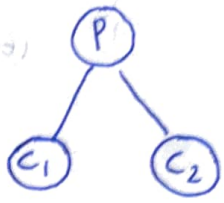
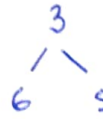
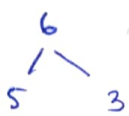
ra. 11. 2022

Binary Heap

- A spl type of tree data structure.
- 2 types – min heap ($P < C_1, C_2$)
  – max heap ($P > C_1, C_2$)



Eg:  is a min heap or 

 or  ⟹ max heap

Operations:

- insert() nlogn
- delete()
- extract-min() nlogn
- build() n

} min heap

in min heap, we add from L to R

Eg: ~~0~~ 5 3 2 7 9 6 8 1 4 [for insertion]

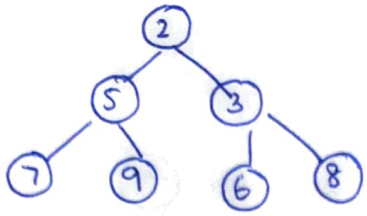
- Initially heap is empty. Add the 1st element

- Add 3.

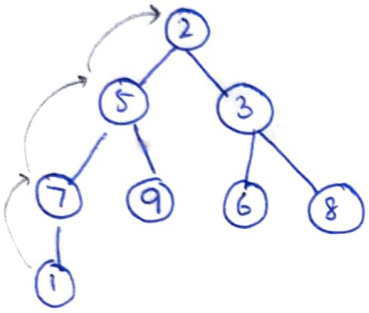
(swapped to satisfy min property)

- Add 2:

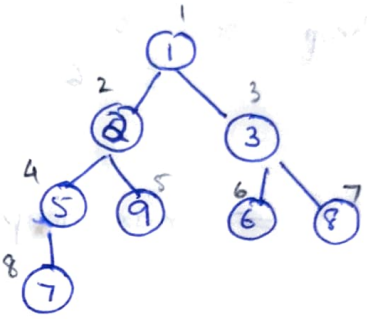
- Add 7, 9, 6, 8



- Add 1



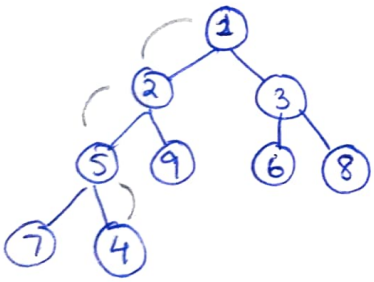
⇒



representing in the form of an array:

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

- Add 4:



⇒

Complexity = $O(n \log n)$

We do swapping along the height of tree.

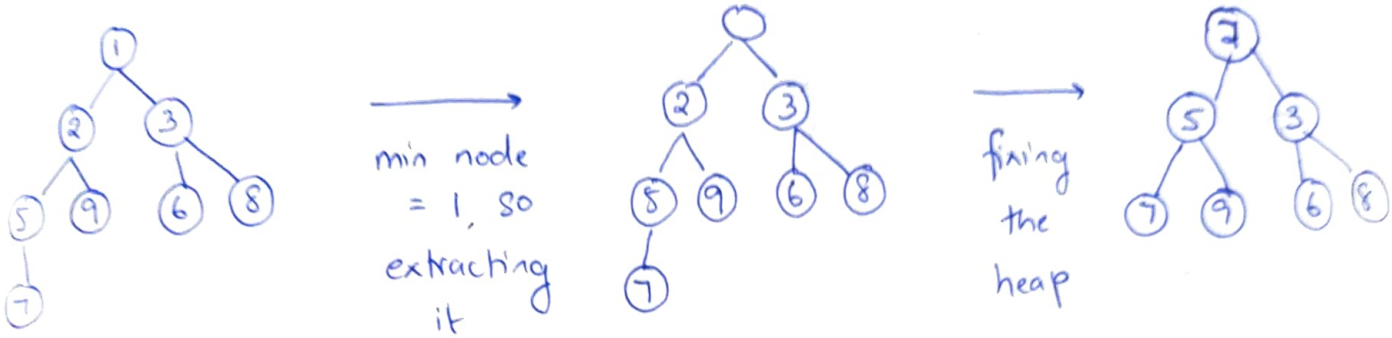
height = $\log n$

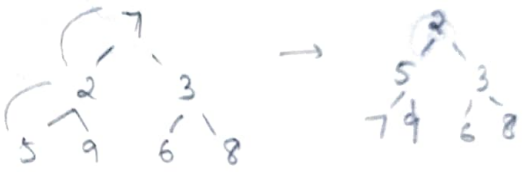
n elements

$\therefore n \log n$

Extract_min()

The min node is the root. Then perform swapping (fixing property) to fix the heap.





Complexity = $O(\log n)$

☆ parent = i

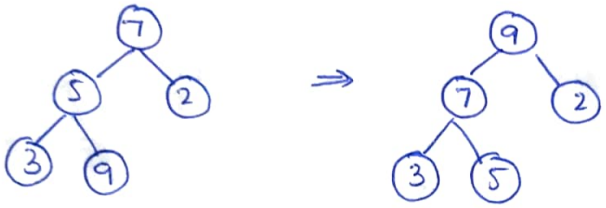
left child = $2i$

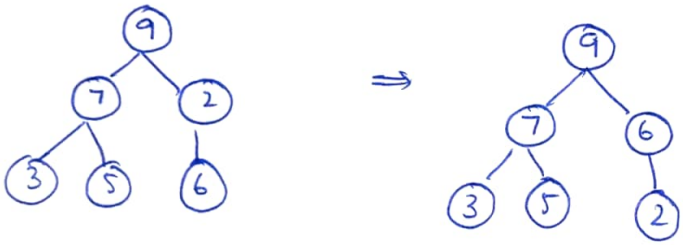
right child = $2i+1$

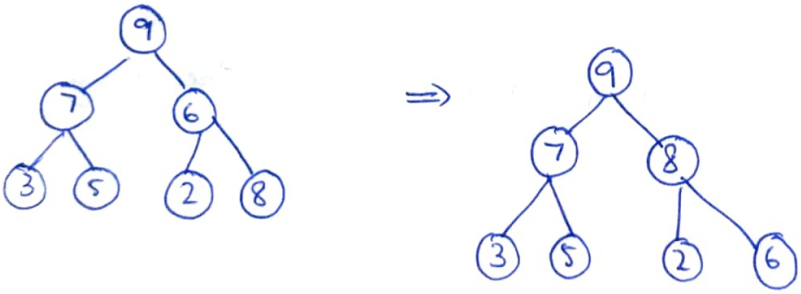
given a child i, it's parent is at position $\lfloor \frac{i}{2} \rfloor$ of the array

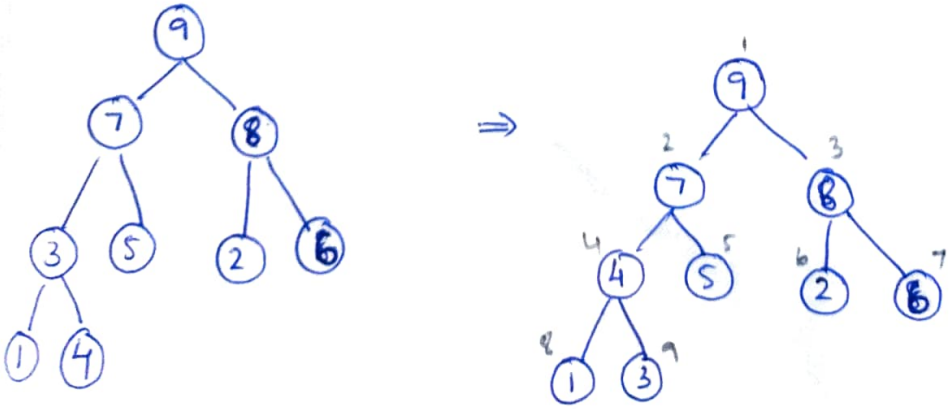
Max heap

5 3 2 7 9 6 8 1 4









Array:

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

Heap Sort

Complexity = $O(n \log n)$

↳ optimal

```
for i = n to 1
    x = extract-max()
    H[i] = x
```

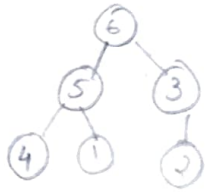
H =

9	7	8	4	5	2	6	1	3

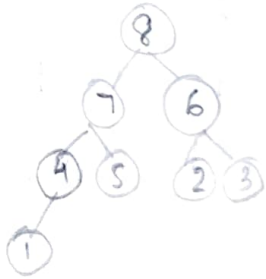
when i = 9, 8 7 6 4 5 2 3 1 | 9

i = 8, 7 5 6 4 1 2 3 | 8 9

i = 7, 6 5 3 4 1 2 | 7 8 9



x = 7



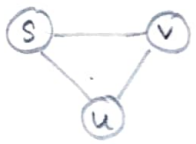
Dijkstra (G, s): → solves the problem of single source shortest path

```
for i = 1 to n
    d[i] = ∞
d[s] = 0
Q = G.V (based on d)
S = { }
While Q ≠ { }:
    u = Extract_min (Q)
    for each (u,v) ∈ E
        Relax ((u,v))
    S = S ∪ u
```

Constraint: graph G shouldn't have -ve weights, i.e. $w > 0$

Relax ((u,v))

```
if d[v] > d[u] + c(u,v)
    d[v] = d[u] + c(u,v)
    v.π = u
```



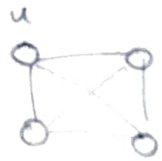
if $d[v] > d[u] + c$ from s then we update it and state that v's parent is now u.

★ in the worst case, v will relax x times, where x is the # possible paths to V.

★ how many paths in a complete graph?

$A^1, A^2, A^3, \ldots A^k$

$1 + 2 + 2 \ldots + 1$

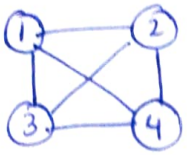


$\ldots 2(n-2)(n-1)$

lab

heap
heapsort
Dijkstra

BST binary tree



$(n-1) \longleftarrow n(E) \longrightarrow {}^{n}C_2$

Close to or is $(n-1)$ — Sparse graph

Close to or is ${}^{n}C_2$ — dense graph

* When we reduce the number of edges, the distance/ length of the shortest path increases.

For a dense graph G, when we ↓ #edges, we want to construct a new ~~graph~~ such that:

$d^*(u,v) \leq k \cdot d(u,v)$ — Multiplicative graph spanner

$d^*(u,v) \leq d(u,v) + c$ — Additive graph spanner.

eg: $10 \leq 2 \times 5$

21.11.2022

Algo Design Methodology

- Brute Force: try all possibilities
- Divide and conquer: divide the problem into subproblems, find sol to the subproblems and combine the solutions.

Eg: quicksort, mergesort

matrix multiplication [originally it takes $O(n^3)$]

↳ under divide and conquer:

- Strassen's algo: $O(n^{2.81})$
- Winogauds: $O(n^{2.14})$
- theoretically proved, but no algo exists: $O(n^2)$

- Greedy method

Greedy Method:

- suitable for optimization problems (but not all of them)
- to be used, the following conditions must be satisfied:
  - optimal substructure
  - greedy choice.

→ **Greedy choice**

in dijkstra, relaxation is a greedy choice ↑

We make a choice based on local maxima or local minima in the hope that we reach global maxima/minima.

→ **Optimal substructure:**

The optimal solution of a problem contains the optimal solution of its subproblems also.

Eg: Amount = 24 Rs

coins available: 1 2 5

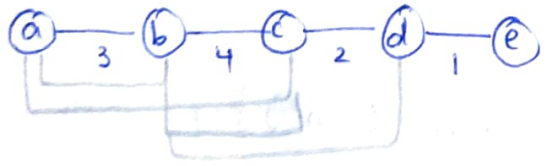
Choosing 4 5Rs coins, we are left with 4 Rs

↓

we need to do the same thing for this amount

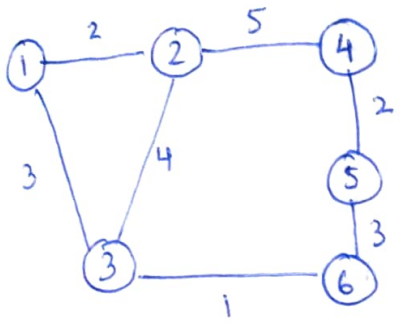
→ its a subprob.

Eg:



Finding the optimal path b/w (a) and (e) is basically finding the optimal path b/w (a)-(b), (b)-(c), (a)-(c), etc...

Minimum Spanning Tree (MST)



Aim: to minimize the total edge cost while finding a tree that connects all the nodes.

→ Kruskal's Algo [select n-1 edges to ensure connectivity with min weight]

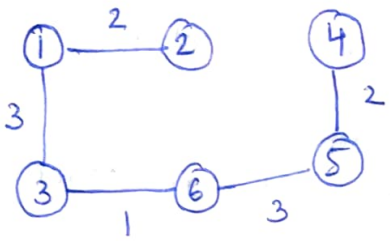
1. sort edges in the asc. order of their weights
2. select the edge with least possible weight and add to MST if it doesn't create cycle.

Step 1:

e	w	
(3,6)	1	✓
(1,2)	2	✓
(4,5)	2	✓
(1,3)	3	✗
(5,6)	3	✓
(2,3)	4	}
(2,4)	5	}

nodes 2, 3, 4 belong to the same component so we cannot consider these edges Otherwise, a cycle will be formed.

Step 2:



Total weight = 2 + 3 + 1 + 3 + 2
= 11

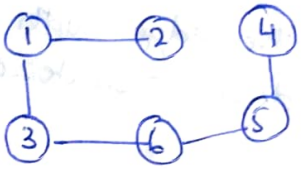
☆ Greedy choice: choose edge with min weight

optimal substructure:  is a subproblem.

MakeSet() : selects a node that represents a subgraph.

Nodes	rep(r)	Adding (3,6) & (1,2) r	Adding 4,5 (r)	Adding 1,3 (r)	Adding 5,6 (r)
1	1	1	1	1	1
2	2	1	1	1	1
3	3	3	3	1	1
4	4	4	4	4	1
5	5	5	4	4	1
6	6	3	3	1	1

Note: Connect 2 nodes only if their rep nodes (r) are different.



★ Are MST and tree in single source shortest path the same?

No, as their objectives are different.

MST wants to minimize total cost while SSSP wants to find the shortest path from source to all other nodes.

12.12.2022

Dynamic Programming

2 properties:

- Optimal substructure:
- Overlapping subproblems

if a problem has these 2 properties, an efficient DP solution can be constructed.

★ Optimal Substructure: optimal sol contains optimal sol of subproblems also. Eg: 

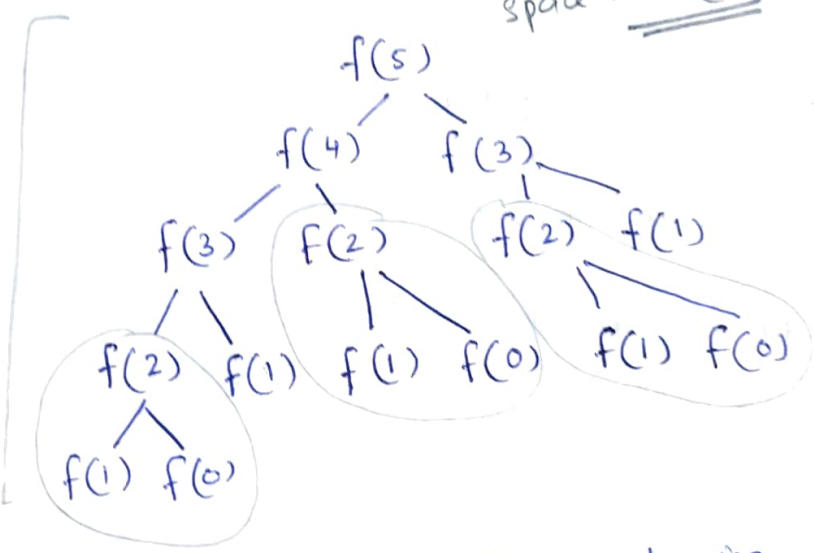
★ Overlapping subproblems:

Eg: f(n) : 0 1 1 2 3 5 8 ...

$$f(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ f(n-1) + f(n-2), & \text{otherwise} \end{cases}$$

space = O(1)

n levels = height [each step it reduces by 1]



2^0

2^1

2^2

2^n recursions

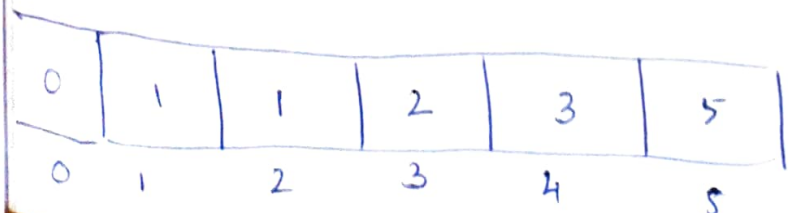
★ Not an optimization prob, ~~but has overlapping~~ → $O(2^n)$

The above problem has repetition of subproblems that are solved again and again.

→ Overlapping subproblems

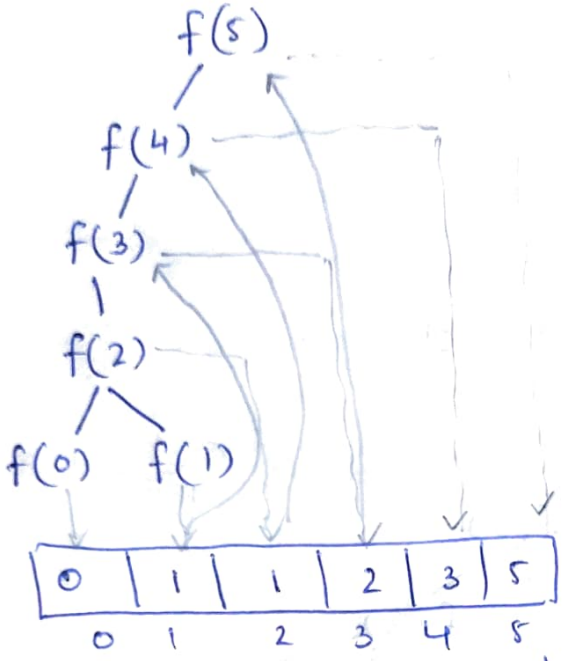
★ This is not efficient, so we store the result to be retrieved later.

→ Time complexity ↓ space complexity ↑ (Trade-off to make it more efficient)



→ Store the values of each subproblem.

Then:



This is a memoization table.

Only 5 recursions
⇒ O(n)

This is called Memoization method

★ We went from an exponential TC to linear TC at the cost of space.

⇒ Space = O(n)

★ Sorting isn't DP because sorting one part of the array isn't the same as sorting another part.

~~DP fib(n)~~:

```
fib(n)
{ arr[n], i = 0
  while (i ≤ n)
  { if (i == 0)
      arr[i] = 0
    else if (i == 1)
      arr[i] = 1
    else
      arr[i] = arr[i-1]
               + arr[i-2]
  } i++
}
```

```
DP- fib(n):
    mem.add(0,0) , mem.add(1,1)
    if mem.lookup(n) ≠ NULL
        return mem.lookup(n)
    else
        x = mem.lookup(n-1)
        if x == NULL then return DP_fib(n-1)

        y = mem.lookup(n-2)
        if y == NULL then return DP.fib(n-2)

        mem.add(n, x+y)
    return x+y
```

13.12.2022

Knapsack Problem

2 variants: fractional knapsack → greedy algo possible (self study)
0|1 knapsack → DP algo

Eg ~~0/1 knapsack~~:

items	1	2	3	4
weight	2	3	4	5
value	3	4	5	6

C = 5

0/1 Knapsack

Let M be the table: ↑ write in asc order

items	weight	value
0	0	0
1	2	3
2	3	4
3	4	5
4	5	6

Knapsack capacity (c) →

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

M[i, j] denotes max values possible when capacity is j and we consider i items.

★ if there are 2 approaches that completely fill the sack, choose the one with greater value.

★ Consider i items and choose the items that completely or almost completely fill the knapsack. Fill the (i, j) cell with the sum of the values of those items.

Eg: (2, 5) → we can choose items 1 and 2, so

$M[2,5] = 3+4 = 7$

★★★ $M[i,j] = \max\left(M[i-1, j],\ M[i-1, j-w(i)] + v(i)\right)$

($M[i-1, j-w(i)]$ ↓ Subproblem)

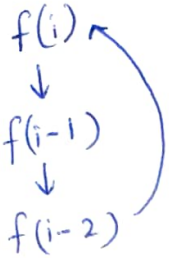
Where $w(i)$ = weight of ith item

$v(i)$ = value of ith item.

★ This is bottom-up approach (just DP) → we solve subprobs that might not be needed in the future.

Fib is top-down (memoization)

Q. $f(i) = \begin{cases} f(i-1) + f(i-2), & \text{if } i \text{ is odd} \\ f(i-1) + f(i-2) + f(i), & \text{if } i \text{ is even} \end{cases}$



Forward ref: refers to a value that hasn't been found yet

Directed cyclic graph ⇒ no DP solution

↓

due to loop, no progress towards base case

★ DP solution exists only if the graph is DAG

→ ~~All values~~ We need all values, required for the current subprob to ~~have been already~~ be computed already.

★ f(5) → f(4) → f(3) → f(2) → f(1)

DP = memoization + recursion

↓

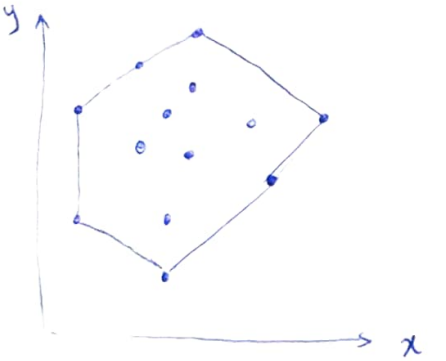
at each recursive step, we progress towards the base case

⇓

progressive recursion.

15.12.2022

Convex Hull



points are rep by p (x, y)

How do we construct this convex hull, given a set of points?

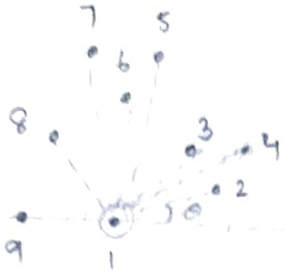
2 methods:

1) Grahams Scan
2) Jarvis method

Given: a set of points:

Steps in Graham's Scan

1. Find the point with the smallest y component.



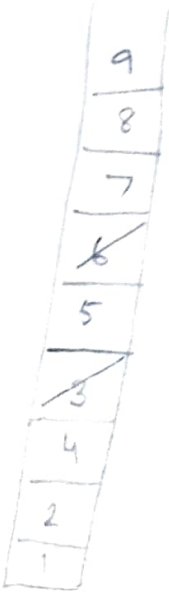
- Find the angle b/w this point and the other points.

  $\max \theta = 180^\circ$

2. Sort the points in ↑ order of angles θ

1
2
4
3
5
6
7
8
9

Add 1st point to stack. Keep adding the other points to the stack, and ensure we are traversing in the counter-clockwise direction. If we encounter a point in the clockwise direction, remove the prev point and add the next.



Add 1, 2, 4, 3

4 to 3 = clockwise → remove 3, add 5

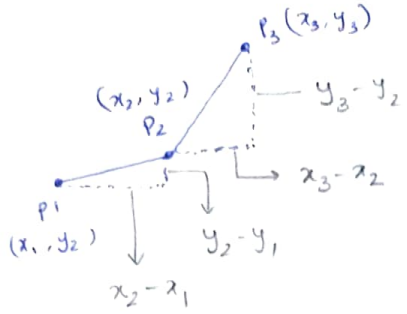
Add 6, ~~4~~

~~5~~ to 6 = clockwise → removed 6, add 7

Add 8, 9

★ To determine direction, we need to find the slope

Eg:



$$\text{Slope of } P_1 P_2 = \frac{y_2 - y_1}{x_2 - x_1} \qquad \text{Slope of } P_2 P_3 = \frac{y_3 - y_2}{x_3 - x_2}$$

if $SP_1 > SP_2$,  clockwise

if $SP_1 = SP_2$, 

if $SP_1 < SP_2$,  anti-clockwise

lab

graham

jarvis

Jarvis Method

For each point, find the θ it makes with the other points. Choose the poi
with the min θ.

Complexity = $O(nh)$

n = total # points

h = # points in the hull.

For each point in the hall that we have discovered, we are finding the angle it makes with the other $(n-1)$ points. [We choose the point with min θ → belongs to the hull, so we do this process with h points]

Q. Show that 2 points that have max distance b/w them is in the hull, given the convex hull.

We have to show that the line passing through them is a dia___ line. Prove using △

if one of the points lie on the boundary, collinear → proved

if not, it will like on the height ($h^2 = b^2 + a^2$) → proved

19-12-2022

Z Algorithm → used to check if a particular pattern is present, else (if specified)

if not specified, used to find the longest substr that matches the prefix

	1	2	3	4	5	6	7	8	9	10
	a	b	b	x	a	b	b	a	b	y
z[i]	0	0	0	0	3	0	0	2	0	0

↑ tells us there is a match of len 3 at this index 5

z[i] = tells how many elements including element at i and after it match the prefix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	a	a	b	x	a	a	b	y	a	a	b	a	a	b
z	0	1	0	0	3	1	0	0	3	1	0	3	1	0

Q. Find out whether a string P is present in another string T.

Logic: Create a new str S = P $ T and use the z algorithm

★ if str is specified, TC = $|P| \cdot |T| = O(mn)$ (m, n)

not specified, TC = $O(n^2)$

Q.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	a	a	b	y	a	a	b	y	d	a	a	b	y	a	a	b	y	a	z
z[i]	0	1	0	0	4	1	0	0	0	8	1	0	0	5	1	0	0	1	0

- We find the repetition of z values.
  can we exploit this to make the algo more efficient?

using z-box:

substring ~~[illegible]~~ at index i, where z[i] > 0

$$|z\text{-box}| = i + z[i] - 1$$

13+4 > 16 so we stop at ind 12.
but 17+1 > 17 so we stop at ind 16

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	a	a	b	y	a	a	b	y	d	a	a	b	y	a	a	b	y	a	z
zbox	0	1	0	0	4	1	0	0	0	8	1	0	0	5	1	0	0	1	0

1+1-1 = 1

4+4-1 = 7

9+8-1 = 16

13+5-1 = 17

from 4 to 7 there is repetition of z values so we don't have to compute z again. We can just copy

★ This is a DP

★ Used alot in compression algorithms.

21.12.2022

Rabin-Carp Algo

(n) Text: c a b c b a b b a c b c a

Pattern: a b c
(P)

goal: to see how many times the pattern occurs in the string/text

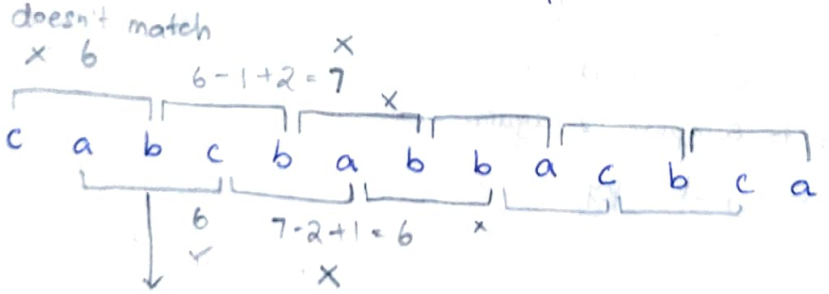
Step 1: find the hashcode for the pattern

apply some numerical code like ASCII, then sum.

a = 1
b = 2
⋮
z = 26

1 2 3
a b c ⟶ 6

Step 2: check for the hashcode in the text. If it occurs, check if the substr and the pattern match.



Old hash value − code (1st char) + code (new char)
↓
excluded
car

★ 5 false matches, to avoid this we can use:

★ $h(P) = c(P[0]) * 10^{|P|-1} + c(P[1]) * 10^{|P|-2} + c(P[2]) * 10^{|P|-3}$

hash fn, code, pattern

$= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 = 123$ (for a b c)

for c a b: $h(p) = 3 \times 10^2 + 1 \times 10 + 2 \times 10^0 = 312$

★ Worst case:

We get a false match, but the mismatch only happens at the last character

Eg: a a a a a = substr

a a a a b = pattern

★ Time complexity = $O(np)$

if we didn't use the sliding window, then $O(n^2)$

★ Any number can be used in $h(p)$ other than 10, but ~~the~~ the number should be greater than the ^{max} numerical code, otherwise a lot of false matches will occur.

Eg: a = 1
b = 2
c = 3
d = 4
⋮
h = 8

Let $h(p) = p[0] \times 4^{|p|-1} + p[i] \times 4^{|p|-2} + \dots + \dots$

Consider a b c and a a g

$1 \times 4^2 + 2 \times 4 + 3 \times 4^0 = 27$

$1 \times 10^2 + 2 \times 10 + 3 \times 10^0 = 123$

$1 \times 4^2 + 1 \times 4 + 7 \times 1 = 27$ → base = 4

$1 \times 10^2 + 1 \times 10 + 7 = 117$ → base = 10

Knuth Morris Prat Algo (KMP)

Avoids the reinitialization of i and j (found) in brute force approach to check if pattern occurs in text.

Checking condition: $T[i] == P[j+1]$, then i++ and j++

Otherwise $j = P[\text{prefix value at } j+1]$

initially, $i = 1$ and $j = 0$

Step 1: ~~check~~ perform prefix matching of the pattern with itself

Eg

	1	2	3	4	5	6	7
P:	a	b	c	a	b	c	a
	0	0	0	1	2	3	4

Step 2: use the checking condition

Q.

		i	1	2	3	4	5	6	7	8	9	10
	T:		a	b	c	a	a	b	c	a	b	b
j		i	1	2	3	4	5					
	P:		a	b	c	a	b					
			0	0	0	1	2					

initially: $j = 0$, $i = 1$

matches occur until $j = 4$ and $i = 5$

then: $j = \text{prefix}[j] = 1$

Now: $j = 1$, $i = 5$

mismatch occurs, so $j = \text{prefix}[\$] = 0$

Now: $j = 0$, $i = 5$

lab

Z

Rabin

KMP

4.1.2023

Computability

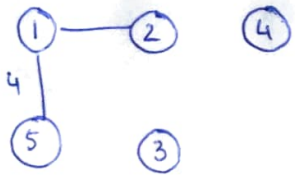
Polynomial time complexity : $O(n^k)$, $k < n$

Such problems $\in$ P class

↳ problems for which there exists a polynomial time algo that gives exact solution.

* There are some problems that can't be solved in P time.

Eg: TSP



Brute Force

1 2 3 4 5
1 2 3 5 4
!

$O(n!)$

$\sim O(n^n)$

time complexity is not P, but exponential

n! ways

NP class

Non-deterministic polynomial class finding the solution isn't deterministic or is unclear

↳ if you are given a solution instance, then you can design a polynomial time algo to verify it.

* automata : ans to q: what is a computable/ non problem?

it's a theoretical machine.

- finite automata
- push down
- Turing machine

} types

Consider:

$$\Sigma = \{a, b\}$$

$\Sigma . \Sigma = \Sigma^2 = (a+b).(a+b) = \{aa, ab, ba, bb\}$

= Set of all strings of length 2

Similarly, Σ^0, Σ^3, ...

$\Sigma^0 = \{\}$ = null set

$\Sigma^3 = \{aaa, \ldots\}$

Σ^* = set of all possible strings

$$\boxed{\Sigma^+ = \Sigma^* - \Sigma^0}$$

Now let $\Sigma = \{a, b, c\}$

Then: $L_1 = \{x \in \Sigma^* \mid x \text{ ends with } a\}$

$= \{a, aa, ba, ca, aaa, aba, \ldots\}$

Now assume that c represents int / char / float,

b represents identifier,

and a represents ;

Then L_1 represents a set of statements that ends with ;

$L_2 = \{y \in \Sigma^* \mid y \text{ starts with } c\}$

$L_1 \cap L_2$ = Set of all strings that start with int / float / char and ends with ;

We are slowly moving towards variable declaration

$L_3 =$

$L_1 \cap L_2 \cap L_3 = \{\text{valid variable declaration in C language}\}$

This becomes a membership checking problem.

↳ such algos are called parsing algos

★ All computable problems can be reduced to membership checking problems

★ A language contains inf strings

↳ abstract machines to accept strings in a given language L

This machine will have:

states: $\{q_0, q_1, \ldots, q_f\}$

$\Sigma = \{a, b, c\}$ → possible symbols in alphabet

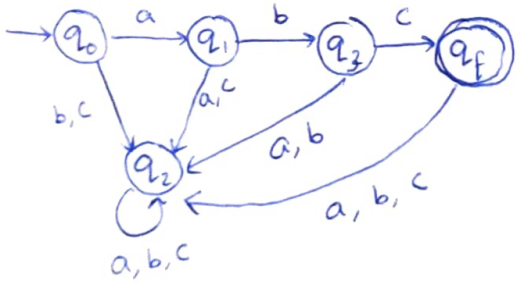
transitions: $\delta(q_i, a) \rightarrow q_j$

} components

q_0 = start state

q_f = final state

Eg: strings that end with c



will accept abc but not acb.

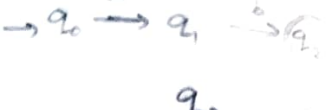
This machine is called deterministic finite automata (DFA)

- deterministic: from every state, all possible transitions are defined (no confusions)
- finite: finite # states

★ Language L = problem

Str ∈ L = instance of problem

★ Can DFA ~~accept~~ solve all problems?



No.

Eg: $L \subseteq \{\Sigma^+ \mid a^n b^n\} = \{ab, aabb, \ldots\}$

~~The~~ We cannot create an automata for this as it's not possible to remember the number of a's and b's.

~~Even~~ If we did create, there would be infinite states which isn't possible for a DFA.

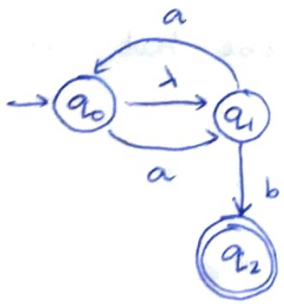
★★ If a = (and b =) ⇒ DFA cannot solve a simple parenthesis matching problem.

★ When does non-determinism come in?

- unspecified transitions
- λ transitions (λ = null ~~stat~~)

Regular language

q_0 can transition to q_1 without reading anything.



$\{\lambda b, ab, ~~aab~~, aaab, aa\lambda b, \ldots\}$

all strings with any number of a's that ends with one b.

This is called NFA

possible to convert NFA to DFA

* To make the machine more powerful, they added a memory component.

Automata definition becomes:

$Q = \{q_0, \ldots, q_f\}$ → states

$M = \{\quad\}$ → memory

$\Sigma = \{\quad\}$

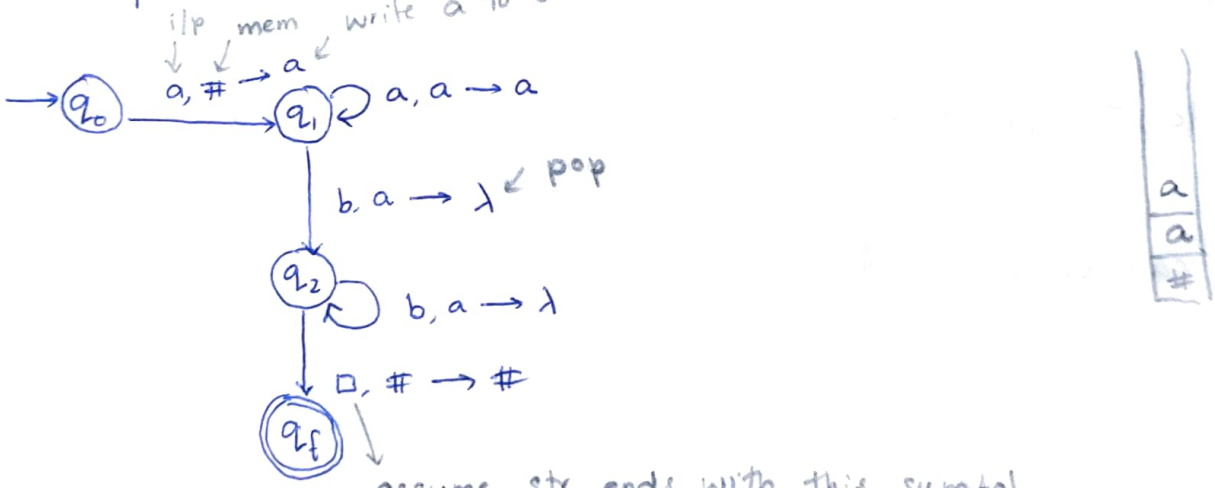
transitions: ~~$\delta(q_i, m) \rightarrow q_j$~~

$\delta(q_i, m, a) \rightarrow q_j$ → next state

- q_i: cur state
- m: top of mem stack
- a: cur symbol

configuration

Q. To Accept $a^n b^n$



* $a^n b^n c^n$ cannot be accepted by the above machine.
i.e. we cannot design a machine for $a^n b^n c^n$

Push Down Automata (PDA)

- Context free languages are accepted by PDA
  ↳ all programming languages are CFL

Sanskrit is CFL

- types: deterministic
  non-deterministic (if λ transitions are used)

Turing Machine (TM)

TOC by Michael Zipser

consists of Random Access Memory (RAM)

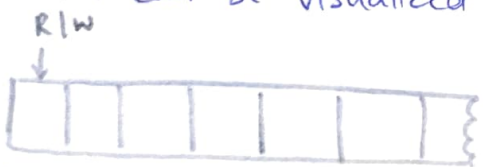
We need some special transitions to move back and forth in the memory.

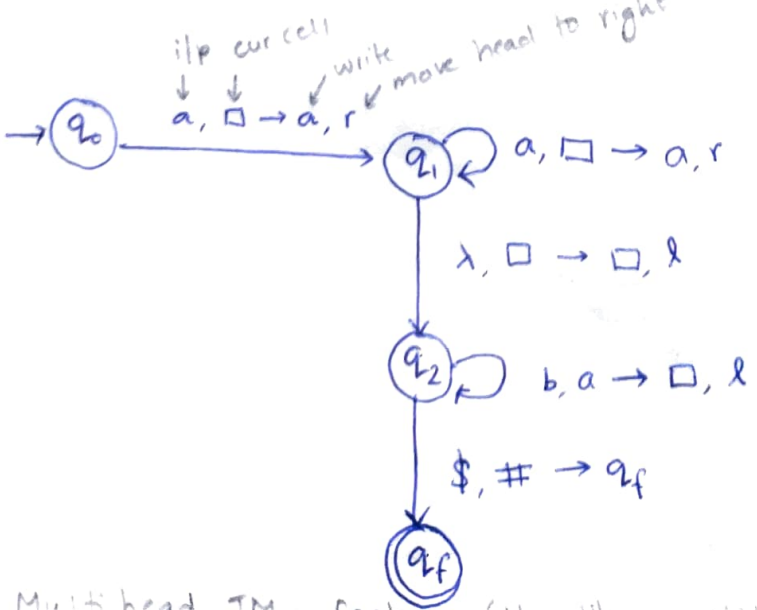
consists of Q, T, M

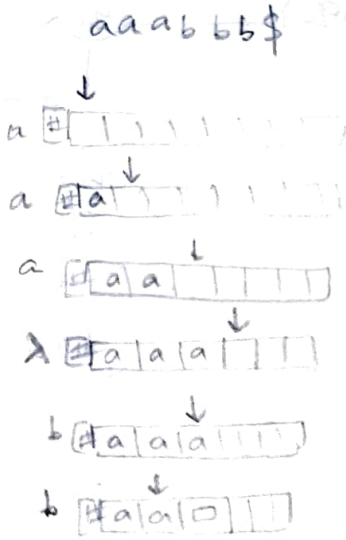
5.1.2023

Q. Create TM for $a^n b^n$

★ TM can be visualized as:







i/p will be $

After parsing all the b's, we will reach # and which denotes that we have parsed the entire str → accepted by TM

★ Multi head TM. faster. (its like multithreading) but similar to TM

★ TM ≡ Algorithm

This means that if we can create a TM for a language, an algorithm exists

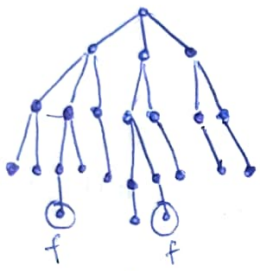
⇒ problem is computable, so algorithm is guarenteed to exist.

aabb

⇓

deterministic

aabb



⇓

non-deterministic

transition graphs of automations

↳ for i/p a, there could be, for ex, 3 possible transitions, out of which only some paths will lead to final states.

↓

complexity is given by the height of the tree.
(# transitions made by TM)

this is dependent on the encoding scheme used (for eg. binary, → refer to 1st class)
on i/p size

5

unary 11111111111111111 → more memory

binary 1111 → logarithmic time

In unary encoding:

magnitude = k, n bits, n = k

$O(k) = O(n)$

In binary, $n = \log k$

$O(k) = O(2^n)$

As the memory increased, complexity became linear (this is wrong?)

↓

of algo

Here, mem ↓, Tc ↑ but this correct info

Hence, we don't use unary

9.1.2023

P: all problems with polynomial time algorithm

NP: all problems for which a solution (certificate is given, verification of certificate can be done in PT)

Problems → optimization problems
Problems → decision problems

Optimization: given a weighted graph, find a shortest tour that starts from ①

decision: given a weighted graph, and k, does there exist a tour of length k?

★ NP consists of all decision problems for which a solution certificate is given.

Subset Sum Problem

$A = \{5, 3, 1, 10, 20, 14, 7, 11\}$

Does there exist a subset of A, such that the sum of the elements in in the subset is k?

→ Brute Force Approach

We have 2^n possibilities:

$S =$ 1 0 0 1 1 1 0 0 → some description of solution
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
⋮
2^n possibilities

∴ complexity = $O(2^n)$

Given a solution, it can be verified in polynomial time

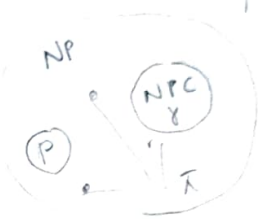
∴ NP

NP complete (NPC)

If a problem $\bar{\Lambda}$ is NP complete, if $\bar{\Lambda} \in NP$ and all problems in NP is PT reducable to $\bar{\Lambda}$.

$\bar{\Lambda} \in NP$

Known NPC $\gamma \leq_p \bar{\Lambda}$ → we want to show this



Reduction

$A \leq_p B$

A is a polynomial time reducable to B

* If I know how to solve B and want to find how to solve A, just reduce A to B.

$A \leq_p B$

* Why is PT reduction very important?

* $A \leq_p B \leq_p C$

If A is reducable to B, B is reducable to C, then A must be reducable to C.

1st problem: SAT problem

SAT is NP complete.

* Cook-Levitin Theorem: 1st problem that is NP complete

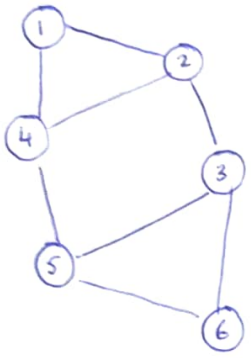
* SAT problem is called Boolean-Satisfiable Problem.

$a \vee \bar{a} \wedge \bar{b} \vee b \vee \bar{c} \vee a$ if this evaluates to true, it is satifiable.

11.1.2023

Q. P.T. the problem of computing hamiltonian path is NPC.

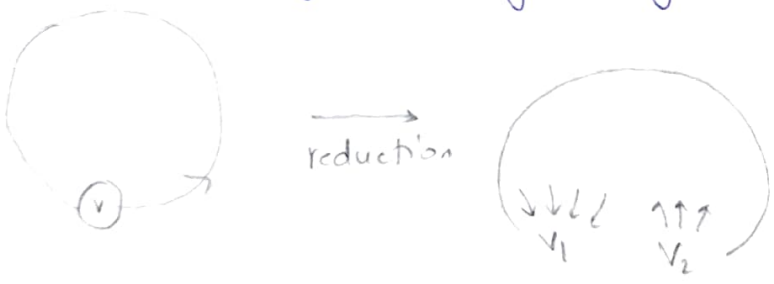
★ Ham. a path that covers all vertices exactly once.



needed to show NPC
- $\pi \in NP$
- Known NPC $\gamma \leq_p \pi$
- reduction
- reverse & forward proofs

★ Ham cycle: a tour that covers all vertices exactly once.
(such a graph is a Ham graph)

Consider a Ham. cycle passing through vertex V.



Split V into V_1 and V_2
↳ we get a Ham. path

★ Steps.
- Take a vertex in HC
- Split it into 2
- cond 1, cond 2

} P.T. reduction

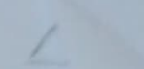
Forward proof
★ If ∃ a HC in graph G, then ∃ a HP in G'. (vertices are split)

Reverse proof
If ∃ a HP in G', ∃ a HC in G (to get this, vertices are merged)

Q. Clique is NPC.

all vertices are connected

* Clique / k-clique: a complete subgraph



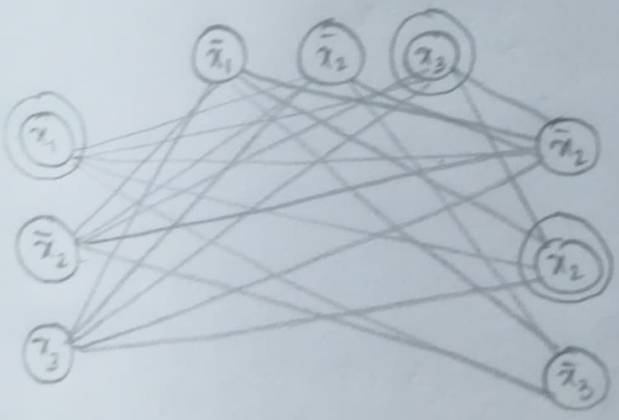
3 CNF SAT is used for reduction. → this is NPC

$\phi = (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_2 \vee x_2 \vee \bar{x}_3)$

For this to be satisfiable, x_1 x_2 & x_3 should be true.

Convert this into a graph. (this construction is done in PT ⇒ atmost n^2 edges)

- Consider each variable in a term as a vertex
- don't connect variables in the same term
- connect variables which aren't complements and belong to diff terms.



x_1, x_2, x_3 are true, then a 3-clique is formed

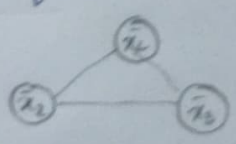
→ Forward

Forward proof: ∃ a satisfiable assignment, then ∃ a k-clique. ($k=3$)

Reverse proof: ∃ k-clique ($k=3$), then ∃ a satisfiable assignment ϕ

↓

Consider a 3-clique



Assign $\bar{x}_2$, $\bar{x}_2$ and $\bar{x}_3$ as False, then ϕ evaluates to true.

∴ ϕ is satisfiable.

* $\in$ NP: proof a clique is done in PT