CS-2103 GF-044.

DESIGN AND ANALYSIS OF ALGORITHMS (DAA) 07-03-2023

sosanraazia@gmail.com, raazia.sosan@dsu.edu.pk.

ALGORITHM:

↳ finite instructions
↳ executed in a specific order.

13-03-23

PROGRAM EFFICIENCY:

minimizing the resources while doing intended task.

Factors Influencing program efficiency:

→ Problem being solved => either problem is understable or can be solved.
→ Programming Language=
→ Compiler, Computer hardware.
→ Programmer ability
→ Programmer effectiveness.
→ Algorithm. ✓

Program to print first 10 natural numbers:

```
for (int i=1 ; i≤10 ; i++) {
    SOUT (i);
}.
```

Algorithm name.
Pre condition
Post condition
Algorithm's body.

ALGO: To print a number.
Pre : -
Post: Print a number.
1) Input i ----------->
2) Output i ----------->
End Algo: To print a number.

SFC Table.

Steps	Frequency	S*F Count.
1	* 1	1
1	* 1	+ 1
	Total Steps=	2

O(1).

ALGO: To print 10 numbers.
Pre :
Post: Output 10 numbers.
1). i ← 1
2) For i←1 to 10
3) Output i
4) End for i
End Algo: To print 10 numbers.

S	F	C
1	1	1
1	10	10
	Total Steps = 11	

O(1)

Algo: To print n numbers

Pre:

Post: Output n numbers.

1) i ← 1
2) For i ← 1 to n.
3) Output i.
4) End For i.

End Algo: To print 10 numbers.

S	F	C
1	1	1
1	N	N

Total steps $1 + N = N$.

$O(N)$.

Algo: Bubble sort an array.

Pre: Given unsorted array.

Post: Sorted array.

1) For i ← 1 to 10
2) For j ← 1 to 10 − i − 1
3) If (arr[j] > arr[j+1])
4) swap arr[j] ↔ arr[j+1]
5) End If
6) End For j.
7) End For i

End Algo: End Bubble sort.

S	F	C
4	100	400
1	100	100

Total steps = 500

for n array

$O(1)$.

4	N^2	$4N^2$
1	N^2	N^2
		$5N^2$

$O(N^2)$.

For i ← 1 to N
  For j ← 1 to N.
    =
  End for j
End for i

} $O(N^2)$.

For k ← 1 to N
End for k

} $O(N)$

$O(N^2 + N)$
$O(N^2)$ higher order

5×100

$5 \times 2 = 10$

$2^2 = 4$

16-03-23

Insertion Sort

ALGO: Insertion Sort.
PRE: Unsorted list
POST: Sorted list.

1) Mark 1st element as sorted
2) For each unsorted el. x.
3) extract the el. x.
4) for j ← last sorted to 0th el.
5) if current el. j > x.
6) move sorted el. to right by 1.
7) end if.
8) break loop & insert x here
9) end for each.

End Algo: Insertion Sort.

O(N) → Best case
O(N²) → worst case.

Asymptotic Notation:

- most significant terms are preferred.
  - → Big Oh Notation, O (Upper-bound)
  - → Big Omega Notation, Ω (Lower-bound).
  - → big Theta Notation, Θ (Average-bound).

Insertion Sort
O(N²)
Θ(N²)
Ω(N)

Saad

Asymptotic Analysis

Yourve to find the bounds of an algorithm.

Growth of a function

$y = mx + c$ Linear order of n
$ax^2 + bx + c = 0$. Quadratic order of n^2

20-03-23

Types of Algorithm
① Probabilistic Algorithm.
e.g. Randomize Quick Sort.
② Hemistic Algorithm
③ Approximate Algorithm.

· Space & Time.
· input size & its composition.

Pcode gunn

→ Experimental Studies of an Algo.
Run the program with varying input sizes & composition.

→ Complexity through Theoretical Analysis::
↳ pesudo code / high-level description of an algo.
↳ running time for input size n.
↳ all possible inputs
↳ evaluateing speed of an algo independent of hardware / software environment.

→ Asymptotic Analysis
Big Oh → worst case / upper bound.
Omega → best case / lower bound.
Theta → average bound.
Representations.
constant $O(1)$.
$O(\log n)$.
$O(n)$
$O(n \log n)$
$O(n^2)$ $O(2^n)$ exponential
$O(n^3)$. $O(n!)$ factorial

Introducing log: 27-03-21

$2^3 = 8$
$\log_2(8) = 3$

$a^x = y$
$\log_a(y) = x$.

Merge Sort: $O(n \log n)$
Insertion Sort: $O(n^2)$

$\log_2 n < n$

Running Time: $T(n)$

$$\sum_{j=1}^{n} j = \frac{n(n+1)}{2} - 1$$

maximum from an array.

Steps	Frequency	Count
c_1	1	c_1
c_2	$n+1$	$c_2(n+1)$
c_3	$n-1$	$c_3(n-1)$
c_4	$n-1$	$c_4(n-1)$
c_5	1	c_5

$$\begin{aligned} T(n) &= c_1 + c_2(n+1) + c_3(n-1) + c_4(n-1) + c_5 \\ &= c_1 + nc_2 + nc_3 - c_3 + nc_4 - c_4 + c_5 \\ &= c_1 + n(c_2 + c_3 + c_4) - c_3 - c_4 + c_5 \rightarrow f(n) \\ &= n \rightarrow g(n) \end{aligned}$$

Asymptotic Analysis.
Types of Analysis:

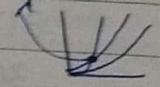
Saad

30-03-23

Asymptotic Notation

$f(n) = n^4 + 100n^2 + 10n + 50$
$g(n) = n^4.$
$f(n) = O(g(n))$
$f(n) = O(n^4)$
$f(n) \leq g(n).$

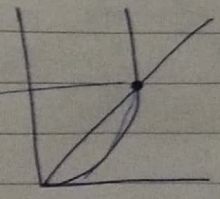
crossover point.



plot $f(n)$ & $g(n)$ on desmos.

05-03-23

- Growth of a function → $O(n)$ or $O(n^2)$.
- Running time → $(f(n))$
- Asymptotic analysis → runing time performance. for best, average & worst case.
- crossoverpoint tells that at this certain point, an algorithms performance is changing!



$f(n) = 4n^2 + 2n + 6$.

we have $g(n) = n^2$.

$f(n) \leq O(g(n))$. → worst case. O

$4n^2 + 2n + 6 \leq O(n^2)$

$f(n) \geq g(n)$ → best case. Ω

$f(n) = g(n)$ → average case. Θ

order of growth, slope in displacement tells the performance of algos.

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

$2n^2 + n <= c\, g.n^2$.

$2n^2 + n <= 3n^2$ let $c = 3$.

$n <= 3n^2 - 2n^2$

$n <= n^2$ ÷ by n

$1 <= n$

$n >= 1$ ✓

Show that $30n + 8$ is $O(n)$. $30n + 8 = O(g(n))$.

$30n + 8 <= c\, g(n)$

$30n + 8 <= cn$ let $c = 31$

$30n + 8 <= 31n$.

$8 <= 31n - 30n$

$8 <= n$.

$n >= 8$.

when $n > 8$, then $31n$ algo will work better.

8 is a crossover point.

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

$2n^2 \leq cn^3$ ÷ by n^2

$2 \leq cn$. we've $c = 1$

$2 \leq n$

or $n \geq 2$.

06-03-23

$1 < \lg n < \sqrt{n} < n < n\log n < n^2 < n^3 \ldots < 2^n < 3^n < \ldots < n^n$

Example:

Find upper bound of running time of quadratic funct.

$f(n) = 3n^2 + 2n + 4$.

$f(n) \leq c \times g(n)$.

$3n^2 + 2n + 4 \leq c \times g(n)$.

$3n^2 + 2n + 4 \leq 3n^2 + 2n^2 + 4n^2$.

$3n^2 + 2n + 4 \leq 9n^2$.

for all $n \geq 1$:

$3n^2 + 2n + 4 \leq 9n^2$.

$3n^2 + 2n + 4 \leq 9n^2$

So, $g(n) = n^2$; $c = 9$; $n_0 = 1$

slope; crossover point

$4n^2 + 3n$ $\quad 5n^2$

$3n$ $\quad 5n^2 - 4n^2$

$\frac{3n}{n}$ $\quad \frac{n^2}{n}$

$3 > n$

$n \leq 3$

Tabular

$3n^2 + 2n + 4 \leq c \cdot g(n)$

$3n^2 + 2n + 4 \leq 4n^2$

$2n + 4 \leq 4n^2 - 3n^2$

$2(n+2) \leq n^2$

$2n + 4 \leq n^2$.

for $n=1 \Rightarrow 2(1)+4 \leq (1)^2 \Rightarrow 6 \leq 1$ ×

$n=2 \Rightarrow 2(2)+4 \leq (2)^2 \Rightarrow 8 \leq 4$ ×

$n=3 \Rightarrow 2(3)+4 \leq (3)^2 \Rightarrow 10 \leq 9$ ×

$\boxed{n=4} \Rightarrow 2(4)+4 \leq (4)^2 \Rightarrow 12 \leq 16$ ✓

So, $g(n) = n^2$, $c = 4$ & $n_0 = 4$

crossover point.

Example.

Find upper bound of running time of quadratic function.

$f(n) = 2n^3 + 4n + 5$.

① $f(n) \leq c \times g(n)$ $\quad \rightarrow n^3$

$2n^3 + 4n + 5 \leq c \times g(n)$

$2n^3 + 4n + 5 \leq 2n^3 + 4n^3 + 5n^3$

$2n^3 + 4n + 5 \leq 11n^3$. (for all $n \geq 1$)

So, $g(n) = n^3$, $c = 11$ & $n_0 = 1$

Highest coefficient method.

② $f(n) \leq c \times g(n)$

$2n^3 + 4n + 5 \leq c \times n^3$

$2n^3 + 4n + 5 \leq 3n^3$

$4n + 5 \leq 3n^3 - 2n^3$

$4n + 5 \leq n^3$. —①.

for $n=1 \Rightarrow 4(1)+5 \leq (1)^3 \Rightarrow 11 \leq 1$ ×

$n=2 \Rightarrow 4(2)+5 \leq (2)^3 \Rightarrow 13 \leq 8$ ×

$\boxed{n=3} \Rightarrow 4(3)+5 \leq (3)^3 \Rightarrow 17 \leq 27$

So, $g(n) = n^3$, $c = 3$ & $n_0 = 3$ ✓ cross over point

least coefficient method.

Gunkai

12-04-23

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \ldots < 2^n < 3^n < \ldots < n^n$

$f(n) = \log n^2$, $g(n) = \log n + 5$

$= 2\log n$

$\log n$ $\log(n)$.

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

$f(n) = n$, $g(n) = \log n^2 = 2\log n$.

$n \geq \log n$

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

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

$\log\log n < \log n$.

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

$f(n) = n$, $g(n) = \log^2 n$.

$n > \log\log n$.

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

$f(n) = n\log n + n$, $g(n) = \log n$.

$n(\log n + 1) > \log n$.

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

$f(n) = 10$, $g(n) = \log 10$.

$10 \quad \log 10$

$1 = 1$

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

$f(n) = 2^n$, $g(n) = 10n^2$

$2^n > n^2$

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

$f(n) = 2^n$, $g(n) = 3^n$

$2^n < 3^n$

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

Lecture - 3: Heap, Heap Sort, Priority Queues. 13-04-23

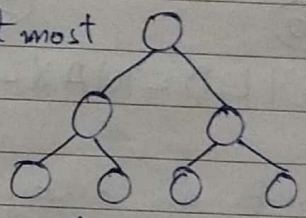
Special Types of Trees.

- Binary Tree atmost 2 child nodes.
- Full Binary tree each node x either a leaf or has degree exactly 2. (2 child).
- Complete Binary Tree all leaves are on same level
- Height of a node no. of edges on the longest path from node down to leaf.
- Level of a node length of a path from root to node.

→ Useful Properties.

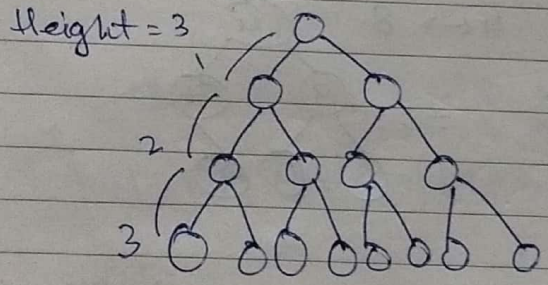
- At most 2^l nodes at level l of B.T
- A B.T with height d has atmost $2^{d+1} - 1$ nodes.

$2^{2+1} - 1 = 7$ nodes



- A B.T with 'n' nodes has height at least $\lceil \lg n \rceil$.

∴ Levels are counted through edges.



HEAP DATA STRUCTURE.

- A heap is a nearly complete B.T

Structural Properties → internal nodes have degree 2.
→ all levels are full, except possibly the last one, which is filled from left to right.

Order (heap) property:
for any node x Parent(x) ≥ x.
root is the maximum element of heap.

Heap Types.

- Max Heap: largest element at root (by default).
- Min Heap: smallest element at root.

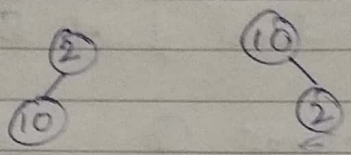
Max: $A[Parent[i]] \geq A[i]$ Min: $A[Parent[i]] \leq A[i]$

Adding / Deleting Nodes.

- new nodes are inserted at the bottom level (left to right).
- nodes are removed from bottom level (right to left).

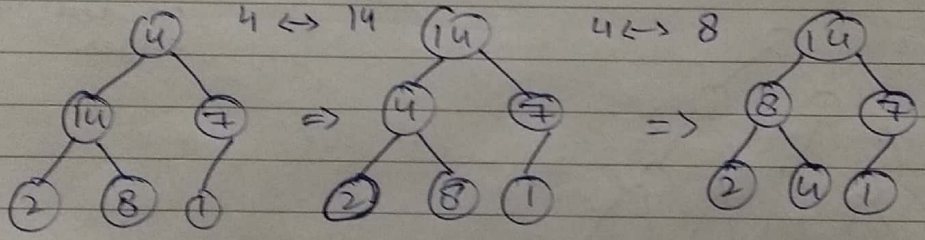
Operations on Heaps.

- MAX-HEAPIFY: maintain / restore the max-heap property



- BUILD-MAX-HEAP: create a max-heap from an unordered array.
- Sort an array in place

Maintaining the Heap Property.

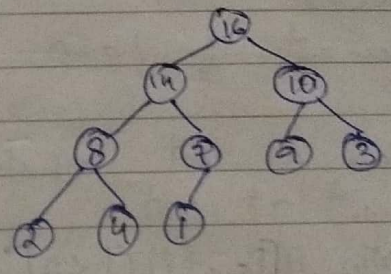


19-04-23.

Array Representation of Heaps.

→ Root of Heap → A[1].
→ Left child of A[i] → A[2i].
→ Right child of A[i] → A[2i+1]
→ Parent of A[i] → A[⌊i/2⌋]
→ Heap Size ≤ length[A].
→ The elements in the subarray A[(⌊n/2⌋+1 ... n] are leaves

⌈ ⌉ ceil
⌊ ⌋ floor



Array Representation

16	14	10	8	7	9	3	2	4	1
1	2	3	4	5	6	7	8	9	10

→ A[6 — 10] are leaves

Heap Types:

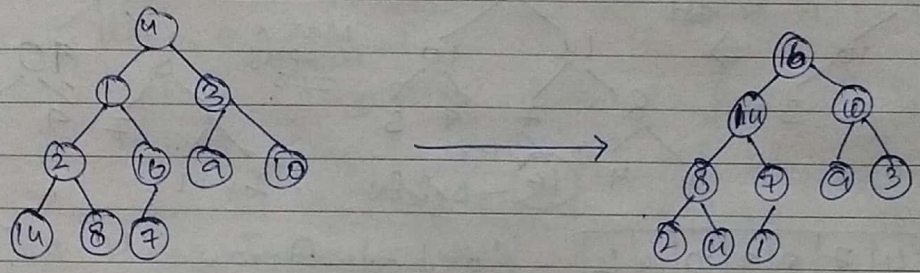
- By Default : Max Heap. Parent will be > child.
- Min heap : child will be > parent.

Max-Heapify Running Time

$O(h)$ or $O(\lg n)$ $\because h = \lg n$.

Build Max Heap.

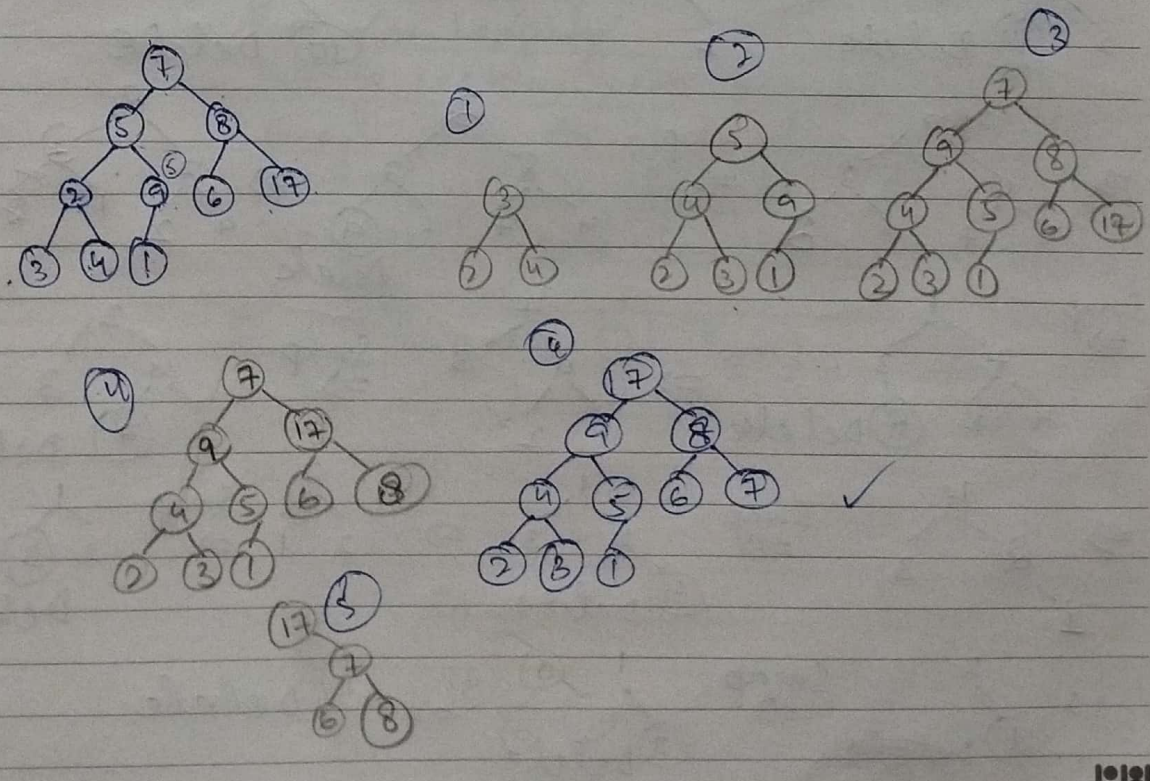
| 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |



Running time of Build Max heap $O(n \lg n)$.

| 7 | 5 | 8 | 2 | 9 | 6 | 17 | 3 | 4 | 1 |

$n/2 \Rightarrow 5$



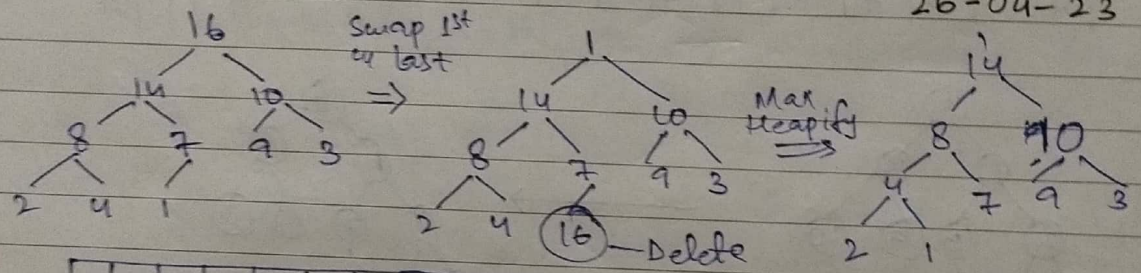
"Sardi Ae Dunya Yaraan De char ton"

The night agent.

Heapsort:

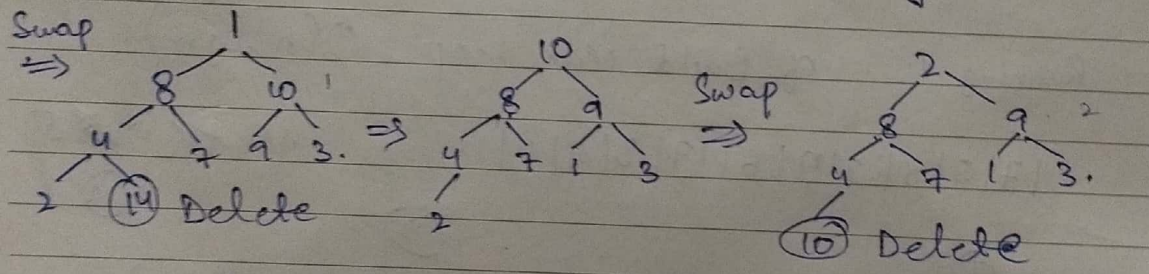
- Goal:
  - Sort an array using heap representation.
- Idea:
  - Build a max heap from the given array.
  - Swap parent w last leaf node.
  - Delete last leaf w store
  - Build a max heap
    repeat until only parent node left.

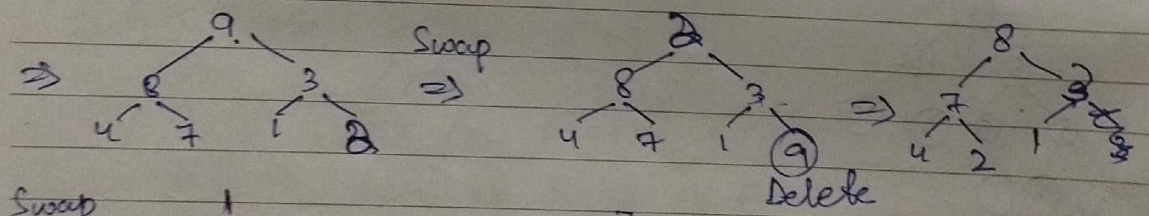
26-04-23

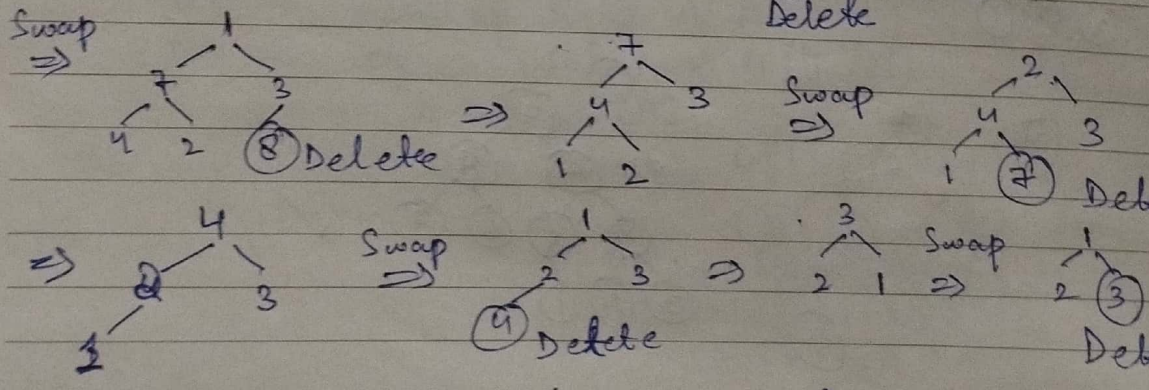


1	2	3	4	7	8	9	10	14	16

Sorted Array.







Running time : $O(n \lg n)$ · worst case
$O(n \lg n)$ · average case
$\Omega(n \lg n)$ · best case

PRIORITY QUEUES

Properties:
- Each element is associated with a value (priority).
- The key with the highest (or lowest) priority is extracted first.

Operations:

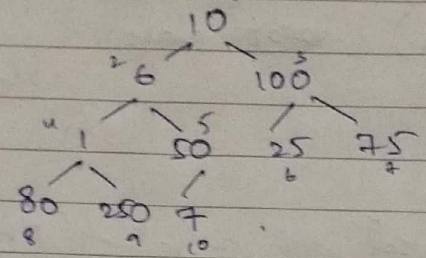
Priority queues can be implemented through heap.

1. Heap-Max : Returns parent element $\Rightarrow O(1)$.
2. Heap-Extract-Max :
   - change parent w last leaf, remove last leaf
   - max-heapify will set the heap again.
   $\Rightarrow O(\lg n)$. $\Rightarrow$ due to max heap w else is constant.
3. Heap-Increase-key :
   Goal: Increases the key (priority) of an element in the heap.

   Idea,
   - Increment the required element's priority.
   - Then max-heapify.
   - Then to restore heap property, compare parent value to child value w swap if needed. (It is from down to up) and max-heapify goes from up to down. There's a difference.

   Running time : $O(\lg n)$.

10, 6, 100, 1, 50, 25, 75, 80, 250, 7.

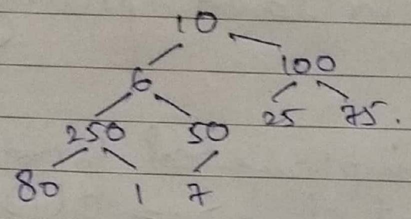


$[\lfloor n/2 \rfloor + 1] \ldots n.$

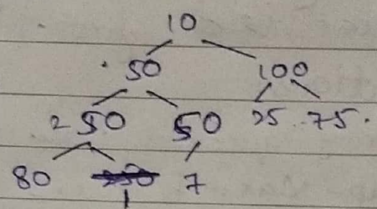
$10/2 + 1 \Rightarrow 6 - 10 \Rightarrow$ leaves

5 → 1
Build max heap.

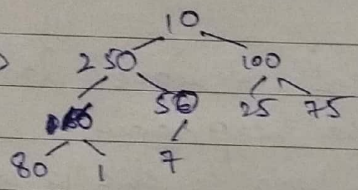
i = 5 no change
i = 4. swap.
i = 3 no change
i = 2



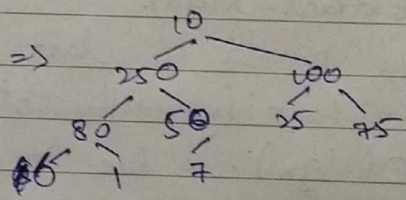
⇒



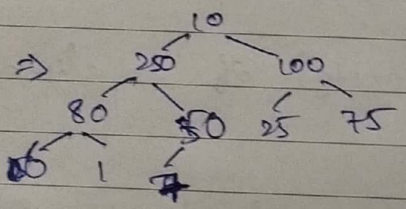
⇒



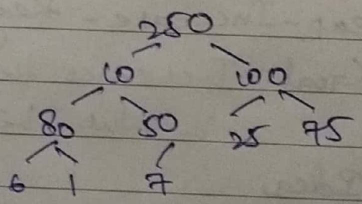
⇒



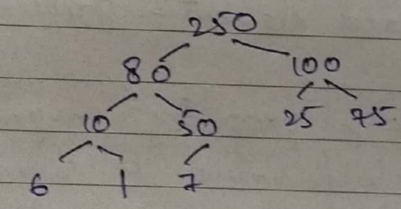
⇒



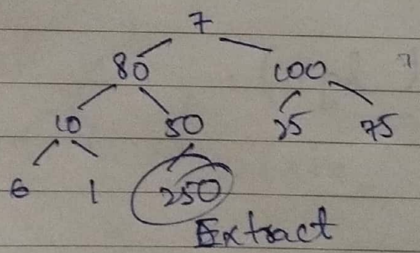
⇒

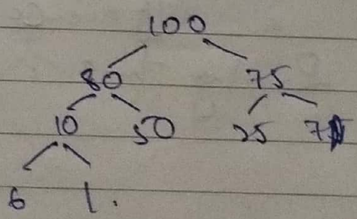


⇒



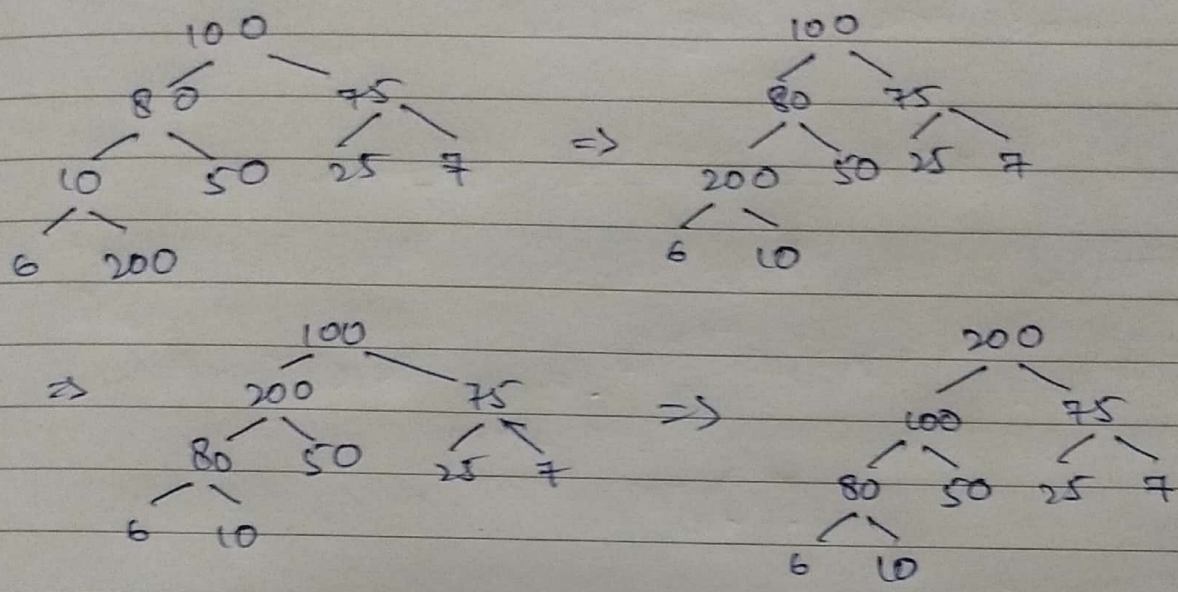
Heap - Extract - Max.





Heap - Increase - Key

1 → 200



Max - Heap - Insert.

- Add a node with value - ∞.
- Run Heap Increase Key.

Running time $O(\log n)$.

FINALS 10-05-22

Lecture-4: RECORRENCE AND SORTING FUNCTION

- MATHEMATICAL INDUCTION:

  S(n) ⇒ successor function; n⇒ natural number.
  ↳ provides next coming natural number.
  → n always starts from 0, 4 successor function will give next natural number.

- Principle of Induction:

  Let P be a claim or a property of natural number. Suppose P holds for zero, then P holds for any natural numbe n, then the successor n+1 holds. If these three are true, the claim P is true for all natural numbers.

- Principle of Induction is a way to prove that a certain property holds for every natural number.

  – To prove, show that it
    → Show that it holds for 0.
    → Show that it holds for some number n.
    → Show that it holds for n+1.

  These steps ~~are~~ is "proof by induction"

  –
    ↳ base case (for 0).
    ↳ induction step.
      ↳ temporary n., assume that P holds for n.
      ↳ show that P holds for n+1

      the assumption that P holds for n → inductive hypothesis.

      - induction → induce → adding → then (+1) → induction
      - hypothesis → we've to prove the assumption 4 H is hypothesis.

- technique to prove the statement S(n) is true for every natural number n (no matter how large.

- Proof

  ↳ Basis step: n = 1 (prove, it is true)
  ↳ Inductive Step: assume S(n) is T 4 prove S(n+1) is T for all n ≥ 1.

- Find case n "within" case n+1.

Example :-

Claim : $1+2+\dots+n = \frac{n(n+1)}{2}$ ——(A).

Basis Step :-

$n = 1$.

$1 = \frac{1(1+1)}{2} \Rightarrow \frac{1(2)}{2} = 1$.

$n = 1$, true. Assume that (A) is true for any ~~n~~ z^+ k. $\Rightarrow n = k$

Inductive Hypothesis

Assume: $1 + 2 + \dots + k = \frac{k(k+1)}{2}$ ——(B).

Inductive Step

Based on B, if B is true ~~fo~~ then A is also true for k+1

$1 + 2 + \dots + k + k+1 = \frac{(k+1)(k+1+1)}{2}$.

$1+2+\dots+k+k+1 = \frac{(k+1)(k+2)}{2}$ ——(C).

Adding k+1 on both sides of eq (B).

$1+2+\dots+k+k+1 = \frac{k(k+1)}{2} + (k+1)$.

$1+2+\dots+k+k+1 = \frac{k^2+k+2k+2}{2}$.

$1+2+\dots+k+k+1 = \frac{k^2+3k+2}{2}$.

$1+2+\dots+k+k+1 = \frac{k^2+2k+k+2}{2} \Rightarrow \frac{k(k+2)+1(k+2)}{2}$

$1+2+\dots+k+k+1 = \frac{(k+1)(k+2)}{2}$ —— proved.

which is equals to eq (C)

Therefore, the sum of first k+1 integers is $(k+1)(k+2)/2$.

Conclusion; By induction, we've shown that the claim is true for all +ve integers.

* Since we've proved, we don't have to use loops we can use this mathematical formula.

RECURRENCES AND RUNNING TIME:

1> Recurrence Relation is for the recursive functions/algorithms to find out their time complexity.

$$T(n) = T(n-1) + n \qquad T(n) \Rightarrow \text{running time.}$$

- Recurrences arise when an algorithm contains recursive calls to itself.
- We get recurrence relationship of a recursive algorithm
- Substitution method, Iteration method, Master theorem, these are the methods to find out time complexities of a recursive algorithms & with that we can find bounds of the algorithms.
- Running time of the Algorithm.
  - → Solve the recurrence.
    - → a clear formula
    - → by solving it for n, we'll find bounds.

Examples

① $T(n) = T(n-1) + n$ $\quad \Theta(n^2)$.
Recursive algorithm that loops through the input to eliminate one item

② $T(n) = T(n/2) + c$ $\quad \Theta(\lg n)$.
Recursive algorithm that halves the input

③ $T(n) = T(n/2) + n$ $\quad \Theta(n)$.
halve the input but must examine every input.

④ $T(n) = 2T(n/2) + 1$ $\quad \Theta(n)$.
Splits the input to halves & does a constant amount of work.

Recurrent Algorithms.

→ BINARY-SEARCH:

for an ordered array A, finds if x (element to be searched) is in the array A[lowest ... highest].

Analalysis of BINARY-SEARCH:

```
Algo: BINARY-SEARCH(A, lo, hi, x).
    if (lo > hi).                          ← c1
        return FALSE.
    mid ← ⌊(lo+hi)/2⌋                      ← c2
    if (x = A[mid]).                       ← c3
        return TRUE.
    if (x < A[mid]).
        BINARY-SEARCH(A, lo, mid-1, x).    ← size: n/2
    if (x > A[mid])
        BINARY-SEARCH(A, mid+1, hi, x).    ← size: n/2.
```

- $T(n) = c + T(n/2)$.
  - ↳ running time for an array of size n.
- at every recursive call, array is halved.

The Iteration Method:.

→ Convert the recurrence into a summation & try to bound it using a known series.
- ↳ iterate recurrence until initial condition is found.
- ↳ use back-substitution to express the recurrence in terms of n & boundary condition.

Method. (Example: Recurrence Relation for Binary Search)

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

$T(1) = 1$

$T(n) = c + T(n/2)$. ∵ $T(n/2) = c + T(n/2/2)$

$= c + (c + T(n/4))$. $= c + T(n/4)$.

$= c + c + T(n/4)$. ∵ $T(n/4) = c + T(n/4/2)$

$= c + c + (c + T(n/8))$. $= c + T(n/8)$.

$= c + c + c + T(n/8)$

$T(n) = c + c + c + T(n/8)$ — c is being added k times

$T(n) = ck + T(n/2^k)$ ... ①

Assume: $n = 2^k$ → since in each recursion it is 2^k, n is being divided

applying log on both side.

$\log n = \log 2^k$

$\log n = k \log 2$

$\log n = k$

① $\Rightarrow T(n) = ck + T(2^k/2^k)$

$= c \log n + T(1)$ ∵ $T(1) = 1$

$= c \lg n + 1$

$T(n) = \lg n$

$O(\lg n)$

Example 2:

$T(n) = n + 2T(n/2)$ $T(n/2) = n/2 + 2T(n/4)$

$= n + 2(n/2 + 2T(n/4))$

$= n + n + 4T(n/4)$ $T(n/4) = n/4 + 2T(n/8)$

$= n + n + 4(n/4 + 2T(n/8))$

$= n + n + n + 8T(n/8)$

$= nk + 2^k T(n/2^k)$ ... ① assume $n = 2^k$

$n = 2^k$ log on b/s.

$\log n = \log 2^k$

$\log n = k \log 2$

$\log n = k$

→ ① $\Rightarrow T(n) = nk + 2^k T(2^k/2^k)$

$T(n) = nk + 2^k T(1)$ ∵ $T(1) = 1$

$T(n) = n \lg n + n$ ∵ $k = \log n$

$T(n) = n \lg n + n$

$O(n \lg n)$

Example 3:

$T(n) = 2T(n/2) + 1$ $T(n/2) = 2T(n/4) + 1$

$= 2(2T(n/4) + 1) + 1$

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

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

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

$7 + n-1$

$T(1) = 1$

$\frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$= 2^k T(n/2^k) + 7$ — ①. assume $n = 2^k$

$n = 2^k$

$\log n = \log 2^k$.

$\log n = k \log 2$.

$\log n = k$.

① $\Rightarrow T(n) = 2^k \cdot T(2^k/2^k) + 7$

$T(n) = 2^k \cdot T(1) + 7$

$T(n) = (2^k \cdot 1) + 7$

$T(n) = 2^k + n - 1 \quad \because n = 2^k$

$T(n) = n + n - 1$

$O(n)$.

$2^k T(n/2^k) + 4 + 2 + 1$

$2^k T(n/2^k) + 2^{k-1} + 2^{k-2} + \cdots + 1$

$n T(2^k/2^k) + (2^k \cdot 2^{-1}) + 2^k \cdot 2^{-2} + \cdots + 1$

$nT(1) + 2^k\left(\frac{1}{2} + \frac{1}{2^2} + \cdots + 1\right)$

$n T(1) + n - 1$

$n + n$

$O(n)$

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

$= (T(n/4) + n/2) + n$.

$= T(n/4) + n/2 + n$.

$= (T(n/8) + n/4) + n/2 + n$.

$= T(n/8) + n/4 + n/2 + n$.

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

$= T(n/4) + n/2$

$T(n/4) = T(n/4/2) + n/4$.

$= T(n/8) + n/4$.

$= T(n/2^k) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \cdots + \frac{n}{2} + n$ assume $n = 2^k$

$n = 2^k$.

$\log n = \log 2^k$

$\log n = k \log 2$.

$\log n = k$.

① $\Rightarrow T(n) = T(n/2^k) + \frac{n}{2^{k-1}} + \frac{n}{2^{k-2}} + \cdots + \frac{n}{2} + n$

$T(n) = T(2^k/2^k) + n\left(\frac{1}{2^{k-1}} + \frac{1}{2^{k-2}} + \cdots + \frac{1}{2} + 1\right)$ $\rightarrow \approx 1$

$T(n) = T(1) + n(1+1)$

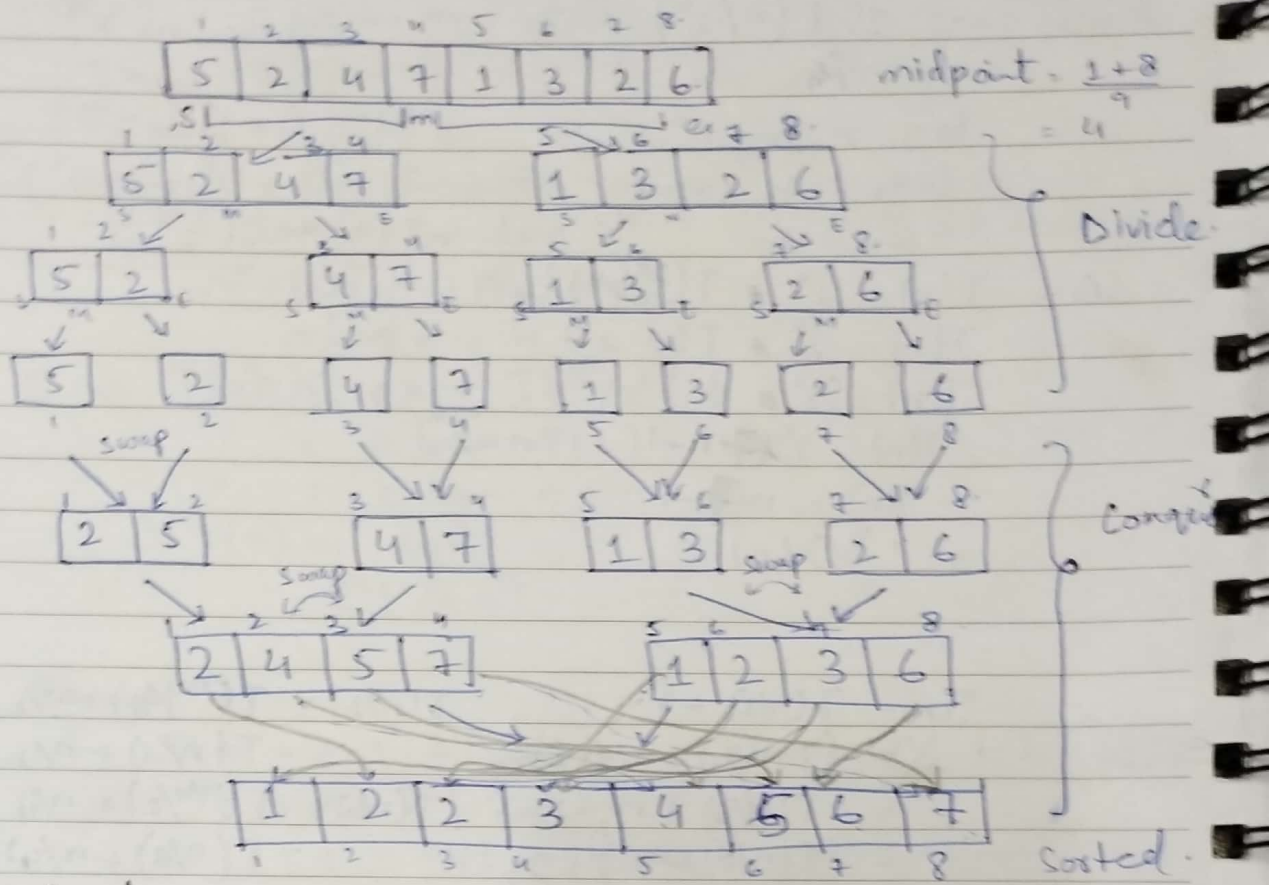
$T(n) = 1 + 2n$.

$T(n) = 2n + 1$

$O(n)$.

MERGE-SORT

11-05-23.



Analyzing Merge Sort.

Algo: MERGE-SORT(A, s, e).

```
if start < end.                                  ⟶ c1
    mid = start + (end - start) / 2.             ⟶ c2
    Merge-SORT(A, start, mid).                   ⟶ n/2.   2T(n/2)
    MERGE-SORT(A, mid+1, end)                    ⟶ n/2
    MERGE(A, start, mid, end).                   ⟶ O(n)
```

- $T(n) = 2T(n/2) + O(n)$.
  $= 2(2T(n/4) + O(n/2)) + O(n)$.
  $= 4T(n/4) + 2O(n/2) + O(n)$
  $= 4T(n/4) + O(n) + O(n)$.
  $= 4(2T(n/8) + O(n/4)) + O(n) + O(n)$.
  $= 8T(n/8) + 4O(n/4) + O(n) + O(n)$
  $T(n) = 8T(n/8) + O(n) + O(n) + O(n)$

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

$\therefore T(n/4) = 2T(n/4/2) + O(n/4)$
$T(n/4) = 2T(n/8) + O(n/4)$

in each recursion $\Rightarrow 2^k$ (away is being divided in 2^k)
$8 = 2^3$.

$T(n) = 2^k T(n/2^k) + O(n) + O(n) + O(n)$.

$T(n) = 2^k T(n/2^k) + k\,O(n)$.

assuming $n = 2^k$, $\log n = k$.

$T(n) = n\,T(n/n) + \log n\, O(n)$.

$T(n) = n(1) + O(n \log n)$.

$T(n) = O(n) + O(n \log n)$.

$T(n) = O(n \log n)$.

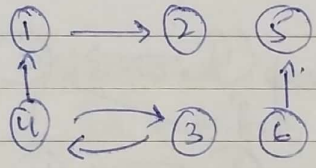
Time complexity for Merge-Sort $= O(n \log n)$.

17-05-23

GRAPHS

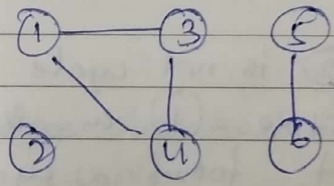
- Set of nodes (vertices) with edges (links) b/w them.
- $G(V, E)$
- Applications: Maps, Schedules, Computer Networks, Hypertext, Circuits.

Directed Graphs (digraphs).



- uni-directional.
- in-degree of v: edges entering v
- out-degree of v: edges leaving v.
- v is adjacent to u if there is an edge (u, v).

Undirected graphs.



- degree of v: edges incident on v
- v is adjacent to u & vice versa if there is an edge (u, v).

Complete graph:

A graph with an edge b/w each pair of vertices (atleast)

Subgraph:
A graph (V', E') such that $V' \subseteq V$ & $E' \subseteq E$

Path from v to w:
A sequences of vertices $\langle v_0, v_1, \cdots, v_k \rangle$ such that $v_0 = v$ & $v_k = w$.

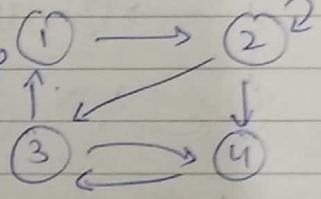
Length of a path
Number of edges in the path
In case of loop, 1 is added to both indegree & out-degree ⇒ (v)↺

w is reacheable from v. (w)
If there is a path from v to w, then destination is reacheable from source. (v)

Simple path
all the vertices in path are distinct (trivial def.)

Cycles.
A path from $\langle v_0, v_1, \cdots, v_k \rangle$ forms a cycle if $v_0 = v_k$ (v_0 = source & v_k = destination) & $k \geq 2$

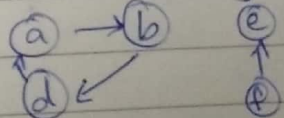
② is not cycle it is a self-loop
if $v_0 = ①$ & $v_k = ①$
it forms a cycle



Acyclic graph
A graph without any cycles.

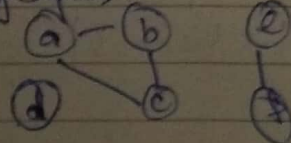
Connected & Strongly Connected.

Directed
- Strongly connected
  every two vertices are reachable from each other
- Strongly connected components
  all possible strongly connected subgraphs.



{a, b, c, d}, {e}, {f}

Undirected
- connected.
  every pair of vertices is connected by paths.
- connected components.
  all possible connected graphs.



{a, b, c}, {d}, {e, f}

Graph Representation.

- Adjacency-List Representation.

Properties of Adjacency-List Representation:
- Memory required : $\Theta(V+E)$.
- Prefered when : The graph is sparse : $|E| << |V|^2$
- Disadvantage : Searching costly.
- Time to decmine $(u,v) \in E \Rightarrow O(\text{degree}(u))$.

- Adjacency-Matrix Representation.

Properties.
- Memory Required $\Rightarrow \Theta(V^2)$.
- Prefered when $\Rightarrow$ The graph is dense $|E|$ is closed to $|V|^2$.
- Disadvantage.
- Time to determine $\Rightarrow O(1)$. if $(u,v) \in E$.
  Time to list all vertices adjacent to u $O(V)$.

Weighted Graph:
In weighted graph, each edge has weight that is a real number. $w(u,v)$.

$w : E \rightarrow R$.

Shortest Path Problems

Input:
Directed Graph $\Rightarrow G(V,E)$.
Weight Function $\Rightarrow w : E \rightarrow R$.
Weight of path $p := \langle v_0, v_1, \ldots, v_k \rangle$.

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

Shortest path
$\delta(u,v) = \min$

Variants.

- Single-source shortest path.
  $G=(V, E)$ ⇒ find a shortest path from a given source node to every other node.

- Single-destination.
  Inverse of single-source.
- Single-pair.
  Shortest path from u to v.
- All pairs shortest path
  shortest path for every pair of nodes.

Negative Weight Edges.

18-05-23

Algorithms:

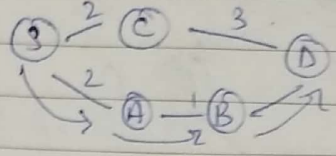
① Dijkstra's:
Negative weights are not allowed.

② Bellman-Ford.
Negative weights are allowed.
Negative cycles are not allowed.
Operations in both
- Initialization
- Relaxation.

Shortest-Path Notations.
For each $v \in V$
$\delta(s, v)$: shortest path weight.
(s: source, v: destination)

$d[v]$: shortest-path weight estimate
- Initially, $d[v] = \infty$ (a bigger value).
- $d[v] = \delta(s, v)$.

- $\pi[v]$: predecessor of v on a shortest path fro. parent of the vertex, we've to remember. s.
  if no $\pi(v) = NIL$.
  π induces in a tree, shortest-path tree.



$\infty + \infty = \nexists \infty$

Initialization.

① Initialize - Single - Source;
for each $v \in V$, put every distance to a node to ∞, make their parents null as the parent of a node will be defined when we'll find shortest distance for the nodes & lastly the distance of source node will be zero.

Relaxation Step:

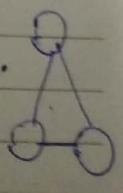
② Relaxing an edge (u,v), we'll test whether the shortest path can be improved from u to v,
if $d[v] > d[u] + w(u,v)$. → RELAX (u,v,w).
improve the shortest pat.
$d[v] > d[u] + w(u,v)$. (distance updated.
$\pi[v] \leftarrow u$ (parent set)

Dijkstra's Algorithm:

- Single-source shortest path problem.
  - No -ve weight edges : $w(u,v) > 0, \forall (u,v) \in E$
- Min-heap will be used.

Dijkstra (G, w, s).

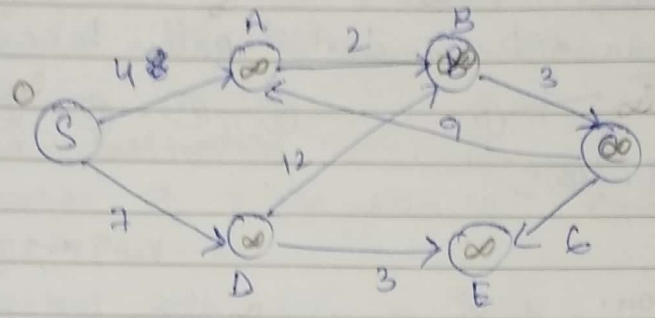
1. INITIALIZE-SINGE-SOURCE(V,S) ← $\Theta(V)$.
2. S ← ∅
3. Q ← V[G] ← O(V) build min-heap
4. while Q ≠ ∅ ← Executed O(V) times } $O(V \lg V)$
5. do u ← EXTRACT-MIN(Q) ← $O(\lg V)$ }
6. S ← S ∪ {u} (Adding). +
7. for each vertex $v \in Adj[u]$ ← O(E) } $O(E \lg V)$
8. do RELAX (u, v, w)
9. UPDATE & (DECREASE-KEY) ← $O(\lg V)$



24-05-23.

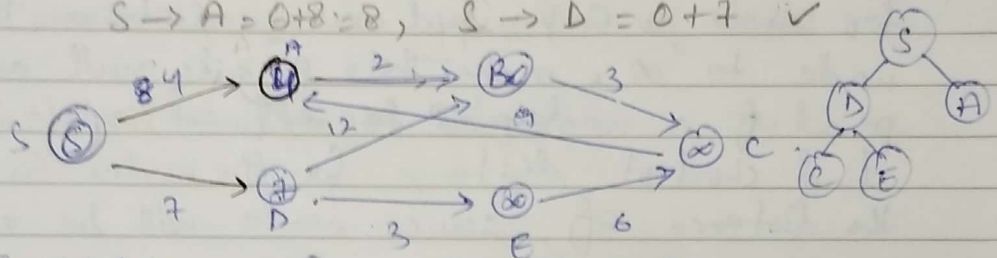
Activity.

- Apply Dijkstra Algorithm on the given graph.

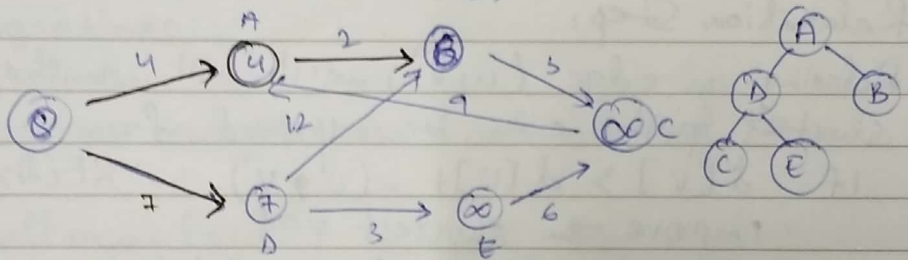


$S = \langle \rangle$ $\quad Q = \langle A, B, C, D, E \rangle$

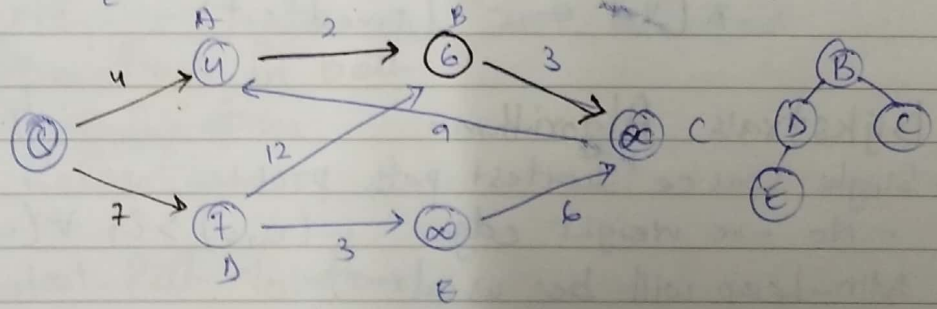
$S \to A = 0+8 = 8$, $\quad S \to D = 0+7$ ✓



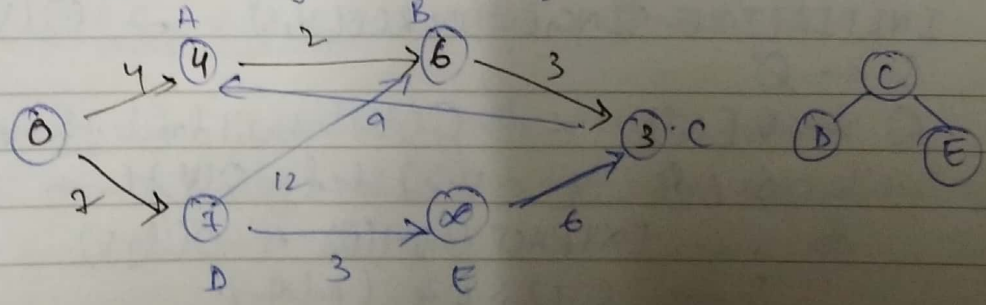
$S = \langle S \rangle$ $\quad Q = \langle A, B, C, D, E \rangle$



$S = \{S, A\}$ $\quad Q = \{B, C, D, E\}$.

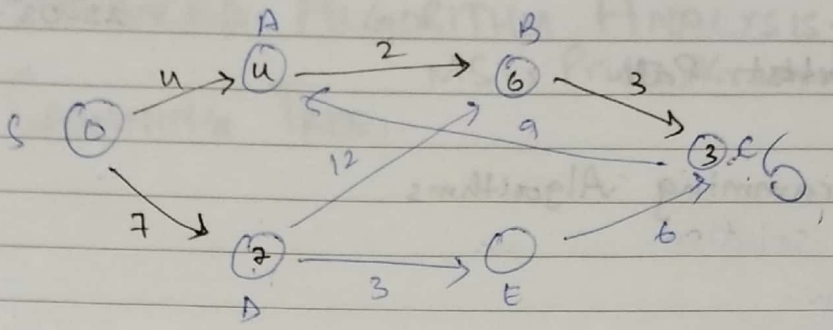


$S = \{S, A, B\}$ $\quad Q = \{C, D, E\}$.



$S = \{S, A, B, C\}$ $\quad Q = \{D, E\}$.

5-23.



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

24-05-23

Running time of Dikstra

$V \log V \times E \log V$

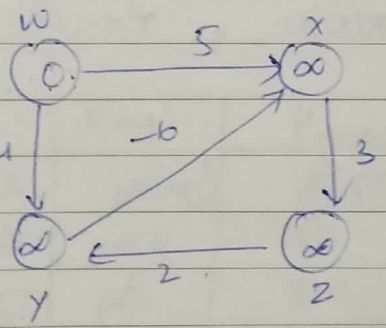
$VE \log^2 V.V$

BELLMAN - FORD ALGORITHM:

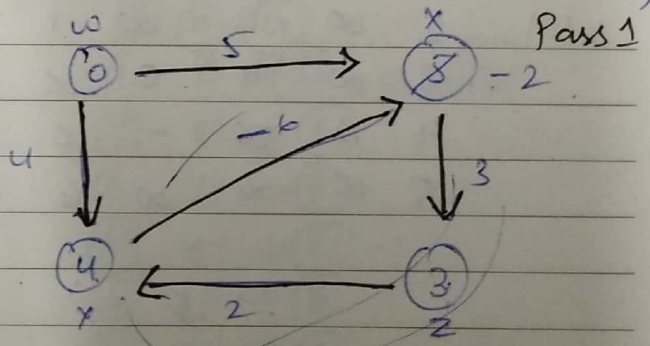
This algorithm can handle negative weights as well.

Running time => $O(VE)$.

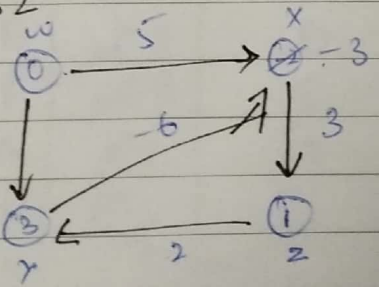
Activity. $E: (w,x), (w,y), (x,z), (y,x), (z,y)$



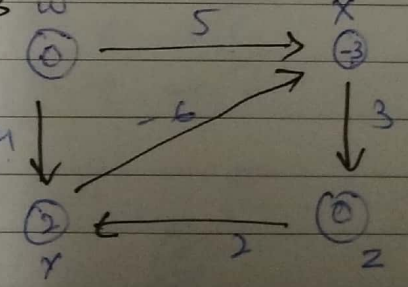
Pass 1



Pass 2



Pass 3



nxn

25-05-23

All-Pairs Shortest Path

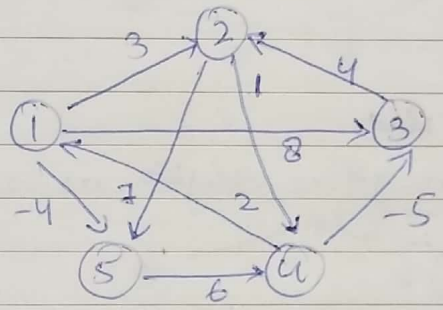
Dynamic Programming Algorithms

- optimum solution
- Recursion
- Back tracking

①

FLOYD-WARSHALL ALGORITHM (FOR A.P.S.P).

- directed graph
- runs in $O(V^3)$.
- -ve weights are handled but not -ve cycles.



$D(0)$ =

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

, $\pi(0)$

Advanced Algorithm Analysis

31-5-23.

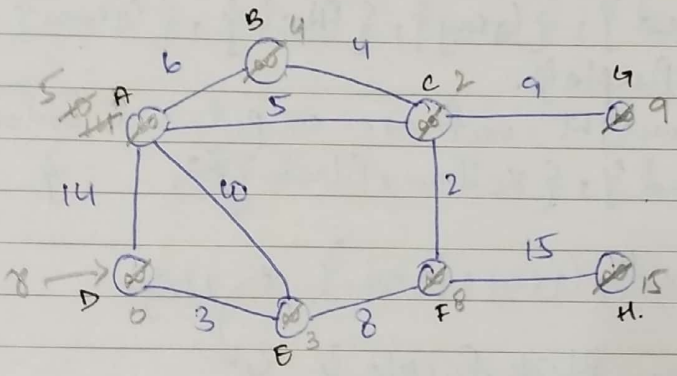
MST, Prims, Kruskul.

Spanning Tree:.

Minimum Spanning Tree:

Prim's Algorithm. => Ends any cycle in the graph.

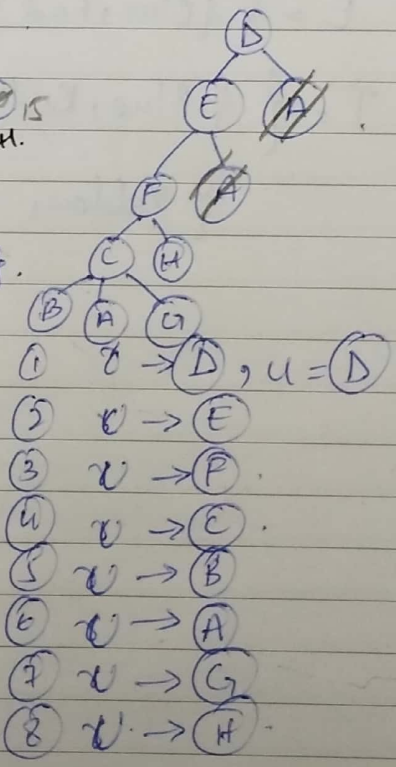
MST(G, w, r) // G => graph, w => weights, r = random source.



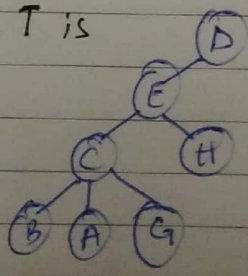
Q = {A, B, C, D, E, F, G, H}.

D -> r , u = r => D.

	Key	Parent
A	~~∞~~ ~~14~~ 5 ~~10~~	~~A~~ E C
B	~~∞~~ 4	C
C	~~∞~~ 2	F
D	0	NULL.
E	~~∞~~ 3	D
F	~~∞~~ 8	E
G	~~∞~~ 9	G
H.	~~∞~~ 15	F



MST is

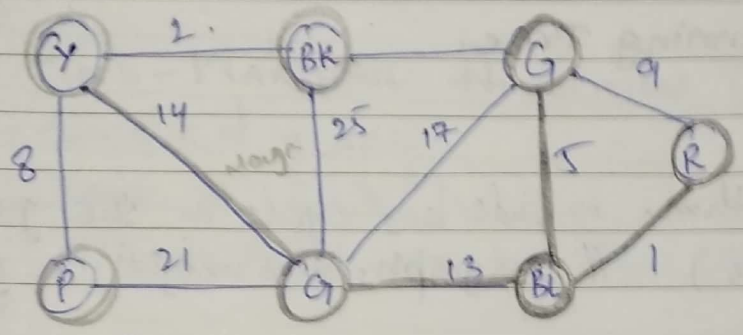


O(E lg V).

Kruskal's Algorithm

Kruskal()

→ Makeset() → yeh har vertex ko ek alg set me rkh raha hai.

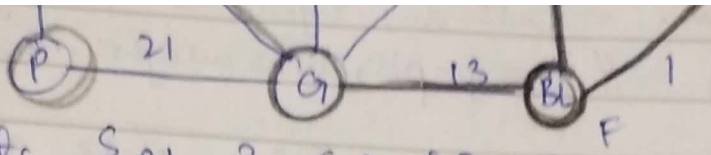


Sets = {Blue}, {Red}, {Gray}, {Black}, {Green} {Yellow}, {Purple}.

check smallest weight vertices. & put in E in sorted order.

E = {{Blue, Red}, {Yellow, Black}, ... }.

T = { {Blue, Red, Grey, Green} ← Merge. {Yellow, Black, Purple} }



Sets = {Blue}, {Red}, {Gray}, {Black}, {Green} {Yellow}, {Purple}.

check smallest weight vertices. & put in E in sorted order.

E = {{Blue, Red}, {Yellow, Black}, ... }.

T = { {Blue, Red, Grey, Green} ← Mega.
{Yellow, Black, Purple} }

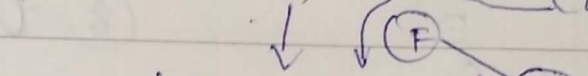
Disjoint Data Set:

07-06-23.

- Union - find data structure.

Union find algorithm:

- Make Set : Make sets which contains 1 element.
- ~~Find Set~~: {{A}, {B}, {C}, {D}, {E}, {F}, {G}}
- Find set: Find({F}) Find({G}).
- Union



Union({F} {G})

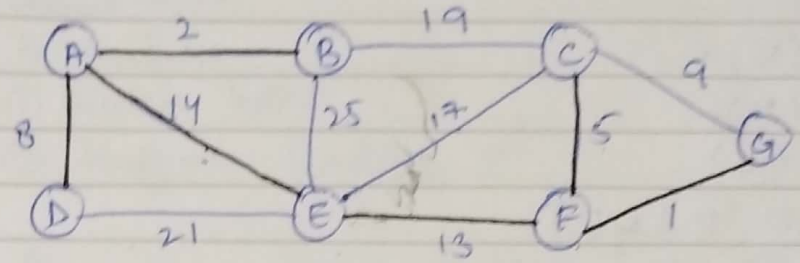
{F, G}

Union(FindSet({A}), FindSet({B})).

{A, B}.

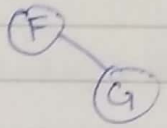
- Union by height
- Union by weight

Kruskal Algorithm

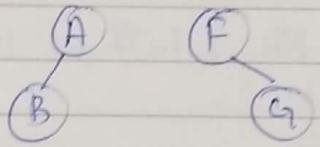


Make Set $\{\{A\},\{B\},\{C\},\{D\},\{E\},\{F\},\{G\}\}$

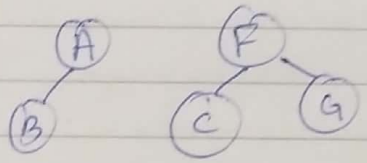
E = $\{\{F,G\},\{A,B\},\{F,C\},\{A,D\},\{C,G\},\{E,F\},$
edge set $\{A,E\},\{C,E\},\{B,C\},\{D,E\},\{B,E\}\}$.



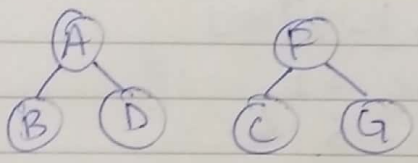
$\{A\},\{B\},\{C\},\{D\},\{E\},\{F,G\}$.



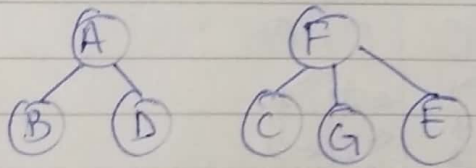
$\{A,B\},\{C\},\{D\},\{E\},\{F,G\}$.



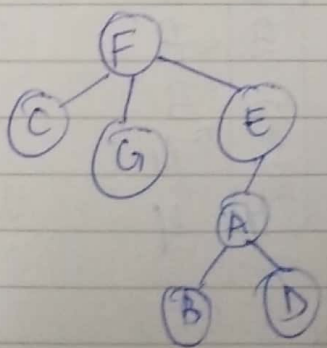
$\{A,B\},\{D\},\{E\},\{F,C,G\}$.



$\{A,B,D\},\{E\},\{F,C,G\}$.



$\{A,B,D\},\{F,C,G,E\}$



$\{A,B,D,F,C,G,E\}$

→ (Union by Weight).

Running Time

Make Set $O(V)$.
Find Set $O(E)$.
Union $O(V)$
Sort edges $O(E \lg E)$

Dynamic Programming.

- Factorial
- Fibonacci series.

Longest Common Sequence.

For X, find longest common subsequence in Y.

Brute - Force

For every subsequence of X, check whether it is a subsequence of Y.

A recursive Solution:

Case 1: $x_i = y_i$

$x_i = \langle A, B, D, E \rangle$

$y_i = \langle Z, B, E \rangle$.

$c[i, y] = c[i-1, j-1] + 1$.

↳ take diagonal and +1

		0	1	2	3	4	5	6
		Y_j	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0	0	0	0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↖2	B	←B
5	D	0	↑1	↖2	↑2	↑2	B	B
6	A	0	↑1	↑2	↑2	↖B	B	↖A
7	B	0	↖1	↑2	↑2	B	A	A

Length = 4

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

BCAB ← LCS

Running Time

Make Set $O(V)$.
Find Set $O(E)$.
Union $O(V)$
Sort edges $O(E \lg E)$

Dynamic Programming.

- Factorial
- Fibonacci Series.

Longest Common Sequence.

For X, find longest common subsequence in Y.

Brute - Force

For every subsequence of X, check whether it is a subsequence of Y.

A recursive Solution:

Case 1: $x_i = y_i$

$x_i = \langle A, B, D, E \rangle$

$y_i = \langle Z, B, E \rangle$.

$c[i, y] = c[i-1, j-1] + 1$.

↳ like diagonal me +1

		0	1	2	3	4	5	6
		Y_j	B	D	C	A	B	A
0	X_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

Length = 4

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

~~BCAB~~ ⇐ LCS ⇒ BCBA

14-06-23

Presentation

3 Members.

① Quick Sort

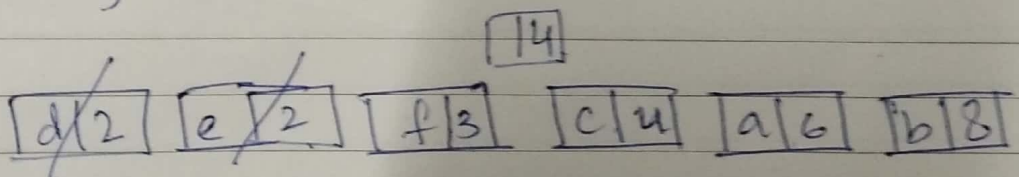
② Huffman Coding. ✓

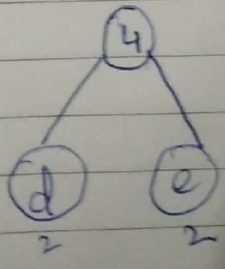
Algorithm uses, pseudo code, example, complexity.

↓ applications

```
Huffman (C)
{
    n = C.length
    Q = C.

    for i=1 to n-1
        z, x, y
        x = Extract Min(Q)
        y = Extract Min(Q).
        z.left = x
        z.right = y.
        z.freq = x.freq + y.freq
        Insert (Q, z).
    return Extract-Min(Q).
}.
```





Recurrence Relations

Binary Search

Merge Sort

Graphs
- Shortest Path Problems.
  - → Dijkstra's Algorithm
  - → Bellman-Ford Algorithm
  - → Floyd-Warshall's Algorithm.

~~Span~~ Trees
- Minimum Spanning Tree.
  - → Prim's Algorithm
  - → Kruskal's Algorithm. (Disjoint Data Structure)

Longest Common Subsequence → Dynamic Programming

Quick Sort →

Huffman Coding → Greedy