20	30
10	20	30

C	S
0	0
1	0
1	0
1	0
⋮	
n-1	

```
void insertionSort(int arr[], int n)
{
        int i, temp, j;
        for (i = 1; i < n; i++)
        {
                temp = arr[i];
                 j = i - 1;

                while (j >= 0 && arr[j] > temp)
                {
                        arr[j + 1] = arr[j];
                        j = j - 1;
                }
                arr[j + 1] = temp;
        }
}
```

Loop Invariants

Definition: A *loop invariant* is a property P that if true before iteration i it is also true before iteration $i+1$

Example:
Computing the maximum

Require: Array of n positive integers A
$m \leftarrow A[0]$
for $i = 1, \ldots, n-1$ **do**
  if $A[i] > m$ **then**
    $m \leftarrow A[i]$
return m

Invariant: Before iteration i: $m = \max\{A[j] \ : \ 0 \leq j < i\}$

Proof: Let m_i be the value of m before iter. i ($\rightarrow m_1 = A[0]$).

- *Base case.* $i = 1$: $m_1 = A[0] = \max\{A[j] \ : \ 0 \leq j < 1\}$ ✓
- *Induction step.*

$$\begin{aligned} m_{i+1} &= \max\{m_i, A[i]\} = \\ &= \max\{\max\{A[j] \ : \ 0 \leq j < i\}, A[i]\} \\ &= \max\{A[j] \ : \ 0 \leq j \leq i\} \ . \ \checkmark \end{aligned}$$

Loop Invariants - More Formally

Main Parts:

- **Initialization:** It is true prior to the first iteration of the loop.

  before iteration $i = 1$: $m = A[0] = \max\{A[j] \ : \ j < 1\}$ ✓

- **Maintenance:** If it is true before an iteration of the loop, it remains true before the next iteration.

  before iteration $i > 1$: $m = \max\{A[j] \ : \ j < i\}$ ✓

- **Termination:** When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

  At the end of the loop m contains the maximum ✓

Loop Invariant of Insertion-sort

Require: n integer
$s \leftarrow 1$
for $j = 2, \ldots, n$ **do**
  $s \leftarrow s \cdot j$
return s

Invariant: At beginning of iteration j: $s = (j-1)!$

- Let s_j be the value of s prior to iteration j
- **Initialization:** $s_2 = 1 = (2-1)!$ ✓
- **Maintenance:** $s_{j+1} = s_j \cdot j = (j-1)! \cdot j = j!$ ✓
- **Termination:** After iteration n, i.e., before iteration $n+1$, the value of s is $(n+1-1)! = n!$ ✓

Algorithm computes the factorial function

Loop Invariant of Insertion-sort

for $j = 1, \ldots, n-1$ **do**
  $v \leftarrow A[j]$
  $i \leftarrow j - 1$
  while $i \geq 0$ **and** $A[i] > v$ **do**
    $A[i+1] \leftarrow A[i]$
    $i \leftarrow i - 1$
  $A[i+1] \leftarrow v$

Loop Invariant: At beginning of iteration j of the outer **for** loop, the subarray $A[0, j-1]$ consists of the elements originally in $A[0, j-1]$, but in sorted order

- **Initialization:** $j = 1$: subarray $A[0]$ is sorted ✓
- **Maintenance:** *Informally*, element $A[j]$ is inserted at the right place within $A[0, j]$. A formal argument would require another loop invariant for the inner loop. ✓
- **Termination:** After iteration $j = n-1$ (i.e., before iteration $j = n$) the loop invariant states that A is sorted. ✓

36

Selection Sort

→ in place
→ not stable

↳ swaps smallest element in array with first position
↳ swaps 2nd smallest element in array with second position

↳ Time complexity

Avg/best/worst case $O(n^2)$

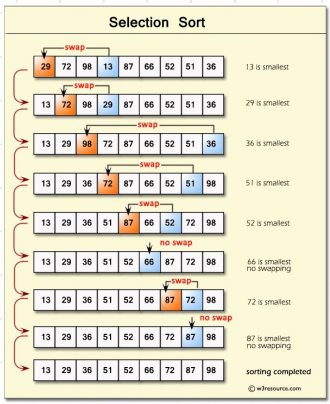
$1 + 2 + 3 + 4 + \dots + n-1$

$T(n) = \frac{n(n-1)}{2}$

comparisons: $O(n^2)$
swaping: $O(n)$
→ bigger → $O(n^2)$

as 1 swap ← 1

$n-1$
$n-2$
$n-3$
$\vdots$
0



Counting Sort

→ is stable

↳ defined range

input:

1	2	3	4	5	6	7
2	1	2	3	1	2	4

↳ Time complexity

Avg/best/worst case

$T(n) = O(n+k)$

scan entire array

range

count:

1	2	3	4	5	6	7	→ range index
2	3	1	1	0	0	0	→ occurence

: 2 (idx 0) 5 (idx 1) 6 (idx 2) 7 7 7

output:

1	2	3	4	5	6	7
1	1	2	2	2	3	4

↳ space complexity

$= O(k)$

CONS

↳ predefined range
↳ huge gap elements

2 3 2000 6

↓

2000 will be extra space

- Pro's:
  - Fast
  - Asymptotically fast - O(*n*+*k*)
  - Simple to code
- Con's:
  - Doesn't sort in place.
  - Elements must be integers. ***countable***
  - Requires O(*n*+*k*) extra storage.

Bucket sort

↳ for floating point numbers

↳ defined range

- Pro's:
  - Fast
  - Asymptotically fast (i.e., O(*n*) when distribution is uniform)
  - Simple to code
  - Good for a rough sort.
- Con's:
  - Doesn't sort in place

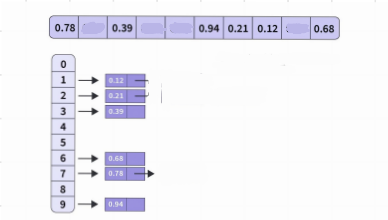
↳ Time complexity

Avg/best case $O(n+k)$

calc to insert | no of elements | range

$O(1) \times n + k$

$O(n+k)$

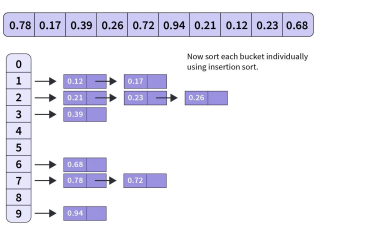


worst case $O(n^2)$

calc to insert | insersion sort

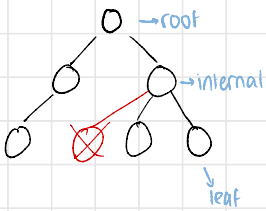
$O(1) \times O(n) \times n$ → no of elements

$\therefore O(n^2)$

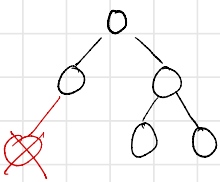


37/ Binary Tree → no loops

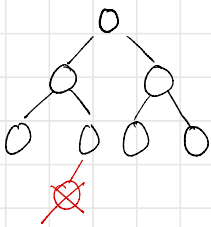
Binary Tree
↳ at most 2 childs



FULL BT
↳ either 0 or 2 childs



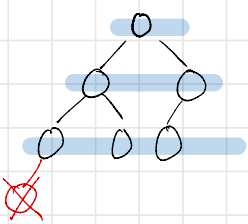
Complete BT
↳ only 2 childs



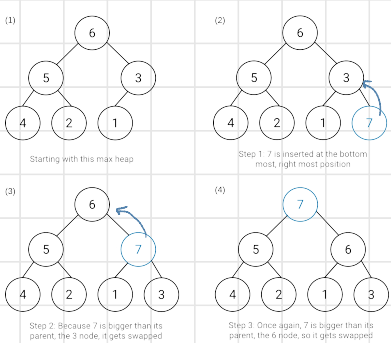
38/ aka ACBT
Heap Tree → max heap (P > C), min heap (P < C)
↳ fill left then right
↳ complete 1 level, then go to next

→ check vide for concept on how its made

Ques) show content after 2 delete operations



Inserting 7 into this heap



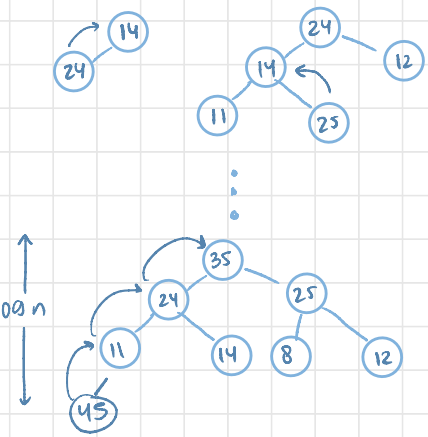
CONCLUSION

- The primary advantage of the heap sort is its efficiency. The execution time efficiency of the heap sort is O(n log n). The memory efficiency of the heap sort, unlike the other n log n sorts, is constant, O(1), because the heap sort algorithm is not recursive.
- The heap sort algorithm has two major steps. The first major step involves transforming the complete tree into a heap. The second major step is to perform the actual sort by extracting the largest element from the root and transforming the remaining tree into a heap.

Heap Tree Insertion

30/ Insert key one by one

14, 24, 12, 11, 25, 8, 35



↳ Time complexity

worst case

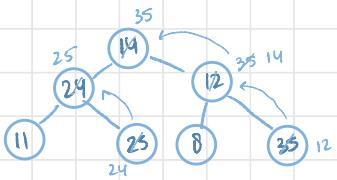
BT height (for adding 1 element) × no of swaps (log n + log n) × n (no of elements to add)

$O(n \log n)$

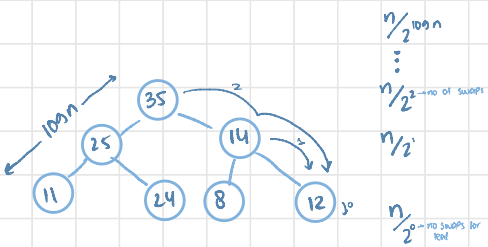
31/ Heapify method

↳ insert all then order

14, 24, 12, 11, 25, 8, 35



Now order using heapify



↳ Time complexity

$O(n)$

comparision in leaf nodes = 0

no of leaf nodes = $n/2$

$$S = \frac{n}{2^0}\times 0 + \frac{n}{2^1}\times 1 + \frac{n}{2^2}\times 2 + \dots + \frac{n}{2^{\log n}}\times \log n \quad \text{(leaf node)} \quad (1)$$

$$S = n\left(\frac{1}{2} + \frac{2}{2^2} + \dots + \frac{\log n}{2^{\log n}}\right)$$

multiply by $\frac{1}{2}$

$$\frac{S}{2} = n\left[\frac{1}{2^2} + \frac{2}{2^3} + \dots + \frac{\log n - 1}{2^{\log n}} + \frac{\log n}{2^{\log n+1}}\right] \quad (2)$$

Subtract eq 2 from 1

$$\frac{S}{2} = n\left[\left(\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots + \frac{1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n+1}}\right]$$

$$\frac{S}{2} = n\left[\left(\frac{\frac{1}{2}\left(1-\frac{1}{2^{\log n}}\right)}{\frac{1}{2}}\right) - \frac{\log n}{2^{\log n+1}}\right]$$

$$= n\left[\left(\frac{2^{\log n}-1}{2^{\log n}}\right) - \frac{\log n}{2^{\log n+1}}\right]$$

$2^{\log n} = n$, $2^{\log n+1} = 2n$

$\therefore S = \frac{a\cdot(1-r^n)}{1-r}$

$\therefore 2^{\log n} = n^{\log 2} = n$

$$\frac{S}{2} = n\left[\frac{n-1}{n} - \frac{\log n}{2n}\right]$$

$S = 2n - 2 - \log n$ → bigger

$O(n)$

LINEAR SORTING ALGORITHMS

NON COMPARISSIONS strategy

Counting Sort

is stable

↳ defined range

Avg/best/worst case

$T(n) = O(n+k)$

n: scan entire array; k: range

↳ space complexity

$= O(k)$

CONS

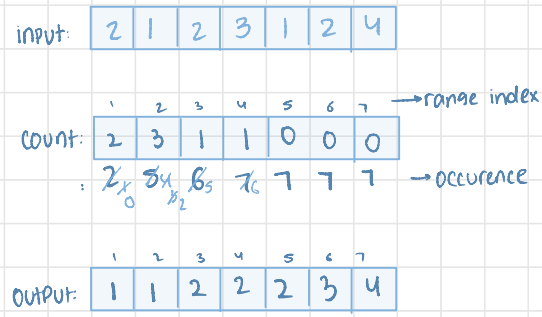
↳ predefined range

↳ huge gap elements

2 3 2000 6

↳ 2000 will be extra space

- Pro's:
  - Fast
  - Asymptotically fast - O(*n*+*k*)
  - Simple to code
- Con's:
  - Doesn't sort in place.
  - Elements must be integers. ***countable***
  - Requires O(*n*+*k*) extra storage.



Bucket Sort

is stable

↳ for floating point numbers

↳ defined range

Avg/best case $O(n+k)$

calc to insert: $O(1)$ × no of elements: n + range: k

$O(n+k)$

worst case $O(n^2)$

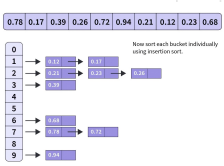
bucket contains all n values of array

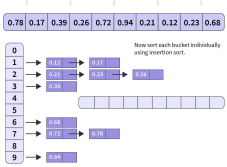
calc to insert: $O(1)$ × insertion sort: $O(n)$ × no of elements: n

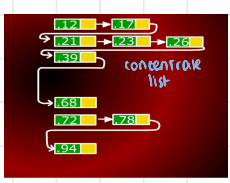
$= O(n^2)$

to make it faster use merge sort $O(n \log n)$ xn

- Pro's:
  - Fast
  - Asymptotically fast (i.e., O(*n*) when distribution is uniform)
  - Simple to code
  - Good for a rough sort.
- Con's:
  - Doesn't sort in place







Radix Sort

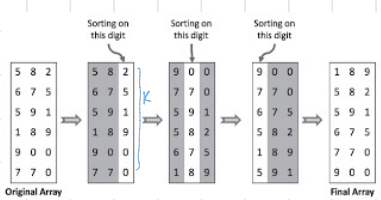
stable

Avg/best/worst case

$T(n) = O(nk)$

n: range; k: k

- Pro's:
  - Fast
  - Asymptotically fast (i.e., O(*n*) when *d* is constant and *k*=O(*n*))
  - Simple to code
  - A good choice
- Con's:
  - Doesn't sort in place
  - Not a good choice for floating point numbers or arbitrary strings.



Stable → duplicates remain in same relative position after sorting

Counting Sort

The algorithm uses three arrays.
1. A[1..n]: Holds the initial input.
2. B[1..n]: Array that holds the sorted output.
3. C[1..k]: Array of integers. C[x] is the rank of x in A, where x ∈ [1..k].

```
5   // C[i] now contains the number of elements = i
6   for i ← 2 to k
7   do C[i] ← C[i] + C[i - 1]   k times
8   // C[i] now contains the number of elements ≤ i
9   for j ← length[A] downto 1
10  do B[C[A[j]]] A[j]
11  C[A[j]] C[A[j]] - 1   n times
```

Bucket Sort

BUCKET-SORT(A)

```
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

Here is the algorithm that sorts A[1..n] where each number is d digits long.

```
RADIX-SORT( array A, int n, int d)
1  for i ← 1 to d
2  do stably sort A w.r.t i^th lowest order digit
```

Algorithm 1: Radix Sort

Data: a - the input integer array with n integers such that $a[j]$ has d digits for all the j.
Result: a is sorted non-decreasingly $(a[1] \leq a[2] \leq \ldots \leq a[n-1] \leq a[n])$.

```
for i ← 1, 2, . . . , d do
|  a ← sort a on digit i using Counting Sort
end
```

TYPES OF ALGOS

1. Brute Force

↳ tries all possibilities

PROS

↳ if solution exists, it will definitely find it

CONS

↳ take a lot of time

2. Divide and Conquer

↳ Optimal substructure → solution of problem can be built from solution of sub problems

↳ divides problems into smaller problems

↳ recursively solves these sub problems

e.g Merge sort, Quick sort

3. Greedy

↳ Optimal substructure

↳ works in phases

↳ at each phase, takes best solution

↳ w/o regard to future consequences

PROS

↳ good for

activity selection problem

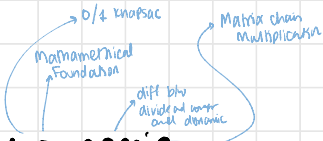
fractional knapsack problem

CONS

↳ bad for

0/1 knapsack problem

coin change problem



4. Dynamic

↳ Optimal substructure

↳ divides problems into smaller overlapping subproblems

↳ repeated subproblems are solved only once

↳ their results are stored

PROS

↳ good for

0/1 knapsack problem

Longest common sub sequence problem

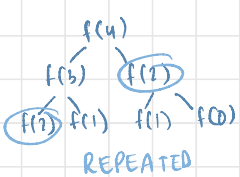
Matrix chain multiplication problem

Coin change problem

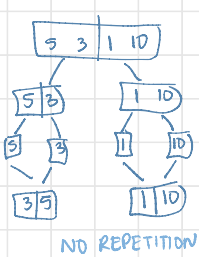
DYNAMIC VS DIVIDE AND CONQUER

↳ overlapping subproblems

↳ stores result of subproblems



↳ no overlapping subproblems



PATTERN MATCHING

→ only concept no algos

Applications: text editors, web search engines, image analysis

1. Brute Force Algo

→ aka Naive

$O((n-m+1)m)$ (string length, pattern length)

$O(nm)$ → when $n \gg m$

↳ check each position in text

↳ backtrack i+1

Q) S = a b c d a b c d f (n = 9)

P = a b c d f (m = 5)

since j[5] ≠ i[5]

then backtrack i

S = a b c d a b c d f

P = a b c d f

S = a b c d a b c d f

P = a b c d f

```
NAIVE-STRING-MATCHER(T, P)
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n − m
4       do if P[1 . . m] = T[s + 1 . . s + m]
5            then print "Pattern occurs with shift" s
```

PROS

↳ fast for large data set → A-Z, a-z, 0-9

CONS

↳ slow for small data set → 0,1 or binary files

↳ backtracking

↳ worst case

S = aaaaaaaaab

P: aaab

2. Knuth Morris Pratt (KMP)

$O(n+m)$

$O(n)$ → for $n \gg m$

↳ moves left to write

↳ no backtracking, use LPS/π

Q) Find LPS → aka π

1. a b c d a b e a b f

0 0 0 0 1 2 0 1 2 0

2. a b c d e a b f a b c

0 0 0 0 0 1 2 0 1 2 3

3. a a b c a d a a b e

0 1 0 0 1 0 1 2 3 0

4. a a a a b a a c d

0 1 2 3 0 1 2 0 0

Q) S = a b a b c a b a b d

P = a b a b d

0 0 1 2 0 → LPS

S = a b a b c a b a b d

P = a b a b d

0 0 1 2 0 → LPS

mismatch so move i to index 2

S = a b a b c a b a b d

P = a b a b d

0 0 1 2 0 → LPS

mismatch so move j to index 0

S = a b a b c a b a b d

P = a b a b d

0 0 1 2 0 → LPS

S = a b a b c a b a b d

P = a b a b d

0 0 1 2 0 → LPS

have reached end of pattern

hence found

3. Boyer Moore

string length, pattern length, and mach table size

$O(nm+s)$

↳ make bad Match Table

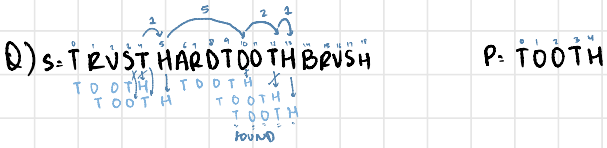
Value: length − index − 1

last letter: length → if not defined before

* : length

↳ compare right to left

↳ If not same move x spaces according to the bad match Table, of last element of string



length = 5

letter	T	O	H	*
value	1	2	5	5

last letter

5-0-1-4
5-3-1=1

5-1-1=3
5-2-1=2

CONS

↳ worst case same as naive

KNAPSACK

↳ fractional KS

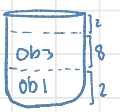
↳ 0/1 KS

1. Greedy Algo

↳ ratio of P/W

↳ doesn't give optimal solution for 0/1 KS

	ob1	ob2	ob3
Profit = {	20	25	60 }
Weight = {	2	4	8 }
P/W = {	10	6.25	7.5



capacity(M) = 12

= 20 + 60

= 80

2. Divide and conquer

↳ partion into subproblems

↳ solve and combine subproblems

CON

↳ subproblems repeated — leads to repeatedly solving common subproblems hence more work done

3. Dynamic O(nm)

↳ subproblems repeated — get solution to subproblem once and store in table for reuse

↳ optimal substructure

↳ extra space taken = (n+1)(m+1)

1. Brute Force Algo $O(2^n)$

↳ check every possibity

↳ 0 - Absence, 1 Present)

ob1	ob2	ob3	
0	0	0	Not possible
0	0	1	60
0	1	0	25
0	1	1	85 → ANS
1	0	0	20
1	0	1	30
1	1	0	45
1	1	1	Not possible

```
KnapSack(v, w, n, W)
{
    for (w = 0 to W) V[0, w] = 0;
    for (i = 1 to n)
        for (w = 0 to W)
            if ((w[i] ≤ w) and (v[i] + V[i − 1, w − w[i]] > V[i − 1, w]))
            {
                V[i, w] = v[i] + V[i − 1, w − w[i]];
                keep[i, w] = 1;
            }
            else
            {
                V[i, w] = V[i − 1, w];
                keep[i, w] = 0;
            }
    K = W;
    for (i = n downto 1)
        if (keep[i, K] == 1)
        {
            output i;
            K = K − w[i];
        }
    return V[n, W];
}
```

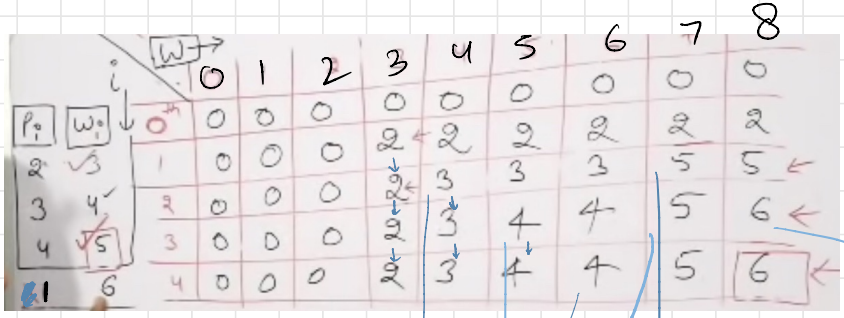
Dynamic 0/1 knapsack

0/1 KNAPSACK(n, m) — n: no of elements, m: capacity

max { 01 KS (n-1), (m - w_n) + P_n → adding in KS (weight, profit)
      01 KS (n-1), (m) → not adding KS (rest passed after item removed)

01 KS (n-1), (m) $w_n > m$ → if weight > capacity

n=0 OR m=0 → termination



max(3 + 0, 2) = 3 (P, 4−4 = w_i, above value)

max(3 + 2, 2) = 5 (7−4=3 at upper row)

max(4 + 0, 3) = 4 (5−5)

max(4 + 0, 5) = 5 (7−5)

max(4 + 2, 5) = 6 (8−5)

max(1 + 0, 4) max(

(4, 8) = 6

x_1	x_2	x_3	x_4	
0	0	1	0	6−4 = 2
1	0	1	0	2−2 = 0

V[i, w] = max { V[i-1, w], V[i-1, w-w[i]] + P[i] }

V[i-1, w]: not adding; V[i-1, w-w[i]]: adding

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

$V[i,w]$	0	1	2	3	4	5	6	7	8	9	10
i / w	0	0	0	0	0	0	0	0	0	0	0
10 5	1	0	0	0	0	0	10	10	10	10	10
40 4	2	0	0	0	0	40	40	40	40	40	50
30 6	3	0	0	0	0	40	40	40	40	50	70
50 3	4	0	0	0	50	50	50	50	90	90	90

→ w

use on 4th, 3rd row

$$V[i,w] = \max\{\underbrace{V[i-1,w]}_{\text{not adding}}, \underbrace{V[i-1,w-w[i]]}_{\text{adding}} + P[i]\}$$

x_1	x_2	x_3	x_4
0	1	0	1

90 - 50 = 40

40 - 40 = 0

$V(4,10) = 90$

Dynamic Matrix Chain Multiplication $O(n^3)$

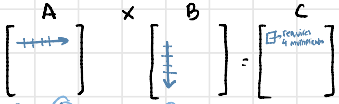
↳ multiplied in pairs

↳ columns of A = rows of B

↳ find pair giving min cost

↳ trying all possibilities ??

↳ tabulation method → bottom approach



columns of A = rows of B

rows of A, columns of B

Q) $A_1 \times A_2 \times A_3 \times A_4$ → which pair to select that will give min multiplications?

A_1: 5x4 A_2: 4x6 A_3: 6x2 A_4: 2x7

$m(1,1)$ → A_1

$m(2,2) = 0$ as not multiplied with anything → A_2

...

$m(1,2)$ s=1 → $A_1 A_2$ → 5x4 4x6 = 120

$m(2,3)$ s=2 → $A_2 A_3$ → 4x6 6x2 = 48

$m(3,4)$ s=3 → $A_3 A_4$ → 6x2 2x7 = 84

m	1	2	3	4
1	0	120	88	158
2		0	48	104
3			0	84
4				0

(i = row, j = column)

s	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

$s(1,4) = 3$ → $(A_1 A_2 A_3)(A_4)$

$s(1,3) = 1$ → $((A_1)(A_2 A_3))A_4$

$s(2,3) = 2$

$m(1,3)$ → $A_1 A_2 A_3$

- $A_1(A_2 A_3)$: 5x4 4x6 6x2 → $m(1,1) + m(2,3) + 5 \times 4 \times 2$ → $0 + 48 + 40$ → 88 (s=1) ↓ min
- $(A_1 A_2)A_3$: 5x4 4x6 6x2 → $m(1,2) + m(3,3) + 5 \times 6 \times 2$ → $120 + 0 + 60$ → 180

$m(2,4)$

- $A_2(A_3 A_4)$: 4x6 6x2 2x7 → $m(2,2) + m(3,4) + 4 \times 6 \times 7$ → $0 + 84 + 168$ → 252
- $(A_2 A_3)A_4$: 4x6 6x2 2x7 → $m(2,3) + m(4,4) + 4 \times 2 \times 7$ → $48 + 0 + 56$ → 104 (s=3) ↓ min

$m(1,4)$: min { $A_1 \times A_2 \times A_3 \times A_4$ (5x4 4x6 6x2 2x7): $m(1,1) + m(2,4) + 5 \times 4 \times 7$ → $0 + 104 + 140$ → 244 ,

$A_1 \times A_2 \times A_3 \times A_4$ (5x4 4x6 6x2 2x7): $[m(1,2) + m(3,4) + 5 \times 6 \times 7]$ → $120 + 84 + 210$ → 414 ,

$A_1 \times A_2 \times A_3 \times A_4$ (5x4 4x6 6x2 2x7): $[m(1,3) + m(4,4) + 5 \times 2 \times 7]$ → $88 + 0 + 70$ → 158 (s=3) ↓ min }

```
MATRIX-CHAIN(p, N)
 1  for i = 1, N
 2  do m[i, i] ← 0
 3  for L = 2, N
 4  do
 5     for i = 1, n − L + 1
 6     do j ← i + L − 1
 7        m[i, j] ← ∞
 8        for k = 1, j − 1
 9        do t ← m[i, k] + m[k + 1, j] + p_{i−1} · p_k · p_j
10           if (t < m[i, j])
11              then m[i, j] ← t;  s[i, j] ← k
```

Analysis: There are three nested loops. Each loop exe
$\Theta(n^3)$.

$A_1 \times A_2 \times A_3 \times A_4$ (5x4 4x6 6x2 2x7)

P_0 P_1 P_2 P_3 P_4

Dynamic Longest common seqvence (LCS) O(mn)

↳ same order, dont have to be together

↳ make table

↳ matching: 1 + left diagonal

↳ not matching: max

String 1: a b c d e f g h i j

String 2 : c d g i → longest

d g i

g i

String 1: a b d a c e

String 2 : b a b c e

LCS 1 : b c e

LCS 2 : a b c e → biggest

String 1: a b c d e f g h i j

String 2 : e c d g i ✗ not same order

LCS 1 : e g i

LCS 2 : c d g i → longest so answer

a b c d b d

Q) String 1: s t o n e

string 2: l o n g e s t

		l	o	n	g	e	s	t
	0	0	0	0	0	0	0	0
s	0	0	0	0	0	0	1	1
t	0	0	0	0	0	0	1	2
o	0	0	1	1	1	1	1	2
n	0	0	1	2	2	2	2	2
e	0	0	1	2	2	3	3	3

→ LCS length = 3

LCS = o n e

↳ only arrows going diagonal are the sequence

Q) A: b d B: a b c d

		a	b	c	d
	0	0	0	0	0
b	0	0	1	1	1
d	0	0	1	1	2

→ length of LCS

LCS = b d = 2

↳ only arrows going diagonal are the sequence

scs

d b a

a b d

The recursive definition of Fibonacci numbers gives us a recursive algorithm for computing them:

```
FIB(n)
1  if (n < 2)
2     then return n
3     else  return FIB(n − 1) + FIB(n − 2)
```

This will take time approximately T(n) = 2T(n-1) + 1 which after solving through iterative substitution, we get exponential time. So this recurrence will be taking huge amount of time. Now we will analyze the tree to avoid un-necessary re-computation.

Fibonacci Sequence: Dynamic Programming

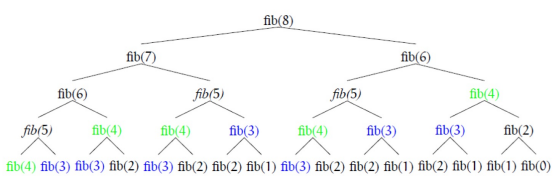
A single recursive call to fib(n) results in one recursive call to fib(n − 1), two recursive calls to fib(n − 2), three recursive calls to fib(n − 3), five recursive calls to fib(n − 4) and, in general, F_{k-1} recursive calls to fib(n − k) For each call, we're recomputing the same fibonacci number from scratch.

We can avoid this unnecessary repetitions by writing down the results of recursive calls and looking them up again if we need them later. This process is called *memoization*. Here is the algorithm with memoization.

```
ITERFIB(n)
1  F[0] ← 0
2  F[1] ← 1
3  for i ← 2 to n
4  do
5     F[i] ← F[i − 1] + F[i − 2]
6  return F[n]
```

Fibonacci Sequence: Dynamic Programming

Recursive tree of Fib(8) :



GEOMETRIC ALGORITHMS

↓
Pseudocode

1. Convex HULL
2. Line Intersection
3. Closest Pair
4. CCW → imp

CONVEX HULL

↳ smallest convex set containing all points

↳ Applications: robot motion planning, collision detection

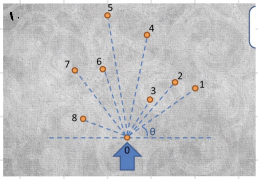
1. Graham scan algo $O(n \log n)$

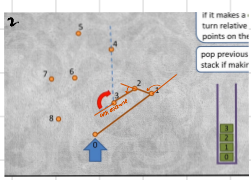
↳ Select point with lowest y coordinate

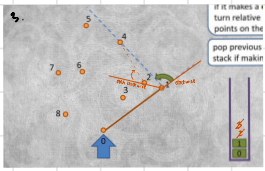
↳ sort points by angle relative to ↑

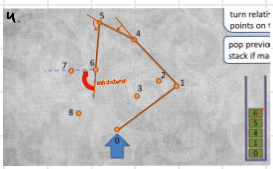
↳ if point counterclockwise relative to prev vertex
then add to stack

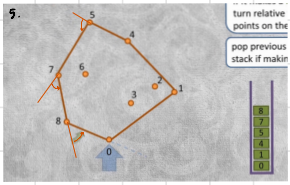
↳ if point anticlockwise relative to prev vertex
then pop off stack till clockwise achieved











Find lowest point p0
Sort all other points angularly about p0,
break ties in favor of closeness to p0;
label them p1, p2, ..., pn-1
Stack S = (pn-1, p0) = (pt-1, pt); t indexes top
i ? 1
while i < n do
if pi is strictly left of (pt-1, pt) then
Push(S, i); i++
else Pop(S)

GRAHAM-SCAN(Q)

1 let p_0 be the point in Q with the minimum y-coordinate, or the leftmost such point in case of a tie
2 let $\langle p_1, p_2, \ldots, p_m \rangle$ be the remaining points in Q, sorted by polar angle in counterclockwise order around p_0 (if more than one point has the same angle, remove all but the one that is farthest from p_0)
3 let S be an empty stack
4 PUSH(p_0, S)
5 PUSH(p_1, S)
6 PUSH(p_2, S)
7 for $i = 3$ to m
8 while the angle formed by points NEXT-TO-TOP(S), TOP(S), and p_i makes a nonleft turn
9 POP(S)
10 PUSH(p_i, S)
11 return S

2. Jarvis March algo $O(nh)$ aka Gift wrapping

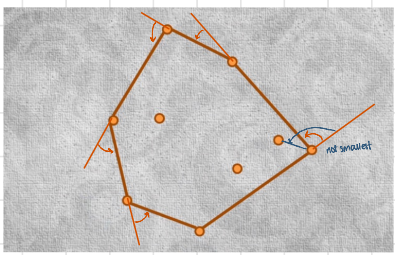
h = no of vertices

$O(n^2)$ worst case when too many vertices

↳ Select point with lowest y coordinate

↳ select point with smallest counterclockwise relative to prev vertex

↳ repeat until starting vertex reached



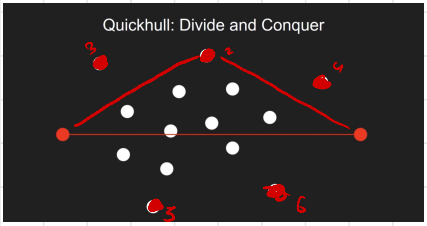
p ? the lowest point p0
repeat
for each q ? P and q ? p do
compute counterclockwise angle ? from previous hull edge
let r be the point with smallest ?
output (p, r) as a hull edge
p ? r
until p = p0

Find the lowest point (smallest y coordinate)
Let i_0 be its index, and set $i \leftarrow i_0$
repeat
for each $j \neq i$ do
compute counterclockwise angle θ from previous hull edge
Let k be the index of the point with the smallest θ
Output (p_i, p_k) as a hull edge
$i \leftarrow k$
until $i = i_0$

3. Quick Hull (Divide and conquer

$O(n \log n) \rightarrow$ best

$O(n^2) \rightarrow$ worst



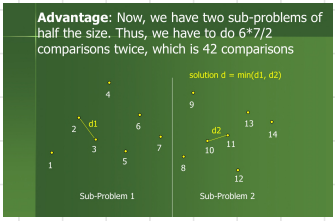
Sol

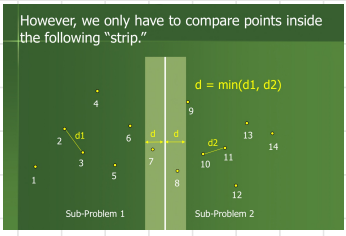
CLOSEST PAIR POINTS

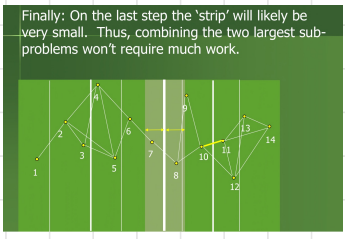
$$d(A,B) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

1. Divide and Conquer $O(n \log n)$

- sort points
- Divide: split points at the middle
- Conquer: find closest pair on each side recursively
  - ↳ continue splitting till 1 comparision left
- Combine: combine solutions
  - ↳ closest pair left $T(n/2)$
  - ↳ closest pair right $T(n/2)$
  - ↳ closest pair at partition < $d = \min(L, R)$ $O(n)$
- Return: best of 3



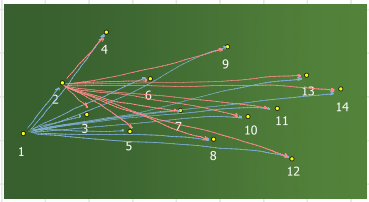




no of comparisions = $\frac{(6.02)(7.02)}{2} = 22$

2. Brute force $O(n^2)$

- ↳ compare every point with every other point
- ↳ no of comparisons = $\sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$



no of comparisions = $\frac{13 \times 14}{2} = 91$

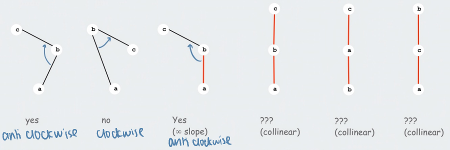
Divide and conquer

```
ClosestPair(Px, Py):
   if |Px| == 2: return dist(Px[1],Px[2])     // base

   δ1 = ClosestPair(FirstHalf(Px,Py))      // divide
   δ2 = ClosestPair(SecondHalf(Px,Py))
   δ = min(δ1,δ2)

   Sy = points in Py within δ of |         // merge
   For i = 1,...,|Sy|:
       For j = (i+1),.....,(i+7)
           δ = min( dist(Sy[i], Sy[j]), δ )
   Return δ
```

```
Closest-Pair(p1, ..., pn)
{
   Compute separation line L such that half the points
   are on one side and half on the other side.                    O(N log N)

   δ1 = Closest-Pair(left half)
   δ2 = Closest-Pair(right half)                                  2T(N / 2)
   δ  = min(δ1, δ2)

   Delete all points further than δ from separation line L        O(N)

   Sort remaining points by y-coordinate.                         O(N log N)

   Scan points in y-order and compare distance between
   each point and next 11 neighbors. If any of these              O(N)
   distances is less than δ, update δ.

   return δ.
}
```

counter clockwise (ccw) → IMP

CCW. Given three point a, b, and c, is a-b-c a counterclockwise turn?
- Analog of comparisons in sorting.
- Idea: compare slopes.

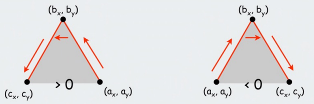


Lesson. Geometric primitives are tricky to implement.
- Dealing with degenerate cases.
- Coping with floating point precision.

CCW. Given three point a, b, and c, is a-b-c a counterclockwise turn?
- Determinant gives twice area of triangle.

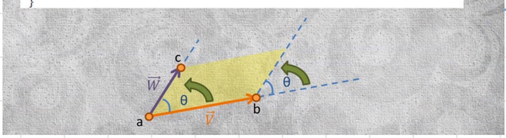
$$2 \times Area(a,b,c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

- If area > 0 then a-b-c is counterclockwise.
- If area < 0, then a-b-c is clockwise.
- If area = 0, then a-b-c are collinear.



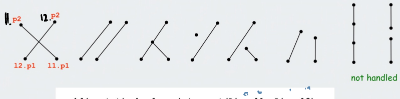

```
int ccw(Point a, Point b, Point c) {
    float area = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);

    if (area < 0) return -1; // clockwise

    if (area > 0) return 1; // counter-clockwise

    return 0; // collinear
}
```

→ used in convex hull

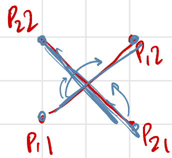


Line Intersection

Intersect: Given two line segments, do they intersect?
- Idea 1: find intersection point using algebra and check.
- Idea 2: check if the endpoints of one line segment are on different "sides" of the other line segment.
- 4 ccw computations.



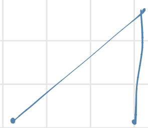

```
public static boolean intersect(Line l1, Line l2)
{
   int test1, test2;
   test1 = Point.ccw(l1.p1, l1.p2, l2.p1)
           * Point.ccw(l1.p1, l1.p2, l2.p2);
   test2 = Point.ccw(l2.p1, l2.p2, l1.p1)
           * Point.ccw(l2.p1, l2.p2, l1.p2);
   return (test1 <= 0) && (test2 <= 0);
}
```



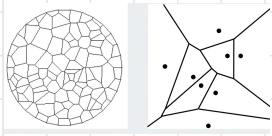
Test 1

ccw (P11, P12, P22) *
ccw (P11, P12, P21)

Test



VORONI DIGRAMS



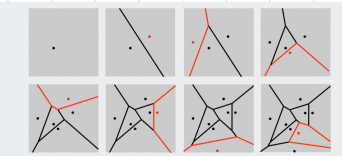
Vornoni Diagram: shows a graph contained in a shaded region

Vornoi Region: set of all points closest to a given point

Construct Vornoi Diagram

↳ draw an edge equidistant to the points

perpendicular bisector



- Running time: O(N log N) on average.

Applications

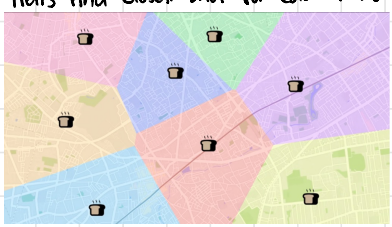
Toxic waste dump problem. N homes in a region. Where to locate nuclear power plant so that it is far away from any home as possible?

looking for largest empty circle (center must lie on Voronoi diagram)

Path planning. Circular robot must navigate through environment with N obstacle points. How to minimize risk of bumping into a obstacle?

robot should stay on Voronoi diagram of obstacles

Helps find closest shop for each region



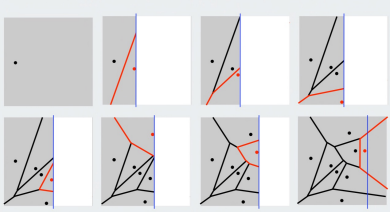
aka sweep line algo

Fortunes Algo O(n log n)

↳ Properly handles degenaracies

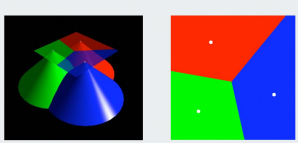
↳ Properly handles floating point computations

Presort points on x-coordinate
Eliminates point location problem



Hoff's algorithm. Align apex of a right circular cone with sites.

- Minimum envelope of cone intersections projected onto plane is the Voronoi diagram.
- View cones in different colors ⇒ render Voronoi.



Implementation. Draw cones using standard graphics hardware!

https://www.cs.unc.edu/~geom/voronoi/siggraph_paper/voronoi.pdf

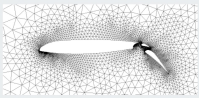
PRIMITIVE OPERATIONS

Primitive operations.

1. Is a point inside a polygon?
2. Compare slopes of two lines.
3. Distance between two points.
4. Do two line segments intersect?
5. Given three points p_1, p_2, p_3, is p_1-p_2-p_3 a counterclockwise turn?

Applications.

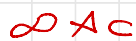
- Data mining.
- VLSI design.
- Computer vision.
- Mathematical models.
- Astronomical simulation.
- Geographic information systems.
- Computer graphics (movies, games, virtual reality).
- Models of physical world (maps, architecture, medical imaging).



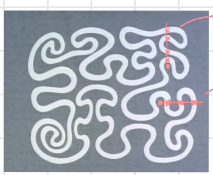
airflow around an aircraft wing

Reference: http://www.ics.uci.edu/~eppstein/geom.html

1. Jordan Curve Theorem

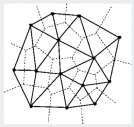
↳ a closed curve  

1. if a line crosses the curve
   - ↳ odd no times → point is inside
   - ↳ even no of times → point lies outside



Delaunay triangulation. Triangulation of N points such that no point is inside circumcircle of any other triangle.

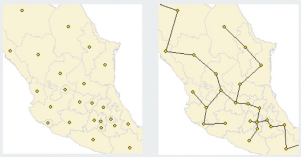
Fact 0. It exists and is unique (assuming no degeneracy).
Fact 1. Dual of Voronoi (connect adjacent points in Voronoi diagram).
Fact 2. No edges cross ⇒ O(N) edges.
Fact 3. Maximizes the minimum angle for all triangular elements.
Fact 4. Boundary of Delaunay triangulation is convex hull.
Fact 5. Shortest Delaunay edge connects closest pair of points.



Delaunay
Voronoi

59

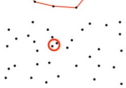
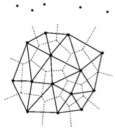
Euclidean MST. Given N points in the plane, find MST connecting them.
- Distances between point pairs are Euclidean distances.



Brute force. Compute $N^2 / 2$ distances and run Prim's algorithm.
Ingenuity.
- MST is subgraph of Delauney triagulation
- Delauney has O(N) edges
- Compute Delauney, then use Prim or Kruskal to get MST in O(N log N) !

60

Geometric algorithms summary: Algorithms of the day

		asymptotic time to solve a 2D problem with N points	
		brute	ingenuity
convex hull		N^2	N log N
closest pair		N^2	N log N
Voronoi/Delauney		N^4	N log N
Euclidean MST		N^2	N log N

62

SEARCHING / SHORTEST PATH

Uniformed search

Breadth-First-Search (BFS)

QUEUE FIFO

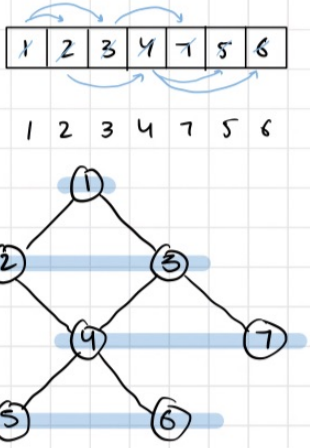
move!

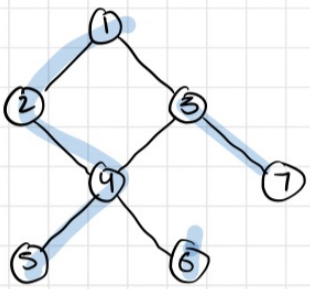
- Start from top, put in Queue
- Put adj vertices in Queue
- take out vertice when explored (take it out of queue)
- move to next vertice in Queue
- Repeat till all vertices explored

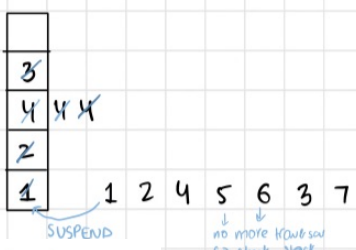
Depth First Search (DFS)

stack LIFO

- Start from top, write it
- move to adj vertex
- suspend prev vertex to stack
- Repeat till no adj vertex
- backtrack by checking stack
- Repeat till no vertices left







- **BFS**(s)

```
visit(s);
queue.insert(s);
while( queue is not empty ){
    u = queue.extractHead();
    for each edge <u, d>{
        if(d has not been visited)
            visit(d);
            queue.insert(d);
    }
}
```

Depth-First Search

1. Start
2. Push Root Node to Stack
   - Mark Root Node as Visited
   - Print Root Node as Output
3. Check Top of the Stack If Stack is Empty, Go to Step 6
4. Else, Check Adjacent Top of the Stack
   - If Adjacent is not Visited
   - Push Node to Stack
   - Mark Node as Visited
   - Print Node as Output
   - Else Adjacent Visited
5. Go to Step 3
6. Stop

- **DFS**(s):

```
s.underDFS = true;   // grey
for each edge <s, d>{
    if(! d.underDFS and d has not been visited)
        DFS(d);
}
Visit(s); // black
```

From the deepest one to the current one

Search Strategy	BFS	DFS
1. Completeness ↳ if there is a solution, will it be found?	yes	no for infinite spaces (loops) yes for finite spaces
2. Time complexity ↳ time to find solution	$O(b^{d+1})$ d: depth of goal b: branching factor	$O(b^m)$ → terrible if $m > d$ m: max depth of state space
3. Space complexity ↳ memory required to search	$O(b^{d+1})$	$O(bm)$
4. Optimality ↳ will best solution be found	yes	NO
5. Suitability	not suitable for large graphs memory req may be excessive	not suitable for large graphs will take too much time to even come close
6. Method	checks all alternatives	goes into one branch until it reaches leaf node uses lesser space than BFS

Suppose the branching factor b=10, and the goal is at depth d=12:

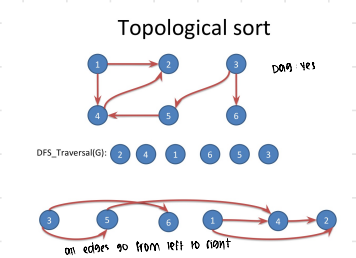
– Then we need $O10^{12}$ time to finish. If O is 0.001 second, then we need 1 billion seconds (31 year). And if each O costs 10 bytes to store, then we also need 1 terabytes. BFS

Topological order

↳ list of vertices in a dag (directed acyclic graph; no cycles)

↳ all edges go left to right

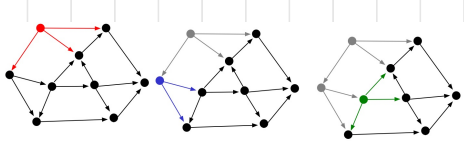
↳ reverse DFS topologically sorts a dag



Topological sort

Implementation

- Start with a list of nodes with in-degree = 0
- Select any edge from list
  - mark as deleted
  - mark all outgoing edges as deleted
  - update in-degree of the destinations of those edges
    - If any drops to zero, add to the list
- Running time? In all, algorithm does work:
  - O(|V|) to construct initial list
  - Every edge marked "deleted" at most once: O(|E|) total
  - Every node marked "deleted" at most once: O(|V|) total
  - So linear time overall (in |E| and |V|)

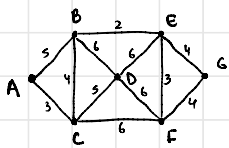


MIN COST OF SPANNING TREE

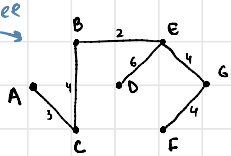
Spanning Tree

- same no of vertices
- edges = n-1
- min edges possible
- connected with every node
- no cycle

e)



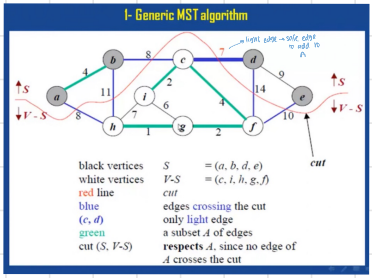
spanning tree



1. Generic Algo

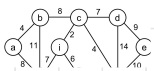
growing a MST

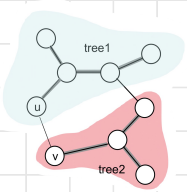
- make a cut → Partition into 2 sets (S, V-S)
- find all edge crossings → edges crossing the cut
- find light edge → lowest edge crossing
- Add light edge to A → Or its a safe edge



Generic MST algorithm

1. A ← ∅
2. **while** A is not a spanning tree
3. **do** find an edge (u, v) that is safe for A
4. A ← A ∪ {(u, v)}
5. **return** A

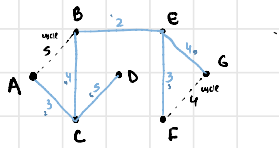




3. Kushkal Algo

→ edge based → greedy Algo

- ↳ strat with smallest edge
- ↳ disconnectedly mark smallest edge
- ↳ no cycles
- ↳ edges : n-1

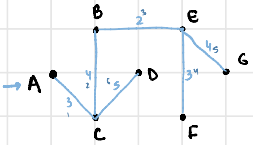


MST = 21

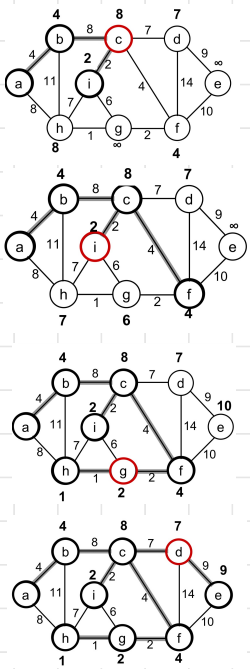
2. Prims Algo

→ node based → greedy Algo

- ↳ start with any node
- ↳ connectedly move to smallest neighbour
- ↳ no cycles
- ↳ edges : n-1



MST = 21



COMPARISIONS	KUSHKALS	PRIMS
1. Sparse graphs, $E = O(V)$ — Less no of edges	$O(V \lg V)$	binary heap: $O(V \lg V)$ Fibonacci heap: $O(V \lg V)$
2. dense graphs, $E = O(V^2)$ — many no of edges	$O(V^2 \lg V)$	binary heap: $O(V^2 \lg V)$ Fibonacci heap: $O(V^2)$
3. Overall running time	$O(E \lg V)$	$O(E \lg V)$
4. Negative weights	Still work	Still work
5. Method	Grows multiple trees at the same time Trees are merged using safe edges	Grows one tree all the time

```
KRUSKAL(G)
1  for all v ∈ V                          ┐ V call to MAKE SET → O(V)
2          MAKESET(v)                     ┘
3  T ← {}
4  sort the edges of E by weight          → O(E lg E)
5  for all edges (u, v) ∈ E in increasing order of weight   → O(E)
6          if FIND-SET(u) ≠ FIND-SET(v)   → 2E calls to Find set
7                  add edge to T                            → O(lg V)
8                  UNION(FIND-SET(u), FIND-SET(v))          → V calls to MAKE UNION
```

```
PRIM(G, r)
1  for all v ∈ V                          ┐
2          key[v] ← ∞                     │ → O(V)
3          prev[v] ← null                 │
4  key[r] ← 0                             ┘
5  H ← MAKEHEAP(key)                      → O(V)
6  while !Empty(H)
7          u ← EXTRACT-MIN(H)             → V calls to Extract min
8          visited[u] ← true
9          for each edge (u, v) ∈ E
10                 if !visited[v] and w(u, v) < key(v)
11                         DECREASE-KEY(v, w(u, v))   → E calls to decrease key
12                 prev[v] ← u
```

Correctness of Kruskal's

- Never adds an edge that connects already connected vertices
- Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST

Correctness of Prim's?

- Can we use the min-cut property?
  - Given a partion S, let edge *e* be the minimum cost edge that **crosses** the partition. *Every* minimum spanning tree contains edge *e*.
- Let S be the set of vertices visited so far
- The only time we add a new edge is if it's the lowest weight edge from S to V-S

Running time

linked lists : $O(VE)$

Running time — same as Dijkstras

Array : $O(V^2)$

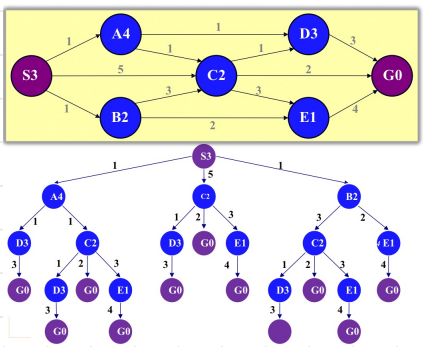
Bin heap : $O(E \log V)$

Fib heap : $O(V \log V + E)$

Search Tree Vs Graph Tree

BASIS FOR COMPARISON	TREE	GRAPH
Path	Only one between two vertices.	More than one path is allowed.
Root node	It has exactly one root node.	Graph doesn't have a root node.
Loops	No loops are permitted.	Graph can have loops.
Complexity	Less complex	More complex comparatively
Traversal techniques	Pre-order, In-order and Post-order.	Breadth-first search and depth-first search.
Number of edges	n-1 (where n is the number of nodes)	Not defined
Model type	Hierarchical	Network

Traversing a Graph as Tree



- A tree is generated by traversing the graph.
- The same node in the graph may appear repeatedly in the tree.
- The arrangement of the tree depends on the traversal strategy (search method)
- The initial state becomes the root node of the tree
- In the fully expanded tree, the goal states are the leaf nodes.
- Cycles in graphs may result in infinite branches

Operations on Disjoint Data Sets

- MAKE-SET(u) – creates a new set whose only member is u
- FIND-SET(u) – returns a representative element from the set that contains u
  - Any of the elements of the set that has a particular property
  - *E.g.:* S_u = {r, s, t, u}, the property is that the element be the first one alphabetically

    FIND-SET(u) = r FIND-SET(s) = r
  - FIND-SET has to return the same value for a given set

Operations on Disjoint Data Sets

- UNION(u, v) – unites the dynamic sets that contain u and v, say S_u and S_v
  - *E.g.:* S_u = {r, s, t, u}, S_v = {v, x, y}

    UNION (u, v) = {r, s, t, u, v, x, y}
- Running time for FIND-SET and UNION depends on implementation.
- Can be shown to be **α(n)=O(lgn)** where **α()** is a very slowly growing function (see **Chapter 21**)

SINGLE SOURCE SHORTEST PATH

Dijkstras Algo → greedy

↳ longest path from s to any node = V-1

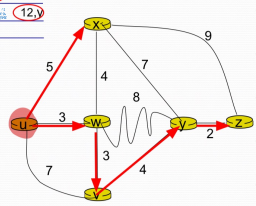
↳ doesn't work for negative weights

Step	N'	D(v)	D(w)	D(x)	D(y)	D(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					

5 vertices so 6 columns

notes:

- construct shortest path tree by tracing predecessor nodes
- ties can exist (can be broken arbitrarily)



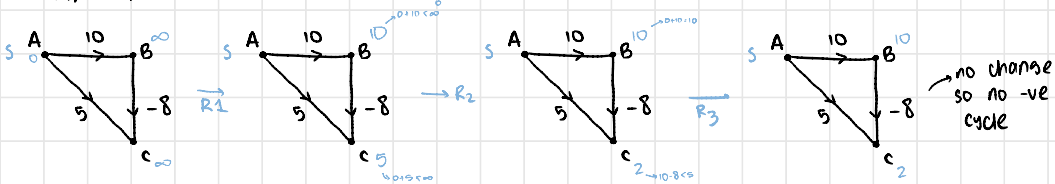
Bellman-Ford Algo

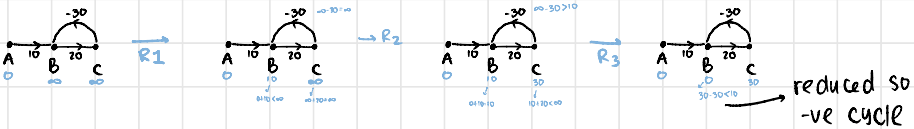
↳ works for -ve weights

↳ relax each node: V-1 times

↳ find if negative cycle: if reduced ans when relaxed V times → means not optimal solution

Relax: 3-1 = 2





Dijkstra's algorithm

```
DIJKSTRA(G, s)
1  for all v ∈ V
2          dist[v] ← ∞
3          prev[v] ← null
4  dist[s] ← 0
5  Q ← MAKEHEAP(V)                    → O(V)
6  while !EMPTY(Q)                    → V iterations
7          u ← EXTRACTMIN(Q)          → V calls
8          for all edges (u, v) ∈ E
9                  if dist[v] > dist[u] + w(u, v)
10                         dist[v] ← dist[u] + w(u, v)
11                         DECREASEKEY(Q, v, dist[v])   O(E) calls
12                         prev[v] ← u
```

Bellman-Ford algorithm

```
BELLMAN-FORD(G, s)
1  for all v ∈ V                      ← initialize all distances
2          dist[v] ← ∞
3          prev[v] ← null
4  dist[s] ← 0
5  for i ← 1 to |V| − 1
6          for all edges (u, v) ∈ E                    ← traverse and update all node/edges
7                  if dist[v] > dist[u] + w(u, v)
8                          dist[v] ← dist[u] + w(u, v)
9                          prev[v] ← u
10 for all edges (u, v) ∈ E            ← check -ve cycles
11         if dist[v] > dist[u] + w(u, v)
12                 return false
```

correctness

- Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v
  - The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex
  - Therefore, there cannot be any other path that hasn't been visited already that would result in a shorter path

correctness

- Loop invariant: After iteration i, all vertices with shortest paths from s of length i edges or less have correct distances

Running time

Array : $O(V^2)$

Bin heap : $O((V+E)\log V)$ $O(E \log V)$

Fib heap : $O(V \log V + E)$

Running time

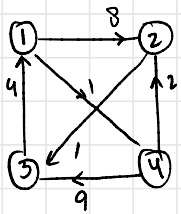
↳ $O(VE)$

```
PRIM(G, r)
1  for all v ∈ V
2          key[v] ← ∞
3          prev[v] ← null
4  key[r] ← 0
5  H ← MAKEHEAP(key)
6  while !Empty(H)
7          u ← EXTRACT-MIN(H)
8          visited[u] ← true
9          for each edge (u, v) ∈ E
10                 if !visited[v] and w(u, v) < key(v)
11                         DECREASE-KEY(v, w(u, v))
12                         prev[v] ← u
```

```
DIJKSTRA(G, s)
1  for all v ∈ V
2          dist[v] ← ∞
3          prev[v] ← null
4  dist[s] ← 0
5  Q ← MAKEHEAP(V)
6  while !EMPTY(Q)
7          u ← EXTRACTMIN(Q)
8          for all edges (u, v) ∈ E
9                  if dist[v] > dist[u] + w(u, v)
10                         dist[v] ← dist[u] + w(u, v)
11                         DECREASEKEY(Q, v, dist[v
12                         prev[v] ← u
```

Floyd Warshall $O(V^3)$

↳ all pairs shortest path



D^0

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	∞	0	∞
4	∞	2	9	0

↓ no direct edge

can use vertex 1

$D^{1=2}$

	1	2	3	4
1	0	8	∞	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	9	0

↓ remains same

can use vertex 1,2

D_2^2

	1	2	3	4
1	0	8	9	1
2	∞	0	1	∞
3	4	12	0	5
4	∞	2	3	0

$D_3^{=1}$

	1	2	3	4
1	0	8	9	1
2	5	0	1	6
3	4	12	0	5
4	7	2	3	0

can use vertex 1,2,3

D_4

	1	2	3	4
1	0	3	4	1
2	5	0	1	6
3	4	7	0	5
4	7	3	2	0

can use 1,2,3,4

Floyd-Warshall

```
FLOYD-WARSHALL(W)
1. n ← rows[W]
2. D(0) ← W
3. for k ← 1 to n
4.     do for i ← 1 to n
5.            do for j ← 1 to n
6.                   d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7. return D(n)
```

- Running time: $O(n^3)$
  - Better than running BF n times!
  - Not really better than running Dijkstra n times.
    - But it's simpler to implement and handles negative weights.

How can we find $D^{(k)}[u,v]$ using $D^{(k-1)}$?

- $D^{(k)}[u,v] = \min\{ D^{(k-1)}[u,v], D^{(k-1)}[u,k] + D^{(k-1)}[k,v] \}$

Case 1: Cost of shortest path through {1,...,k-1}

Case 2: Cost of shortest path from u to k and then from k to v through {1,...,k-1}

COMPUTATIONAL COMPLEXITY, NP COMPLETENESS

SOLUTION OF ALGORITHM

1. Polynomial Time
2. Non Polynomial Time

Polynomial Time $O(n^k)$ → constant

Exponential time $O(2^n)$

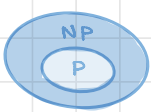
Polynomial Time	Exponential Time
Linear Search — n	0/1 Knapsack — 2^n
Binary Search — $\log n$	Traveling SP — 2^n
Insertion Sort — n^2	Sum of Subsets — 2^n
Merge Sort — $n \log n$	Graph Coloring — 2^n
Matrix Multiplication — n^3	Hamiltonian Cycle — 2^n

Solves way faster than exponential

show similarities so if 1 problem solved, all problem solved

P Problems → is also NP, $P \subseteq NP$

- Deterministic in Polynomial time
- Problems solved in Polynomial time



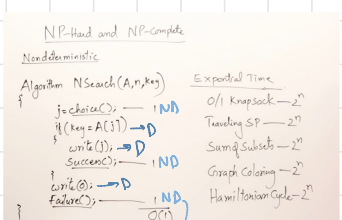
- easy to solve
- easy to verify
- polynomial time
- tractable

NP Problems

- Non-Deterministic Polynomial time
- Problems not solved but verified in polynomial time

- hard to solve
- easy to verify
- exponential time
- intractable

Polynomial time ND Algo → NP



Solved by someone else in future

- Polynomial-time verification can be used to easily tell if a problem is a NP problem
- E.g.:
  - Sorting, n-integers
    - A candidate: an array
    - Verification: scan it once
  - Max-heapify, n-nodes:
    - A candidate: a complete binary search tree
    - Verification: scan all the nodes once
  - Find all the sub sets of a given set A, |A|=n
    - A candidate: a set of set
    - Verification: check each set

Proving P = NP, then

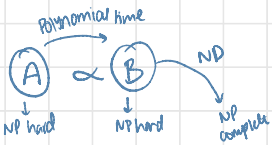
↳ Problem can be solved in polynomial time

Proving P ≠ NP, then

↳ there are some problems that can never be solved

Reduction Properties

1. if $A \rightarrow B$, $B \rightarrow P$ (P: reducable)
   then $A \rightarrow P$
2. $A \not\rightarrow P$ then
   $B \not\rightarrow P$



NP Hard Problems

↳ every NP problem can be polynomial reduced to it

↳ $A \propto B$ (A: NP hard)

↳ harder/equal to NPC problems

↳ e.g. max, min → optimization problems
e.g. shortest path

NP Complete problem (NPC)

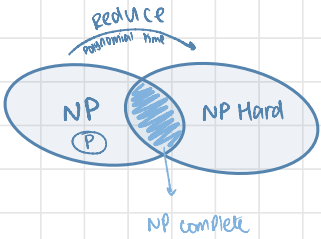
↳ a hardest NP problem

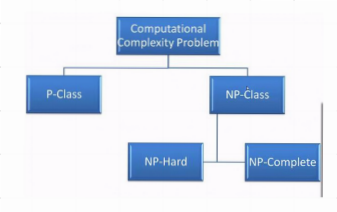
↳ it has no polynomial time algos

↳ harder/equal to NP problems

↳ intersection of NP and NP hard

↳ e.g. decisions graph





Why we study NPC?

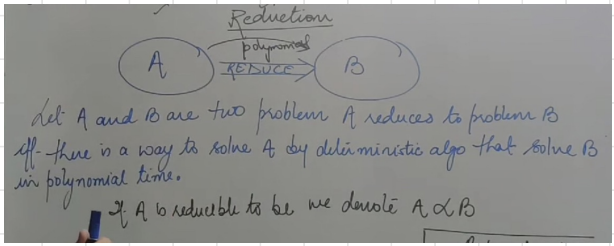
- One of the most important reasons is:
  - If you see a problem is NPC, you can stop from spending time and energy to develop a fast polynomial-time algorithm to solve it.
- Just tell your boss it is a NPC problem
- How to prove a problem is a NPC problem?

How to prove a problem is a NPC problem?

- A common method is to prove that it is not easier than a known NPC problem.
- To prove problem A is a NPC problem
  - Choose a NPC problem B
  - Develop a **polynomial-time algorithm translate A to B**
- A **reduction algorithm**
- If A can be solved in polynomial time, then B can be solved in polynomial time. It is contradicted with B is a NPC problem.
- So, A cannot be solved in polynomial time, it is also a NPC problem.

What if a NPC problem needs to be solved?

- Buy a more expensive machine and wait
  - (could be 1000 years)
- Turn to approximation algorithms
  - Algorithms that produce near optimal solutions



APPROXIMATION ALGOS FOR NPC

Approximation ratio P(n)

↳ These algos known as P(n)-approximation algo

$$P(n) \geq \max\left[\frac{C}{C^*}, \frac{C^*}{C}\right]$$

C^*: cost of optimal solution

C: cost of optimal solution by approx algo

a) Total weight of MST graph = 20

Total weight of MST graph with algo < 25

approximation ratio = $\frac{25}{20} = 1.25$

This is a 1.25 approximation algo

b) P(n) = 1

This is an algo that can always find a optimal solution

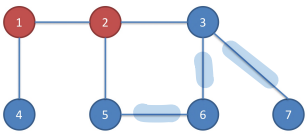
- If a problem is NP-complete, there is very likely no polynomial-time algorithm to find an optimal solution
- The idea of approximation algorithms is to develop polynomial-time algorithms to find a near optimal solution

- E.g.: develop a greedy algorithm without proving the greedy choice property and optimal substructure.
- Are those solution found near-optimal?
- How near are they?

VERTEX COVER PROBLEM

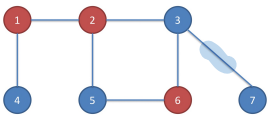
↳ given a undirected graph

↳ find a vertex cover with min size



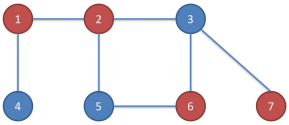
Are the red vertices a vertex-cover?

↳ NO as edges (5,6), (3,6), (3,7) aren't covered by them



Are the red vertices a vertex-cover?

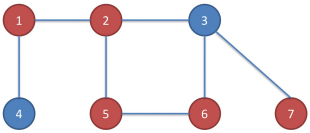
↳ NO as edge (3,7) isn't covered by it



Are the red vertices a vertex-cover?

↳ yes

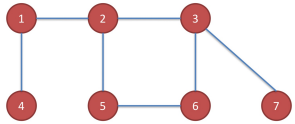
↳ size: 4



Are the red vertices a vertex-cover?

↳ yes

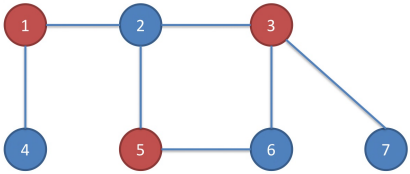
↳ size: 5



Are the red vertices a vertex-cover?

↳ yes

↳ size: 7



A minimum vertex-cover

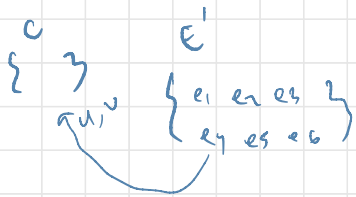
size: 3

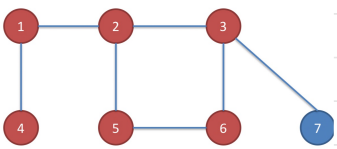
* Vertex cover problem is NP-complete
↳ has 2 approx polynomial time algo

2 Approximation Algorithm of Vertex Cover

- **APPROX-VERTEX-COVER**(G) → AVC

```
C = ∅;
E'=G.E;
while(E' ≠ ∅ ){
    Randomly choose a edge (u,v) in E', put u and v into C;
    Remove all the edges that covered by u or v from E'
}
Return C;
```

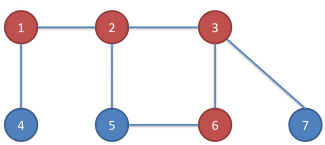




It is then a vertex cover

Size? 6

How far from optimal one? Max(6/3, 3/6) = 2



It is then a vertex cover

Size? 4

How far from optimal one? Max(4/3, 3/4) = 1.33

↳ Approx-vertex-cover (G) = 2 approximation algorithm
↳ as when the size of min vertex is s,
the vertex cover produced by AVC is at most 2s

Vertex-cover problem and a 2-approximation algorithm

Proof:

- Assume a minimum vertex-cover is U*
- A vertex-cover produced by **APPROX-VERTEX-COVER**(G) is U
- The edges chosen in **APPROX-VERTEX-COVER**(G) is A
- A vertex in U* can only cover 1 edge in A
  - So $|U^*| \geq |A|$
- For each edge in A, there are 2 vertices in U
  - So $|U| = 2|A|$
- So $|U^*| \geq |U|/2$
- So $\frac{|U|}{|U^*|} \leq 2$

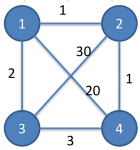
U^* = min VC
U = VC by AVC
A = edges chosen in AVC
↳ $|U^*| \geq |A|$
↳ $|U| = 2|A|$
$\therefore |U^*| \geq \frac{|U|}{2}$

$$\frac{|U|}{|U^*|} \leq 2$$

Traveling salesman problem (TSP)

↳ given a weighted undirected graph

↳ find min route



* TSP is NP-complete

↳ doesn't have polynomial time approx algo with constant approx ratio

↓

HOW TO SOLVE NPC ? ⟶ **Triangle inequality**

↳ $w(u,v) \leq w(u,v) + w(w,v)$ (weight)

- E.g.:
  - If all the edges are defined as the distance on a 2D map, the triangle inequality is true

* For TSPs where triangle inequality = true

↳ there is a 2 approx polynomial time algo

↓

2 Approximation Algorithm of TSP

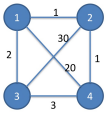
APPROX-TSP-TOUR(G)

Find a MST m;

Choose a vertex as root r;

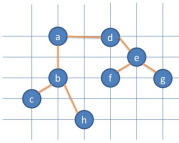
return preorderTreeWalk(m, r);

Can we apply the approximation algorithm on this one?



No. The triangle inequality is violated.

Traveling-salesman problem

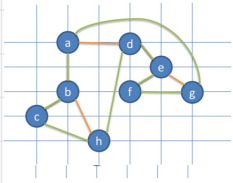


Use Prim's algorithm to get a MST

Choose "a" as the root

Preorder tree walk

a b c h d e f g



For any pair of vertices, there is a edge and the weight is the Euclidean distance

Triangle inequality is true, we can apply the approximation algorithm

The route is then...

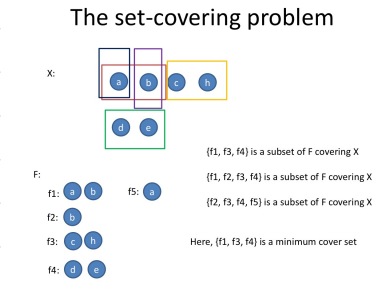
Because it is a 2-approximation algorithm

A TSP solution is found, and the total weight is at most twice as much as the optimal one

Set covering Problem

↳ Given Set X, family F of subsets of X

↳ find subset of F that cover X, with min size



✘ Set covering problem is NP complete

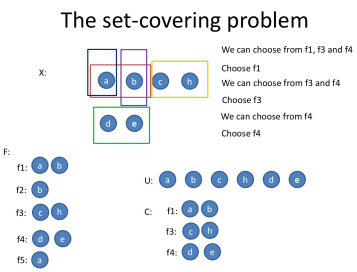
- If the size of the largest set in F is m, there is a $\sum_{i=1}^{m} 1/i$ - approximation polynomial time algorithm to solve it.

↳ there is a 2 approx polynomial time algo

2 Approximation Algorithm of Set covering Problem

GREEDY-SET-COVER(X, F)

```
U=X;
C=Ø;
While(U≠ Ø){
    Select S∈F that maximizes |S∩U|;
    U=U-S;
    C=CU{S};
}
return C;
```



Set Cover and its generalizations and variants are fundamental problems with numerous applications. Examples include:

- selecting a small number of nodes in a network to store a file so that all nodes have a nearby copy,
- selecting a small number of sentences to be uttered to tune all features in a speech-recognition model [11],
- selecting a small number of telescope snapshots to be taken to capture light from all galaxies in the night sky,
- finding a short string having each string in a given set as a contiguous sub-string.