GATE CSE NOTES

by

Joyoshish Saha

Asymptotic Analysis

Introduction.

* The Game of Life
  — J.H. Conway 1970

* Top-down Refinement —
  Division of work into functions & using them in the main().

* Stubs - dummy functions for testing purpose.

* Structured Work-through
  — Explaining the program to other programmers starting from main() followed by the functions.

* Scaffolding
  — Snapshots of program execution by inserting printf surrounded by #ifdef's at key points.

* Static Analyzer
  — Program that examines the source program looking for uninitialised or unused variables, etc.

  eg. Unix utility 'Lint'.

* Principles of Program Testing —
  1. Black-box method —
     Easy values, typical-realistic, extreme values, Illegal values.

2. Glass-Box Method -

Checking all the paths of program execution. Not practicable for large programs.

* Variables. (x, y)
* Data types - Primitive (int, float, char, double, etc.)
  User-defined (structures, unions)
* Data Structure - Way of storing & organising data in a computer so that it can be used efficiently.

Depending on the organisation of the elements,

1. Linear data structure - Elements are accessed in a sequential order, but not compulsory to store all elements sequentially.

   eg. Linked lists, stack, queue.

2. Non-linear data structure - Elements stored/accessed in a non-linear order.

   eg. Tree, Graph.

* ADT — Declaration of Data
        Declaration of Operations.

→ Commonly used ADTs —

Linked list, stack, queue, priority queues, binary trees, dictionaries, disjoint sets (union & find), hash tables, graph etc.

* The goal of analysis of algorithms is to compare algorithms mainly in terms of running time but also in terms of other factors like memory, developer's effort.

* Running Time Analysis. —

Process of determining how processing time increases as the size of the problem (input size) increases.

Common types of input —

Size of an array, polynomial degree, no. of elements in a matrix, vertices & edges in a graph.

* Algorithm — Sequence of computational procedure that transform input into the output.

[Word Algorithm comes from Persian author — Abu Jafar Mohammed ibn Musa al Khawarizmi]

Mathematically algorithm is a function.

$$f : I \rightarrow O$$

Where I is the set of inputs & O is the set of outputs.

* Types of algorithms —

1. Top-down approach (Iterative algo)
2. Bottom-up approach (Recursive algo)

eg. finding a factorial —

a) Iterative.

```
Fact(n)
{
    for i = 1 to n
        fact = fact * i ;
    return fact ;
}
```

calculated as $1 \times 2 \times 3 \times \ldots \times (n-1) \times n$

b) Recursive.

```
Fact(n)
{
    if n = 1
        return 1 ;
    else
        return n * Fact(n-1) ;
}
```

calculated as $n \times (n-1) \times \ldots \times 3 \times 2 \times 1$.

* Analysis of Algorithms involves the following measures —

1. Correctness.

Providing a proof of correctness of the algorithm.

2. Amount of work done - Time Complexity

Time complexity does not refer to the actual running time of an algorithm in terms of millisec. The actual running time depends on the system config. Time complexity must be defined in terms of step count: the number of times each statement in the algo is executed. We express the step count (time complexity) as a function of input size as the tc increases with problem size.

3. Space Complexity.

An indicator of problem size. Related complexity measure that refers to the amount of space used by an algo.

4. Optimality.

Finding the best possible solution.

* Step-count Method.

Time & space complexity are calculated using step-count method.

Some basic assumptions —

i) No count for { and }.

ii) Basic statement like assignment & return have a count of 1.

iii) If a basic statement is iterated, then multiply by the no. of times the loop is run.

iv) The loop statement is iterated n times; it has a count of (n+1).

eg.

1. Sum of elements in array.

```
Algo Sum(a,n)                    | T.C.      | Space
{                                |           |
  sum = 0;                       | 1         | 1 word for sum
  for i = 1 to n                 | n+1       | 1 word for each i, n
    sum = sum + a[i];            | n         | n words for array
  return sum;                    | 1         |
}                                |           |
                                 | 2n+3      | 1+2+n = n+3.
```

2. Adding two matrices of order m & n.

```
Algo Add(a, b, c, m, n)          | T.C.
{                                |
  for i = 1 to m                 | m+1
    for j = 1 to n               | (n+1) m
      c[i,j] = a[i,j] + b[i,j];  | m.n
}                                |
                                 | 2mn + 2m + 2
```

3. Fibonacci Series.

```
algo Fibonacci (n)                          T.C.
{
    if n <= 1                                1
      output 'n'
    else {
      f2 = 0 ;                               1
      f1 = 1 ;                               1

      for i = 2 to n                         n
      {
          f = f1 + f2                        n-1
          f2 = f1                            n-1
          f1 = f                             n-1
      }
    output 'f';}                             1
}                                          ------
                                           4n+1.
```

4. Recursive sum of elements on an array.

```
algo Sum (a, n)                             Step count
{
    if n <= 0                                  1
      return 0 ;                               1
    else
      return Sum (a, n-1) + a[n] ;             2 + sc of
                                                 recursive
}                                                call.
```

If step count of array sized n, is T(n),

$$T(n) = 3 + T(n-1) \quad , n > 0$$

$$T(n) = 2 \quad , \quad n \leq 0$$

$$\Rightarrow \quad T(n) = 3n + 2.$$

1

lg lg n

√lg n

lg n

(lg n)^c c≥1

n^c 0<c<1

n = 2^lg n

n log* n

n lg n = lg n!

n^2

n^c c>2

c^n c>1

n!

2^2^n

* Order/Rate of Growth.

The rate at which the running time increases as a fun^n of i/p is called rate of growth.

It gives a simple characterization of the algorithm's efficiency by identifying relatively significant term on the step count. For a step count $2n^2+3n+1$, the order of growth depends on $2n^2$ for large n.

→ Commonly used Rate of Growths (In decreasing order)

2^{2^n} → n! → 4^n → 2^n → n^2 → n log n

Highest

log(n!) ← n ← $2^{\log n}$ ← $\log^2 n$ ← $\sqrt{\log n}$

→ log log n → 1. lowest.

→ Rate of growths for different operations.

i) Adding element to front of linked list - 1

ii) Finding element in sorted array - log n

iii) Finding element in unsorted array - n

iv) Sorting n items by divide & conquer - Mergesort - n log n

v) Shortest path between two nodes in a graph – n^2

vi) Matrix multiplication – n^3

vii) Tower of Hanoi Problem – 2^n.

* Types of Analysis. –

1. Worst Case – defines input for which the algo takes long time, i/p is the one for which the algo runs the slowest.

2. Best case – defines input for which the algo takes lowest time; i/p is the one for which the algo runs the fastest.

3. Average case – provides a prediction about the running time of the algorithm.

* Asymptotic Notation.

Asymptotic analysis focuses analysis on the significant term. To represent lower, upper bounds we need some kind of syntax – asymptotic notations.

1. Big-oh Notation – (O) To express an upper bound on TC. as a f^n of i/p size.

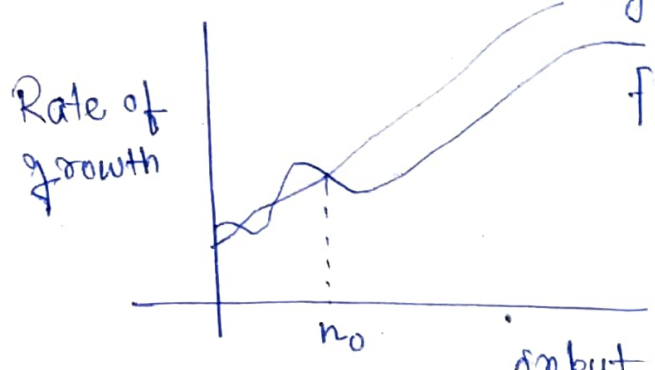
2. Omega (Ω) – Express a lower bound.

3. Theta (Θ) – Express the tight bound on TC.

→ Asymptotic Upper Bounds.

- **Big-oh Notation.**

Gives the tight upper bound of the given function.



$f(n) = O(g(n))$ if & only if ∃ positive constants c, n_0 such that $f(n) \leq cg(n)$ $\forall n \geq n_0$

- It gives the smallest rate of growth $g(n)$ that is greater than or equal to given algo's rate of growth. $f(n)$.
- Big-oh captures the worst case analysis.
- There are no unique set of values for n_0 & c in providing asymptotic bounds.

eg. i) $f(n) = 100n + 6$

$100n + 6 \leq 100n + n = 101n$

$c = 101$, $n_0 = 6$ $\quad f(n) \leq c\,g(n)$

∴ for all $n \geq 6$,

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

ii) $f(n) = n^4 + 100n^2 + 50.$

$f(n) \leq 2n^4 \quad \forall \; n \geq 11$

$f(n) = O(n^4)$ with $c = 2,\ n_0 = 11.$

iii) $f(n) = 6 \cdot 2^n + n^2.$

$f(n) \leq 7 \cdot 2^n \quad \forall \; n \geq 7.$

$f(n) = O(2^n)$ with $c = 7$ & $n_0 = 7.$

iv) $f(n) = n.$

$f(n) \leq n \quad \forall \; n \geq 1.$

$f(n) = O(n)$ with $c = 1,\ n_0 = 1.$

v) $f(n) = 410$

$410 \leq 410 \quad \forall \; n \geq 1.$

$410 = O(1)$ with $c = 1,\ n_0 = 1.$

vi) $f(n) = n!$

$n! \leq c\, n^n \quad \forall \; n \geq 2.$

$n! = O(n^n)$ with $c = 1,\ n_0 = 2.$

. Stirling's approximation – $n! \approx \sqrt{2n\pi}\, n^n e^{-n}.$

- **o-notation.** (Little oh).

$f(n) = o(g(n))$ if for any positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that $0 \leq f(n) < c\,g(n)$ for all $n \geq n_0$.

- Captures the loose upper bounds.

eg. i) $2n = o(n^2)$

n^2 is polynomially larger than $2n$ by n^{ϵ}, $\epsilon = 1$.

ii) $10n^2 + 4n + 2 = o(n^3)$

n^3 is polynomially larger than $10n^2 + 4n + 2$ by n^{ϵ}, $\epsilon = 1$.

iii) $n^3 + n + 5 = o(n^{3.1})$

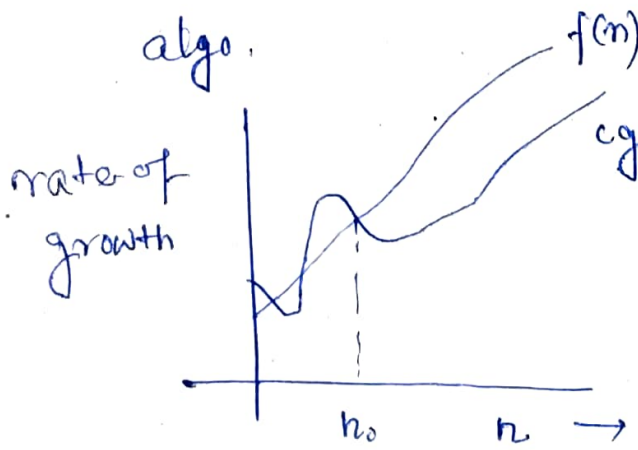
$\epsilon = 0.1$.

- $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$. $\quad f(n) < g(n)$

→ **Asymptotic Lower Bounds**

- **Omega Notation (Ω)**

Gives the tight lower bound of given algo.



$f(n) = \Omega(g(n))$ iff $\exists$ $c, n_0 > 0$, s.t. $f(n) \geq cg(n) \geq 0$ $\forall n \geq n_0$.

- denotes best case analysis of an algo.

• Gives the largest rate of growth $g(n)$ that is less than or equal to given algo's rate of growth $f(n)$.

eg. i) $f(n) = 5n^2$.

$\exists\ c, n_0$ s.t. $0 \le cn^2 \le 5n^2 \Rightarrow cn^2 \le 5n^2$

$\Rightarrow \quad c = 6$ & $n_0 = 1$.

$\therefore 5n^2 = \Omega(n^2)$ with $c = 6$ & $n_0 = 1$.

ii) $f(n) = 100n + 5$

$100n + 5 \geq n \quad \forall\ n \geq 1$.

$f(n) = \Omega(n)$ with $c = 1$, $n_0 = 1$.

iii) $f(n) = n^3 + n + 5$.

$n^3 + n + 5 \geq n^3 \quad c = 1 \ \forall\ n \geq 0$.

$f(n) = \Omega(n^3)$.

iv) $f(n) = 2n^2 + n\log n + 1$.

$f(n) \geq 2n^2$, $c = 2 \ \forall\ n \geq 1$

$f(n) = \Omega(n^2)$

v) $f(n) = 6 \cdot 2^n + n^2 = \Omega(2^n)$

$6 \cdot 2^n + n^2 \geq 2^n$, $c = 1 \ \forall\ n \geq 1$.

$f(n) = \Omega(n^2) \Rightarrow f(n) = \Omega(n) \Rightarrow f(n) = \Omega(1)$.

$\Omega(2^n)$ is tight lower bound ; others are loose lower bounds.

• ω - Notation (Little Omega).

$f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

or in other more formal way,

$\omega(g(n)) = \{ f(n) :$ for any +ve cons. $c > 0$,
$\exists \; n_0 > 0$ s.t. $0 \leq cg(n) < f(n)$
$\forall \; n \geq n_0 \}$.

• $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$. $\quad f(n) > g(n)$

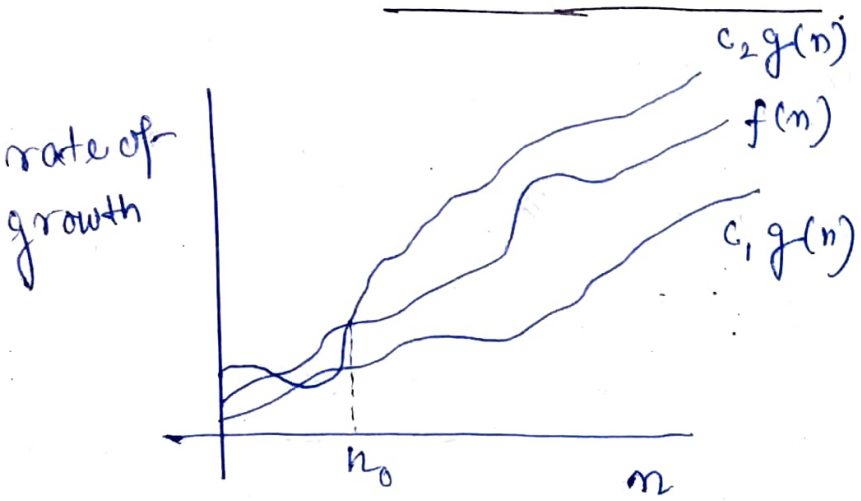
eg. i) $n^2 = \omega(n)$, but $n^2 \neq \omega(n^2)$

ii) $3n + 2 = \omega(\log n)$.

iii) $2n^2 + n\log n + 1 = \omega(n^{1.9})$.

→ Asymptotic Tight Bound.

• Theta Notation.



$\Theta(g(n)) = \{ f(n) : \exists$ positive constants c_1, c_2,
n_0 s.t. $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\forall \; n \geq n_0 \}$

• $g(n)$ is lower bound as well as upper bound of $f(n)$.

- $f(n) = \Theta(g(n))$ if & only if $f(n) = O(g(n))$ & $f(n) = \Omega(g(n))$.

eg. i) $f(n) = 3n + 10^{10}$.

$3n \leq 3n + 10^{10} \leq 4n \quad \forall\ n \geq 10^{10}$.

$f(n) = \Theta(n)$, with $c_1 = 3$, $c_2 = 4$, $n_0 = 10^{10}$

ii) $f(n) = \frac{n^2}{2} - \frac{n}{2}$.

$\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2. \quad \forall\ n \geq 1.$

$f(n) = \Theta(n^2)$ with $c_1 = 1/5$, $c_2 = 1$, $n_0 = 1$

iii) Prove $n \neq \Theta(\log n)$.

$c_1 \log n \leq n \leq c_2 \log(n)$.

$\Rightarrow c_2 \geq \frac{n}{\log n} \quad \forall\ n \geq n_0.$

Not possible.

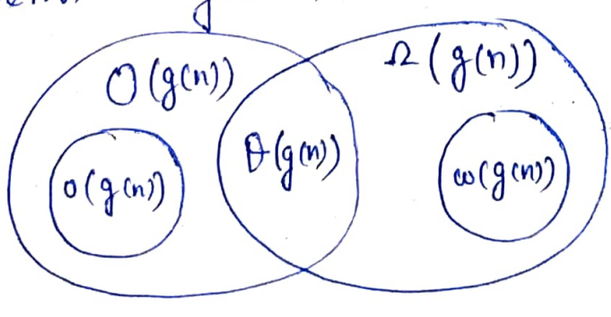
iv) $f(n) = n\sqrt{n} + n\log n + 2$

$n\sqrt{n} \leq n\sqrt{n} + n\log n + 2 \leq 5\, n\sqrt{n}.$

$\forall\ n \geq 2, \quad c_1 = 1, \quad c_2 = 5.$

$f(n) = \Theta(n\sqrt{n})$.

* On a Venn Diagram,



→ Properties of Asymptotic Notation.

1. Reflexivity.

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

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

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

2. Symmetry

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

proof.

Necessary part.

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

$c_1 g(n) \leq f(n) \leq c_2 g(n)$. $\forall\, n \geq n_0$

$\Rightarrow g(n) \leq \frac{1}{c_1} f(n)$ and $g(n) \geq \frac{1}{c_2} f(n)$

$\Rightarrow \frac{1}{c_2} f(n) \leq g(n) \leq \frac{1}{c_1} f(n)$.

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

as $1/c_1$, $1/c_2$ are well-defined.

Sufficiency part.

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

$c_1 f(n) \leq g(n) \leq c_2 f(n)$. $\forall\, n \geq n_0$

$\Rightarrow f(n) \leq \frac{1}{c_1} g(n)$ & $f(n) \geq \frac{1}{c_2} g(n)$.

$\Rightarrow \frac{1}{c_2} g(n) \leq f(n) \leq \frac{1}{c_1} g(n)$.

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

3. Transitivity.

- $f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n))$

$\Rightarrow f(n) = \Theta(h(n))$.

Proof.

$f(n) = \Theta(g(n))$ & $g(n) = \Theta(h(n))$.

$\Rightarrow f(n) = \Theta(h(n))$.

$f(n) \leq c_1 g(n)$. $\forall n \geq n_0$.

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

$g(n) \leq c_2 h(n)$.

$f(n) \leq c_1 c_2 h(n)$.

$f(n) \leq c h(n)$. , $c = c_1 c_2$.

$\therefore f(n) = \Theta(h(n))$.

- Θ & Ω also satisfy transitivity.

4. Transpose Symmetry.

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

Proof.

Necessity.

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

$g(n) \geq \frac{1}{c} f(n)$.

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

Sufficiency

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

$\Rightarrow f(n) \leq \frac{1}{c} g(n)$

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

→ More Observations.

- Lemma 1. Let $f(n)$ & $g(n)$ be two asymptotic non-negative functions.

Then $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

* Proof. Assume $f(n) \leq g(n)$

$\Rightarrow \max(f(n), g(n)) = g(n)$.

* Consider, $g(n) \leq \max(f(n), g(n)) \leq g(n)$

$\Rightarrow g(n) \leq \max(f(n), g(n)) \leq f(n) + g(n)$.

$\Rightarrow \frac{1}{2} g(n) + \frac{1}{2} g(n) \leq \max(f(n), g(n))$

$\leq f(n) + g(n)$

Now, we can write based on assumptions made,

$$\frac{1}{2} f(n) + \frac{1}{2} f(n) \leq \max(f(n), g(n)) \leq f(n) + g(n).$$

$$\Rightarrow \frac{1}{2}\left(f(n) + g(n)\right) \leq \max\left(f(n), g(n)\right) \leq f(n) + g(n).$$

$$\therefore \max\left(f(n), g(n)\right) = \Theta\left(f(n) + g(n)\right).$$

- **Lemma 2.** For two asymptotic functions $f(n)$ & $g(n)$,

$$O(f(n)) + O(g(n)) = O\left(\max(f(n), g(n))\right).$$

proof. Assume $f(n) \leq g(n)$.

$$\Rightarrow O(f(n)) + O(g(n)) \leq c_1 f(n) + c_2 g(n)$$

Now, we can write

$$O(f(n)) + O(g(n)) \leq c_1 g(n) + c_2 g(n).$$
$$\leq (c_1 + c_2) g(n)$$
$$\leq c\, g(n).$$
$$\leq c \max\left(f(n), g(n)\right).$$

$$\therefore O(f(n)) + O(g(n)) = O\left(\max(f(n), g(n))\right)$$

→ (1) If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$, $c \in R^+$ then

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

(2) If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$, $c \in R$ (c can be 0)

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

(3) If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = O(g(n))$

& $g(n) \neq O(f(n))$. $f(n) = o(g(n))$ (small 'o')

(4) If $\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c$, $c \in R$ (c can be ∞)

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

(5) If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then $f(n) = \Omega(g(n))$

& $g(n) \neq \Omega(f(n))$. $f(n) = \omega(g(n))$

• **Lemma 3.** Show that $\log n = O(\sqrt{n})$ however $\sqrt{n} \neq O(\log n)$.

Proof. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}}$.

Applying L'Hôpital rule,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{1/n}{\frac{1}{2} n^{-1/2}}$$

$$= \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

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

$\Rightarrow \log n = O(\sqrt{n})$.

Proof for $\sqrt{n} \neq O(\log n)$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\sqrt{n}}{\log n}$$

$$= \lim_{n\to\infty} \frac{\frac{1}{2} n^{-1/2}}{1/n}$$

$$= \lim_{n\to\infty} \frac{\sqrt{n}}{2} = \infty$$

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

$\Rightarrow \sqrt{n} = \Omega(\log n)$

$\Rightarrow \sqrt{n} \neq O(\log n)$.

- NTB. There are asymptotic functions that cannot be compared using these notations.

eg. i) $f(n) = n$ & $g(n) = n^{1+\sin(n)}$

ii) $f(n) = n\cos^2(n)$ & $g(n) = n\sin^2(n)$.

$g(n)$ is a periodic fun^n that oscillates between 1 & n^2.

* Commonly used logarithms & Summations.

- $a^{\log_b x} = x^{\log_b a}$
- $\sum_{K=0}^{n} x^K = \frac{x^{n+1} - 1}{x - 1} \quad (x \neq 1)$.
- $\sum_{K=1}^{n} \frac{1}{K} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} \approx \log n$.
- $\sum_{K=1}^{n} \log K \approx n \log n$.
- $\sum_{K=1}^{n} K^p \approx \frac{1}{p+1} n^{p+1}$.

* Guidelines for Asymptotic Analysis

1. $O(1)$: Time complexity of a f^n is considered as $O(1)$ if it doesn't contain loop, recursion & call to any other non-constant time function.

2. $O(n)$: TC of a loop is $O(n)$ if the loop variables are incremented by a constant amount.

3. $O(n^c)$: TC of nested loops is equal to the no. of times the innermost statement is executed.

4. $O(\log n)$: TC of a loop is $O(\log n)$ if loop variable is divided/multiplied by a constant amount.

5. $O(\log \log n)$: If loop variable is reduced/increased exponentially by a constant amount.

→ When there are consecutive loops, we calculate time complexity as sum of TC's of individual loops.

→ When there are many if-else statements inside loops, we evaluate the situation when values in if-else conditions cause maximum number of statements to be executed.

Q. G'91.

```
A() {
    i = 1, s = 1 ;
    while (s <= n)
    {   i++ ;
        s = s + i ;
        print - ;
    }
}
```

n^th S is first n natural numbers' sum.

$1+2+3+\cdots = \frac{K(K+1)}{2}$

S: 1 → 3 → 6 10 15 21 . . . <= n

i: 1 2 3 4 5 6 . . . . K

Assume after K iterations the loop stops (s > n).

$s = \frac{K(K+1)}{2}$ by observation

$\frac{K(K+1)}{2} \not\leq n.$

$\Rightarrow \frac{K^2+K}{2} \not\leq n.$

$\therefore K = O(\sqrt{n})$. (Ans).

Q.
```
A() {
    i = 1 ;
    for ( i = 1 ; i <= √n ; i++)
        print -
}
```

$f(n) = O(\sqrt{n})$

$f(n) = \Omega(\sqrt{n})$

$f(n) = \underline{\Theta(\sqrt{n})}$.

Q.
```
A() {
    int i, k, j, n;
    for ( i = 1 ; i <= n ; i++)
    {   for (j = 1 ; j <= i ; j++)
        {   for (k = 1, k <= 100 ; k++)
                print -
        }
    }
}
```

$i = 1$
$j = 1$ time
$K = 100$ times

$i = 2$
$j = 2$ times
$K = 2 \times 100$ times

. . . .

$i = n$
$j = n$ times
$K = n \times 100$ times

So, the innermost print statement gets executed

$$100 + 2\times100 + 3\times100 + \cdots + n \times 100 \text{ times}$$

$$= 100 \frac{n(n+1)}{2} \text{ times.}$$

$$\therefore f(n) = O(n^2). \quad \text{(Ans.)}$$

Q.
```
A() {
    int i, j, k, n;
    for (i = 1; i <= n; i++)
    {
        for (j = 1; j <= i^2; j++)
        {
            for (k = 1; k <= n/2; k++)
                print -
        }
    }
}
```

$i = 1$
$j = 1$ time
$K = n/2$ times

$i = 2$
$j = 4$ times
$k = 4 \cdot n/2$ times.

. . .

$i = n$
$j = n^2$ times,
$K = n^2 \cdot n/2$ times

$$f(n) = n/2 + 4 \cdot n/2 + \cdots + n^2 n/2$$

$$= n/2 \left(1 + 4 + 9 + \cdots + n^2\right) = \frac{n}{2} \frac{n(n+1)(2n+1)}{6}.$$

$$f(n) = O(n^4)$$

Q. A() {

for(i=1; i<n; i= i*2) — Multiplied by constant amount.

print -

}

i = 1, 2, 4, ... , n

Assume no of iterations = K.

$i = 2^0, 2^1, 2^2, \ldots, 2^K$ steps

$2^K = n \Rightarrow K = \log_2 n$

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

Note - in the loop when the variable is incremented by multiple, $i = K * i$, then $f(n) = O(\log_K n)$

Q. A() {

int i, j, K, n;

for(i= n/2; i<=n; i++) — n/2

for (j=1; j<=n/2; j++) — n/2

for (K=1; K<=n; K=K*2) — $\log_2 n$

print -

}

$f(n) = \frac{n}{2} \times \frac{n}{2} \times \log_2 n = O(n^2 \log_2 n)$.

Q. A() {

for(i= n/2; i<=n; i++) — n/2

for(j=1; j<=n; j=2*j) — $\log_2 n$

for (K=1; K<=n; K=K*2) — $\log_2 n$

print -

}

$f(n) = O(n(\log_2 n)^2)$

$O(n(\log_2 n)^2)$

Q. Assuming $n \geq 2$

```
A() {
    while (n > 1)
        n = n/2 ;
}
```

$n = 2^k$

$n = 2^1$ — 1 time

$n = 2^2$ — 2 times

$n = 2^3$ — 3 times

⋮

$n = 2^k$ — $\log_2 n$ times

when $n \neq 2^k$

$f(n) = O(\lfloor \log_2 n \rfloor)$.

Q.
```
A() {
    for (i=1 ; i<=n ; i++)          — n
        for (j=1 ; j<=n ; j=j+i)
            print —
}
```

$i = 1$	$i = 2$	$i = 3$	...	$i = k$
$j = n$ times	$j = n/2$ times	$j = n/3$ times	...	$j = n/k$ times

$f(n) = n + \frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n}$

$= n\left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right)$.

$= n \log n$

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

Oct 4, 20
10.46 pm
memberinveg
oMm

Q.
```
A() {
    int n = 2^(2^k) ;
    for (i=1 ; i<=n ; i++) {        — n
        j = 2 ;
        while (j<=n)
            { j = j^2 ;
              print — }
    }
}
```

$2, 2^2, 2^4, 2^8, \ldots, 2^{2^k}$

$k = \log(\log n)$

$\varepsilon (\log \log n)$

$n = 2^{2^k}$

$\Rightarrow k = \log\log n$

$k = 1$
$n = 4$.
$j = 2$ times
print 4×2 times

$k = 2$
$n = 16$
$j = 3$ times
print 16×3 times

Innermost statement execution

$n(k+1)$ times.

$\Rightarrow n(\log\log n + 1)$

$f(n) = O(n \log\log n)$

→ Time Complexity for recursive algorithm.

eg. A(n) {
    if (n > 1)
    return A(n-1)
}

$T(n) = 1 + T(n-1), \quad n > 1$

$= 1, \quad n = 1$

☑. By Back-substitution —

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

$T(n-2) = 1 + T(n-3)$.

⋮

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

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

$= k + T(n-k)$. for k iterations

To stop $n - k = 1 \Rightarrow k = n - 1$.

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

$T(n) = n. = O(n)$, Ans.

---

eg. $T(n) = n + T(n-1), \quad n > 1$

$= 1, \quad n = 1$

By back-substitution,

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

$T(n-2) = (n-2) + T(n-3)$

⋮

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

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

$= n + (n-1) + (n-2) + T(n-3)$

$= n + (n-1) + (n-2) + \cdots + (n-k) + T(n-(k+1))$.

We need to remove the $T(\cdot)$ expression from RHS & express $T(n)$ in terms of n only (by base condition or when the code halts).

Base condition,

$n - (k+1) = 1 \Rightarrow k = n-2$.

$T(n) = n + (n-1) + (n-2) + \cdots + (n-n+2) + T(1)$.

$= n + (n-1) + (n-2) + \cdots + 2 + 1$.

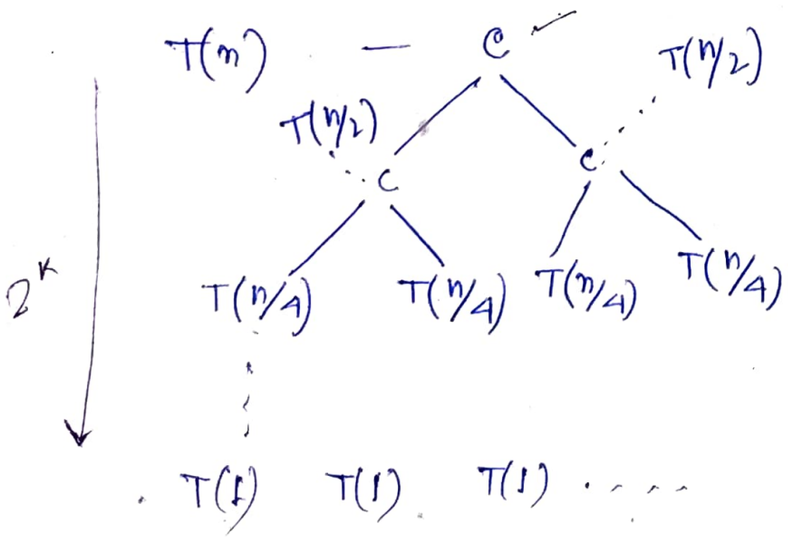
$= \frac{n(n+1)}{2}$

$T(n) = O(n^2)$. (Ans.)

Recursion Tree Method.

eg. $T(n) = 2T(n/2) + c$, $n > 1$

$= c$, $n = 1$.



$T(n) = c + 2c + \cdots + nc$

$= c(1 + 2 + 4 + \cdots + n)$

assuming $n = 2^k$.

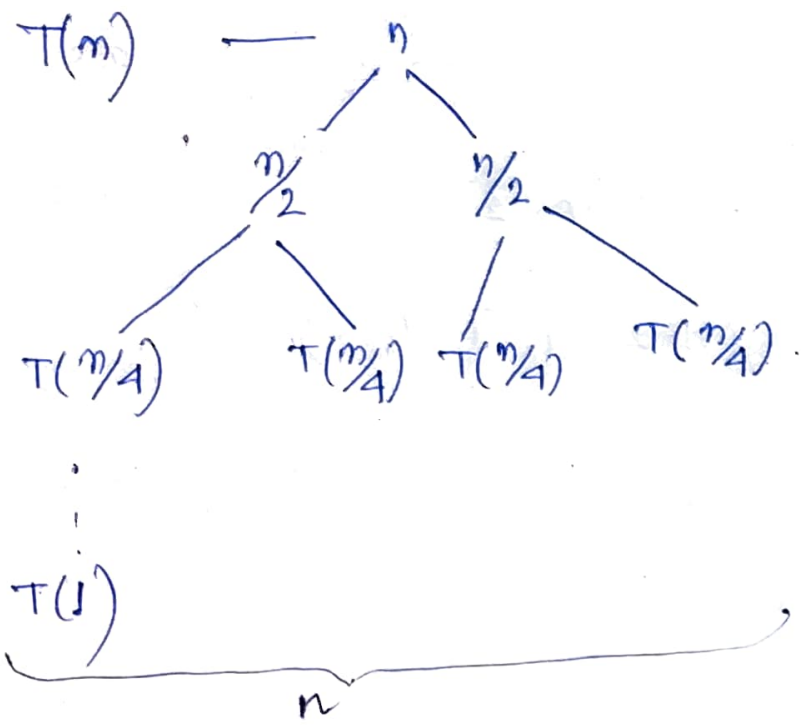
$= c \cdot \frac{1(2^{k+1} - 1)}{2-1}$

$= c(2n - 1)$.

$T(n) = O(n)$ (Ans)

eg. $T(n) = 2T(n/2) + n \quad , n > 1$

$= 1 \quad , n = 1.$



n	$n/2^0$
n	$n/2^1$
n	$n/2^2$
n	$n/2^K = 1$

$(K+1)$ levels.

$n = 2^K$

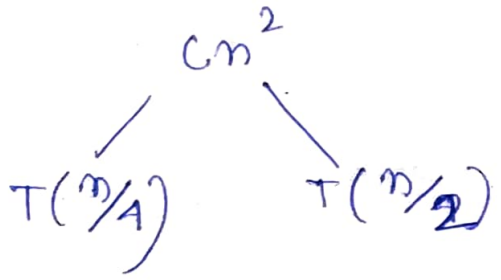
$K = \log_2 n$.

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

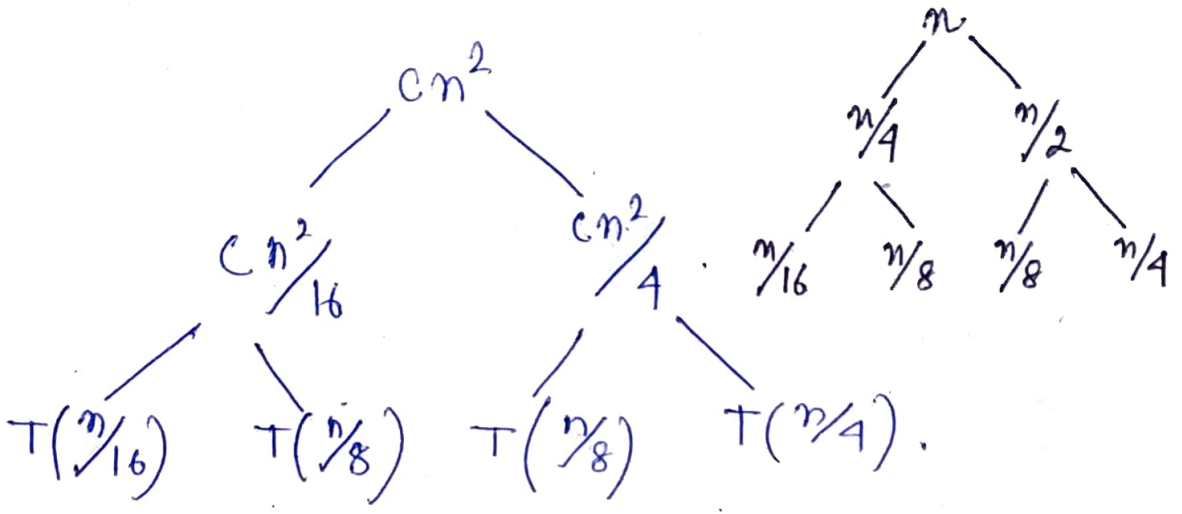
$= O(n \log n)$.

Note: We draw a recurrence tree & calculate the time taken by every level of tree. finally, we sum the work done at all levels. To draw, the recurrence tree, we start from the given recurrence & keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

eg. $T(n) = T(n/4) + T(n/2) + cn^2$



Further breaking down $T(n/4)$, $T(n/2)$,



cn^2

$c\frac{n^2}{16} + c\frac{n^2}{4}$

$c\frac{5}{16}n^2$

To know value of $T(n)$, we need to calculate sum of tree nodes level by level.

$$T(n) = c\left(n^2 + \frac{5n^2}{16} + \frac{25n^2}{256} + \cdots\right)$$

$$= c\,\frac{n^2}{1 - 5/16}$$

$$T(n) = O(n^2).$$

$c\left(\frac{n}{16}\right)^2 + 2c\left(\frac{n}{8}\right)^2 + c\left(\frac{n}{4}\right)^2$

$= cn^2\left(\frac{1}{256} + \frac{1}{32} + \frac{1}{16}\right)$

$= cn^2 \cdot \frac{25}{256}$

* Comparing various functions

eg. 2^n $\quad n^2$

$n\log_2 2$ $\quad 2\log_2 n$

n $\quad 2\log_2 n$

for $n = 2^{100}$,

2^{100} $\quad 200$.

$\therefore 2^n$ is asymptotically larger.

eg. 3^n 2^n

$n \log 3$ $n \log 2$.

$\log 3$ $\log 2$

3^n is greater.

We can say $2^n = O(3^n)$.

eg. n^2 $n \log n$.

n $\log n$.

n^2 larger.

eg. n $(\log n)^{100}$

$\log n$ $100 \log \log n$.

for $n = 2^{128}$

128 700.

n larger after some big values.

for $n = 2^{1024}$

1024. 1000

eg. $n^{\log n}$ $n \log n$

$\log n \cdot \log n$ $\log n + \log \log n$

$n^{\log n}$ larger

$n = 2^{1024}$

1024×1024 $1024 + 10 = 1034$.

$n = 2^{2^{20}}$

$2^{20} \cdot 2^{20}$ $2^{20} + 20$

eg. $\sqrt{\log n}$ $\log\log n$

$\frac{1}{2}\log\log n$ $\log\log\log n$. $\sqrt{\log n}$ larger.

$n = 2^{2^{10}}$

5 $\log_2 10$

eg. $n^{\sqrt{n}}$ $n^{\log n}$

$\sqrt{n}\log n$ $\log n \cdot \log n$ $n^{\sqrt{n}}$ larger.

$\sqrt{n}$ $\log n$.

$\frac{1}{2}\log n$ $\log\log n$.

$n = 2^{128}$

64. 7.

eg: $f(n) = \begin{cases} n^3 & 0 < n < 10000 \\ n^2 & n \geq 10000 \end{cases}$

$g(n) = \begin{cases} n & 0 < n < 100 \\ n^3 & n > 100 \end{cases}$

	0 - 99	100 - 9999	10000 - ∞
$f(n)$	n^3	n^3	n^2
$g(n)$	~~n^3~~ n	n^3	n^3.

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

$n_0 = 10000$.

eg. $f_1 = 2^n$, $f_2 = n^{3/2}$, $f_3 = n\log n$, $f_4 = n^{\log n}$

① 2^n $\quad$ $n^{3/2}$

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

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

$n = 2^{128}$

2^{128} $\quad$ 3×64.

2^n larger.

② 2^n $\quad$ $n\log n$

$n\log 2$ $\quad$ $\log n + \log\log n$

n $\quad$ $\log n + \log\log n$

$n = 2^{128}$

2^{128} $\quad$ $128 + 7$

2^n larger.

③ 2^n $\quad$ $n^{\log n}$.

$n\log 2$ $\quad$ $\log n \cdot \log n$

n $\quad$ $(\log n)^2$

$n = 2^{128}$

2^{128} $\quad$ 128×128

2^n larger.

④ $n^{\log n}$ $\quad$ $n^{3/2}$

$(\log n)^2$ $\quad$ $\frac{3}{2}\log n$

$\log n$ $\quad$ $\frac{3}{2}$.

$n^{\log n}$ larger

f_1 is the largest

⑤ $n^{\log n}$ $\quad$ $n\log n$.

$(\log n)^2$ $\quad$ $\log n + \log\log n$

$n = 2^{128}$

128^2 $\quad$ $128 + 7$

$n^{\log n}$ larger.

f_4 is second largest

⑥ $n^{3/2}$ $\quad$ $n\log n$

$\frac{3}{2}\log n$ $\quad$ $\log n + \log\log n$

$n = 2^{128}$

$\frac{3}{2}\times 128$ $\quad$ $128 + 7$

$n = 2^{1024}$

$\frac{3}{2}(1024)$ $\quad$ $1024 + 7$.

$n^{3/2}$ is larger.

$\therefore f_1 > f_4 > f_2 > f_3$ (Ans)

* Master Theorem. (Divide & Conquer)

Formula for solving recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$, $b > 1$ & $f(n)$ is asymptotically positive ($f(n)$ is positive for all sufficiently large n).

This recurrence describes an algo that divides a problem of size n into a subproblems each of size n/b & solves them recursively.

- n/b might not be an integer.

But, replacing $T(n/b)$ with $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ does not affect the asymptotic behaviour of the recurrence.

- **Theorem**

Let $a \geq 1$, $b > 1$ be constants, let $f(n)$ be a function & let $T(n)$ be defined on the nonnegative integers by the recurrence $T(n) = aT(n/b) + f(n)$, where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then

$$T(n) = \Theta(n^{\log_b a}).$$

2. If $f(n) = \Theta(n^{\log_b a})$, then

$$T(n) = \Theta(n^{\log_b a} \log n).$$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ & if $af(n/b) \leq cf(n)$ for some constant $c < 1$ & all sufficiently large n, then $T(n) = \Theta(f(n))$.

- Another form of Theorem. (Extended Master)

If the recurrence is of the form

$T(n) = aT(n/b) + \Theta(n^k \log^p n)$, where $a \geq 1$, $b > 1$, $k \geq 0$ & p is a real number, then:

1. If $a > b^k$, then $T(n) = \Theta(n^{\log_b a})$
2. If $a = b^k$,
   i) If $p > -1$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$
   ii) If $p = -1$, then $T(n) = \Theta(n^{\log_b a} \cdot \log\log n)$

$n^{\lg_b a} \cdot \lg^{p+1} n$

$\begin{cases} T(n) = T(n/2) + 2^n \Rightarrow T(n) = O(2^n) \\ T(n) = 2T(n/2) + n! \Rightarrow T(n) = O(n!) \end{cases}$

- Inadmissable equations (can't be solved using Master):

i) $T(n) = 2^n T(n/2) + n^n$ ii) $T(n) = 2T(n/2) + \frac{n}{\lg n}$ (can apply extended Master) iii) $T(n) = 0.5T(n/2) + n$ iv) $T(n) = 64T(n/8) - n^2 \lg n$ — not +ve

iii) If $p < -1$, then

$$T(n) = \Theta\left(n^{\log_b a}\right).$$

3. If $a < b^k$,

i) If $p \geq 0$, then

$$T(n) = \Theta\left(n^k \log^p n\right)$$

ii) If $p < 0$, then

$$T(n) = O\left(n^k\right).$$

eg. $T(n) = 3T\left(n/2\right) + n^2$.

$a = 3, \quad b = 2, \quad k = 2, \quad p = 0.$

$a < b^k$

$3 < 2^2$

$\therefore T(n) = \Theta\left(n^2\right).$

by 3.i.

eg. $T(n) = 4T\left(n/2\right) + n^2$.

$a = 4, \quad b = 2, \quad k = 2, \quad p = 0. > -1$

$a = b^k$

$4 = 2^2$

$\therefore T(n) = \Theta\left(n^{\log_2 4} \log n\right)$

$T(n) = \Theta\left(n^2 \log n\right).$

by 2.i.

v) $T(n) = 2T(n/2) + n(2 - \cos n)$ vi) $T(n) = \sin n$

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

$a = 1, b = 2, k = 2, p = 0. \geq 0$

$a < b^k$

$1 < 2^2$

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

by 3.i.

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

$a = 2^n$ a not constant

Master Theorem not applicable.

eg. $T(n) = 16\,T(n/4) + n$.

$a = 16, b = 4, k = 1, p = 0.$

$a > b^k$. $\therefore T(n) = \Theta(n^{\log_4 16})$

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

eg. $T(n) = 2T(n/2) + n\log n$.

$a = 2, b = 2, k = 1, p = 1. > -1.$

$a = b^k$

$\therefore T(n) = \Theta(n^{\log_2 2} \log^2 n)$

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

eg. $T(n) = 2T(n/2) + n/\log n$.

$a = 2,\ b = 2,\ K = 1,\ p = -1.$

$a = b^K.$

$T(n) = \theta\left(n^{\log_2 2} \log\log n\right)$

$T(n) = \underline{\theta\left(n \log^2 n\right)}.$

eg. $T(n) = 2T(n/4) + n^{0.51}.$

$a = 2,\ b = 4,\ K = 0.51,\ p = 0.$

$a < b^K.\quad p \geq 0$

$T(n) = \theta\left(n^{0.51}\right)$

eg. $T(n) = 0.5T(n/2) + 1/n$

$a = 0.5,\ b = 2$

$a \geq 1$ Not applicable.

eg. $T(n) = 6T(n/3) + n^2 \log n.$

$a = 6,\ b = 3,\ K = 2,\ p = 1. \geq 0$

$a < b^K \quad p \geq 0$

$T(n) = \theta\left(n^2 \log n\right).$

eg. $T(n) = 64T(n/8) - n^2 \log n$.

function not +ve. Not applicable.

eg. $T(n) = 7T(n/3) + n^2$

$a = 7, \; b = 3, \; k = 2, \; p = 0$

$a < b^k \therefore T(n) = \Theta(n^2)$.

---

eg. $T(n) = 4T(n/2) + \log n$.

$a = 4, \; b = 2, \; k = 0, \; p = 1$.

$a > b^k$

$T(n) = \Theta(n^{\log_2 4})$.

$T(n) = \underline{\Theta(n^2)}$.

eg. $T(n) = 16T(n/4) + n!$

$a = 16, \; b = 4, \; p = 0$.

$T(n) = \underline{\Theta(n!)}$.

eg. $T(n) = \sqrt{2}\,T(n/2) + \log n$.

$a = \sqrt{2}, \; b = 2, \; k = 0, \; p = 1$.

$a > b^k$

$T(n) = \Theta(n^{\log_2 \sqrt{2}}) = \underline{\Theta(\sqrt{n})}$.

eg. $T(n) = 3T(n/2) + n$

$a = 3, \quad b = 2, \quad k = 1, \quad p = 0$

$a > b^k$

$T(n) = \theta(n^{\log_2 3})$.

---

eg. $T(n) = 2T(n/2) + \sqrt{n}$.

$a = 2, \quad b = 2, \quad k = 1/2, \quad p = 0$

$a > b^k$

$T(n) = \underline{\theta(n)}$.

---

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

$a = 4, \quad b = 2, \quad k = 1, \quad p = 0$.

$a \geq b^k$

$T(n) = \underline{\theta(n^2)}$.

---

eg. $T(n) = 3T(n/4) + n\log n$.

$a = 3, \quad b = 4, \quad k = 1, \quad p = 1. \geq 0$

$a < b^k$

$T(n) = \underline{\theta(n \log n)}$.

---

* Master Theorem for Subtract & Conquer Recurrences. (Master Theorem)

Let $T(n)$ be a function defined on positive n & having the property

$$T(n) = \begin{cases} c & , \text{if } n \leq 1 \\ aT(n-b) + f(n) & , n > 1 \end{cases}$$

for some constants c, $a > 0$, $b > 0$, $k \geq 0$ & function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k) & , a < 1 \\ O(n^{k+1}) & , a = 1 \\ O(n^k a^{n/b}) & , a > 1 \end{cases}$$

• Variant of the theorem.

The soln to the equn

$T(n) = T(\alpha n) + T((1-\alpha)n) + \beta n$,

where $0 < \alpha < 1$ & $\beta > 0$ are constants

is $O(n \log n)$.

eg. $T(n) = 3T(n-1)$ $\quad n > 0$

$= c$ $\quad n <= 0$

$a = 3$, $b = 1$, $f(n) = 0 \Rightarrow f(n) = O(n^0)$

$k = 0$

$T(n) = O(n^0 3^{n/1}) = \underline{O(3^n)}$

Shortcut.

$$T(n) = a\,T(n-b) + O(n^k)$$

$$a > 0 \; ; \; b \geq 1 \; ; \; k \geq 0$$

case 1 : if $a > 1$ then $T(n) = O(n^k a^{n/b})$

case 2 : if $a = 1$ then $T(n) = O(n^{k+1})$

case 3 : if $a < 1$ then $T(n) = O(n^k)$

eg. $T(n) = T(n-1) + n(n-1)$, $n \geq 2$

$= 1$, $n = 1$.

$a = 1,\ b = 1\ ,\ f(n) = n(n-1)$

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

$k = 2$.

$T(n) = \underline{O(n^3)}$.

eg. $T(n) = 2T(n-1) - 1$. $n > 0$

$= 1$ $n \leq 0$.

Not applicable.

* Method of Guessing & Confirm.

for recurrences:

Considering recurrence $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$.

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

For upper bound,

$T(n) = \sqrt{n}\, T(\sqrt{n}) + n$.

$\leq \sqrt{n} \cdot c\sqrt{n} \log \sqrt{n} + n$.

$= nc \log \sqrt{n} + n$.

$= nc \frac{1}{2} \log n + n$

$\leq cn \log n$.

Assuming $1 \leq c \frac{1}{2} \log n$ that correct for large n & any constant c.

For lower bound,

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n.$$

$$\geq \sqrt{n} \cdot k\sqrt{n} \log \sqrt{n} + n$$

$$= nk \log \sqrt{n} + n$$

$$= nk \cdot \frac{1}{2} \log n + n$$

$$\geq kn \log n.$$

Assuming $1 \geq k \cdot \frac{1}{2} \log n$, which is incorrect if n is sufficiently large & for any constant k.

So, $\Theta(n \log n)$ is too big. Now, taking $\Theta(n)$.

For lower - bound,

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \quad \geq n$$

For upper bound,

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

$$\leq \sqrt{n} \cdot c\sqrt{n} + n$$

$$= nc + n$$

$$= n(c+1)$$

$$\not\leq cn$$

Now, $\Theta(n)$ is too small.

Taking $\Theta(n\sqrt{\log n})$,

for upper bound,

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + n$$
$$\leq \sqrt{n} \cdot c\sqrt{n}\sqrt{\log\sqrt{n}} + n$$
$$= nc\frac{1}{\sqrt{2}}\log\sqrt{n} + n$$
$$\leq cn\log\sqrt{n}.$$

for lower bound,

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + n$$
$$\geq \sqrt{n} \cdot k\sqrt{n}\sqrt{\log\sqrt{n}} + n$$
$$= nk\frac{1}{\sqrt{2}}\log\sqrt{n} + n$$
$$\not\geq kn\log\sqrt{n}.$$

Taking $\Theta(n\log\log n)$,

upper bound, $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$
$$\leq \sqrt{n}\,c\sqrt{n}\log\log\sqrt{n} + n$$
$$= nc\log\log n - cn + n$$
$$\leq cn\log\log n, \text{ if } c \geq 1$$

lower bound, $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$
$$\geq \sqrt{n}\,k\sqrt{n}\log\log\sqrt{n} + n$$
$$= nk\log\log n - kn + n$$
$$\geq kn\log\log n, \text{ if } k \leq 1$$

$\therefore T(n) = \Theta(n\log\log n)$.

* Analysing Space Complexity.

→ Iterative Algo.

eg. Algo (A, n) {
    int i ; ——— 1
    for (i = 1 to n)
        A[i] = 0 ;
}

$S(n) = O(1)$

eg. Algo (A, n) {
    int i ; ——— 1
    create B[n] ; ——— n
    for (i = 1 to n)
        B[i] = A[i] ;
}

$S(n) = O(n+1)$
$= O(n).$

eg. Algo (A, n) {
    create B[n, n] ; — n^2
    int i, j ; — 2
    for (i = 1 to n)
        for (j = 1 to n)
            B[i, j] = A[i] ;
}

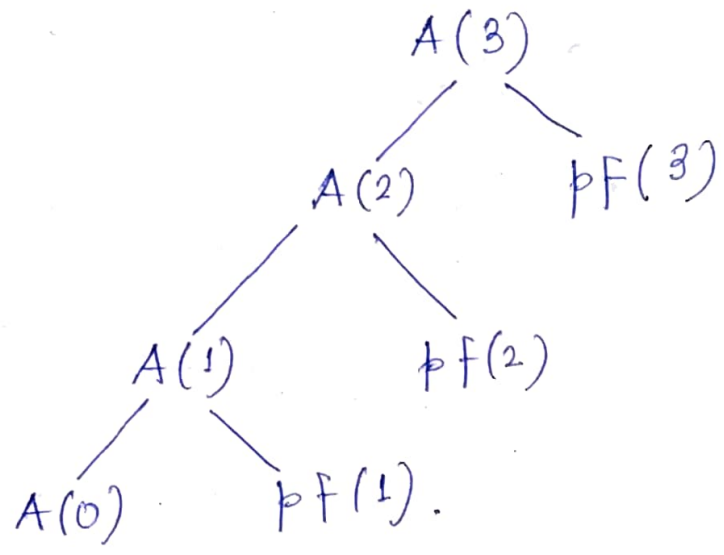
$a = 1$

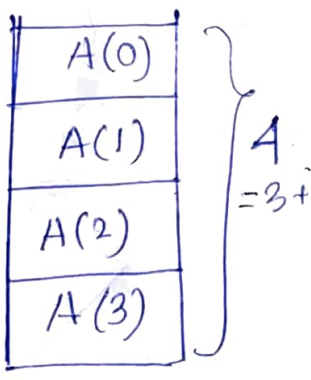
$S(n) = O(n^2+2)$
$= O(n^2).$

→ Recursive Algo. (Tree Method)

eg.
```
A(n) {
    if (n >= 1) {
        A(n-1);
        pf(n);
    }
}
```

For A(3),





Every first time we visit a recursive call, we push it to stack & the last time we visit the recursive call we pop it from stack. The statements gets executed once it is visited.

Space required for stack = $(n+1)k$

$S(n) = O(nk) = O(n)$

k → cells required for every recursive call (constant)

For time complexity,

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

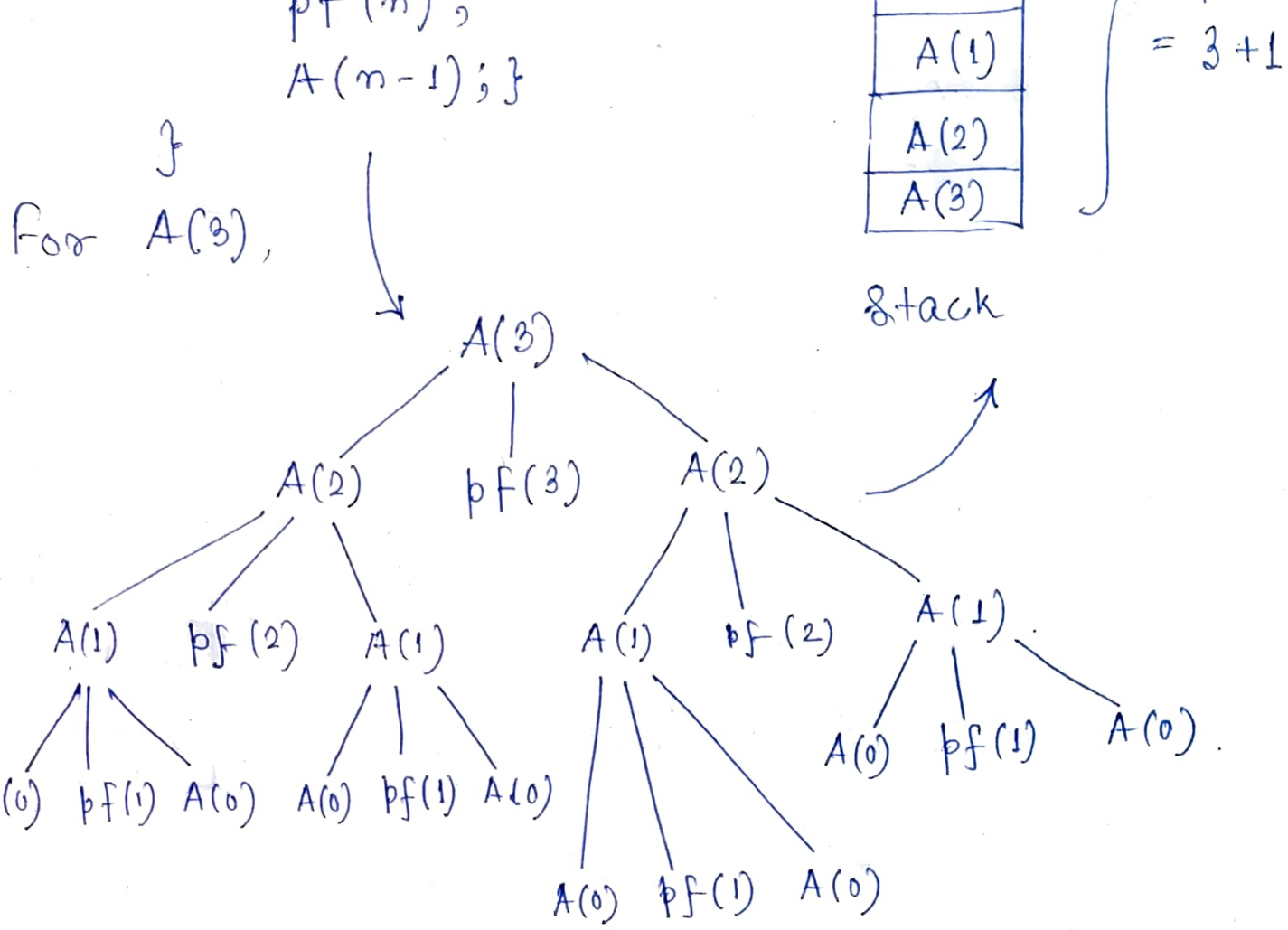
By Muster theorem,

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

eg.

```
A(n){
    if (n >= 1) {
        A(n-1);
        pf(n);
        A(n-1); }
}
```

For A(3),



No. of recursive calls for $n = 3$ is 15
$= 2^{3+1} - 1$

No. of n n n $n = 2$ is 7
$= 2^{2+1} - 1$

No. of recursive calls for n is $2^{n+1} - 1$.

$S(n) = O((n+1)K) = \underline{O(n)}$.

For TC, $T(n) = 2T(n-1) + 1$.

by Master Theorem,

$T(n) = \underline{O(2^n)}$.

- Not in-place sorting : Merge
- Stable : Insertion, Bubble., Merge
- Adaptive sorting : Bubble, Insertion, Quick
- Non-adaptive : Selection, merge, heap
- Selection sorting algo makes min. no. of memory writes.

Quick sort preferred for arrays.

Merge sort for Linked lists.

Quick sort is cache friendly.

Quick sort has good locality of reference.

↳ best sorting algo

Commonly used complexities.

for (i=0; i<n; i++)	O(n)
for (i=0; i<n; i=i+2)	O(n)
for (i=n; i>1; i--)	O(n)
for (i=1; i<n; i*2)	$O(\log_2 n)$
for (i=1; i<n; i*3)	$O(\log_3 n)$
for (i=n; i>1; i=i/2)	$O(\log_2 n)$
for (i=n; i>1; i=i/3)	$O(\log_3 n)$
for (i=2; i<n; i=i^c)	$O(\log \log n)$

↳ $2, 2^c, 2^{c^2}, \ldots 2^{c^K}$

$2^{c^K} < n$

$\Rightarrow K < \log_c \log_2 n$

prev. pg

- for (i=0; i<n; i++)
  for (j=i; j<n; j++)
    count (n-n)

↓

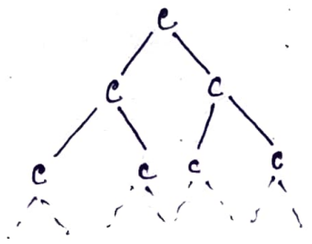
inner loop stmt. $n, n-1, \ldots, 1$

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

$\theta(n^2)$.
as no best case.

- eg. recursion tree.

→ $T(n) = 2T(n-1) + c$.



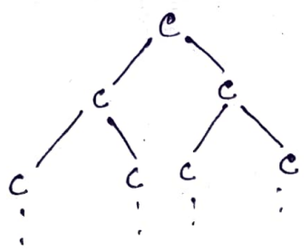
c
2c
4c
⋮
n times.

$c + 2c + 4c + 8c + \ldots$

$= c \cdot (2^n - 1)$.

$= \theta(2^n)$.

→ $T(n) = 2T(n/2) + c$.



c
2c
4c
⋮
$\log_2 n$ times.

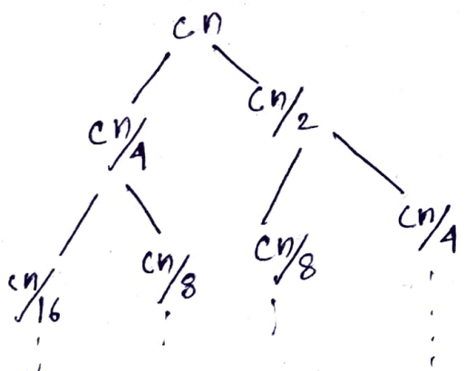
$c + 2c + 4c + \ldots$ $\log_2 n$ times

$= c(2^{\log_2 n} - 1)$

$= \theta(n)$

→ Reduction of trees at different speed - examples

→ $T(n) = T(n/4) + T(n/2) + cn$.



cn
$\frac{3cn}{4}$
$\frac{9cn}{16}$

finishes fast but assume full tree

$\log_2 n$ times as max height $\log_2 n$.

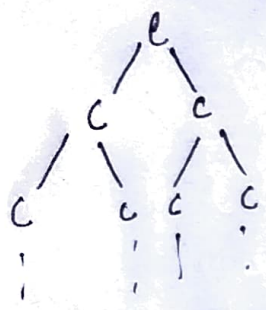
$cn + \frac{3cn}{4} + \frac{9cn}{16} + \ldots$ $\log_2 n$ times

$= cn\left(\frac{(3/4)^{\log_2 n} - 1}{\frac{3}{4} - 1}\right)$ $r < 1$.

$= cn \cdot \frac{1}{1 - 3/4} = O(n)$

not θ for assumption of full tree.

→ $T(n) = T(n-1) + T(n-2) + c$ fibonacci no.



c
2c
4c

$c + 2c + 4c + \cdots$ n times

$= c \cdot (2^n - 1)$

$= O(2^n)$

assuming full tree

=> max height ~~log~~ n

* Amortized Analysis.

→ In an amortized analysis, we average the time required to perform a sequence of data structuree operations over all the operations performed.

→ Using an amortized analysis, we can show that the average cost of an operation is small, even though a single operation within the sequence might be expensive.

→ Used to analyse time complexities of hash tables, Splay trees & disjoint sets.

→ Techniquees used in amortized analysis —

1. aggregate method
2. accounting method
3. potential method.

- ### Aggregate Analysis.

eg. Hash Table Insertions.

Increase the size of table whenever it becomes full.:

— Allocate memory for a table of size double the old table.

— Copy contents of old table.

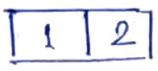
— free the old table.

Initially empty table & size is 0.

Insert 1.

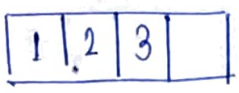
(overflow)

Insert 2



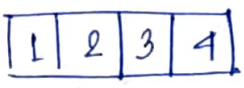
(overflow)

Insert 3



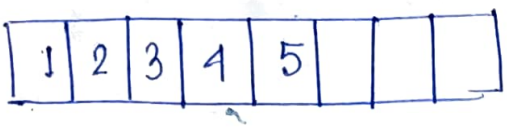
(overflow)

Insert 4



Insert 5

(overflow)



Item No.	1	2	3	4	5	6	7	8	9	10
Table Size	1	2	4	4	8	8	8	8	16	16
Cost	1	2 1+1	3 1+2	1	5 1+4	1	1	1	9 1+8	1

2 copies
1 insertion

8 copies
1 insertion.

Total amortised cost =

$$\left(\frac{1+2+3+1+5+1+1+1+9+\cdots}{n} \right)$$

$$\begin{cases} 1+2^K < n \\ K < \lg_2 (n-1) \end{cases}$$

$$= \frac{(1+1+1+\ldots \text{ } n \text{ times}) + (\overbrace{1+2+4+8+\ldots}^{\lfloor \log_2 n - 1 \rfloor + 1 \text{ times}})}{n}$$

$$= \frac{n + 2n}{n} = 3. \qquad O(n)$$

$O(1)$ (Ans.)

$1 + 2^K = n$

$K = \lg_2 (n-1).$

$1 + 2 + 2^2 + 2^3 + \ldots \lg_2 (n-1)$ times.

$$\frac{2^{\lg_2 (n-1) - 1} - 1}{2-1} = \frac{2^{\lg_2 (n-1)}}{2} - 1.$$

$$= \frac{n-1}{2} - 1$$

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

---

* NK Examples : Great!

Q Time complexity :

```
f(int n) {
1.  for(i=0; i<n; i++) {
2.      for(j=i; j<i*i; j++) {
            if (j%i == 0) {
3.              for (k=0; k<j; k++)
                    printf("*");
            }
        }
}
```

→ executes j times when $j \% i == 0$.

Total # iterations for each i $O(n^2)$.

Executes j times $= n^2$ times.

(when $i = n \Rightarrow j = n^2$

$n^2 \% n == 0$

$k = 0$ to n^2)

1. n times
2. $j = n$ to n^2

   $n^2 - n = n(n-1)$ times $\approx O(n^2)$
3. $O(n^2)$

$\Rightarrow O(n^5)$. Loose bound

- For tighter bound,

```
// for(k=0 ; k < j ; k++)
       printf ("*");
```

] This has TC $\Theta(j)$.

```
// for (j=i ; j < i*i ; j++)
     if (j%i==0) {
         do Θ(j) work
     }
```

] This loop runs $i^2 - i = \Theta(i^2)$ times. Most iterations will do $O(1)$ work, as $j \% i \neq 0$ for most of the cases. But, every ith iteration will do $\Theta(j)$ work (as $(i + ki) \% i == 0$).

We divide the work for this loop as 2 parts :

i) Baseline $\underline{\Theta(i^2)}$ work (for looping from i to i^2 : $i^2 - i = i(i-1)$ times)

ii) Added with (i) the extra work $^{\Theta(j)}$ done when $j \% i == 0$.

(ii) happens every ith iteration.

⇒ how many times $j \% i == 0$ becomes true?

$i, i+1, i+2i, i+3i, \ldots, i+pi$

$i + pi \leq i^2 \Rightarrow p \leq i-1$ times

Total work =

$i + 2i + 3i + 4i + \ldots + i^2$

$= i(1 + 2 + 3 + \ldots + i) = \underline{\Theta(i^3)}$

As i^3 dominates i^2,
So work done for this loop is $\Theta(i^3)$

For the outermost loop,

```
for (i=0; i<n; i++)
    do Θ(i^3) work
```

$\Theta(n^4)$

Ans

(Stackoverflow)

Q $T(n) = 2T(\sqrt{n}) + \lg n$

→ Say $n = 2^m$

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

Say, $S(m) = T(2^m)$

$S(m/2) = T(2^{m/2})$

$\Rightarrow S(m) = 2S(m/2) + m$

$= O(m \lg m)$

$m = \lg n$

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

Q. $T(n) = T(\sqrt{n}) + 1$

→ Same logic

$S(m) = S(m/2) + 1$

$= O(\lg m)$

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

Q $T(n) = 2T(\sqrt{n}) + 1$

$S(m) = 2S(m/2) + 1$

$= O(m)$

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

Q.
```
fun (int n){
    if (n < 2) return;
    else counter = 0;
    for i = 1 to 8 do
        fun (n/2);
    for i = 1 to n^3 do
        counter++;
}
```

$\Rightarrow T(n) = 8T(n/2) + n^3 + 1.$

$a = 8 \quad b = 2 \quad k = 3 \quad p = 0$

$a = b^k \quad p = 0 > -1.$

$\Theta(n^3 \lg n).$

Q. temp = 1
repeat
  for i = 1 to n
    temp++
  n /= 2
until n ≤ 1

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

$$n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + \cdots + \frac{n}{2^k} = 1$$
$$= O(n).$$

Q.
```
fun (int n) {
  int i = 1;
  while (i < n) {
    int j = n;
    while (j > 0)
      j = j/2;
    i = 2 * i;
  }
}
```

$n, n/2, n/2^2, n/2^3, \ldots n/2^k = 1$
$\lg_2 n$ times

$1, 2, 2^2, 2^3, \ldots, 2^k = n$
$\lg_2 n$ times

$\Rightarrow O(\lg^2 n)$.

Note. no work is done inside any loop, only the loop runs & comparision is done.

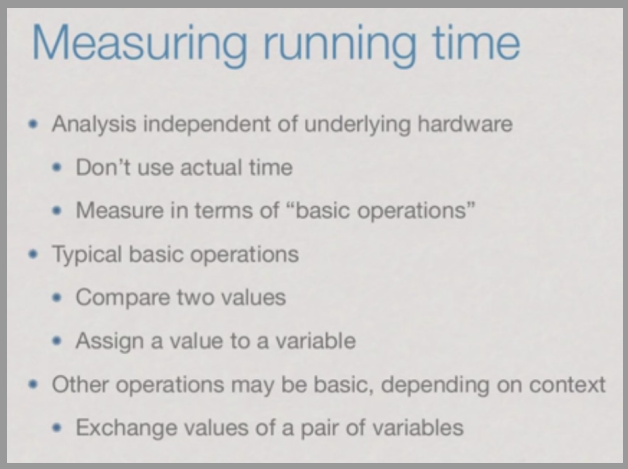
Asymptotic analysis

Not only the correctness of the code but also the running time for that code is important.

Running time of a program depends upon:

1. Single or multiprocessor
2. 32 or 64 bit
3. Read/Write speed to memory
4. Input (We consider only this while calculating time complexity of a program)

We are interested in the rate of growth of time taken with respect to the input.

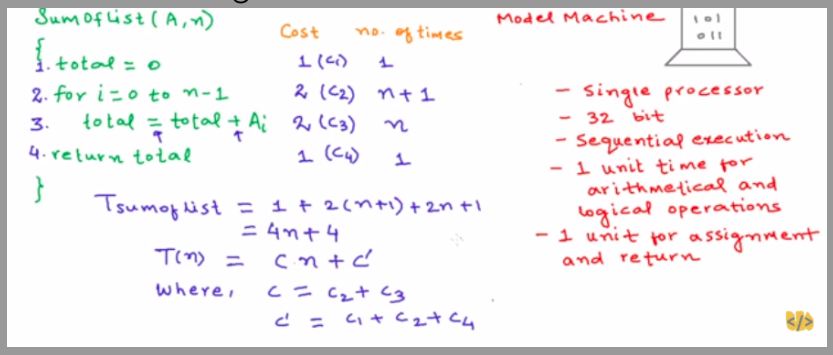


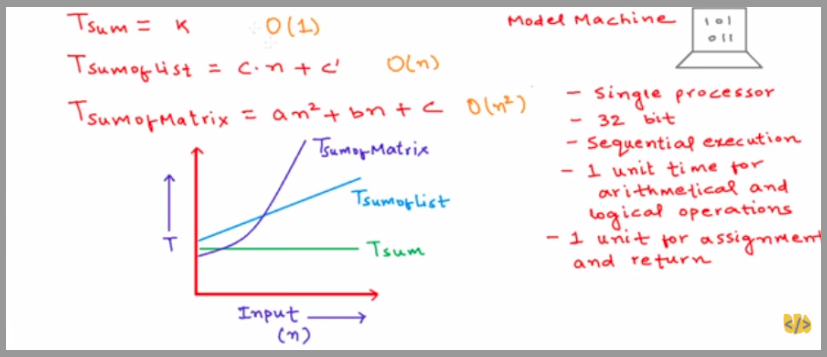
2 kinds of algorithm:

1. Constant time algorithm:

```
Sum(int a, int b) {
        return (a+b);
}
```

2. Variable time algorithm:





Asymptotic notations:

Formally:

"$T(n)$ is $O(f(n))$" iff for some constants c and n_0, $T(n) <= cf(n)$ for all $n >= n_0$

"$T(n)$ is $\Omega(f(n))$" iff for some constants c and n_0, $T(n) >= cf(n)$ for all $n >= n_0$

"$T(n)$ is $\Theta(f(n))$" iff $T(n)$ is $O(f(n))$ AND $T(n)$ is $\Omega(f(n))$

"$T(n)$ is $o(f(n))$" iff $T(n)$ is $O(f(n))$ AND $T(n)$ is NOT $\Theta(f(n))$

Informally:

"$T(n)$ is $O(f(n))$" basically means that $f(n)$ describes the upper bound for $T(n)$

"$T(n)$ is $\Omega(f(n))$" basically means that $f(n)$ describes the lower bound for $T(n)$

"$T(n)$ is $\Theta(f(n))$" basically means that $f(n)$ describes the exact bound for $T(n)$

"$T(n)$ is $o(f(n))$" basically means that $f(n)$ is the upper bound for $T(n)$ but that $T(n)$ can never be equal to $f(n)$

Another way of saying this:

"$T(n)$ is $O(f(n))$" growth rate of $T(n)$ <= growth rate of $f(n)$

"$T(n)$ is $\Omega(f(n))$" growth rate of $T(n)$ >= growth rate of $f(n)$

"$T(n)$ is $\Theta(f(n))$" growth rate of $T(n)$ == growth rate of $f(n)$

"$T(n)$ is $o(f(n))$" growth rate of $T(n)$ < growth rate of $f(n)$

Definition

$f(n) \in O(g(n))$ if there exists constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$

- "Big-O" notation
- $O(g(n))$ is an upper-bound on the growth of a function, $f(n)$.

For the formal definition, suppose $f(x)$ and $g(x)$ are two functions defined on some subset of the real numbers. We write

$$f(x) = O(g(x))$$

(or $f(x) = O(g(x))$ for $x \to \infty$ to be more precise) if and only if there exist constants N and C such that

$$|f(x)| \leq C\,|g(x)| \quad \text{for all } x>N.$$

Intuitively, this means that f does not grow faster than g.

If a is some real number, we write

$$f(x) = O(g(x)) \quad \text{for } x \to a$$

if and only if there exist constants $d > 0$ and C such that

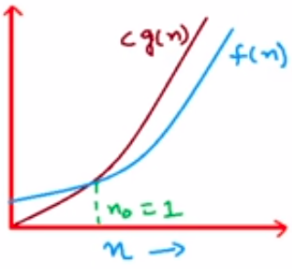
$$|f(x)| \leq C\,|g(x)| \quad \text{for all } x \text{ with } |x-a| < d.$$

The first definition is the only one used in computer science (where typically only positive functions with a natural number n as argument are considered; the absolute values can then be ignored), while both usages appear in mathematics.

Time Complexity analysis – asymptotic notations

O – "big-oh" notation → upper bound

$O(g(n)) = \{ f(n) :$ there exist constants C and n_0, $f(n) \leq Cg(n)$, for $n \geq n_0 \}$

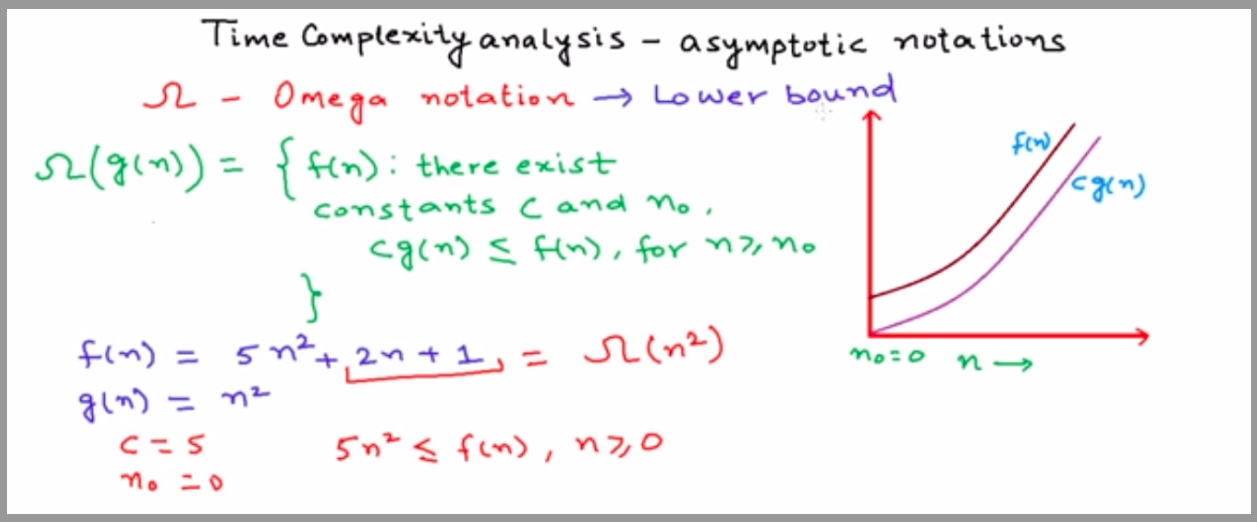


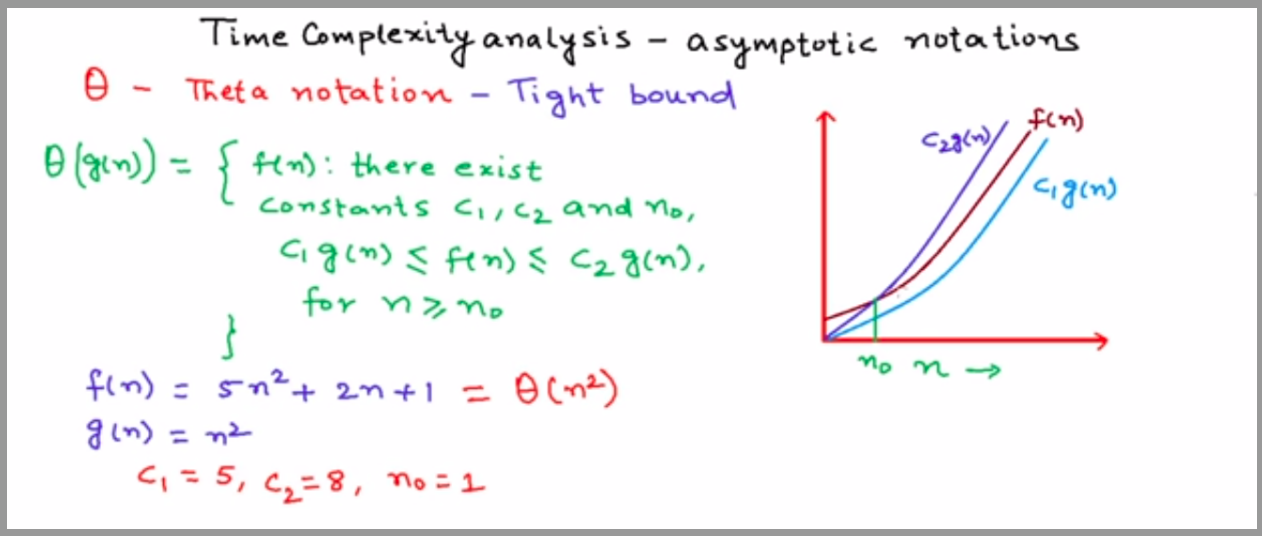
$f(n) = 5n^2 + 2n + 1 = O(n^2)$

$g(n) = n^2$

$C = 8, \quad f(n) \leq 8n^2, \quad n \geq 1$

$n_0 = 1$





Asymptotic Analysis of Algorithms

GFG

- Behavior of a function as $n \to \infty$.

- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then $f(n) = \Omega(g(n))$
  - "f is bigger than g", or "f dominates g"
- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) = O(g(n))$
  - "f is smaller than g", or "g dominates f"
- if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ then $f(n) = \Theta(g(n))$
  - "f is the same as g"

n^2 = Omega(n)

- Show that $n^2 = \Omega(n)$.
- Evaluate $\lim_{n\to\infty} \frac{f(n)}{g(n)}$
- $\lim_{n\to\infty} \frac{n^2}{n}$
- $\lim_{n\to\infty} n = \infty$ so $n^2 = \Omega(n)$

n = Omega(logn)

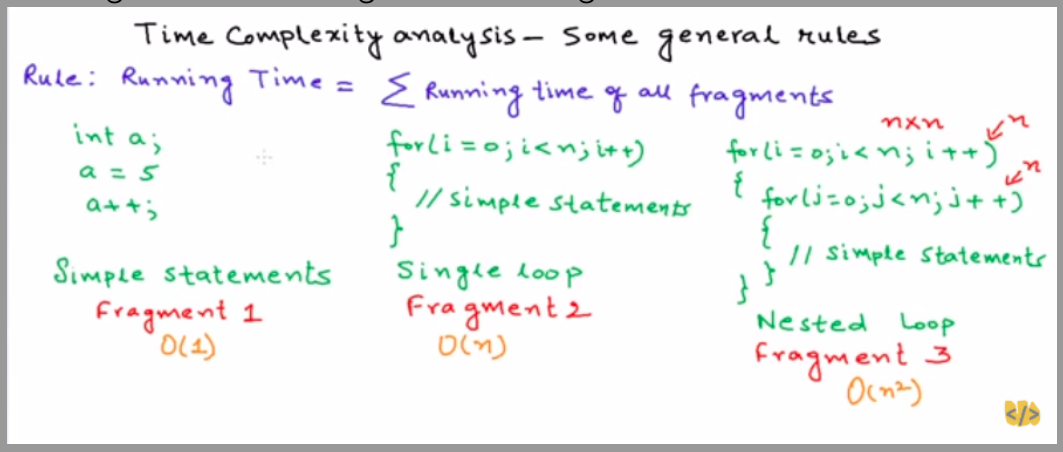
- Show that $n^2 = \Omega(n)$.
- Evaluate $\lim_{n\to\infty} \frac{f(n)}{g(n)}$
- $\lim_{n\to\infty} \frac{n}{\log n}$
- l'Hopital's rule. (from Calculus)
- if $lim_{n\to c} f(n) = \infty$ and $lim_{n\to c} g(n) = \infty$, then $\lim_{n\to c} \frac{f(n)}{g(n)} = \lim_{n\to c} \frac{f'(n)}{g'(n)}$
- $\lim_{n\to\infty} \frac{n}{\log n} = \lim_{n\to\infty} \frac{1}{1/n}$
- $\lim_{n\to\infty} \frac{n}{\log n} = \lim_{n\to\infty} n = \infty$
- So, $n = \Omega(\log n)$

Some general rules:

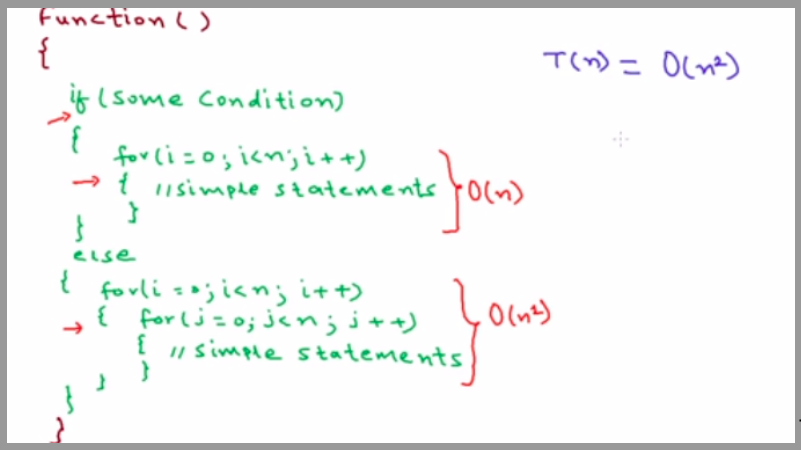
We analyse time complexity for very large input size and as a worst case scenario(? Murphy's Law).

1. Drop lower order terms. Take the highest growing term.

2. Drop constant multiplier
3. Running time = Σ Running time of all fragments



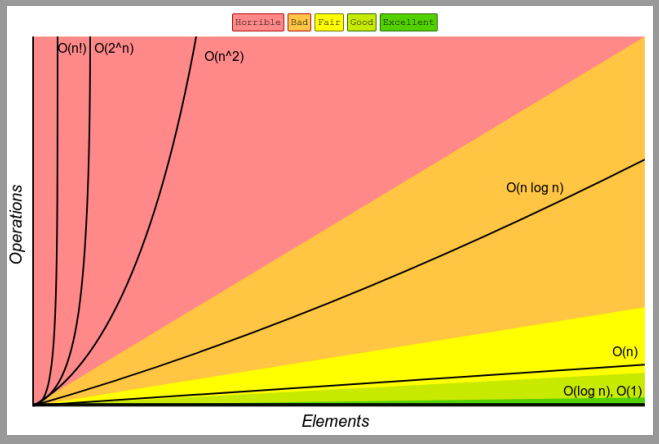
4. Maximum running time of the program is the overall running time of the program.



Taking the worst case.

O(1) + O(1) = c1 + c2 = c3 = c3 * 1 = O(1)

O(2n^2) = (2n^2)*c1 + … = (c1*2)n^2 + … = O(n^2)



Notation	Name
$O(1)$	constant
$O(\log \log n)$	double logarithmic
$O(\log n)$	logarithmic
$O(n^c),\ 0 < c < 1$	fractional power
$O(n)$	linear
$O(n \log^* n)$	n log-star n
$O(n \log n) = O(\log n!)$	linearithmic, loglinear, or quasilinear
$O(n^2)$	quadratic
$O(n^c),\ c > 1$	polynomial or algebraic
$L_n[\alpha, c],\ 0 < \alpha < 1 = e^{(c+o(1))(\ln n)^\alpha (\ln \ln n)^{1-\alpha}}$	L-notation or sub-exponential
$O(c^n),\ c > 1$	exponential
$O(n!)$	factorial

Arithmetic series

$$\sum_{K=1}^{n} K = (1+2+\dots+n) = \frac{n(n+1)}{2}$$

Geometric Series

$$\sum_{K=0}^{n} x^K = 1 + x + x^2 + \dots x^n = \frac{x^{n+1}-1}{x-1} \quad (x \neq 1)$$

Harmonic Series

$$\sum_{K=1}^{n} \frac{1}{K} = 1 + \frac{1}{2} + \frac{1}{3} + \dots \frac{1}{n} \approx \log n$$

other important formulae

$$\sum_{K=1}^{n} \log K \approx n \log n$$

$$\sum_{K=1}^{n} K^p = 1^p + 2^p + \dots n^p = \frac{1}{p+1} n^{p+1}$$

Proof

- $f_1(n) \leq c_1 g_1(n)$ for all $n > n_1$
- $f_2(n) \leq c_2 g_2(n)$ for all $n > n_2$

$n_3 = \max(n_1, n_2)$ $\quad c_3 = \max(c_1, c_2)$

$n \geq n_3$ (n_0)

$$\begin{aligned} f_1 + f_2 &\leq c_1 g_1 + c_2 g_2 \\ &\leq c_3 g_1 + c_3 g_2 \\ &\leq c_3 (g_1 + g_2) \\ &\leq c_3 (2 \cdot \max(g_1, g_2)) \\ &\leq (2c_3)(\max(g_1, g_2)) \end{aligned}$$

(c)

- If
  - $f_1(n)$ is $O(g_1(n))$
  - $f_2(n)$ is $O(g_2(n))$
- then $f_1(n) + f_2(n)$ is $O(\max(g_1(n), g_2(n)))$

- f(n) = O(g(n)) means g(n) is an upper bound for f(n)
  - Useful to describe limit of worst case running time for an algorithm
- f(n) = Ω(g(n)) means g(n) is a lower bound for f(n)
  - Typically used for classes of problems, not individual algorithms
- f(n) = Θ(g(n)): matching upper and lower bounds
  - Best possible algorithm has been found

Let's take a closer look a the formal definition for big-O analysis

"$T(n)$ is $O(f(n))$" if for some constants c and n_0, $T(n) <= cf(n)$ for all $n >= n_0$

The way to read the above statement is as follows.

- n is the size of the data set.
- $f(n)$ is a function that is calculated using n as the parameter.
- $O(f(n))$ means that the curve described by $f(n)$ is an upper bound for the resource needs of a function.

This means that if we were to draw a graph of the resource needs of a particular algorithm, it would fall under the curve described by $f(n)$. What's more, it doesn't need to be under the exact curve described by $f(n)$. It could be under a constant *scaled* curve for $f(n)$... so instead of having to be under the n^2 curve, it can be under the $10n^2$ curve or the $200n^2$ curve. In fact it can be any constant, as long as it is a constant. A constant is simply a number that does not change with n. So as n gets bigger, you cannot change what the constant is. The actual value of the constant does not matter though.

The other portion of the statement $n >= n_0$ means that $T(n) <= cf(n)$ does not need to be true for all values of n. It means that as long as you can find a value n_0 for which $T(n) <= cf(n)$ is true, and it never becomes untrue for all n larger than n_0, then you have met the criteria for the statement $T(n)$ is $O(f(n))$

In summary, when we are looking at big-O, we are in essence looking for a description of the growth rate of the resource increase. The exact numbers do not actually matter in the end.

$O(2^N)$

$O(2^N)$ denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an $O(2^N)$ function is exponential - starting off very shallow, then rising meteorically. An example of an $O(2^N)$ function is the recursive calculation of Fibonacci numbers:

```
int Fibonacci(int number)
{
    if (number <= 1) return number;

    return Fibonacci(number - 2) + Fibonacci(number - 1);
}
```

Logarithms

Logarithms are slightly trickier to explain so I'll use a common example:

Binary search is a technique used to search sorted data sets. It works by selecting the middle element of the data set, essentially the median, and compares it against a target value. If the values match it will return success. If the target value is higher than the value of the probe element it will take the upper half of the data set and perform the same operation against it. Likewise, if the target value is lower than the value of the probe element it will perform the operation against the lower half. It will continue to halve the data set with each iteration until the value has been found or until it can no longer split the data set.

This type of algorithm is described as **O(log N)**. The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase e.g. an input data set containing 10 items takes one second to complete, a data set containing 100 items takes two seconds, and a data set containing 1000 items will take three seconds. Doubling the size of the input data set has little effect on its growth as after a single iteration of the algorithm the data set will be halved and therefore on a par with an input data set half the size. This makes algorithms like binary search extremely efficient when dealing with large data sets.

Big-O notation just describes asymptotic bounds, so it is correct to say something like, for example, "Quicksort is in O(n!)," even though Quicksort's actual worst-case running time will never exceed O(n^2). All Big-O is saying is "for an input of size n, there is a value of n after which quicksort will always take less than n! steps to complete." It does not say "Quicksort will take n! steps in the worst case."

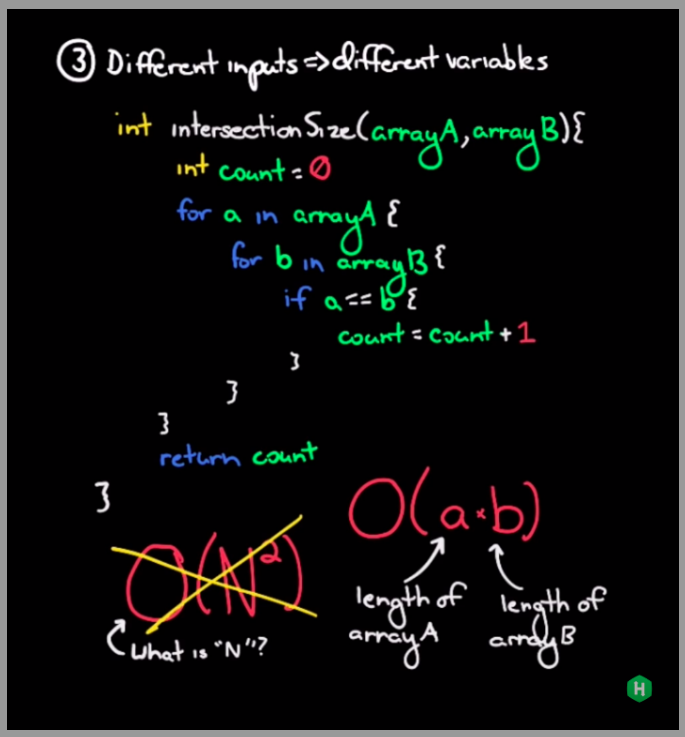
***http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/index.html* (Different families of graph)**

https://web.archive.org/web/20171215122943/http://eniac.cs.qc.cuny.edu/andrew/csci700/lecture2.pdf

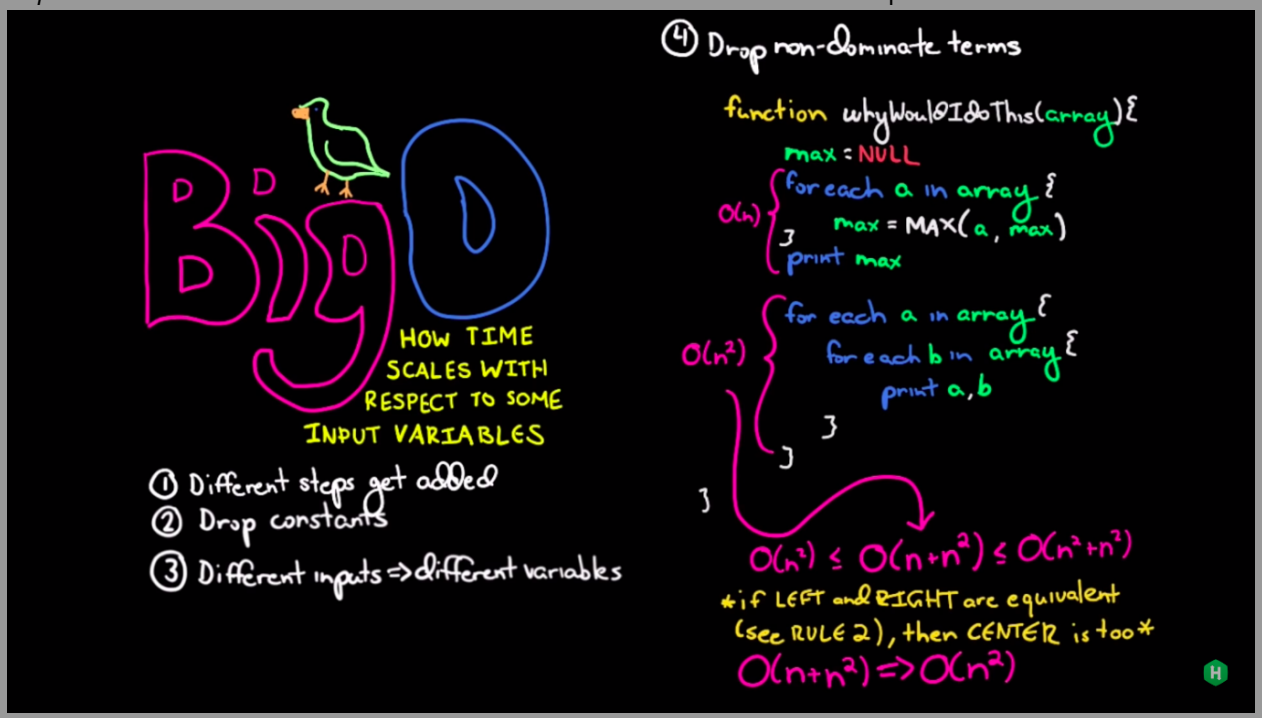
Practise problems on Time complexity of an algorithm

http://www.iitk.ac.in/esc101/08Jul/lecnotes/practise_sol.pdf

Rules:



Important! We need to take different variables for different inputs.



Here is a list of classes of functions that are commonly encountered when analyzing algorithms. The slower growing functions are listed first. c is some arbitrary constant.

notation	name
$O(1)$	constant
$O(\log(n))$	logarithmic
$O((\log(n))^c)$	polylogarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial
$O(c^n)$	exponential

Note that $O(n^c)$ and $O(c^n)$ are very different. The latter grows much, much faster, no matter how big the constant c is. A function that grows faster than any power of n is called *superpolynomial*. One that grows slower than an exponential function of the form c^n is called *subexponential*. An algorithm can require time that is both superpolynomial and subexponential; examples of this include the fastest algorithms known for integer factorization.

Note, too, that $O(\log n)$ is exactly the same as $O(\log(n^c))$. The logarithms differ only by a constant factor, and the big O notation ignores that. Similarly, logs with different constant bases are equivalent.

The above list is useful because of the following fact: if a function $f(n)$ is a sum of functions, one of which grows faster than the others, then the faster growing one determines the order of $f(n)$.

Example: If $f(n) = 10 \log(n) + 5 (\log(n))^3 + 7 n + 3 n^2 + 6 n^3$, then $f(n) = O(n^3)$.

One caveat here: the number of summands has to be constant and may not depend on n. This notation can also be used with multiple variables and with other expressions on the right side of the equal sign. The notation:

$$f(n,m) = n^2 + m^3 + O(n+m)$$

represents the statement:

$$\exists C \exists N \forall n,m>N : f(n,m) \leq n^2+m^3+C(n+m)$$

Related notations

In addition to the big O notations, another Landau symbol is used in mathematics: the little o. Informally, $f(x) = o(g(x))$ means that f grows much slower than g and is insignificant in comparison.

Formally, we write $f(x) = o(g(x))$ (for $x \to \infty$) if and only if for every $C>0$ there exists a real number N such that for all $x > N$ we have $|f(x)| < C\ |g(x)|$; if $g(x) \neq 0$, this is equivalent to $\lim_{x\to\infty} f(x)/g(x) = 0$.

Also, if a is some real number, we write $f(x) = o(g(x))$ for $x \to a$ if and only if for every $C>0$ there exists a positive real number d such that for all x with $|x - a| < d$

we have $|f(x)| < C\ |g(x)|$; if $g(x) \neq 0$, this is equivalent to $\lim_{x \to a} f(x)/g(x) = 0$.

Big O is the most commonly-used of five notations for comparing functions:

Notation	Definition	Analogy
$f(n) = O(g(n))$	see above	$\leq$
$f(n) = o(g(n))$	see above	$<$
$f(n) = \Omega(g(n))$	$g(n)=O(f(n))$	$\geq$
$f(n) = \omega(g(n))$	$g(n)=o(f(n))$	$>$
$f(n) = \theta(g(n))$	$f(n)=O(g(n))$ and $g(n)=O(f(n))$	$=$

Be careful to differentiate between:

1. Performance: how much time/memory/disk/... is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code.
2. Complexity: how do the resource requirements of a program or algorithm scale, i.e., what happens as the size of the problem being solved gets larger?

Complexity affects performance but not the other way around.

The time required by a function/procedure is proportional to the number of "basic operations" that it performs. Here are some examples of basic operations:

- one arithmetic operation (e.g., +, *).
- one assignment (e.g. x := 0)
- one test (e.g., x = 0)
- one read (of a primitive type: integer, float, character, boolean)
- one write (of a primitive type: integer, float, character, boolean)

Some functions/procedures perform the same number of operations every time they are called. For example, StackSize in the Stack implementation always returns the number of elements currently in the stack or states that the stack is empty, then we say that StackSize takes ***constant time***.

Other functions/ procedures may perform different numbers of operations, depending on the value of a parameter. For example, in the BubbleSort algorithm, the number of elements in the array, determines the number of operations performed by the algorithm. This parameter (number of elements) is called the ***problem size***/ ***input size***.

When we are trying to find the complexity of the function/ procedure/ algorithm/ program, we are ***not*** interested in the ***exact*** number of operations that are being performed. Instead, we are interested in the relation of the ***number of operations*** to the ***problem size***.

Typically, we are usually interested in the ***worst case***: what is the ***maximum*** number of operations that might be performed for a given problem size. For example, inserting an element into an array, we have to move the current element and all of the elements that come after it one place to the right in the array. In the worst case, inserting at the beginning of the array, ***all*** of the elements in the array must be moved. Therefore, in the worst case, the time for insertion is proportional to the number of elements in the array, and we say that the worst-case time for the insertion operation is ***linear*** in the number of elements in the array. For a linear-time algorithm, if the problem size doubles, the number of operations also doubles.

How to Determine Complexities

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

Sequence of statements

```
statement 1;
statement 2;
  ...
statement k;
```

The total time is found by adding the times for all statements:

total time = time(statement 1) + time(statement 2) + ... + time(statement k)

If each statement is "simple" (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).

If-Then-Else

```
if (cond) then
   block 1 (sequence of statements)
else
   block 2 (sequence of statements)
end if;
```

Here, either block 1 will execute, or block 2 will execute. Therefore, the worst-case time is the slower of the two possibilities:

max(time(block 1), time(block 2))

If block 1 takes O(1) and block 2 takes O(N), the if-then-else statement would be O(N).

Loops

```
for I in 1 .. N loop
   sequence of statements
end loop;
```

The loop executes N times, so the sequence of statements also executes N times. If we assume the statements are O(1), the total time for the for loop is N * O(1), which is O(N) overall.

Nested loops

```
for I in 1 .. N loop
   for J in 1 .. M loop
      sequence of statements
   end loop;
end loop;
```

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the complexity is O(N * M).

In a common special case where the stopping condition of the inner loop is `J < N` instead of `J < M` (i.e., the inner loop also executes N times), the total complexity for the two loops is $O(N^2)$.

Statements with function/ procedure calls

When a statement involves a function/ procedure call, the complexity of the statement includes the complexity of the function/ procedure. Assume that you know that function/ procedure *f* takes constant time, and that function/procedure *g* takes time proportional to (linear in) the value of its parameter *k*. Then the statements below have the time complexities indicated.

f(k) has O(1)
g(k) has O(k)

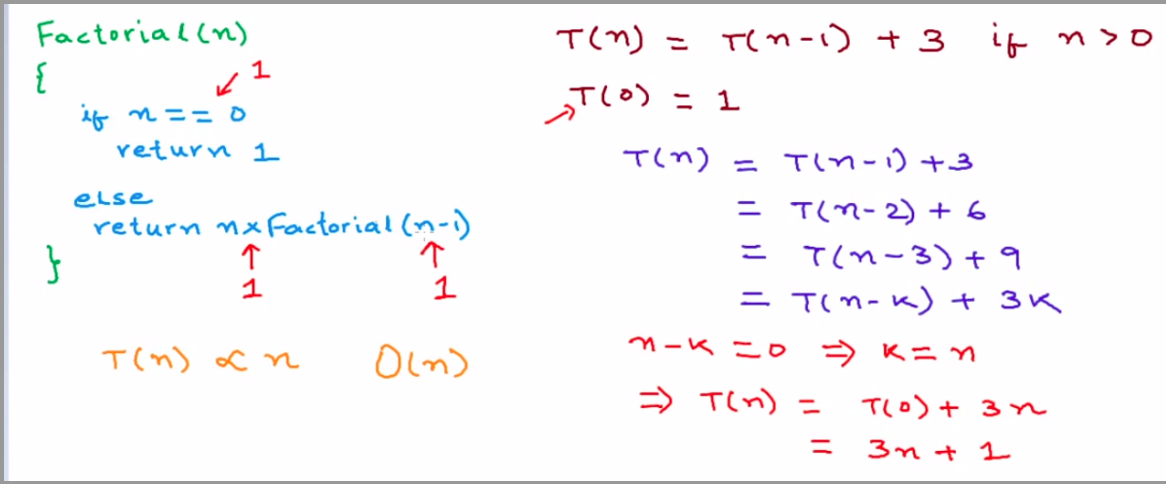
When a loop is involved, the same rule applies. For example:

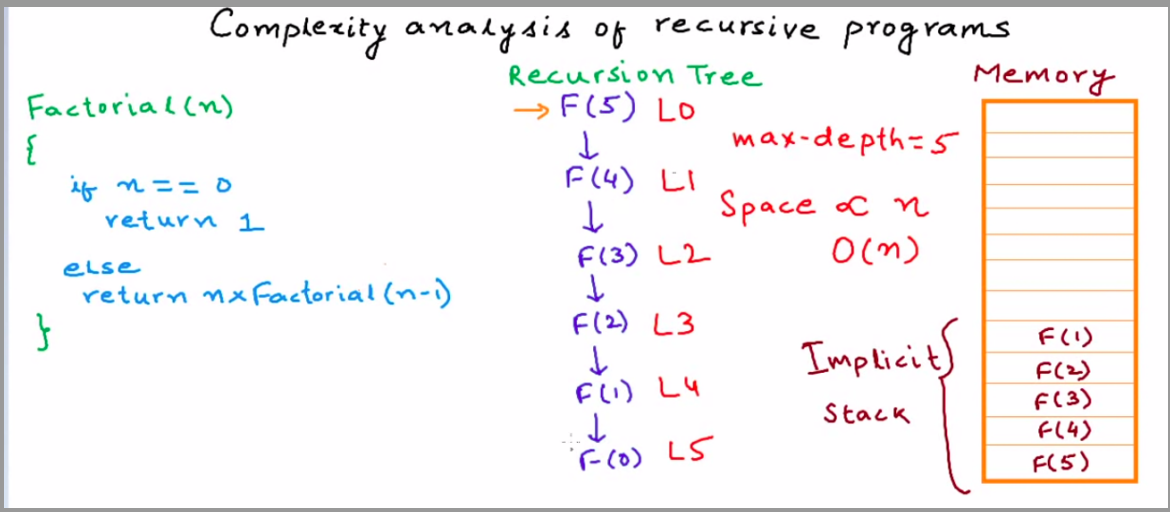
```
for J in 1 .. N loop
    g(J);
end loop;
```

has complexity (N^2). The loop executes N times and each function/procedure call `g(N)` is complexity O(N).

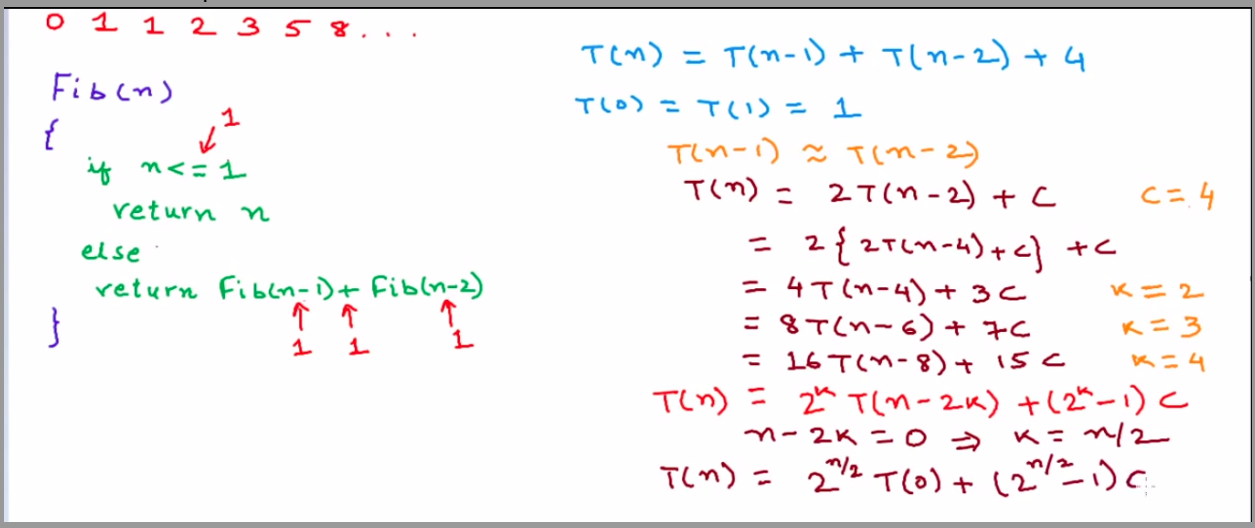
Complexity analysis of recursive algorithms:

Factorial





Fibonacci Sequence



```
Fib(n)
{
   if n<=1          (1)
     return n
   else
    return Fib(n-1)+ Fib(n-2)
}                    (1)  (1)   (1)
```

$T(n) \propto 2^{n/2}$ (Lower bound)

$T(0) = T(1) = 1$

$T(n-1) \approx T(n-2)$

$T(n) = 2T(n-2) + C \quad C = 4$

$= 2\{2T(n-4)+C\} + C$

$= 4T(n-4) + 3C \quad k = 2$

$= 8T(n-6) + 7C \quad k = 3$

$= 16T(n-8) + 15C \quad k = 4$

$T(n) = 2^k T(n-2k) + (2^k - 1)C$

$n - 2k = 0 \Rightarrow k = n/2$

$T(n) = 2^{n/2} T(0) + (2^{n/2} - 1)C$

$= (1+C) \times 2^{n/2} - C$


```
Fib(n)
{
   if n<=1          (1)
     return n
   else
    return Fib(n-1)+ Fib(n-2)
}                    (1)  (1)   (1)
```

$T(n) \propto 2^{n/2}$ (Lower bound)

$T(n) \propto 2^n$ (Upper bound)

$T(0) = T(1) = 1$

$T(n-2) \approx T(n-1)$

$T(n) = 2T(n-1) + C \quad C = 4$

$= 4T(n-2) + 3C$

$= 8T(n-3) + 7C$

$= 2^k T(n-k) + (2^k - 1)C$

$n - k = 0 \Rightarrow k = n$

$T(n) = 2^n T(0) + (2^n - 1)C$

$\Rightarrow T(n) = (1+C)2^n - C$


```
Fib(n)
{
   if n<=1          (1)
     return n
   else
    return Fib(n-1)+ Fib(n-2)
}                    (1)  (1)   (1)
```

$T(n) \propto 2^{n/2}$ (Lower bound)

$T(n) \propto 2^n$ (Upper bound)

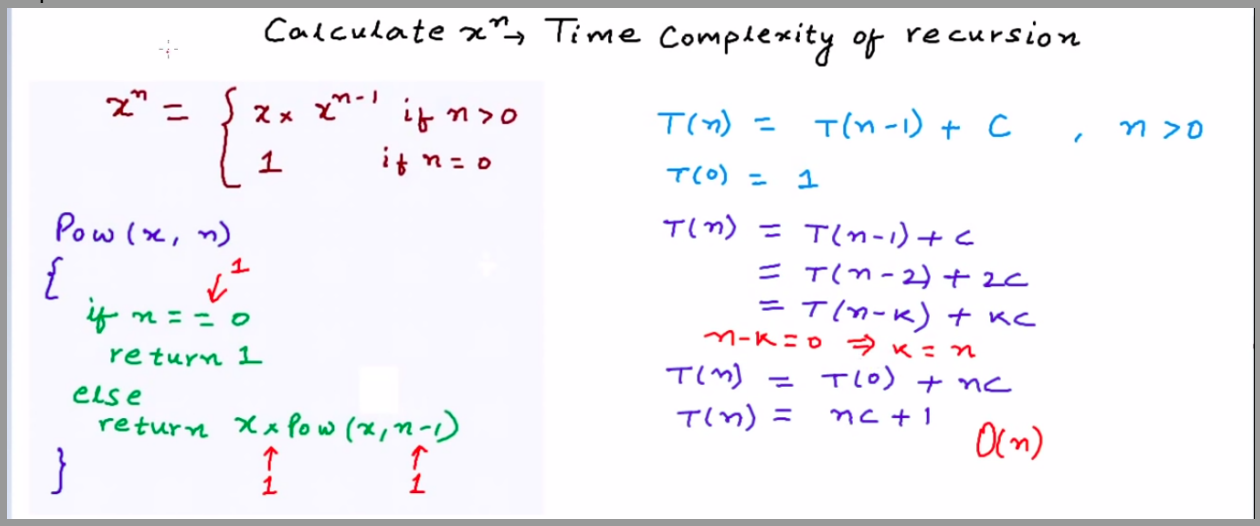
→ exponential time algorithm

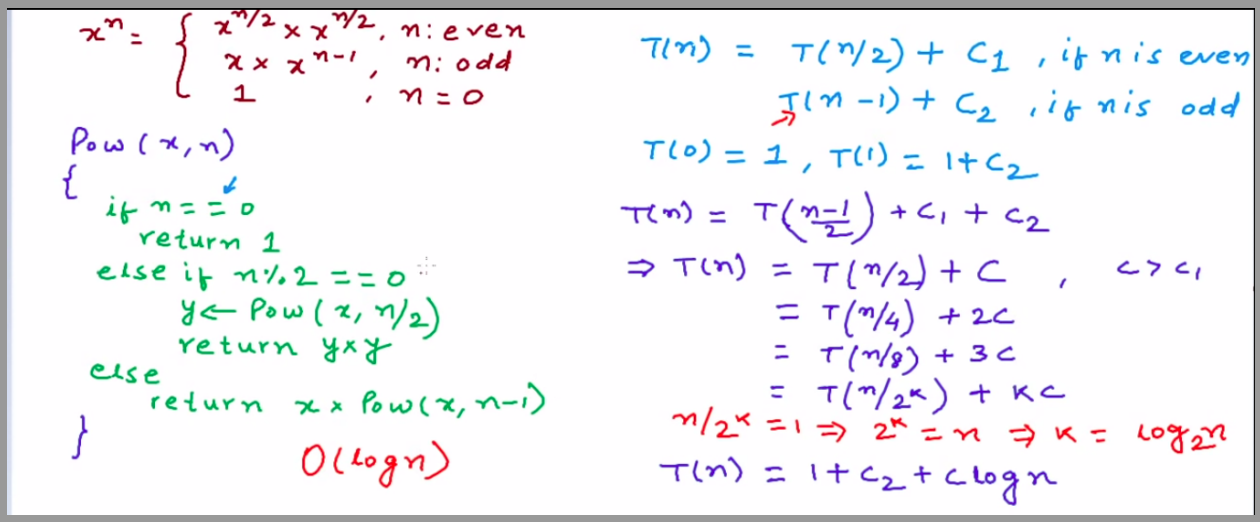
$O(2^n)$ → Fib (recursion)

$O(n)$ → Fib (Iterative)

→ Linear Time algorithm

Exponentiation





- $f = \Theta(h)$ and $g = \Theta(h)$ then $f + g = \Theta(h)$
  - $f = \Theta(h) = c_1 h + slack.$
  - $g = \Theta(h) = c_2 h + slack.$
  - $f + g = (c_1 + c_2)h + slack.$

- $\Theta(f + g) = \Theta(\max(f, g))$
  - $f = n^3$, $g = n \log n$, $h = f + g = n^3 + n \log n$
  - $\Theta(h) = \Theta(f + g) = \Theta(\max(f, g)) = \Theta(n^3)$

- ***Formally prove that***

  Θ(max(f,g))=Θ(f+g)

Just hope this clarifies a bit more:

You want to show that $h(x) = \max(f(x), g(x)) = \Theta(f(x) + g(x))$. Observe two cases:

Case1:

$h(x) = f(x)$, hence $f(x) \geq g(x)$. Then $2h(x) = 2f(x) \geq f(x) + g(x)$,

$\Leftrightarrow h(x) \geq \frac{f(x)+g(x)}{2}$

$\Leftrightarrow h(x) = \Omega(f(x) + g(x))$ This gives the lower bound.

Next, $h(x) = f(x) \leq f(x) + g(x) = O(f(x) + g(x))$

Hence, $h(x) = \Theta(f(x) + g(x))$

Case 2:

$h(x) = g(x)$. Similar to Case 1.

- *If f(n) = theta(g(n)) and g(n) = theta(h(n)), then why h(n) = theta(f(n))*

Expanding the Big-O notation by its defination usually makes things easy.

$$\because F(n) = \Theta(G(n))$$
$$\therefore \exists p, q(pG(n) \leqslant F(n) \leqslant qG(n))$$

$$\because G(n) = \Theta(H(n))$$
$$\therefore \exists r, s(sH(n) \leqslant G(n) \leqslant tH(n))$$

$$\therefore \frac{s}{p}H(n) \leqslant F(n) \leqslant \frac{t}{q}H(n)$$
$$\therefore \frac{q}{t}F(n) \leqslant H(n) \leqslant \frac{p}{s}H(n)$$

$$\therefore H(n) = \Theta(F(n))$$

.2. Simplify your asymptotic terms as much as possible

- O(f(n)) + O(g(n)) = O(f(n)) when g(n) = O(f(n)). If you have an expression of the form O(f(n) + g(n)), you can almost always rewrite it as O(f(n)) or O(g(n)) depending on which is bigger. The same goes for Ω or Θ.
- O(c f(n)) = O(f(n)) if c is a constant. You should never have a constant inside a big O. This includes bases for logarithms: since $\log_a x = \log_b x / \log_b a$, you can always rewrite O(lg n), O(ln n), or $O(\log_{1.4467712} n)$ as just O(log n).
- But watch out for exponents and products: $O(3^n n^{3.1178} \log^{1/3} n)$ is already as simple as it can be.

https://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/

Recursion & Backtracking.

* A recursive method solves a problem by calling a copy of itself to work on a smaller problem.

* Format of a recursive function.

```
if (test for base case)
        return some base case value
else if (test for another base case)
        return some base case value
else // the recursive case.
        return (some work & a recursive call)
```

(*) • The recursion has a base case - we always include a conditional statement as the first statement in the program that has a return.

• Recursive calls must address subproblems that are smaller in some sense, so that recursive calls converge to the base case.

• Recursive calls should not address subproblems that overlap.

Fibonacci, recurrence rel^n,
$T(n) = T(n-1) + T(n-2) + 1$; $T(0) = 0$ $T(1) = 1$

eg. Calculating factorial.

```
int Fact (int n) {
    if (n == 1)
        return 1;
    else if (n == 0)
        return 1;
    else
        return n * Fact (n-1);
}
```

$$T(n) = T(n-1) + 1 \quad n > 0$$
$$= 1 \quad n = 0$$

- Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends the copy of the returning method is removed from memory.

for

* Recursion vs. Iteration.

Iteration	Recursion.
1. Terminates when a cond^n is proven to be false.	1. Terminates when base case is reached.
2. Each iteration doesn't require extra space.	2. Each recursive call requires extra space on the stack frame.
3. An infinite loop could loop forever as there is no extra memory created.	3. If we get infinite recursion, program may run out of memory & give stack overflow.

- Generally iterative solutions are more efficient than recursive solutions (due to overhead of function calls).
- Any problem that can be solved recursively can also be solved iteratively & vice-versa.

* Example algorithms of Recursion.

Fibonacci Series, Factorial
Merge Sort, Quick sort
Binary search
Tree Traversals
Graph Traversals (DFS, BFS)
Dynamic Programming
Divide & conquer algos
Tower of Hanoi
Backtracking.

* Tower of Hanoi puzzle.

Algo. 1. Move the top n-1 disks from source to auxiliary tower.

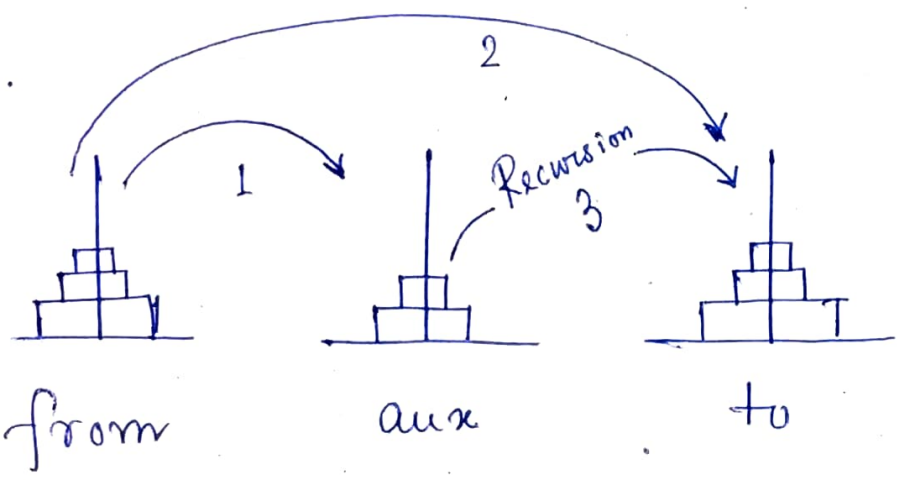
2. Move the nth disk from source ~~disk~~ to destination tower.

3. Move the $n-1$ disks from ~~source to~~ auxiliary tower to destination tower.

4. Transferring the top $n-1$ disks from source to auxiliary tower can be thought as a fresh problem & solved in same manner.

Code

```
void TOH (int n, char frompeg, char topeg,
                 char auxpeg)
{
    if (n == 1) {
    printf (" Move dosk 1 from peg %c
            to peg %c ", frompeg, topeg);
    return; }
TOH (n-1, frompeg, auxpeg, topeg);
printf (" Move disk %d from peg %c
        to peg %c ", n, frompeg,
                     topeg );
TOH ( n-1, auxpeg, topeg , frompeg);
}
```



Eg. Checking whether the array is in sorted order or not.

```
int F(int A[], int n) {
    if (n == 1)
        return 1;
    return (A[n-1] < A[n-2])? 0 : F(A, n-1);
}
```

Recursion

* **Addition** (tail recursive)

```
int add (int i, int j) {
    if (i == 0)
        return j;
    else
        return add (--i, ++j);
}
```

add (2, 3)
↓
add (1, 4)
↓
add (0, 5)
↓
return 5

* **GCD** : Euclid's Algorithm. $O(\lg_2(\min(m,n)))$

For $m \geq n > 0$, $gcd(m,n) = \begin{cases} n & \text{if } n \text{ divides } m \text{ with no remainder} \\ gcd(n, rem(m/n)) & \text{otherwise} \end{cases}$

Why? $m = n\lfloor m/n \rfloor + rem(m/n)$

$n) m (\lfloor m/n \rfloor$
$\overline{rem(m/n)}$

Now, any divisor d common to m & n must divide the first term with no remainder, since it is product of n & an integer. Therefore, d must also divide the second term since d divides m & m is the sum of 2 terms. Since any divisor common to m & n must divide the rem. of m/n we know that, in particular, the gcd does, since it is a common divisor. It just happens to be the greatest such divisor.

```
int gcd (m,n) {
  if ( m % n == 0)
      return n;
  else
      return gcd (n, m%n);
}
```

* GCD : Dijkstra's Algo

for $m, n > 0$, $gcd(m,n) = \begin{cases} m & \text{if } m = n \\ gcd(m-n, n) & \text{if } m > n \\ gcd(m, n-m) & \text{if } m < n \end{cases}$

If $m > n$, $gcd(m,n)$ is same as $gcd(m-n, n)$

Why? If m/d & n/d both leave no remainder, then $\left(\frac{m-n}{d}\right)$ leaves no remainder.

```
int gcd (m, n) {
    if (m == n)   return m;
else if (m > n)   return gcd (m-n, n);
    else return gcd (m, n-m);
}
```

* Time complexity of recursive fibonacci program:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Linear recursive function for fibonacci number,

$$F(n) = F(n-1) + F(n-2)$$

$$a_n + \alpha a_{n-1} + \beta a_{n-2} = 0$$

$$x^2 + \alpha x + \beta = 0$$

$$a_n = a r_1^n + b r_2^n$$

r_1, r_2 2 distinct roots

Characteristic eqn:

$$x^2 - x - 1 = 0$$

$\alpha = -1$, $\beta = -1$

$$x = \frac{1 \pm \sqrt{5}}{2}$$

$$F(n) = \left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n$$

$T(n)$, $F(n)$ both asymptotically same.

$$\therefore T(n) = O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

$$= O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) = O(1.6180^n)$$

golden ratio

Tight upper bound.

* Back-tracking.

Method of exhaustive search using divide & conquer.

Eg. Binary string generation

Generating k-ary string.

The Knapsack problem

Generalised Strings

Hamiltonian Cycles.

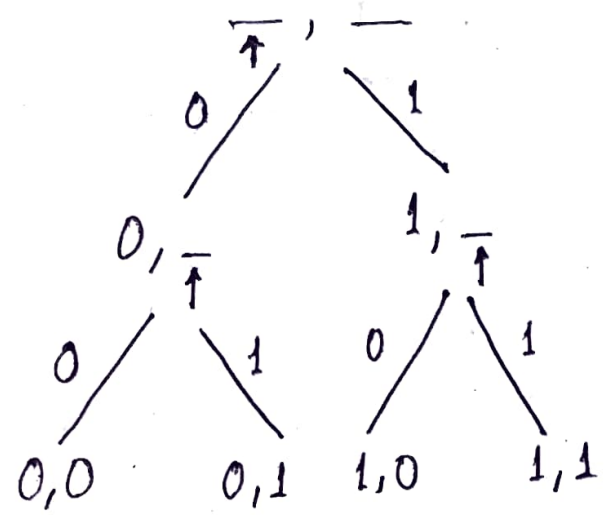
Graph colouring Problem.

Eg. Generate all strings of n bits.

Assume A[0...n-1] is an array of size n.

```
void Binary (int n) {
    if (n < 1)
        printf ("%.s", A);
                // Array A global var.
    else {
        A [n-1] = 0;
        Binary (n-1);
        A [n-1] = 1;
        Binary (n-1);
    }
}.
```

eg. n = 2



$$T(n) = \begin{cases} c & \text{if } n < 0 \\ 2T(n-1) + d & \text{otherwise.} \end{cases}$$

by Master theorem, $T(n) = O(2^n)$.

```
print_n_bit_string (int k, char a[], int len)
{
    if (k == len)
        printf ("\n %s", a);
    else {
        a[k] = '0';
        print_n_bit_string (k+1, a, len);

        a[k] = '1';
        print_n_bit_string (k+1, a, len);
    }
}
```

- Common problems with recursion : i) Omitting the base case
  ii) Blowing out the stack - when recursion is too deep iii) Failure to make progress
- Tail recursion : When a recursive f^n calls itself only once & as the last thing it does.

↳ A tail recursive fun^n is better than non-tail recursive f^n, as tail-recursion can be optimized by modern compilers (using tail call elimination).

eg. Quicksort is tail recursive; but Mergesort is not. (Quicksort performs better)

→ Quick Sort.

```
void quickSort (int arr[], int low, int high)
{  if (low < high){
      int k= partition (arr, low, high);
      quick Sort (arr, low, k-1);
      quickSort (arr, k+1, high);
   }
}
```

tail call elimination using goto:

```
void quickSort (int arr[], int low, int high)
{
   start:
   if (low < high) {
      int k = partition (arr, low, high);
      quicksort (arr, low, k-1);
      low = k+1;
      high = high;
      goto start;
   }
}
```

→ Factorial calculation

```
int fact (int n) {
   if (n==0) return 1;
   return n * fact(n-1);
}
```

→ non-tail-recursive

→ tail call elimination

accumulator

```
int fact (int n, int a) {
    if (n == 0) return a;
    return ( fact (n-1, n*a) );
}
```

Idea is to use an accumulator to accumulate the factorial value. When n reaches 0, return the accumulated value.

- Function Stack frame management in tail call elimination. :

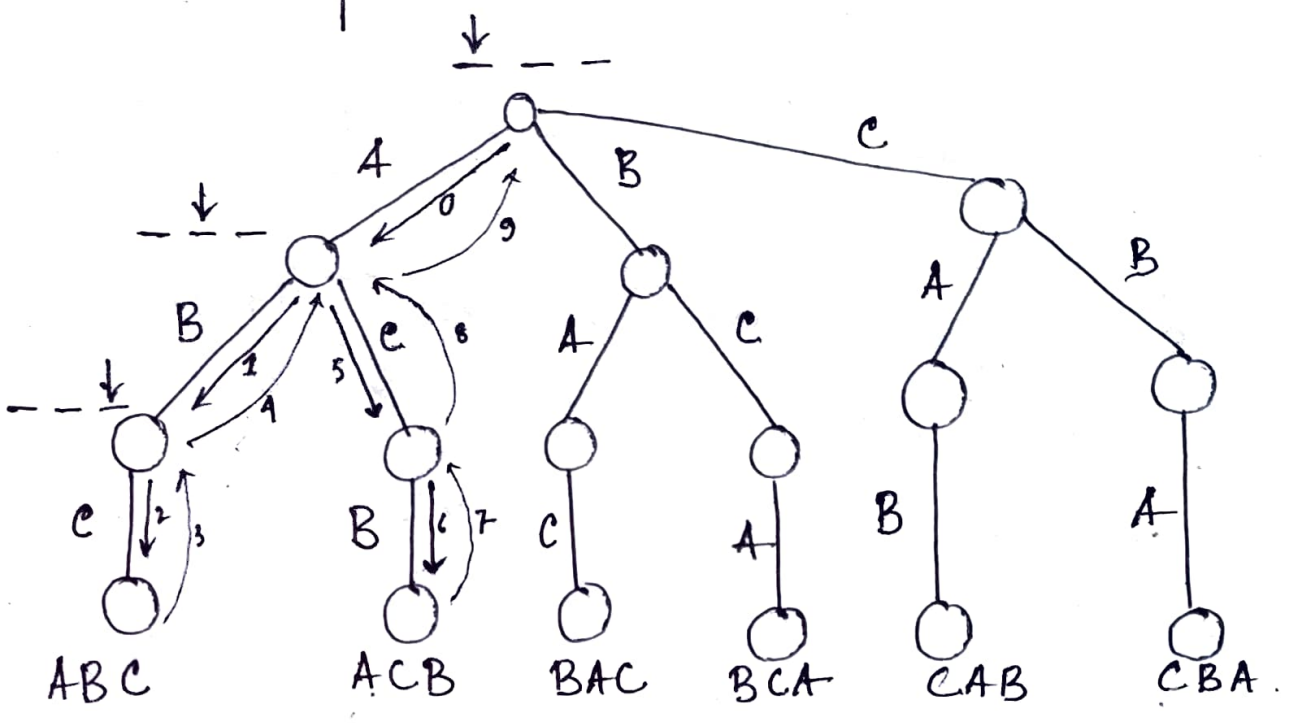
There are no statements needed to be performed after function call statement. So, preserving state (local var, data of f call) & frame of parent f^n is not reqd. Child f^n is called & finishes immediately. It doesn't have to return control back to the parent f^n. Hence, f^n executes in constant memory space. This makes tail recursion faster & memory-friendly.

So, in tail recursion, compiler may be able to collapse the stack down to one entry, so we save stack space.

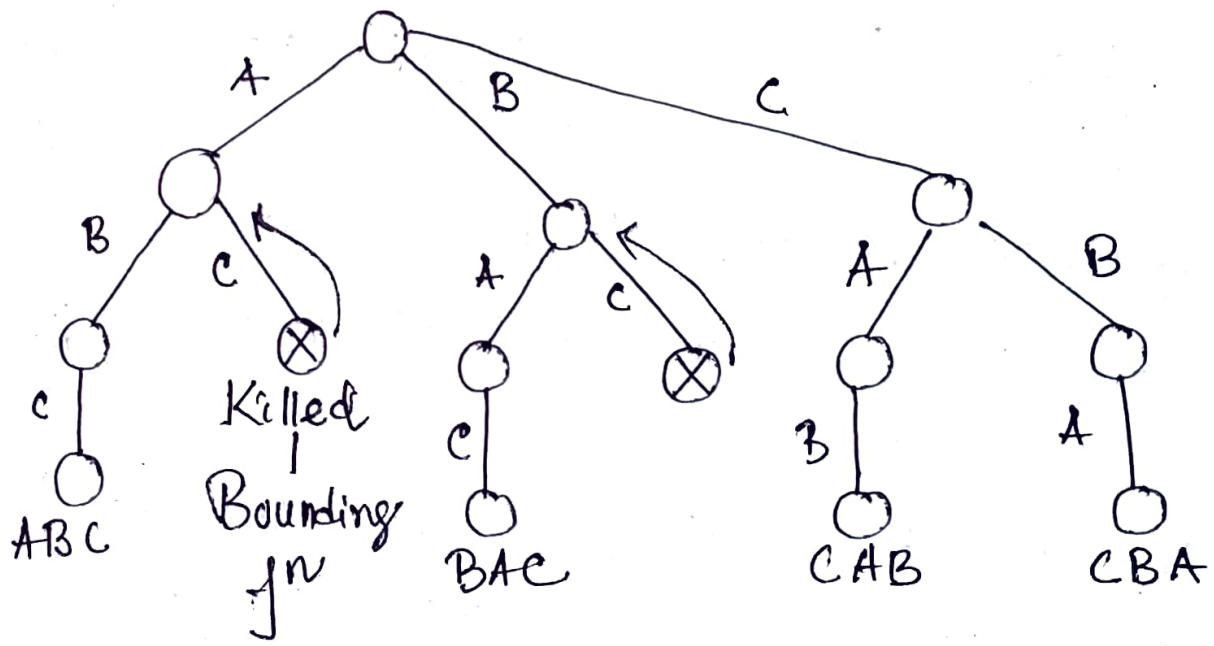
*Backtracking Problem:

- A, B, C seating arrangements.

State - space tree.



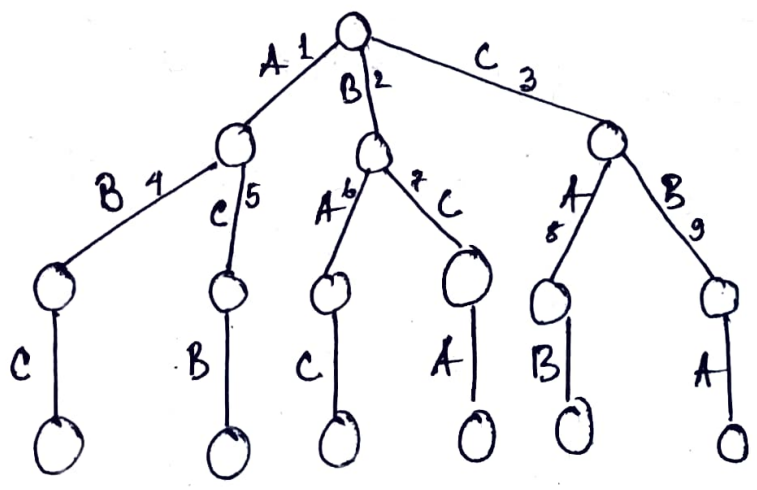
- C should not be in middle.



→ In backtracking, DFS

In branch & bound, BFS.

↓ A, B, C arrangement



Sorting

* Insertion sort algorithm and analysis.

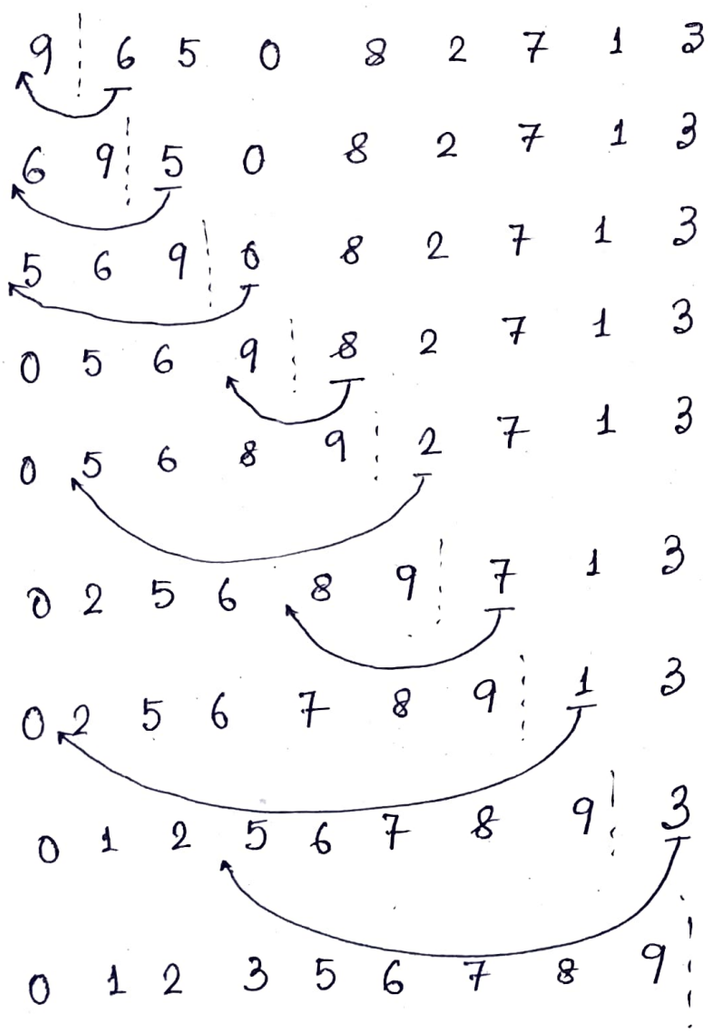
(inplace, stable, online

Iterative version

```
Insertion_sort (A) {
    for j = 2 to A.length
        key = A[j]
        // insert A[j] into sorted sequence A[1...j-1]
        i = j-1
        while (i > 0 and A[i] > key)
            A[i+1] = A[i]
            i = i-1
        A[i+1] = key
}
```

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

eg.



Input sequence

array before marker → sorted part

O/P sequence.

Sorted.

✓ → Insertion sort is basically inserting keys one by one into the sorted subarray so that the key is placed at the correct place in the sorted subarray.

	no. of comparisons	no. of movements
$j = 2$	1	2
$j = 3$	2	4
$j = 4$	3	6
⋮		
$j = n$	$n-1$	$2(n-1)$

If reverse sorted, worst case.

$+ = 2(1) + 2(2) + 2(3) + \cdots + 2(n-1)$

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

Worst case time complexity $O(n^2)$

Best case n n $\Omega(n)$ [1 comparison for each element

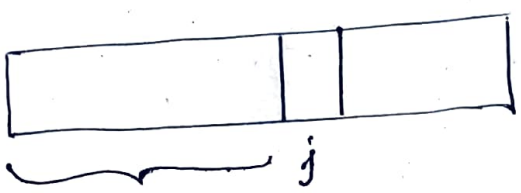
(in sorted order)

Space complexity = $O(1)$ key, i, j, n

When space complexity is $O(1)$, that ~~sorting~~ algo is called in place.

- No of comparisons

1. $O(\log n)$ if used binary search



No. of movements

$O(n)$ when binary search

⇒ No improvement in time complexity even when binary search is used.

2. Linked list (doubly).

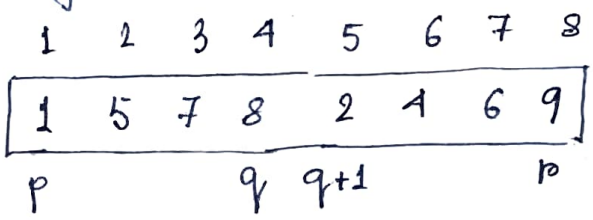
$O(n)$ $O(1)$.

⇒ No improvement.

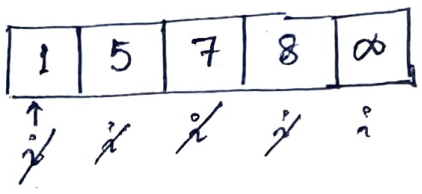
- Max. no. of swaps = $O(n^2)$ [Reverse sorted]

* Merge sort algorithm and analysis

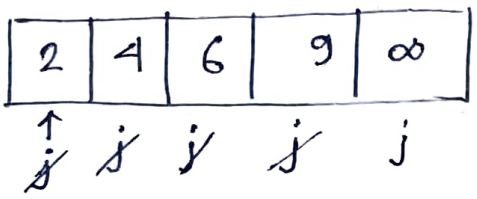
Merging

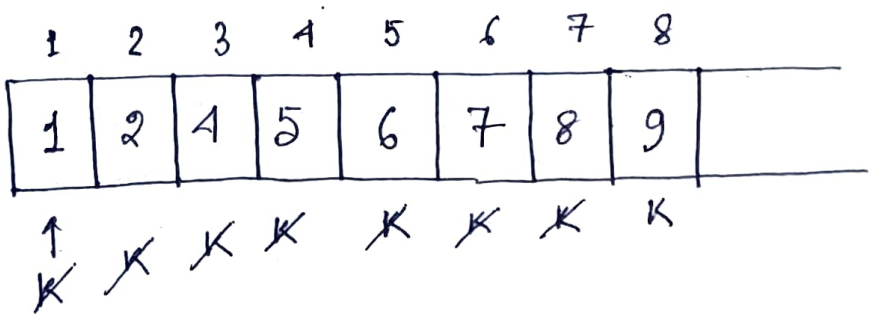


L



R





merge_routine (A, p, q, r) {

$n_1 = q - p + 1$ // # elems in 1st list

$n_2 = r - q$ // # " " 2nd list

let $L[1 \ldots n_1+1]$ & $R[1 \ldots n_2+1]$ be new arrays

for (i = 1 to n_1) $O(n)$

$L[i] = A[p+i-1]$. ✓

for (j = 1 to n_2) $O(n)$

$R[j] = A[q+j]$ ✓

$L[n_1+1] = \infty$

$R[n_2+1] = \infty$

$i = 1$, $j = 1$

for ($k = p$ to r) $O(n)$.

if ($L[i] \leq R[j]$)

$A[k] = L[i]$

$i = i+1$

else $A[k] = R[j]$

$j = j+1$.

$(q+1)+j-1$

Total time complexity

$O(n)$.

Space complexity $O(n)$.

- If 2 sorted lists are of size n & m, Time complexity is $O(n+m)$.

- ∞ added at last of L & R to copy the last element of either L or R successfully (always less than ∞.)

- merge_sort (A, p, r) {

  if p < r

  q = ⌊(p+r)/2⌋

  merge_sort (A, p, q)

  merge_sort (A, q+1, r)

  merge_routine (A, p, q, r)

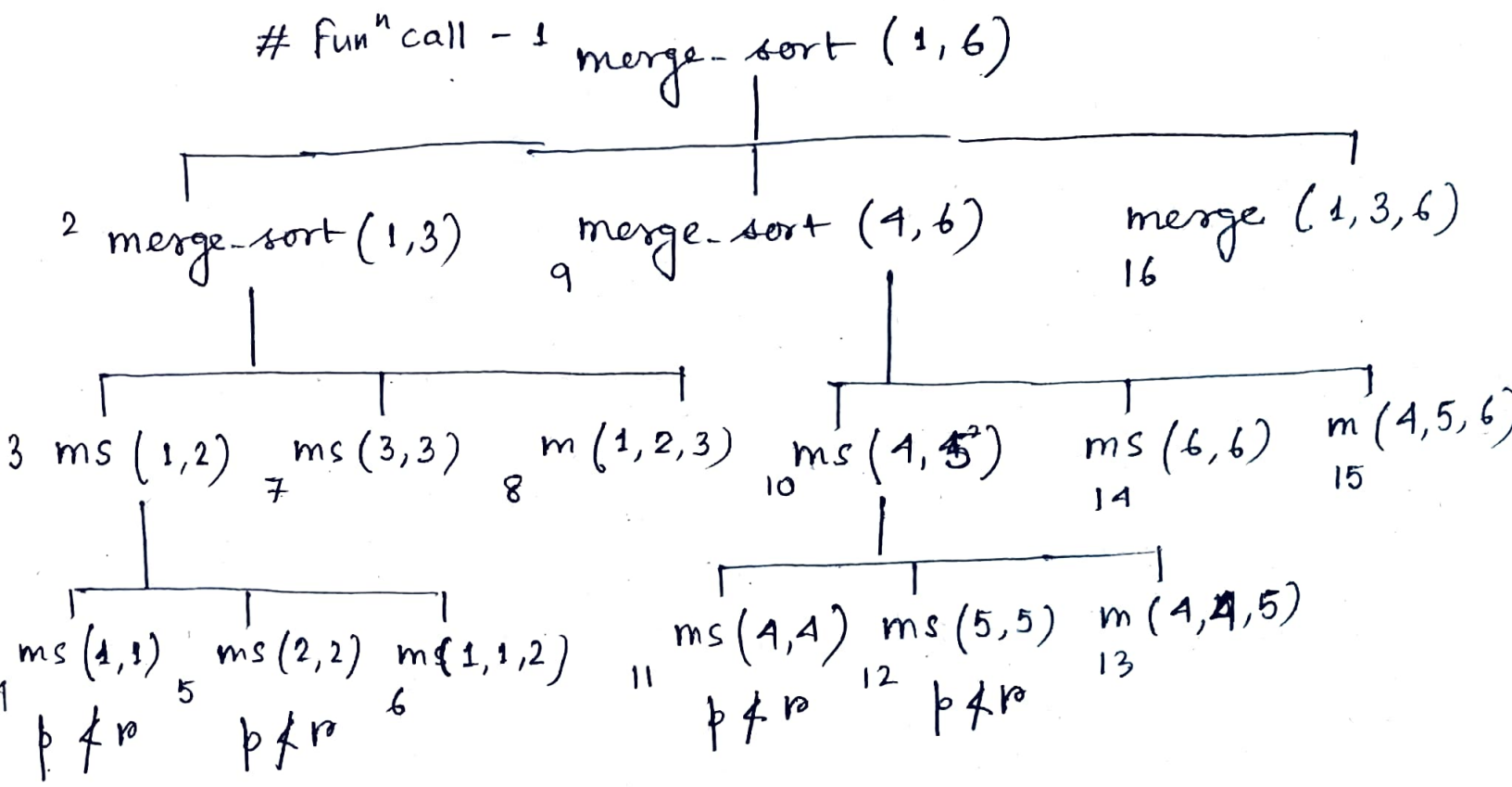
}.

[Divide & Conquer]

	1	2	3	4	5	6
A	9	6	5	0	8	2

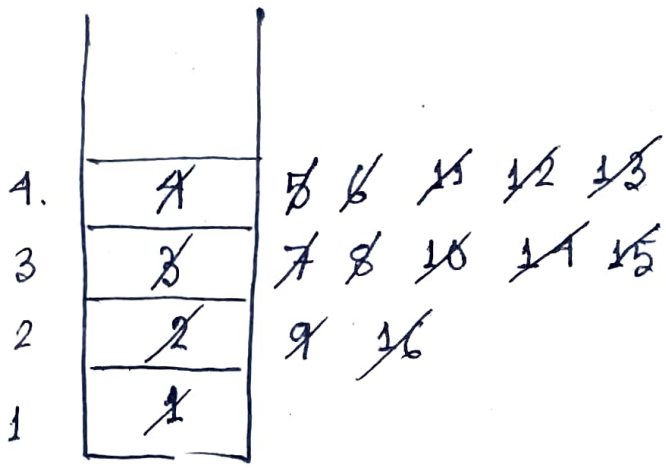
eg.

$\left\lfloor \frac{1+6}{2} \right\rfloor = 3$



Fun^n call sequence – 1, 2, 3, ..., 16

{ ms merge_sort
{ m merge_routine



Stack frame for recursion.

no. of level n

$(\lceil \log n \rceil + 1)$

Size of stack

$(\lceil \log n \rceil + 1)\, K.$

$= O(\log n).$

- For merging $O(n)$.
- Space complexity of merge sort $= O(n) + O(\lg n)$

$= O(n).$

- Time complexity.:

```
ms (A, p, r) {
    if p < r
        q = ⌊(p+r)/2⌋
    ms (A, p, q)        T(n/2)
    ms (A, q+1, r)      T(n/2)
    m (A, p, q, r)      O(n)
}
```

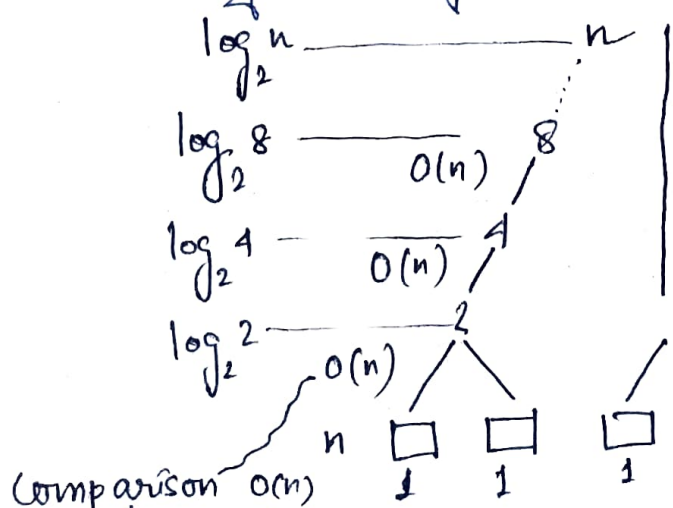
$T(n) = 2T(n/2) +$ (Time to merge 2 arrays of size $n/2$) $\downarrow O(n)$

$T(n) = 2T(n/2) + O(n). \quad = \Theta(n \log n)$

(Master's theorem) or Recursion tree

Q. Given n elements, merge them into one sorted list using merge procedure.

$O(n \log_2 n)$



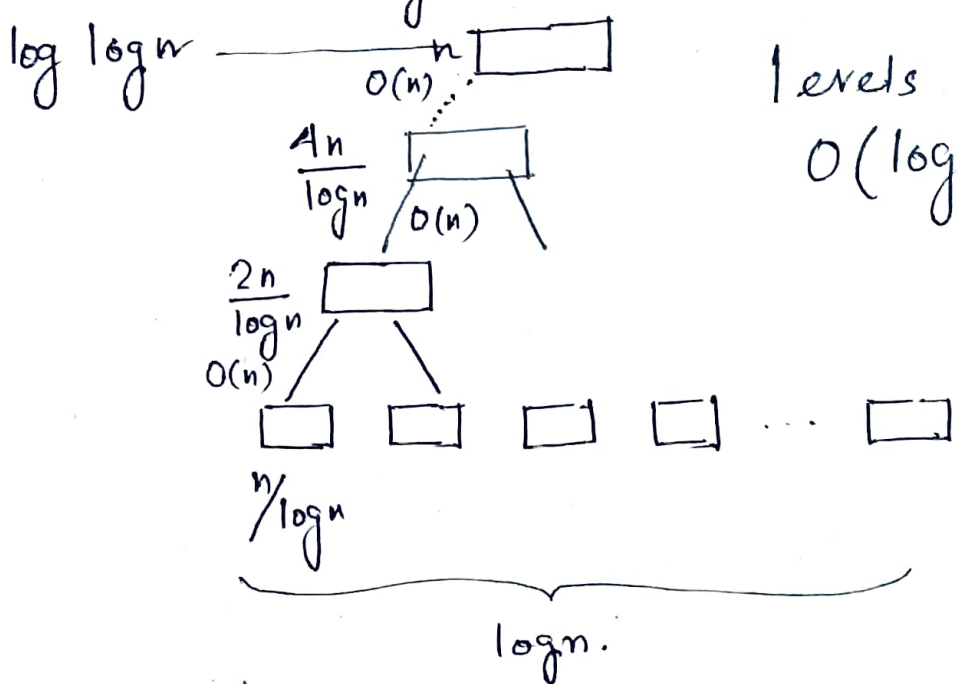
Levels

$O(\log(\text{no. of elems. in last level}))$

$= O(\log n)$

Q Given 'logn' sorted lists each of size $n/\log n$, what is the total time required to merge them into one single list?

→ # elems = $\log n \times \frac{n}{\log n} = n$.

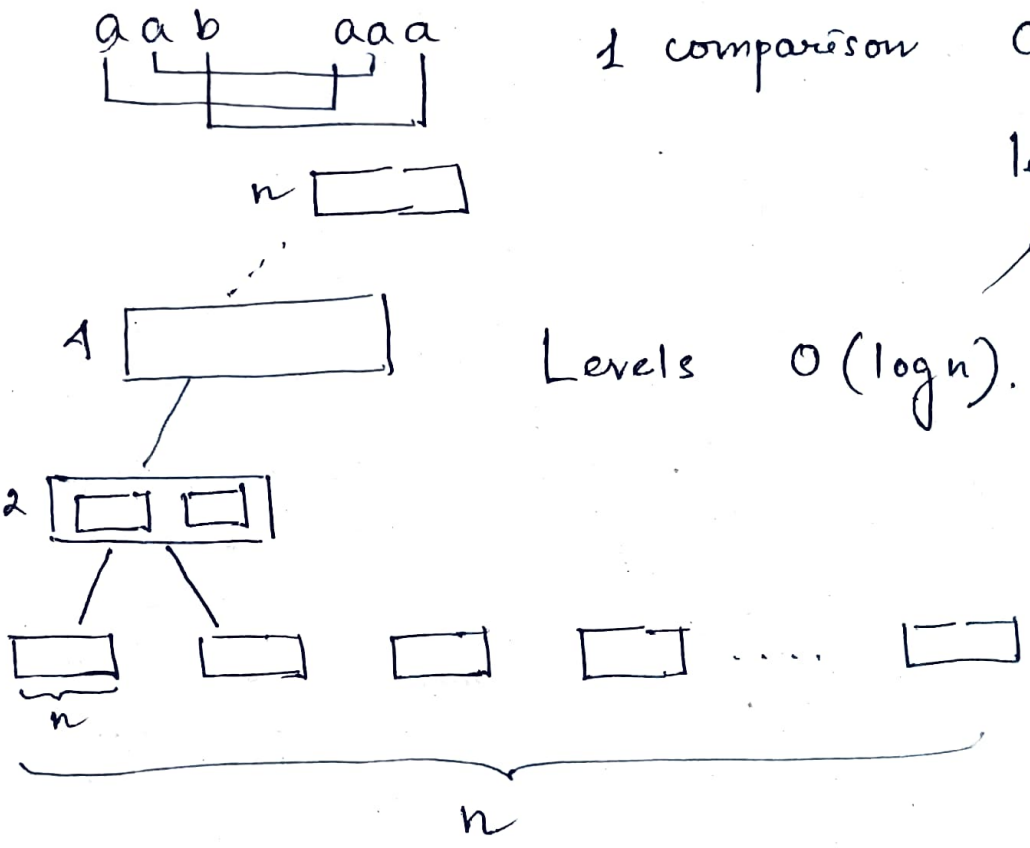


O(n log logn).

plug in place of logn → log logn / #levels

Q n strings each of length n are given then what is the time taken to sort them?

→



O ($n^2 \log n$). total time complexity

→ Expected no. of comparisons in a merge step when each sorted array has n/2 elems — $\frac{n^2}{n+2} \approx n-2$

(cs.stackexchange – explanation)

- Merge sort uses divide & conquer.
- Merging 2 sorted lists of size m, n
  total time taken $O(m+n)$
- 2-way merging.

- External merge sort – Divide the data into reasonable size, sort them using any sorting algo in RAM, then k-way merge. (applied gg).

* Quick Sort

Takes last elem as pivot, places the pivot at its correct position in sorted array & places all smaller (than pivot) to left of pivot & all greater to right.

```
partition (A, p, r) {
    x = A[r]   // pivot.
    i = p - 1
    for (j = p to r-1) {
        if (A[j] ≤ x) {
            i = i+1
            exchange A[i] with A[j]
        }
    }
    exchange A[i+1] with A[r]
    return i+1
}
```

O(n)

init i = p-1
j = p
x = A[r] pivot

eg.

	p					r
	50	23	9	18	61	32
	i, j					
	50	23	9	18	61	32
✗	i	j				
	23	50	9	18	61	32
	i	j				
	23	50	9	18	61	32
	✗	i	j			
	23	9	50	18	61	32
		i	j			
	23	9	50	18	61	32
		✗	i	j		
	23	9	18	50	61	32
			i	j		

50 ≰ 32

23 ≤ 32

9 ≤ 32

18 ≤ 32

not stable

23 9 18 50 61 32
i, i+1, j
61 ≰ 32 → out of for loop

23 9 18 32 61 50.
≤32 | pivot (i+1) | >32
Repeat | Repeat

23 9 18 — 23 ≠ 18

i j

23 9 18 — 9 ≤ 18

i j

9 23 18

i j

9 18 23

61 50 — 61 ≠ 50

i j

50 61.

9 18 23 32 50 61 Sorted

- Quicksort (A, p, r) {
  if (p < r) {
  q = partition (A, p, r)
  Quicksort (A, p, q-1)
  Quicksort (A, q+1, r)
  }

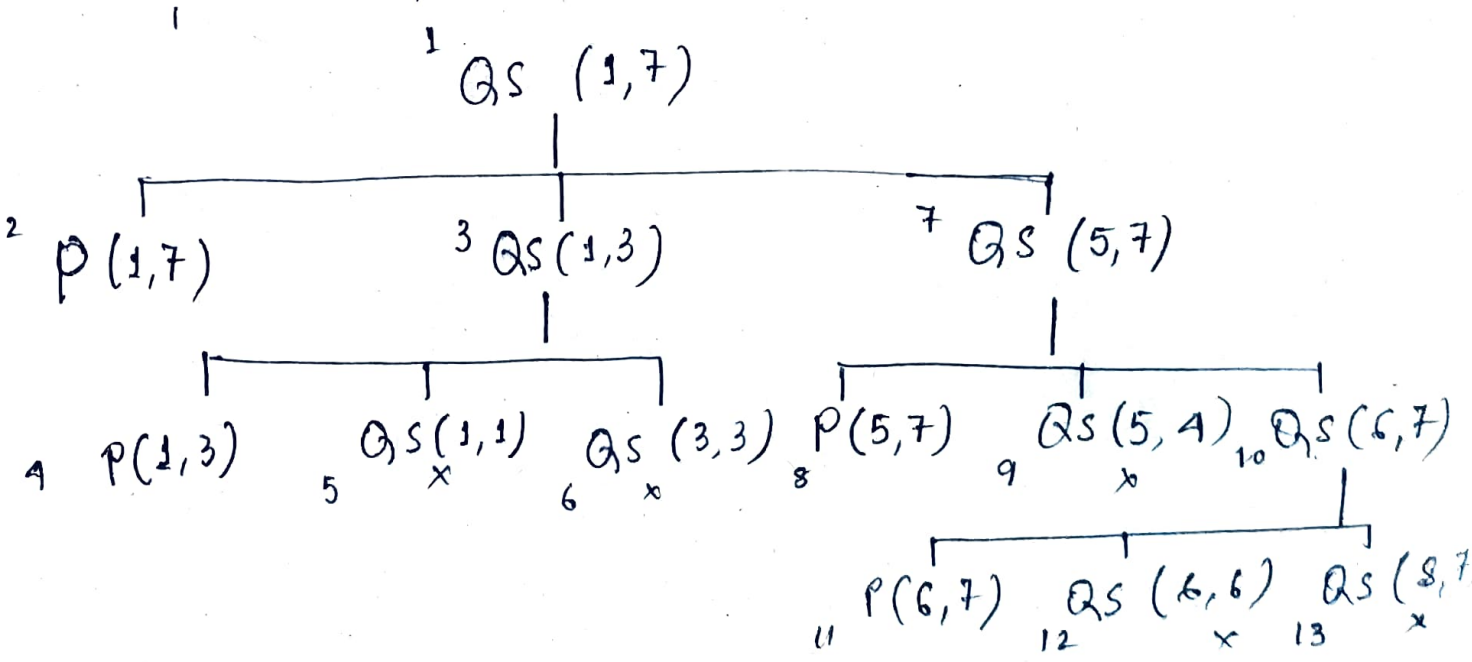
Best	Worst
$\Theta(n)$	$O(n)$
$T(n/2)$	$T(0)$
$T(n/2)$	$T(n-1)$

- 1 2 3 4 5 6 7

  5 7 6 1 3 2 4

  1 3 2 4 7 6 5

  1 2 3 4 5 6 7



- No. of stack entries = No. of levels reqd.
- Space complexity $O(\log n)$ when balanced.
  $O(n)$.
- Best case time complexity

  Partitions the array into 2 equal parts.

  as in merge sort $\longrightarrow T(n) = 2T(n/2) + O(n)$
  $= \Theta(n \log n)$.

Worst case

$T(n) = T(n-1) + O(n)$ | All elems < pivot or All elems > pivot
$= T(n-1) + cn$
$= T(n-2) + c(n-1) + cn$
$\vdots$
$= c + c \cdot 2 + c \cdot 3 + \ldots + cn$
$= O(n^2)$

pivot puts element at front or end

- Input sequence is in ascending order.

  1 2 3 4 5 6

  $T(n) = O(n) + T(n-1) = O(n^2)$

- Input sequence is in descending order.

  6 5 4 3 2 1.

  $T(n) = O(n) + T(n-1) = O(n^2)$.

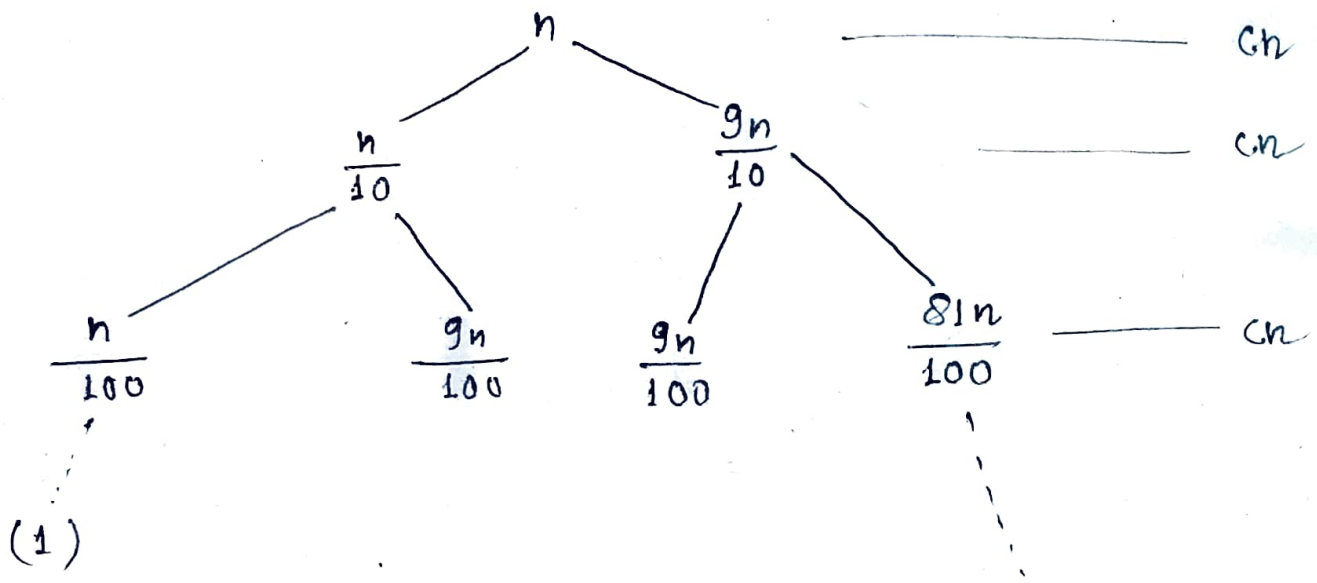
- All elems equal.

  2 2 2 2 2 2.

  i j

  $T(n) = O(n) + T(n-1) = O(n^2)$.

- Splitting in ratio 1 : 9. / or 1 : 99 or 1 : 9g



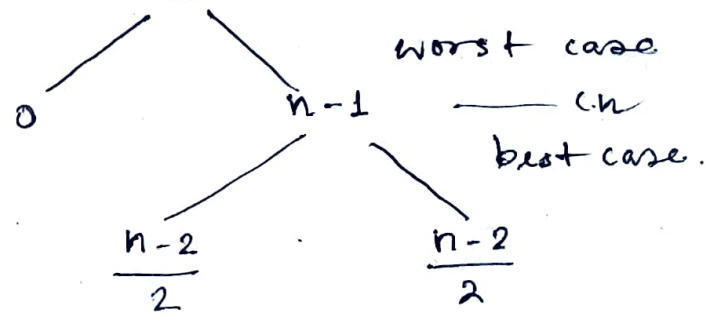
$n - \frac{n}{10/9} - \frac{n}{(10/9)^2} - \ldots - 1$

$\frac{n}{(10/9)^k} = 1 \Rightarrow k = \log_{10/9} n$

$\log_{10/9} n \rightarrow \Theta(\log_2 n)$ # levels.

Total work $\Theta(n \log_2 n)$ Best case TC.

- alternating best & worst cases



$T(n) = cn + cn + 2T\left(\frac{n-2}{2}\right)$

$= 2cn + 2T\left(\frac{n-2}{2}\right)$

$\leq O(n) + 2T(n/2)$

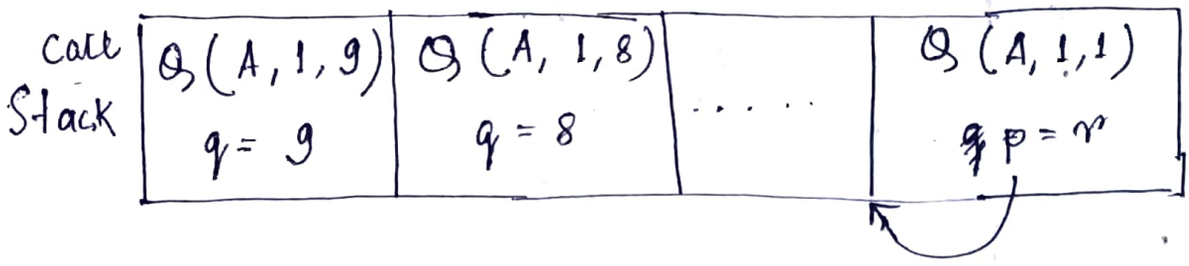
TC $\Theta(n \log n)$.

* Space complexity of Quick Sort using Call-stack.

A	1	2	3	4	5	6	7	8	9
	p								r

q : pivot

```
QS(A,p,r)
if (p<r)
  q = partition(A,p,r)
  QS(A,p,q-1)
  QS(A,q+1,r)
```



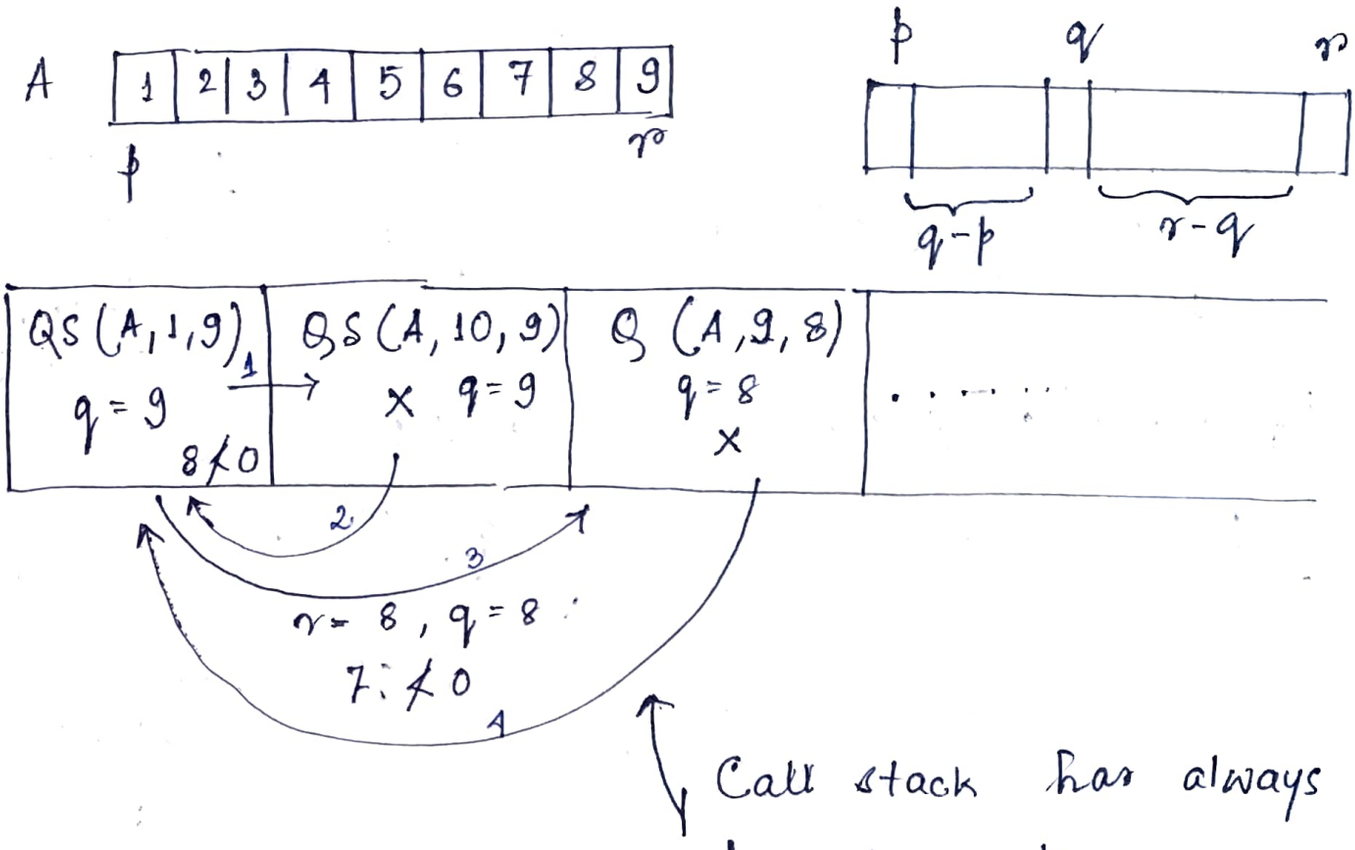
max 9 entries in call stack

Worst case space complexity O(n).

When all elems in increasing order

- Sedgewick's Solution.

```
QS(A,p,r)
While (p<r)
    q = partition(A,p,r)
    if q-p < r-q
    // recurse on smaller sub-array first
        QS(A,p,q-1)
        p = q+1
    else
        QS(A,q+1,r)
        r = q-1.
```



1. Process smaller subarray first.
2. Update variables after going back to the calling f^n. ($p = q+1$ or $r = q-1$)
3. Process larger subarray.

Worst case happens here when pivot breaks the array into exactly 2 halves. Then call stack depty $\lg n$. $n \to n/2 \to n/4 \to \cdots$

So, here space complexity $O(\lg n)$.

5. The median of n elements can be found in $O(n)$ time. Which is correct about complexity of quick sort in which median is selected as pivot?

a) $O(n^2 \log n)$
b) $O(n^2)$
c) $O(n \log n)$

→ After the partition algorithm it will go in the middle of the array having half of the elems to left & half in the right. Thus after the partition the array will be divided into 2 equal parts of $n/2$ elems each.

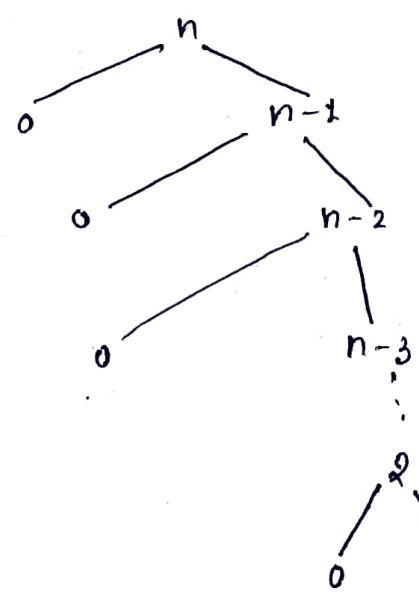
$$T(n) = \underbrace{O(n)}_{\text{selecting median}} + \underbrace{O(n)}_{\text{for partition}} + T(n/2) + T(n/2)$$

$$= O(n) + 2T(n/2) = O(n \log n).$$

The median value will allow the algo to split into 2 approximately equal parts, so the runtime will speed up.

- Worst case running time

QS has most unbalanced partitions possible. The original call takes cn time for some c, recursive call on $n-1$ elems takes $c(n-1)$ time, recursive call on $n-2$ elems takes $c(n-2)$ time & so on.



Sum = $c\left((n+1)\frac{n}{2} - \right.$

QS worst case running time $\Theta(n^2)$.

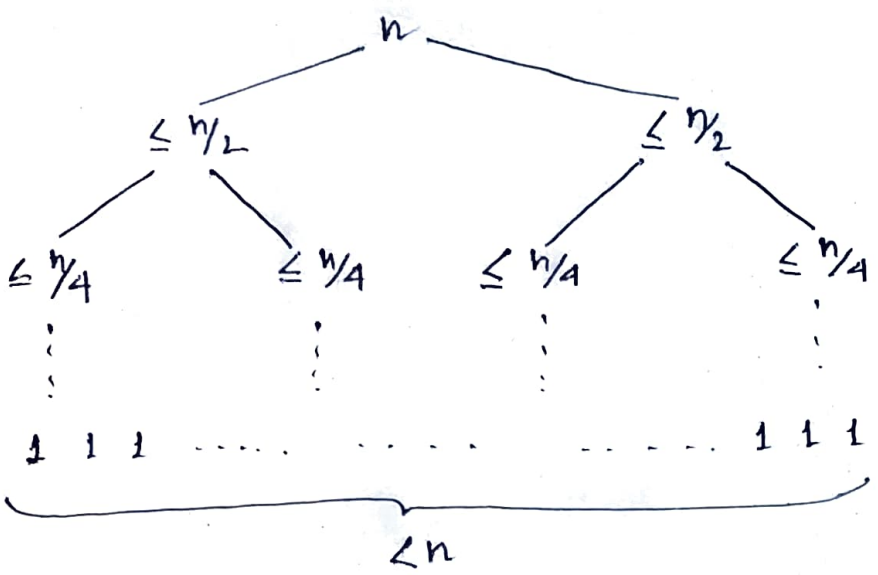
QS Time Complexity Analysis.

- Best case running time.

Partitions are as evenly balanced as possible. Sizes are equal or are within 1 of each other.

Subproblem size

Total partitioning time for all subproblems of this size.



cn

$\leq 2 \cdot \frac{cn}{2} = cn$

$\leq 4 \cdot \frac{cn}{4} = cn$

$\leq n \cdot c = cn$

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

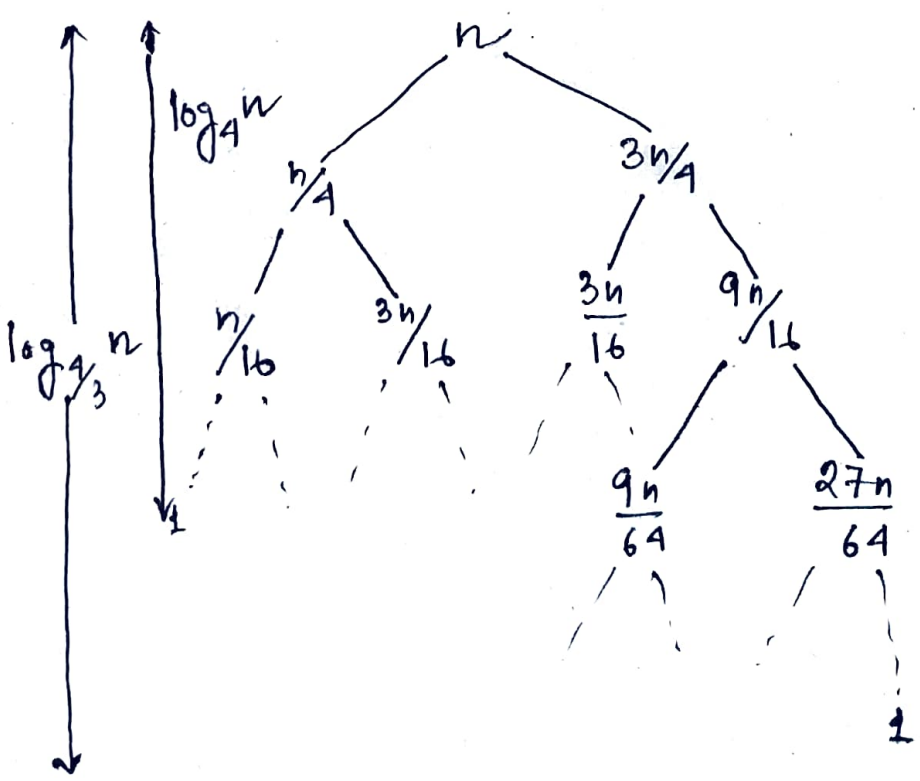
$$= \theta(n \log_2 n)$$

- Average case running time.

Not evenly balanced partitions (3-to-1 split).

Subproblem size

Part. time



Left child of each node represents a subproblem size 1/4 as large & the right child represents a subproblem size 3/4 as large. By following a path of left children, we get from the root down to a subproblem of size 1 faster than any other path. After $\log_4 n$ levels, we get to subproblem of size 1. Going down a path of right children it takes $\log_{4/3} n$ levels to get down to a subproblem of size 1.

In each of the first $\log_4 n$ levels, there are n nodes & so the total partitioning time for each of these levels is cn. For the rest of the levels, each has fewer than n nodes & so the partitioning time for every level is at most cn. Altogether there are $\log_{4/3} n$ levels, yielding total partitioning time $O(n \log_{4/3} n) = O(n \log_2 n)$.

albeit with a larger hidden constant factor than the best case running time.

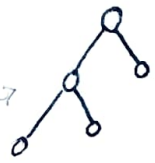
$\log_{4/3} n = \frac{\log_2 n}{\log_2 (4/3)}$

---

* Not QS related

	Insert	Search	Find min	Delete min
Unsorted array.	$O(1)$	$O(n)$	$O(n)$	$O(n)$.
Sorted array (non-desc.)	$O(n)$.	$O(\log n)$.	$O(1)$	$O(n)$.
LL	$O(1)$	$O(n)$	$O(n)$.	$O(n)$
Heap. (min)	$O(\log n)$		$O(1)$	$O(\log n)$

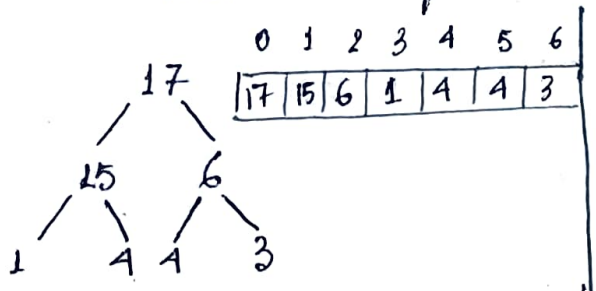
* Heap. Generally n-ary tree.



Heap is almost complete.

Basic requirement of a heap is that the value of a node must be $\geq$ (or $\leq$) to the values of its children (heap property). All leaves should be at h or $h-1$ levels ($h > 0$).

— Max Heap : Value of a node must be greater than or equal to the values of its children.



locations in array

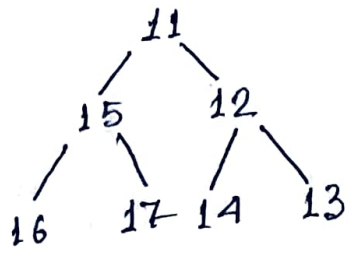
l_child = $2l+1$

r_child = $2l+1+1$ = $2l+2$.

parent = $\lfloor \frac{l}{2} \rfloor$ $\lfloor \frac{l-1}{2} \rfloor$

When indexing starts at 0

— Min Heap :



- If array in descending order, it's max heap.
  If array in ascending " " min heap.
- Height of any binary heap $\log_2 n$.
  n — no. of nodes.

$\lfloor \frac{n}{2} \rfloor$ to $n-1$ all leaves

```
max_heapify (A, i) {        [Refer DS Note]
    l = 2i+1;
    r = 2i+2;
    if (l ≤ A.heap_size and A[l] > A[i])
        largest = l;
    else largest = i;
    if (r ≤ A.heap_size and A[r] > A[largest])
        largest = r;
    if (largest ≠ i)
        exchange A[i] with A[largest];
        max_heapify (A, largest);
}
```

TC $O(\log n)$.

SC $O(\log n)$

- build_max_heap (A) {

  A.heap_size = A.length;
  for (i = ⌊A.length/2⌋ down to 1)
      max_heapify (A, i);

}

TC O(n).
SC O(logn)

- heap_extract_max (A) {

  if A.heap_size < 1
      error 'heap_underflow'
  max = A[1]
  A[1] = A[A.heap_size]
  max_heapify (A, 1)
  return max

}

O(logn) TC
O(logn) SC.

- heap_increase_key (A, i, key) {

  if (key < A[i])
      error.
  A[i] = key
  while (i > 1 and A[i/2] < A[i])
      exchange A[i] and A[i/2].
      i = i/2.

}

TC O(logn).
SC O(1)

- max_heap_insert (A, key)

  heap_size[A] ← heap_size[A] + 1
  A[heap_size[A]] ← −∞
  heap_increase_key (A[heap_size[A]], key).

TC O(logn).

-

	Find max	Delete max	Insert	increase	decrease
Max heap	O(1)	O(logn)	O(logn)	O(logn)	O(logn)

Find min	delete	Search
O(n)	O(n)	O(n)

- Heap_sort (A) { TC = O(nlogn)

  build_max_heap (A)

  for (i = A.length down to 2)

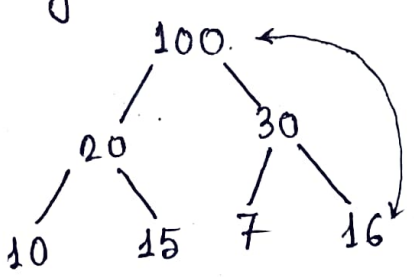
  exchange A[1] with A[i]

  A.heap_size = A.heap_size − 1

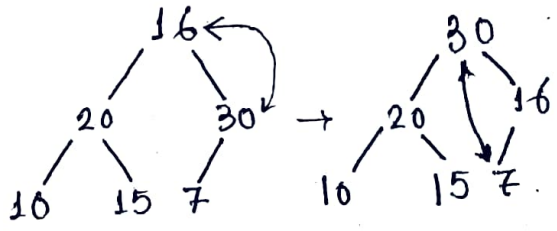
  max_heapify (A, 1)

}

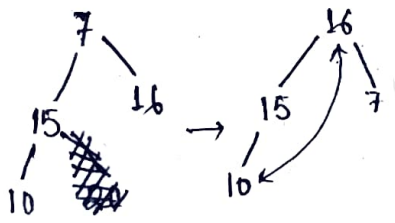
eg.



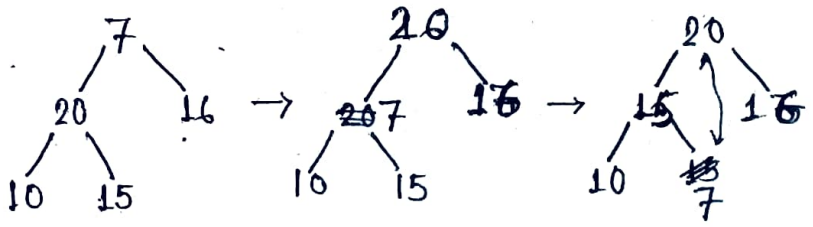
100 20 30 10 15 7 16



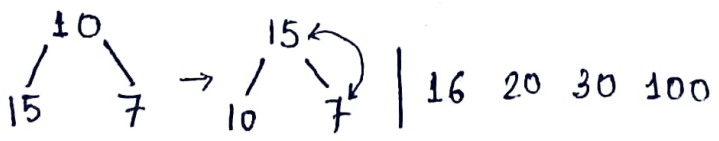
30 20 16 10 15 7 | 100



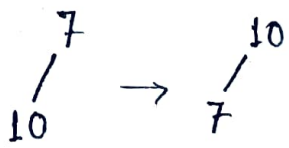
16 15 7 10 | 20 30 100



20 15 16 10 7 | 30 100



16 20 30 100 ⇒



10 7 | 15 16 20 30 100

7 10 15 16 20 30 100

Q. In a heap with n elems with the smallest elem at the root, the 7th smallest elem can be found in time

a) $\Theta(n \log n)$ b) $\Theta(n)$ c) $\Theta(\log n)$ ✓d) $\Theta(1)$.

7th smallest elem present in any of 7 levels.
Total possible elems to check = 1+2+4+6+8+16+32 = 69 = O(1)

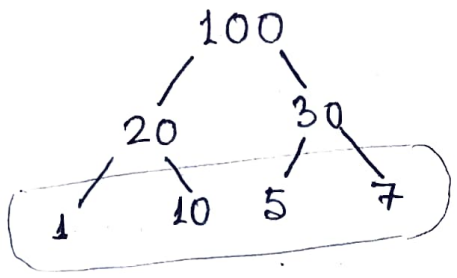
delete_min 1st — $\Theta(\log n)$
delete_min 2nd — $O(\log n)$
⋮
delete_min 6th — $O(\log n)$
find_min — $O(1)$
insert — $6 \times O(\log n)$

$6 \times O(\log n) + O(1) + 6 \times O(\log n)$
$= O(\log n)$

Q. In a binary max heap containing n numbers, the smallest elem can be found in time

✓a) $\Theta(n)$ b) $\Theta(\log n)$ c) $\Theta(\log \log n)$ d) $\Theta(1)$.

→ Smallest elem must be in the leaves.



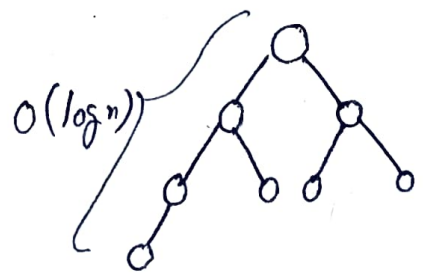
Q. Inserting an elem into a max heap. If we perform a binary search on the path from new leaf to root to find the position of newly inserted elem, no of comparisons performed are -

due to binary search

$O(\log \log n)$

$\lfloor n/2 \rfloor$ to $n-1$.

#leaves $O(n/2)$
$= O(n)$.



Q. We have a binary heap on n elems and wish to insert n more elems (not necessarily one after another) into this heap. Total time reqd. for this is —

a) $\theta(\log n)$ ✓b) $\theta(n)$ c) $\theta(n \log n)$ d) $\theta(n^2)$

→ First add n elems at the end of array.

Build-heap on 2n elems.

↳ $O(2n) = O(n)$

Q. G' Assume that the algorithms considered here sor
16 the input sequences in ascending order. If th
input is already in ascending order, which is
true?

✓i) Quick sort runs in $\theta(n^2)$ time

ii) Bubble sort runs in $\theta(n^2)$ time n

iii) Merge sort runs in $\theta(n)$ time nlgn

✓iv) Insertion sort runs in $\theta(n)$ time.

a) i & iv only e) ii & iv only

b) i & iii) only ✓d) i & iv only

→ Worst case for quick sort.

Best case for bubble sort.

Merge sort runs for nlogn time

Insertion sort runs in $\theta(n)$ time.

Q. G'16. Worst case running time of Insertion sort, merge sort & quick sort are -

a) $\theta(n\log n)$, $\theta(n\log n)$, $\theta(n^2)$

~~b)~~ $\theta(n^2)$, $\theta(n\log n)$, $\theta(n^2)$.

* Bubble Sort.

If pairwise sorted ⇒ then sorted

```
Bubble_sort (int a[], n) {
    int swapped, i, j;
    for (i = 0; i<n; i++) {
        swapped = 0;
        for (j=0; j < n-i-1; j++) {
            if (a[j] > a[j+1]) {
                Swap (a[j], a[j+1]);
                swapped = 1;
            }
        } if (swapped == 0) break;
    }
}
```

Time complexity -

No. of swaps required
⇓
No. of comparisons

For n elements first we have to make $n-1$ comparisons [$i=0$]

Provided swapped = 1 always [worst case],

$(n-1)+(n-2)+(n-3)+\cdots+1 = \frac{(n-1)(n-2)}{2} = O(n^2)$.

Worst case - reverse sorted array

Best case time complexity - Sorted array
Iteration for only $i=0$.

$O(n-1) = O(n)$

Bubble_sort (int a[], int n)

- do {

  swapped = false // iter = n-1

  for i = 0 to iter

    if (arr[i] > arr[i+1])

      swap (arr[i], arr[i+1])

      swapped = true

  iter = iter - 1

} while (swapped).

- iter = n-1 before do-while

	0	1	2	3	4
A	15	16	6	8	5

$n = 5$.

pass 1:

15 16 6 8 5

15 16 6 8 5

15 6 16 8 5

15 6 8 16 5

15 6 8 5 | 16 | Sorted portion

16 at its correct position

pass 2:

15 6 8 5 | 16

6 15 8 5 16

6 8 15 5 16

6 8 5 | 15 16 | Sorted

pass 3:

6 8 5 | 15 16

6 8 5 | 15 16

6 5 | 8 15 16 | sorted.

pass 4:

6 5 8 15 16

5 | 6 8 15 16 | sorted

If n elems, sort only n-1 elems.

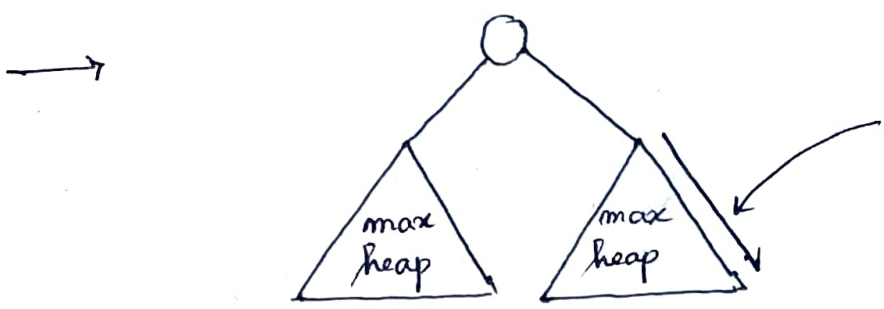
If in some pass, there is no swapping, no need to go for further passes.

Q. G'15 Consider a complete binary tree where the left and right subtrees of the root are max heaps. The lower bound for the no. of operations to convert the tree to a heap is —

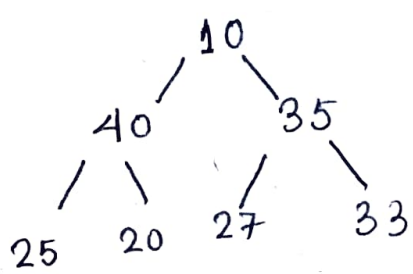
a) $\Omega(\log n)$ b) $\Omega(n)$ c) $\Omega(n \log n)$

d) $\Omega(n^2)$

Lower bound - best algorithm

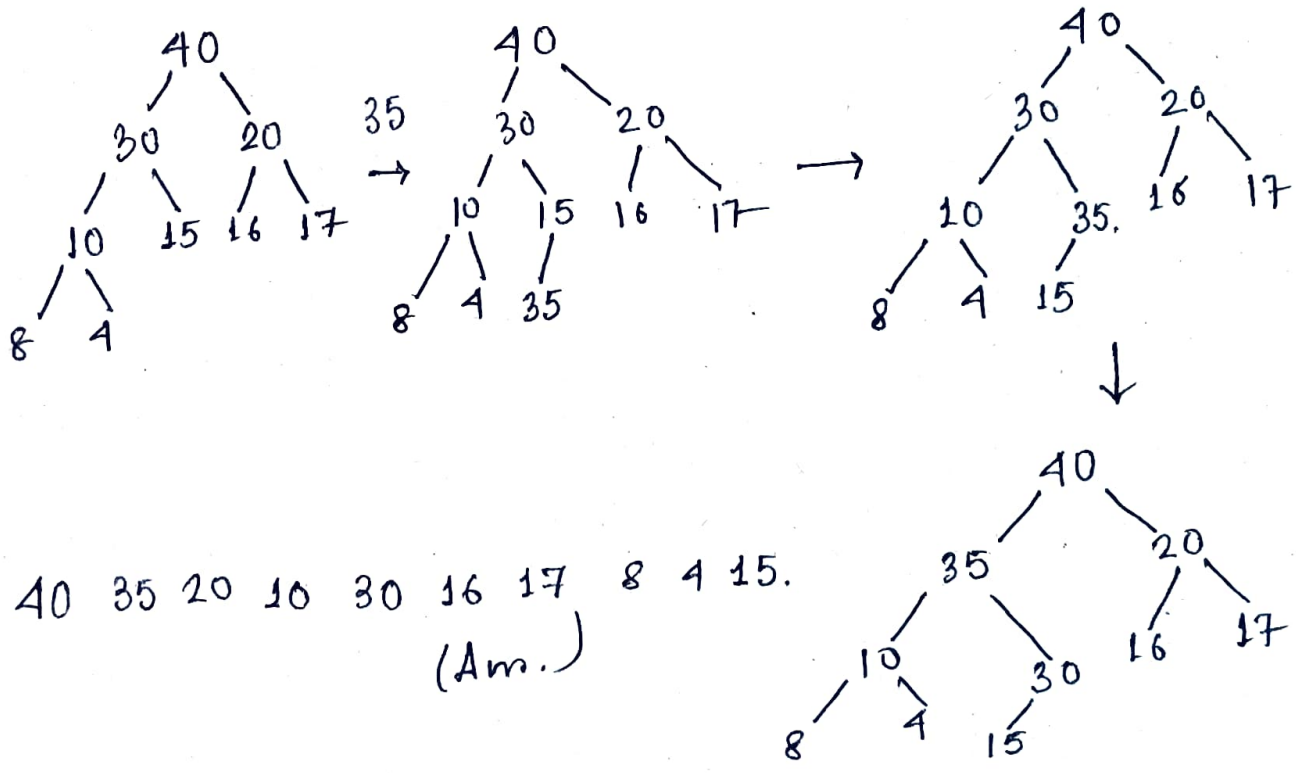


max_heapify (root)



Best time complexity for the worst case input.

Q. G'15 Consider a max heap represented by the array 40, 30, 20, 10, 15, 16, 17, 8, 4. Now consider that a value 35 is inserted into this heap. After insertion the new heap is —



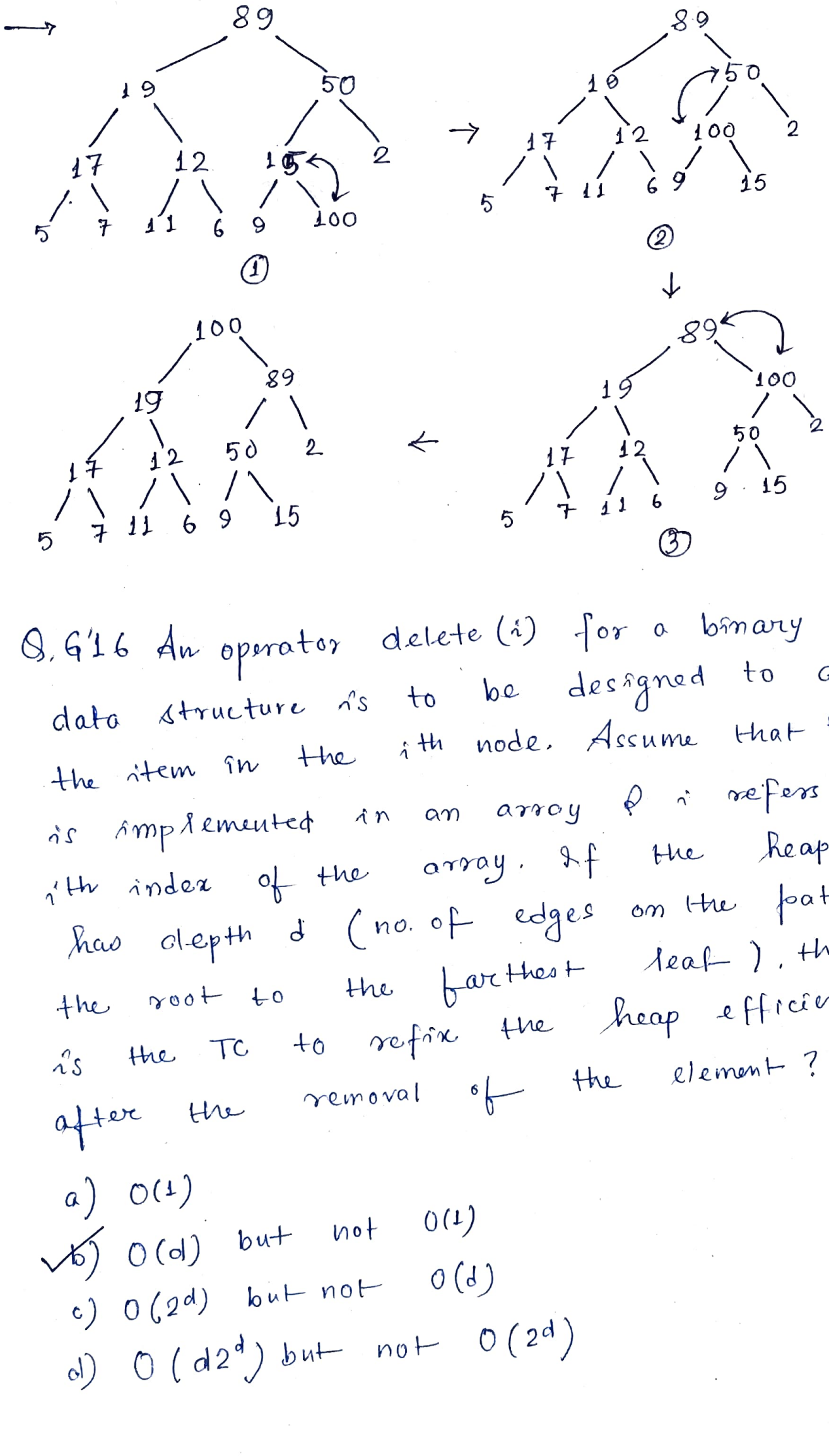
40 35 20 10 30 16 17 8 4 15.

(Am.)

Q.15 Consider the following array of elem

<89, 19, 50, 17, 12, 15, 2, 5, 7, 11, 6, 9, 100>

The min. no. of interchanges needed to convert it into a max heap is —



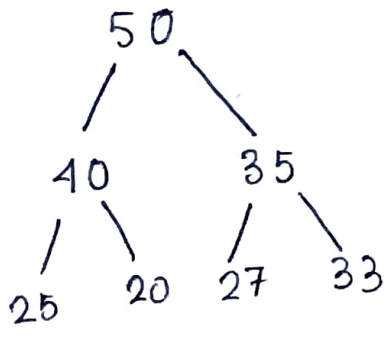
Q.16 An operator delete (i) for a binary heap data structure is to be designed to delete the item in the ith node. Assume that the heap is implemented in an array s.t. i refers to the ith index of the array. If the heap tree has depth d (no. of edges on the path from the root to the farthest leaf), then what is the TC to refix the heap efficiently after the removal of the element?

a) O(1)

✓b) O(d) but not O(1)

c) O(2^d) but not O(d)

d) O(d2^d) but not O(2^d)



d = 2 max_heapify()

In worst case root i.e. 50 will be deleted. So, max 2 swap needed.

For d, O(d) will be needed.

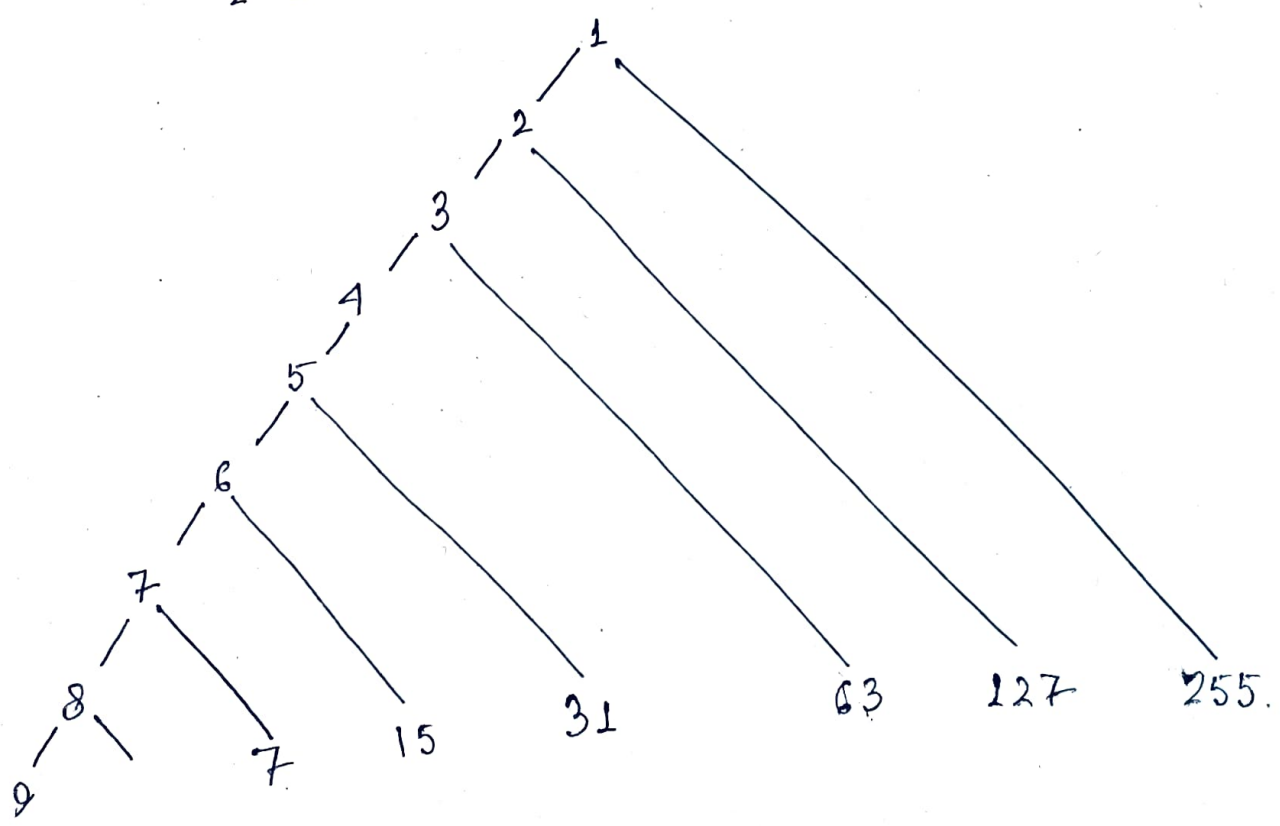
GO Q. G'16 A complete binary min-heap is made by including each integer in [1, 1023] exactly once. The depth of a node in the heap is the length of the path from the root of the heap to that node. Thus the root is at depth 0. The maximum depth at which integer 9 can appear is ___.

$\sum_{}^{K}$ No. of elems in depth K = 1023

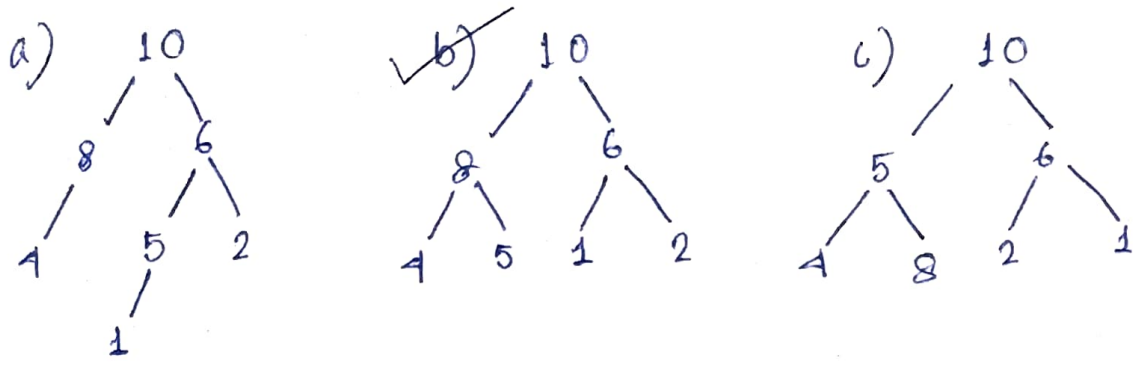
K = 0

$\Rightarrow 2^0 + 2^1 + 2^2 + \cdots + 2^K = 1023$

$\Rightarrow \frac{1\left(2^{K+1}-1\right)}{2-1} = 1023 \Rightarrow K = 9.$ Max ~~depth~~ levels



S, G'11 Which is a max heap?

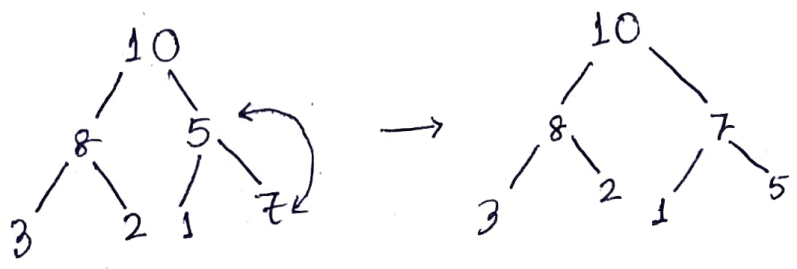


→ a) Not complete.

c) Not satisfying heap property.

S, G'14, 05 A priority queue is implemented as a max heap. Initially, it has 5 elems. The level order traversal of the heap is 10, 8, 5, 3, 2. Two new elems 1 and 7 are inserted into the heap in that order. The level order traversal of the heap after insertion —

→



10 8 7 3 2 1 5

* Bucket Sort.

Sorting large array of float values lying uniformly b/w the lower & upper bound.

Bucket sort runs in linear time on average.

Bucket sort considers that the i/p is generated by a random process that distributes elems uniformly over the interval $\mu = [0, 1]$

To sort n input numbers,

1. Partition μ into n non overlapping intervals called Buckets.
2. Put each input number into its bucket.
3. Sort each bucket using a simple algorithm (Insertion sort)
4. Concatenate the sorted lists.

Bucket sort considers that the input is an n element array A that each elem A[i] in the array satisfies $0 \leq A[i] < 1$. The code depends upon an auxiliary array B[0..n-1] of linked lists & considers that there is a mechanism for maintaining such lists.

```
Bucket_sort(A)
    n ← length(A)
    for i ← 1 to n
        do insert A[i] into list B[n*A[i]]
    for i ← 0 to n-1
        do sort list B[i] with insertion sort
    Concatenate the lists B[0], B[1], .., B[n-1]
                                    together in order.
```

index (size * array[i])

B[⌊nA[i]⌋]

B[n*A[i]] — index in bucket

TC - O(n+k) for best & average case

O(n²) for worst case

SC - O(nk) for worst case.

* Counting Sort.

Linear time sorting algorithm, that works faster by not making a comparison. It assumes that the numbers to be sorted os in range 1 to k where k is small. (also whole numbers only, non-negative)

Uses 3 arrays -

1. A[1...n] input array
2. B[1...n] sorted output
3. C[0...k] temporary working storage.

Counting-Sort (A, B, k)

```
for i ← 0 to k                    ] Θ(k)
    do C[i] ← 0
for j ← 1 to length[A]            ] Θ(n)
    do C[A[j]] ← C[A[j]] + 1
// C[i] now contains no. of elems equal to i
for i ← 1 to k                    ] Θ(k)
    do C[i] ← C[i] + C[i-1]
// C[i] now contains the no. of elems ≤ i
for j ← length[A] downto 1        ] Θ(n)   ---- maintain stability
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```

1.

	1	2	3	4	5	6	7	8
A	1	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

2.

	0	1	2	3	4	5
C	2	2	4	7	7	8

3.

	1	2	3	4	5	6	7	8
B							3	

	0	1	2	3	4	5
C	2	2	4	6	7	8

4.

	1	2	3	4	5	6	7	8
B		0					3	

	0	1	2	3	4	5
C	1	2	4	6	7	8

5.

	1	2	3	4	5	6	7	8
B		0				3	3	

	0	1	2	3	4	5
C	1	2	4	5	7	8

6.

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

Overall running time $\Theta(n+k)$

- We use counting sort When we have $k = O(n)$, in which case running time is $\Theta(n)$

Auxiliary space $O(n+k)$

Bucket Sort: $\{1.2, 0.22, 0.43, 0.36, 0.39, 0.27\}$

Scatter & gather

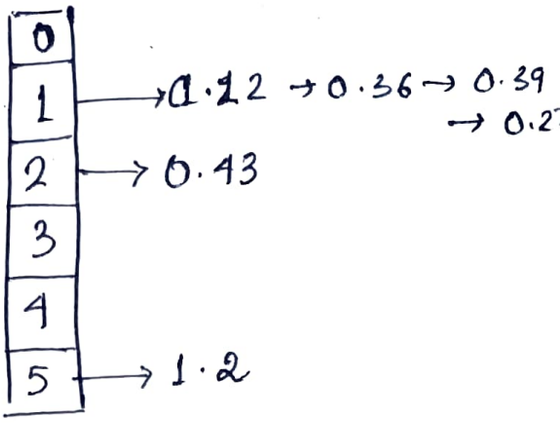
Optimal size of each bucket = $\frac{\text{largest elem}}{\text{array size}} = \frac{1.2}{6} = 0.2$

By dividing each elem with 0.2, we'll get an index for each elem's respective bucket.

$\frac{1.2}{0.2} = 6$ → -1 for largest elem only

$\frac{0.22}{0.2} = 1.1$

$\frac{0.43}{0.2} = 2.15$



Now, sort each non-empty bucket. Then concatenate.

→ Tc: Worst ⇒ If the collection has a short range it's common to have many elems in a single bucket, whereas lot of buckets are empty. Now, the complexity depends on the algo we use to sort the contents of bucket. If we're using insertion sort, it's $O(n^2)$

Best ⇒ Having all the elems already sorted. Additionally the elems are uniformly distributed i.e Each bucket would have the same #elems. Creating buckets takes $O(n)$ & insertion sort takes $O(k)$, k - range of numbers.

Average case ⇒ When the elems are distributed randomly in the array. $O(n)$

- Types of Sorting algos : i) Comparison sort
  ii) Integer sort (Determine for each elem x how many elems are $< x$. If there are 14 elems that are $< x$, then place x at the 15th place.)

- Selection sort in unstable as it may change order of elems with same value.

* Radix Sort

Radix_sort (A, d)
  for i ← 1 to d
    do use a stable sort to sort array A on digit i

Each key on A[1...n] is a d-digit integer.
Digits numbered 1 to d from right to left.

(counting sort) range known.

[Each elem in the n-elem array A has d digits, where digit 1 is the lowest order digit and d the highest]

eg.

329	720	720	329
457	355	329	355
657	436	436	436
839 →	457 →	839 →	457
436	657	355	657
720	329	457	720
355	839	657	839.

- Use counting sort as range is known.
  For decimal number system [0, 9]
  For binary n n [0, 1]

- Time complexity:

Let there be d digits in input integers. Radix sort takes $O(d(n+b))$ time where b is the base for representing numbers (b = 10 for decimal). If k is the max. value possible, then $d = O(\log_b(k))$. So, overall time complexity is $O((n+b)\log_b k)$.
Let $k \leq n^c$, c = constant. In that case,

complexity becomes $O(n \log_b(n))$. It still does not beat comparison based sorting algorithms.

If we have $\log_2 n$ bits for every digit, the running time of radix appears to be better than Quick sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for radix sort & quick sort uses H/W caches more effictively. Also, radix sort uses counting sort as a subroutine & counting sort takes extra space to sort numbers.

* Selection Sort.

(in-place algo.)

Minimum # swaps among the comparison based sorting algos (Cycle sort - best, min. swaps)

```
selection-sort (A)
  n ← length [A]
  for j ← 1 to n-1
      do smallest ← j
         for i ← j+1 to n
             do if A[i] < A[smallest]
                   then smallest ← i
         exchange A[j] ↔ A[smallest]
```

After the first n-1 elems, the subarray A[1... n-1] contains the smallest n-1 elems, sorted, & therefore element A[n] must be the largest elem.

findMinIndex (Arr[], start, end)

```
min_index = start
for i from (start + 1) to end :
    if Arr[i] < Arr[min_index] :
        min_index = i                    TC  O(n)
return min_index.                        SC  O(1)
```

SelectionSort (Arr[], arr_size)

```
for i from 1 to arr_size :
    min_index = findMinIndex (Arr, i, arr_size)
    if i != min_index :
        swap (Arr[i], Arr[min_index])
```

Time complexity : Suppose there are n elems in the array. At worst case there can be n iterations in findMinIndex() for start = 1 & end = n.

No auxiliary space used.

Total iterations =

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

$TC = O(n^2)$

$SC = O(1)$

$\begin{cases} \frac{n(n-1)}{2} & \text{comparisons} \\ n & \text{swaps} \end{cases}$

→ Good thing about selection sort is it never makes more than O(n) swaps & can be useful when memory write is a costly operation.

→ Bubble, insertion, quick.
non-adaptive: selection, merge, heap.

Adaptive: If the algo takes advantage of already sorted elements in the list that is to be sorted. That is, while sorting if the source list has some elems already sorted, adaptive algo will take into account & will try not to reorder them.

Inplace sort: Uses constant amt of extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of elems within the list. eg. Insertion, selection sort. While typical merge sort implementation, counting sort are not in-place.

↳ O(lgn) or O(n) space may be allowed for its operation. (wiki)

Stable sort: Any sorting algo that is not stable by nature, can be modified to be stable. Repeated elements appear in the same order as in the input. eg. Bubble, Insertion, Merge, Count sort etc.

Online sorting: Processes its input piece-by-piece in a serial fashion i.e. in the order that the input is fed to the algorithm, without having the entire input available from the beginning.

Selection, merge sort, bubble sort are offline as they require access to the entire input. Insertion sort considers one i/p elem per iteration & produces a partial sorted o/p w.r.t considering future elems. Thus, insertion sort is online.

Sorting Techniques.

1. Insertion Sort

Best: O(n) [Sorted array] Worst: O(n^2) [Reverse sorted array] Space: O(1)

```
void insertionSort(int arr[], int n) {
   int i, key, j;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      /* Move elements of arr[0..i-1], that are
        greater than key, to one position ahead
        of their current position */
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      }
      arr[j + 1] = key;
   }
}
```

Insertion sort is used when the number of elements is small. It can also be useful when the input array is almost sorted, only a few elements are misplaced in a complete big array.

-Binary Insertion Sort
We can use binary search to reduce the number of comparisons in normal insertion sort. Binary Insertion Sort uses binary search to find the proper location to insert the selected item at each iteration. In normal insertion, sorting takes O(i) (at ith iteration) in the worst case. We can reduce it to O(logi) by using binary search. The algorithm, as a whole, still has a running worst case running time of O(n2) because of the series of swaps required for each insertion. Refer this for implementation.

2. Merge Sort

TC: 2T(n/2) + O(n) = Θ(nlogn) SC: [Stack O(logn)] + [For merge procedure O(n)] = O(n)

```
void merge(int arr[], int l, int m, int r) {
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;
   /* create temp arrays */
   int L[n1], R[n2];
   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];
   /* Merge the temp arrays back into arr[l..r]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      }
      else {
         arr[k] = R[j];
         j++;
      }
      k++;
   }
   /* Copy the remaining elements of L[], if there
     are any */
   while (i < n1) {
      arr[k] = L[i];
      i++;
      k++;
   }
   /* Copy the remaining elements of R[], if there
     are any */
   while (j < n2) {
      arr[k] = R[j];
      j++;
      k++;
   }
}
```

```
void mergeSort(int arr[], int l, int r)
{
   if (l < r) {
      // Same as (l+r)/2, but avoids overflow for
      // large l and h
      int m = l + (r - l) / 2;
      // Sort first and second halves
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);
      merge(arr, l, m, r);
   }
}
```

Merging 2 sorted lists of size m, n: O(m+n)

Time complexity of Merge Sort is Θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Applications of Merge Sort

-Merge Sort is useful for sorting linked lists in O(nLogn) time.In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
-Inversion Count Problem
-Used in External Sorting

→ Sorted or Reverse sorted array

3. Quick Sort TC: Best Θ(nlogn) Partitions are equal in size T(n) = 2T(n/2) + O(n) Same as Merge Sort
Worst: $O(n^2)$ All elems < pivot or All elems > pivot Divides into n-1 & 0 T(n) = T(n-1) + O(n)
SC: #stack entries Worst O(n) Divides into n-1 and 0 Best: O(logn) Balanced partitioning

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high];    // pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

```
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

T(n) = T(k) + T(n-k-1) + Θ(n) The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

Sorted input (or reverse sorted): T(n) = T(n-1) + O(n) = $O(n^2)$
All elems equal in input: T(n) = T(n-1) + O(n) = $O(n^2)$
Splitting in ratio 1:9 O(nlogn)
Alternating best and worst cases: 0,n-1 → (n-1)/2, (n-1)/2
Θ(nlogn)

→ Why QuickSort is preferred over MergeSort for sorting Arrays

Quicksort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and deallocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both types have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.

Most practical implementations of Quick Sort use randomized versions. The randomized version has expected time complexity of O(nLogn). The worst case is possible in the randomized version also, but the worst case doesn't occur for a particular pattern (like sorted array) and randomized Quicksort works well in practice. Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays. Quick Sort is also tail recursive, therefore tail call optimizations are done.

→ Why MergeSort is preferred over QuickSort for Linked Lists?

In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike arrays, in linked lists, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.

In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked lists. Quick Sort requires a lot of this kind of access. In the linked list to access i'th index, we have to travel each and every node from the head to the i'th node as we don't have a continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

→ QuickSort Tail Call Optimization (Reducing worst case space to Log n)

```
/* This QuickSort requires O(Log n) auxiliary space in
   worst case. */
void quickSort(int arr[], int low, int high)
{
    while (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        int pi = partition(arr, low, high);

        // If left part is smaller, then recur for left
        // part and handle right part iteratively
        if (pi - low < high - pi) {
            quickSort(arr, low, pi - 1);
            low = pi + 1;
        }
        // Else recur for right part
        else
        {
            quickSort(arr, pi + 1, high);
            high = pi - 1;
        }
    }
}
```

If the left part becomes smaller, then we make a recursive call for the left part. Else for the right part. In the worst case (for space), when both parts are of equal sizes in all recursive calls, we use O(Log n) extra space.

→ Randomised partition function

```
int rand_partition ( int A[ ] , int start , int end ) {
   //chooses position of pivot randomly by using rand() function .
    int random = start + rand( )%(end-start +1 ) ;
    swap ( A[random] , A[start]) ;       //swap pivot with 1st element.
    return partition(A,start ,end) ;     //call the above partition function
}
```

4. **Heap Sort** TC: O(nlogn) heapify has complexity O(logn), build_maxheap has complexity O(n) and we run heapify n-1 times in heap_sort function

SC: O(1) Arrange the array as heap

```
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

```
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

5. **Bubble Sort** TC: Worst [Reverse sorted] $O(n^2)$ n-1 + n-2 + n-3 + ... + 1 = $O(n^2)$ SC: O(1)

Best [Sorted] O(n) Run one iteration only

```
void bubbleSort(int arr[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)
     // Last i elements are already in place
     for (j = 0; j < n-i-1; j++)
        if (arr[j] > arr[j+1])
           swap(&arr[j], &arr[j+1]);
}
```

6. **Selection Sort** TC: $O(n^2)$ always n(n-1)/2 comparisons + n swaps at max SC: O(1)

```
void selectionSort(int arr[], int n)
{
   int i, j, min_idx;
   // One by one move boundary of unsorted subarray
   for (i = 0; i < n-1; i++)
   {
      // Find the minimum element in unsorted array
      min_idx = i;
      for (j = i+1; j < n; j++)
        if (arr[j] < arr[min_idx])
          min_idx = j;
      // Swap the found minimum element with the first element
      swap(&arr[min_idx], &arr[i]);
   }
}
```

Bucket Sort: Put each elem of the i/p array into its individual bucket n * A[i]; then sort individual bucket using insertion sort. At last concatenate all buckets.

7. **Bucket Sort** TC: Best [Uniform distribution] O(n+k) Worst $O(n^2)$ Average O(n) SC: O(nk)
k: range of numbers in a bucket

```
void bucketSort(float arr[], int n)
{
  // 1) Create n empty buckets
  vector<float> b[n];

  // 2) Put array elements in different buckets
  for (int i = 0; i < n; i++) {
    int bi = n * arr[i]; // Index in bucket
    b[bi].push_back(arr[i]);
  }

  // 3) Sort individual buckets
  for (int i = 0; i < n; i++)
    sort(b[i].begin(), b[i].end());

  // 4) Concatenate all buckets into arr[]
  int index = 0;
  for (int i = 0; i < n; i++)
    for (int j = 0; j < b[i].size(); j++)
      arr[index++] = b[i][j];
}
```

When all elems belong to same bucket.

8. **Counting Sort** Counting sort takes O(n + k) time and O(n + k) space, where n is the number of items we're sorting and k is the number of possible values. When k = O(n), TC = O(n).

```
void countSort(char arr[])
{
  // The output character array that will have sorted array
  char output[strlen(arr)];
  // Create a count array to store count of individual
  // characters and initialize count array as 0
  int count[RANGE + 1], i;
  memset(count, 0, sizeof(count));
  // Store count of each character
  for (i = 0; arr[i]; ++i)                          ] O(n)
    ++count[arr[i]];

  // Change count[i] so that count[i] now contains actual
  // position of this character in output array
  for (i = 1; i <= RANGE; ++i)                      ] O(K)
    count[i] += count[i - 1];
  // Build the output character array
  for (i = 0; arr[i]; ++i) {                        ] O(n)
    output[count[arr[i]] - 1] = arr[i];
    --count[arr[i]];
  }

  /*                                                TC O(n+K)
   For Stable algorithm
   for (i = sizeof(arr)-1; i>=0; --i)
   {
     output[count[arr[i]]-1] = arr[i];
     --count[arr[i]];
   }
   */

  // Copy the output array to arr, so that arr now
  // contains sorted characters
  for (i = 0; arr[i]; ++i)
    arr[i] = output[i];
}
```

Space: Count O(K)
Output O(n)
O(n+k)

In many cases, K = O(n) i.e. # items to be sorted is not asymptotically different than the # values those items can take on. Because of this, counting sort is often said to be O(n) time & space.

CountSort: Assumes that each of n elems has a key value ranging from 0 to K. For each elem, determine the no. of elems that are less than it. Then using this info place the elem in the correct place of sorted array.

→ In many implementations, d = 64 (bits). , K also constant (2 - binary). With these assumptions, Radix sort is O(n) time & space.

9. Radix Sort TC: Best, Average, Worst O(nk) n is the number of input data and k is the maximum element in the input data
SC: O(n+k)

```
void radixsort(int arr[], int n)
{
   // Find the maximum number to know number of digits
   int m = getMax(arr, n);

   // Do counting sort for every digit. Note that instead
   // of passing digit number, exp is passed. exp is 10^i
   // where i is current digit number
   for (int exp = 1; m / exp > 0; exp *= 10)
      countSort(arr, n, exp);
}
```

TC O(d(n+k)) d-#digits | K-base
as we call counting sort for each of the d digits in the input numbers (countsort O(n+K))

SC O(n+k) ao for counting sort

```
// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
   int output[n]; // output array
   int i, count[10] = { 0 };
   // Store count of occurrences in count[]
   for (i = 0; i < n; i++)
      count[(arr[i] / exp) % 10]++;
   // Change count[i] so that count[i] now contains actual
   //  position of this digit in output[]
   for (i = 1; i < 10; i++)
      count[i] += count[i - 1];
   // Build the output array
   for (i = n - 1; i >= 0; i--) {
      output[count[(arr[i] / exp) % 10] - 1] = arr[i];
      count[(arr[i] / exp) % 10]--;
   }
   // Copy the output array to arr[], so that arr[] now
   // contains sorted numbers according to current digit
   for (i = 0; i < n; i++)
      arr[i] = output[i];
}
```

Radix sort: For each significant place (digit) sort them using count sort.

4. COMPARISON OF VARIOUS SORTING ALGORITHMS

In this section, compare sorting algorithms according to their complexity, method used by them and also mention their advantages and disadvantages. In the following table, n represent the number of elements to be sorted. The column Average and worst case give the time complexity in each case. These all are comparison sort.

Table 1
Comparison of Comparison Based Sort

Name	*Average Case*	*Worst Case*	*Method*	*Advantage/Disadvantage*
Bubble Sort	$O(n^2)$	$O(n^2)$	Exchange	1. Straightforward, simple and slow. 2. Stable. 3. Inefficient on large tables.
Insertion Sort	$O(n^2)$	$O(n^2)$	Insertion	1. Efficient for small list and mostly sorted list. 2. Sort big array slowly. 3. Save memory
Selection Sort	$O(n^2)$	$O(n^2)$	Selection	1. Improves the performance of bubble sort and also slow. 2. Unstable but can be implemented as a stable sort. 3. Quite slow for large amount of data.
Heap Sort	$O(n \log n)$	$O(n \log n)$	Selection	1. More efficient version of selection sort. 2. No need extra buffer. 3. Its does not require recursion. 4. Slower than Quick and Merge sorts.
Merge Sort	$O(n \log n)$	$O(n \log n)$	Merge	1. Well for very large list, stable sort. 2. A fast recursive sorting. 3. Both useful for internal and external sorting.

Name	*Average Case*	*Worst Case*	*Method*	*Advantage / Disadvantage*
				4. It requires an auxiliary array that is as large as the original array to be sorted.
In place-merge Sort	O(n log n)	O(n log n)	Merge	1. Unstable sort. 2. Slower than heap sort. 3. Needs only a constant amount of extra space in addition to that needed to store keys.
Shell Sort	O(n log n)	O(n log$^2 n$)	Insertion	1. Efficient for large list. 2. It requires relative small amount of memory, extension of insertion sort. 3. Fastest algorithm for small list of elements. 4. More constraints, not stable.
Cocktail Sort	O(n^2)	O(n^2)	Exchange	1. Stable sort. 2. Variation of bubble sort. 3. Bidirectional bubble sort.
Quick Sort	O(n log n)	O(n^2)	Partition	1. Fastest sorting algorithm in practice but sometime Unbalanced partition can lead to very slow sort. 2. Available in many slandered libraries. 3. O (log n) space usage. 4. Unstable sort and complex for choosing a good pivot element.
Library sort	O(n log n)	O(n^2)	Insertion	1. Stable 2. It requires extra space for its gaps.
Gnome sort	O(n^2)	O(n^2)	Exchange	1. Stable 2. Tiny code size. 3. Similar to insertion sort.

The following table described sorting algorithm which are not comparison sort. Complexities below are in terms of n, the number of item to be sorted, k the size of each key and s is the chunk size use by implementation. Some of them are based on the assumption that the key size is large enough that all entries have unique key values, and hence that $n << 2^k$.

Comparison of Non-Comparison Sort

Name	*Average Case*	*Worst Case*	$n << 2^K$	*Advantage / disadvantage*
Bucket Sort	$O(n.k)$	$O(n^2.k)$	No	1. Stable, fast. 2. Valid only in range o to some maximum value M. 3. Used in special cases when thekey can be used to calculate the address of buckets.
Counting Sort	$O(n + 2^k)$	$O(n + 2^k)$	Yes	1. Stable, used for repeated value. 2. Often used as a subroutine in radix sort. 3. Valid in the rang (o k]
Radix Sort	$O(n.\frac{k}{s})$	$O(n.\frac{k}{s})$	No	Where k is some integer. 1. Stable, straight forward code. 2. Used by the card-sorting machines. 3. Used to sort records that are keyed by multiple fields like date (Year, month and day).

Contd.

Name	*Average Case*	*Worst Case*	$n << 2^K$	*Advantage / disadvantage*
MSD Radix Sort	$O(n.\frac{k}{s})$	$O(n.\frac{k}{s})$	No	1. Enhance the radix sorting methods, unstable sorting. 2. To sort large computer files very efficiently without the risk of overflowing allocated storage space. 3. Bad worst-case performance due to fragmentation of the data into many small sub lists. 4. It inspects only the significant characters.
LSD Radix Sort	$O(n.\frac{k}{s})$	$O(n.\frac{k}{s}.2s)$	No	1. It inspects a complete horizontal strip at a time. 2. It inspects all characters of the input. 3. Stable sorting method.

5. PROBLEM DEFINITION AND SORTING METHODS

All the sorting algorithms are problem specific. Each sorting algorithms work well on specific kind of problems. In this section we described some problems and analyses that which sorting algorithm is more suitable for that problem.

Table 3
Sorting Algorithms According to Problem

Problem Definition	Sorting Algorithms
The data to be sorted is small enough to fit into a processor's main memory and can be randomly accessed i.e. no extra space is required to sort the records (Internal Sorting).	Insertion Sort, Selection Sort, Bubble Sort
Source data and final result are both sorted in hard disks (too large to fit into main memory), data is brought into memory for processing a portion at a time and can be accessed sequentially only (External Sorting).	Merge Sort (business application, database application)
The input elements are uniformly distributed within the range [0, 1).	Bucket Sort
Constant alphabet (ordered alphabet of constant size, multiset of characters can be stored in linear time), sort records that are of multiple fields.	Radix Sort
The input elements are small.	Insertion Sort
The input elements are too large.	Merge Sort, Shell Sort, Quick Sort, Heap Sort
The data available in the input list are repeated more times i.e. occurring of one value more times in the list.	Counting Sort
The input elements are sorted according to address (Address Computation).	Proxmap Sort, Bucket Sort
The input elements are repeated in the list and sorted the list in order to maintain the relative order of record with equal keys.	Bubble Sort, Merge Sort, Counting Sort, Insertion Sort
The input elements are repeated in the list and sorted the list so that their relative order are not maintain with equal keys.	Quick Sort, Heap Sort, Selection Sort, Shell Sort
A sequence of values, $a_0, a_1, \ldots, a_{n-1}$, such that there exists an i, $0 \le i \le n-1$, a_0 to a_i is monotonically increasing and a_i to a_{n-1} is monotonically decreasing. (sorting network)	Biotonic-merge Sort

Sorting Algorithms	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	Ω(N)	θ(N^2)	O(N^2)	O(1)
Selection Sort	Ω(N^2)	θ(N^2)	O(N^2)	O(1)
Insertion Sort	Ω(N)	θ(N^2)	O(N^2)	O(1)
Quick Sort	Ω(N log N)	θ(N log N)	O(N^2)	O(N)
Merge Sort	Ω(N log N)	θ(N log N)	O(N log N)	O(N)
Heap Sort	Ω(N log N)	θ(N log N)	O(N log N)	O(1)

Sorting Algorithms	Special Input Condition	Time Complexity			Space Complexity
		Best Case	Average Case	Worst Case	Worst Case
Counting Sort	Each input element is an integer in the range 0- K	Ω(N + K)	θ(N + K)	O(N + K)	O(K)
Radix Sort	Given n digit number in which each digit can take on up to K possible values	Ω(NK)	θ(NK)	O(NK)	O(N + K)
Bucket Sort	Input is generated by the random process that distributes elements uniformly and independently over the interval [0, 1)	Ω(N + K)	θ(N + K)	O(N^2)	O(N)

If stability is important and space is available, the merge sort might be the best choice for the implementation.
Like insertion sort, quick sort has tight code and some hidden constant factor in its running time is small.
When the input array is almost sorted or input size is small, then the insertion sort can be preferred.
We can prefer merge sort for sorting a linked list.

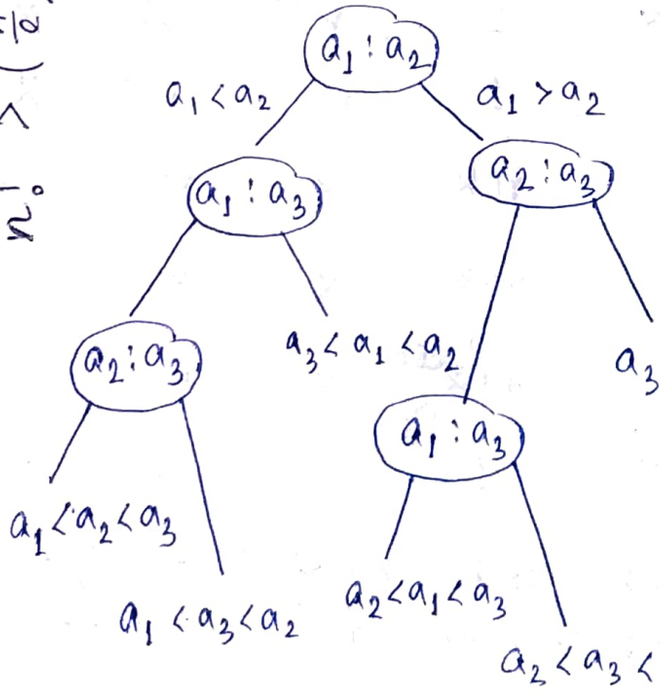
→ Where to use which sorting algorithm ?

- Insertion sort is preferable when we have an almost sorted array. Insertion sort best case $O(n)$ for sorted array.
- Merge sort : When the whole array does not fit into the main memory (use sort merge strategy)
- Quick sort : General purpose sorting algo, use randomised version to prevent the worst case ($O(n^2)$)
- Counting & Radix Sort : When range is known (0 to k). → can give linear time sorting.
- Bubble sort : Should be avoided.

→ Lower bound for comparison based sorting :

$\langle a_1, a_2, a_3 \rangle$ 3! permutations available

Decision tree — (internal node - decision, leaf nodes - result of computation)



Height of tree = h

$\Rightarrow$ # comparisons = h

A tree with height h has at max 2^h levels.

Let l be the # leaves in the tree.

$$n! \leq l \leq 2^h$$

$$h \geq \lg(n!) = \Omega(n \lg n)$$

$n! > (n/e)^n$ Stirling's approx

comparisons ≡ height of decision tree.

- $\log(n!) = \log(1) + \log(2) + \cdots + \log(n)$

$\leq \log(n) + \log(n) + \cdots + \log(n)$

$= n \log n$

Lower bound:

$\log(1) + \cdots + \log(n/2) + \cdots + \log(n)$

$\geq \log(n/2) + \cdots + \log(n) = \log(n/2) + \log(n/2 + 1) + \cdots + \log(n)$

$\geq \log(n/2) + \cdots + \log(n/2)$

$= n/2 \log(n/2) = \frac{n}{2} \log n - \frac{n}{2} \log 2.$

$\Rightarrow \log(n!) = \Omega(n \log n).$

- Summary for sorting algo.

	Time Best	Avg	Worst	Adaptive	Memory/ space	Inplace	Stable	Online
Quick	$n \lg n$	$n \lg n$	n^2	✓	$\lg n$	✓	X	X
Insertion	n	n^2	n^2	✓	1	✓	✓	✓
Merge	$n \lg n$	$n \lg n$	$n \lg n$	X	n	X	✓	X
Heap	$n \lg n$	$n \lg n$	$n \lg n$	X	1	✓	X	X
Bubble	n	n^2	n^2	✓	1	✓	✓	X
Bucket	$n + r^{\phi}$	n	n^2		nk	X	✓	
Counting	$n + r^{\Delta}$	$n + r$	$n + r$		$n + r$ $\Theta(n)$	X	✓	
Radix	nr^*	nr	nr		$n + 2^r$ $\Theta(n)$	X	✓	
Selection	n^2	n^2	n^2	X	1	✓	X	X

Δ max possible elem / size of count array

$*$ max #digits.

ϕ r = #buckets

Searching

* Sequential / Linear Search.

```
int seqSearch (int a[], int n, int key) {
    int i;
    for (i = 0; i < n ; i++)
        if (a[i] == key) break;
    if (i == n) return NOTFOUND;
    return i;
}
```

Best case : key = a[0]

Worst case : key = a[n-1]
No match

Recursive version –

```
int seqSearch (int a[], int n, int key) {
    int temp;
    if (a[0] == key) return 0;
    if (n == 1) return NOTFOUND;
    temp = seqSearch (a+1, n-1, key);
    if (temp == NOTFOUND) return NOTFOUND;
    return temp + 1;
}
```

TC = O(n)

Space usage : No. of stack frames (activ^n records) used by the recursive f^n is $\propto n$ in the worst case. The iterative f^n uses constant amount of extra space.

* Binary Search — Works only for sorted array.

Recursive version :

BinSrch (a, i, l, x)

```
// given an array a[i:l] of elems in non-
// decreasing order, 1 ≤ i ≤ l, determine whether
// x is present & if so, return j such that
// x = a[j], else return 0.
{
    if (l = i) then
    {
        if (x = a[i]) then return i;
        else return 0;
    }
    else
    { // reduce into a smaller subproblem
        mid := ⌊(i+l)/2⌋ ;
        if (x = a[mid]) then return mid;
        else if (x < a[mid]) then
            return BinSrch (a, i, mid-1, x);
        else return BinSrch (a, mid+1, l, x);
    }
}
```

Iterative version :

```
BinSrch (a, n, x)
// array a[1:n] non-decreasing order, n ≥ 0.
{
    low := 1 ;
    high := n;
```

```
while ( low ≤ high) do
{
    mid := ⌊(low + high) / 2⌋ ;
    if (x < a[mid] then high := mid - 1 ;
    else if (x > a[mid] ) then low := mid + 1 ;
    else return mid ;
}
return 0 ;
}
```

TC = $O(\log n)$

for n elems,

$$T_n = \begin{cases} c_0 & \text{if } n = 1 \\ T_{n/2} + c_1 & \text{if } n > 1. \end{cases}$$

Using iterative method,

$T_{n/2} = T_{n/2^2} + c_1$ | $T(n) = T(n/2) + \Theta(1)$

$T_{n/2^2} = T_{n/2^3} + c_1$

$\vdots$

$T_1 = c_0$

$$T_n = c_0 + c_1 \log_2 n = O(\log_2 n)$$

```
Iter_Bin_Srch (A, v, low, high)
    while low ≤ high
        do mid ← ⌊(low+high)/2⌋
        if v = A[mid] then
            return mid
        if v > A[mid] then
            low ← mid + 1
        else high ← mid - 1
    return NIL
```

```
Recur_Bin_Srch ( A, v, low, high)
    if low > high
        then return NIL
    mid ← ⌊(low + high) / 2⌋
    if v = A[mid] then
        return mid
    if v > A[mid] then
        return Recur_Bin_Srch ( A, v, mid+1, high
    else return Recur_Bin_Srch (A, v, low, mid-1)
```

Divide & Conquer. (Binary search, Min-Max, Merge/Quick sort, Strassens matrix multiplication)

- **Min Max.** (Recursive) $T(n) = O(n)$

1	2	3	4	5
5	20	3	9	10

l = 1, h = 5

$mid = \frac{1+5}{2} = 3$

```
MinMax (l, h, max, min) {
  if (l == h)        // 1 elem
    max = min = a[l]
  else if (l = h-1) // 2 elems
  {
    if (a[l] > a[h]) {
        max = a[l]
        min = a[h]
    }
    else {
        max = a[h]
        min = a[l]
    }
  }
  else {
    mid = (l+h)/2 ; // divide
    MinMax (l, mid, max, min)
    MinMax (mid+1, h, max1, min1) // conquer
    if (max < max1) then  max = max1
    if (min > min1) then  min = min1
  }
}
```

5	20	3
1	2	3

m = 2

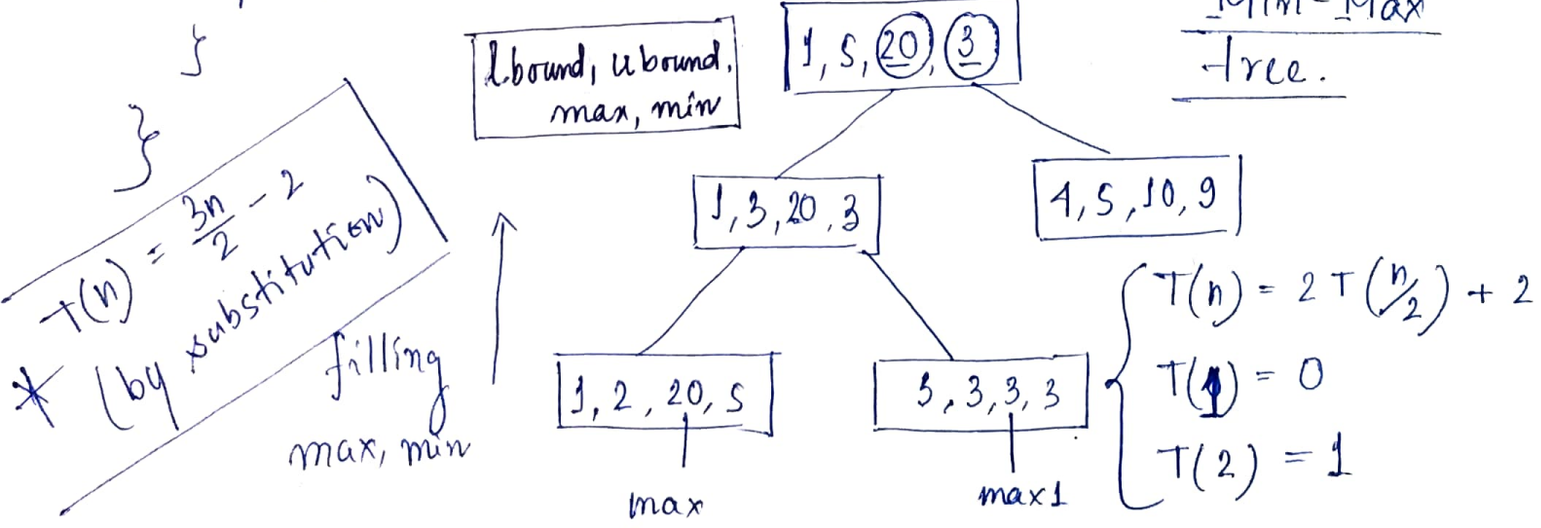
9	10
4	5

m = 4

5	20		3		9	10
min = 5		min = 3		min = 9		
max = 20		max = 3		max = 1		

Base cases:

1. 1 elem. (l = h)
   min = max = elem
2. 2 elems (l = h-1)
   single comparison to find min, max



Min-Max tree.

Strassen's Matrix Multiplication

$$c_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

gfg

(size in powers of 2).

$$A = \left(\begin{array}{cc:cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hdashline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right) \quad B = \left(\begin{array}{cc:cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hdashline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right)$$

A_{11} A_{12} A_{21} A_{22} B_{11} B_{12} B_{21} B_{22}

$$p = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$q = (A_{21} + A_{22}) B_{11}$$

$$r = A_{11} (B_{12} - B_{22})$$

$$s = A_{22} (B_{21} - B_{11})$$

$$t = (A_{11} + A_{12}) B_{22}$$

$$u = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$v = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$c_{11} = p + s - t + v$$

$$c_{12} = r + t$$

$$c_{21} = q + s$$

$$c_{22} = p + r - q + u.$$

7 multiplications instead of typical 8

$$T(n) = \begin{cases} 7T(n/2) + n^2 & n > 2 \\ 1 & n \leq 2 \end{cases}$$

$$T(n) = O(n^{\log_2 7})$$

$$= O(n^{2.81})$$

Greedy Algorithms

"The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, & captures the essence of the evolutionary spirit."

— Gordon Gekko

---

* Fundamental types of algorithms.

1. Recursive algorithms
2. Dynamic programming
3. Backtracking algorithm
4. Divide and Conquer algorithm
5. Greedy algorithm
6. Brute force algorithm
7. Randomized algorithm.
8. Branch & bound algorithm

* Greedy algorithm.

It makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

Greedy is a strategy that works well on optimization problems with the characteristics:

1. Greedy-choice property.

A global optimum can be arrived at by selecting a local optimum.

2. Optimal substructure

An optimal solution to the problem contains an optimal solution to the subproblems.

→ Greedy algorithms are a commonly used paradigm for combinatorial algorithms. Combinatorial problems intuitively are those for which feasible solutions are subsets of a finite set (typically from items of input). Therefore, in principle, these problems can always be solved optimally in exponential time by examining each of those feasible solutions. The goal of a greedy algorithm is find the optimal by searching only a tiny fraction.

→ A greedy algorithm makes greedy choices at each step to ensure that the objective function is optimised. The greedy algorithm has only one shot to compute the optimal solution so that it never goes back

and reverses the decision.

→ It's quite easy to come up with a greedy algorithm.

→ Analyzing the run time for greedy algorithms will generally be much easier than for other techniques.

→ The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues. Proving that a greedy algorithm is correct is more of an art than a science!

*→ Most greedy algorithms are not correct.

* Knapsack Problem.

Given a knapsack with limited weight capacity, few items having some weight and value.

Problem states which items should be placed into the knapsack such that the value or profit obtained by putting the items into the knapsack is maximum and also the weight limit of the knapsack does not exceed.

Knapsack problem has the following variants ~

1. Fractional knapsack problem
2. 0/1 knapsack problem.

Fractional Knapsack problem.

Items are divisible here. We can put fraction of any item into the ks if taking the complete item is not possible. It's solved using greedy method.

Given n objects and a ks. Object i has a weight w_i and the ks has capacity m. If a fraction x_i, $0 \le x_i \le 1$, of object i is placed into the ks, then a profit of $p_i x_i$ is earned. Objective is

$$\text{maximise} \sum_{1 \le i \le n} p_i x_i$$

$$\text{Subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and} \quad 0 \le x_i \le 1, \quad 1 \le i \le n$$

Profits and weights are positive numbers.

eg. $n = 3$, $m = 20$, $(p_1, p_2, p_3) = (25, 24, 15)$,
$(w_1, w_2, w_3) = (18, 15, 10)$

Greedy about		(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
profit	1.	$(1, 2/15, 0)$	20	28.2
$\frac{\text{profit}}{\text{weight}}$	2.	$(0, 1, 1/2)$	20	31.5 maximum among all

$(1.4, 1.6, 1.5)$

Greedy-Knapsack_frac

```
S1  O(n)   for i = 1 to n
           {
               compute p_i / w_i
           }

S2  O(nlogn) — Sort objects in non-increasing order of p/w
   merge sort

           for i = 1 to n  from sorted list
           {
S3  O(n)       if (m > 0 && w_i <= m)
               {
                   m = m - w_i
                   p = p + p_i
               }
               else
                   break;
           }

S4  O(1)   if (m > 0)
               p = p + p_i (m / w_i)   // the obj in S3 part
```

if any whole object can't be put in the KS (as $w_i > m$), then there must be only one object that can be put in the KS in fraction. The index of that object is the object that fails at last.

Time complexity = $O(n \log n)$.

→ Building heap based on p/w

⇒ will also take $O(n \log n)$

→ Applications ~

Cognitive radio networks

Resource management.

* Huffman Coding Algorithm.

Huffman coding implements a rule known as prefix rule to prevent the ambiguities while decoding. It ensures that the code assigned to any character is not a prefix of the code assigned to any other character.

Huffman coding uses variable length encoding. It assigns variable length code to all the characters. The code length of a character depends on how frequently it occurs in the given text. The character which occurs most frequently gets the smallest code.

Steps in huffman coding ~

1. Building a huffman tree from the Anput characters.
2. Assigning code to the characters by traversing the huffman tree.

Huffman tree.

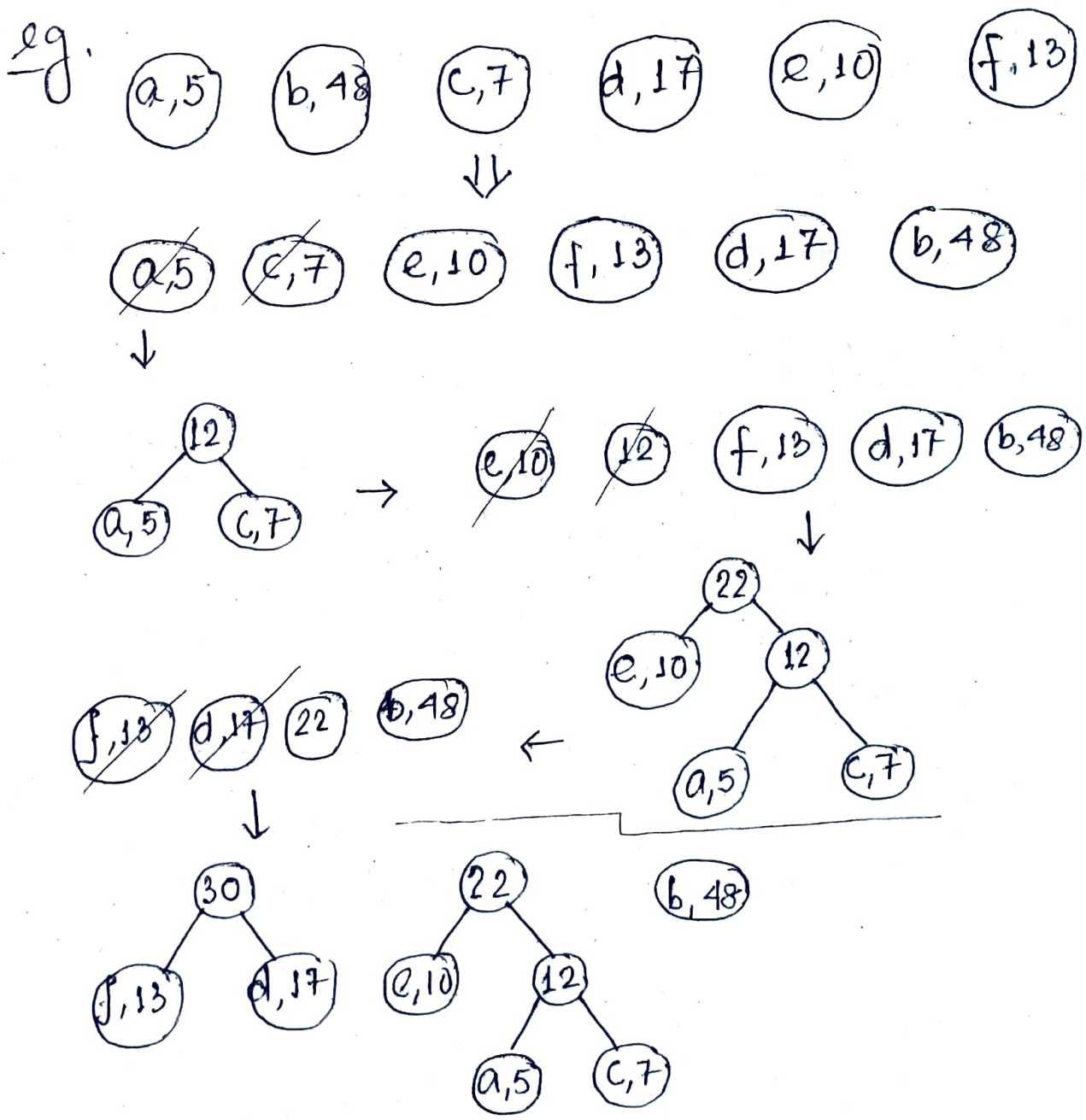
1. Create a leaf node for each character of the text. Leaf node of a character contains the occuring frequencies of character.
2. Arrange all the nodes in increasing order of their frequency value.

3. Considering the first 2 nodes having minimum frequency,

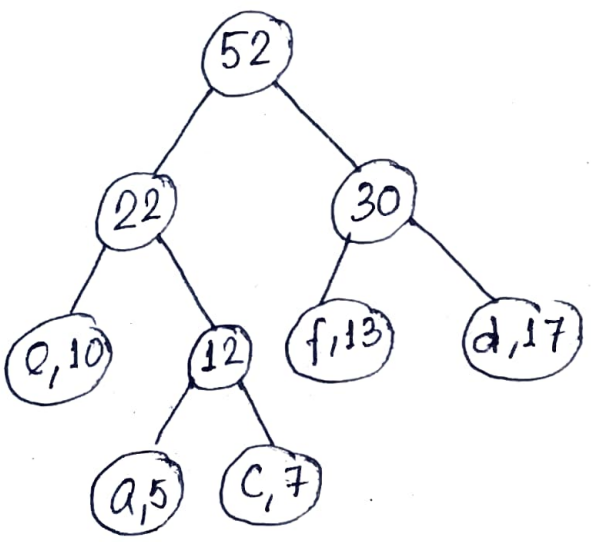
- create a new internal node
- the frequency of this new node is the sum of frequency of those two nodes
- make the first node as a left child & the other node as a right child of the newly created node.

4. Keep repeating 2 and 3 until all the nodes form a single tree.

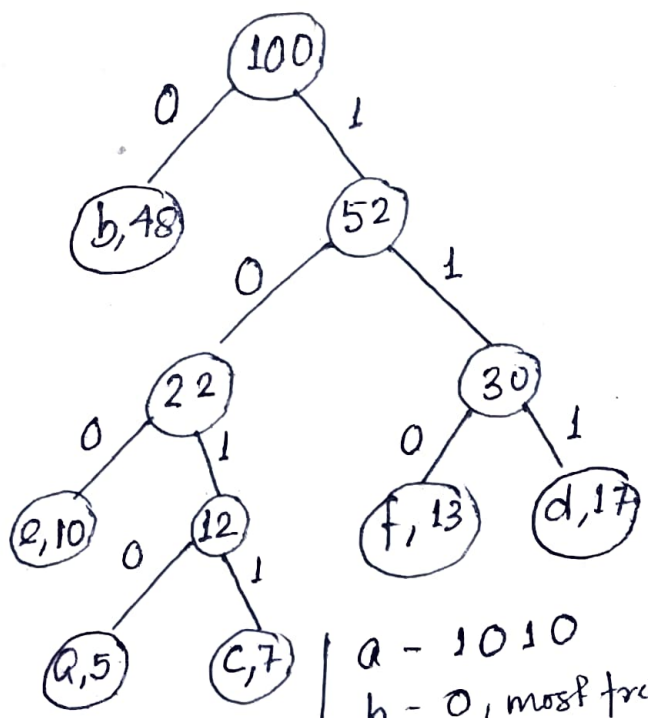
eg.



→ (22) (30) (b,48) → (b,48) (52)



→



a - 1010
b - 0, most freq
c - 1011
d - 111
e - 100
f - 110

Codes from the tree.—

1. 0 for Left child, 1 for right child
2. OR, 0 for right child, 1 for left

Always follow single convention in a tree, also in the decode tree.

Huffman (C)

| Z new internal tree node.

```
n = |C|
make a min heap Q with C                    // O(n)
for i = 1 to n-1
    Z.left  = x = extract_min (Q)           } 2 log n
    Z.right = y = extract_min (Q)           }
    Z.freq = x.freq + y.freq
    Insert (Q, Z)
return (extract_min (Q))                    // O(1)
```

| space complexity O(n)

Time complexity: extract_min() is called $2 \times (n-1)$ times if there are n nodes. As extract_min() calls minHeapify(), which takes O(log n) time, overall TC is **O(n log n)**

→ Average code length per character =

$$\frac{\Sigma\,(freq_i \times \text{code length}_i)}{\Sigma\, freq_i}$$

$$= \Sigma\,(probability_i \times \text{code length}_i)$$

→ Total no. of bits in Huffman encoded message = total no. of chars in the message × (average code length / character)

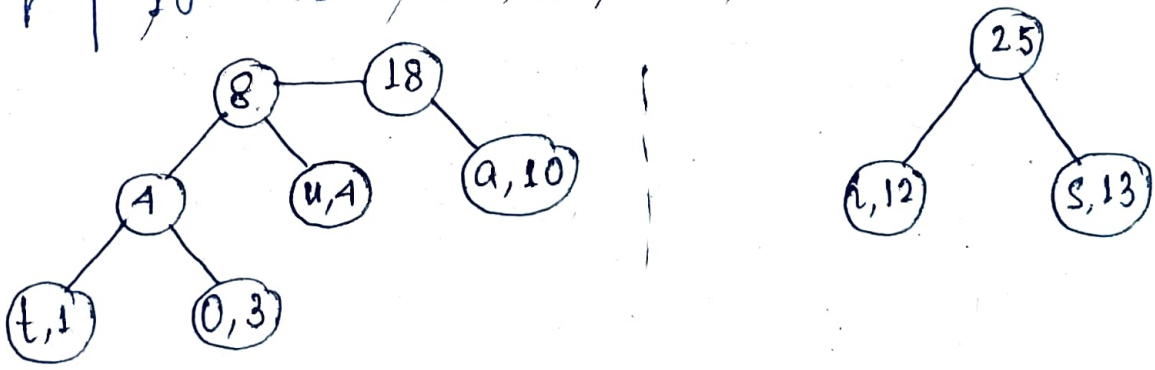
$$= \Sigma\,(freq_i \times \text{code length}_i)$$

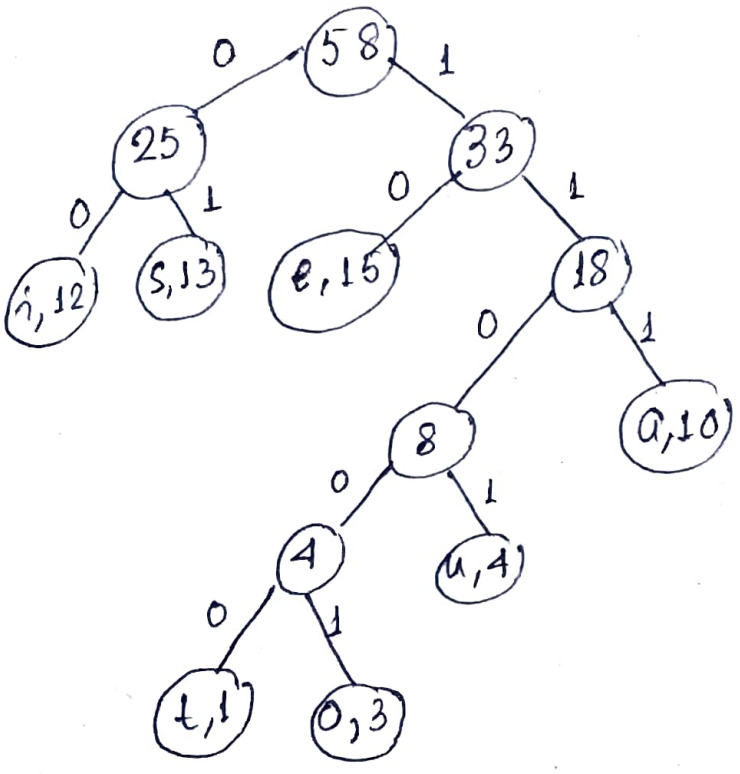
Q. A file contains the following characters with the frequencies as shown. If Huffman coding is used for compression, determine-

1. Huffman codes
2. Average code length
3. Length of Huffman encoded message in bits.

Char	a	e	i	o	u	s	t
freq	10	15	12	3	4	13	1

→





a 111

b 10

i 00

o 11001

u 1101

s 01

t 11000

Average code length =

$$\frac{10\times3 + 15\times2 + 12\times2 + 3\times5 + 4\times4 + 13\times2 + 1\times5}{10 + 15 + 12 + 3 + 4 + 13 + 1}$$

$= 2.52.$

Length of Huffman encoded message =

$(10 + 15 + 12 + 3 + 4 + 13 + 1) \times 2.52$

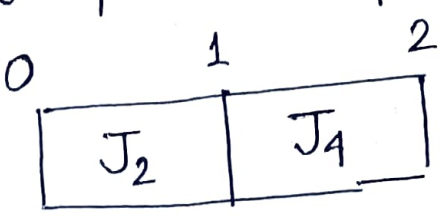
$= 146.16 \approx 147$ bits.

* Job sequencing with deadlines

eg

Jobs	J_1	J_2	J_3	J_4
~~Duration~~ Deadline	2	1	1	2
Profit	6	8	5	10

Highest profit as far as possible on the timeline.



10+8 = 18 profit

- **Job sequencing with deadlines.**

The sequencing of jobs on a single processor with deadline constraints.

- We're given a set of jobs; each job having a deadline/~~duration~~ & some profit.
- The profit of job is attained only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs & the processor takes one unit of time to complete a job.

Problem: How can the total profit be maximized if only one job can be completed at a time?

* Greedy algorithm

i) Sort all the given jobs in decreasing order of profit.

ii) Check the value of maximum deadline.

Draw a Gantt chart where max time on Gantt chart is the value of max deadline.

iii) Pick up the jobs one by one.

Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline

* Example.

Jobs	J1	J2	J3	J4	J5	J6
Deadline	5	3	3	2	4	2
Profit	200	180	190	300	120	100.

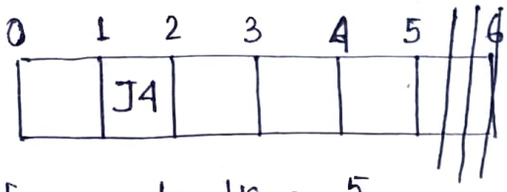
⇒ 1. Sort. acc. to profit.

	J4	>	J1	>	J3	>	J2	>	J5	>	J6
Deadline	2		5		3		3		4		2

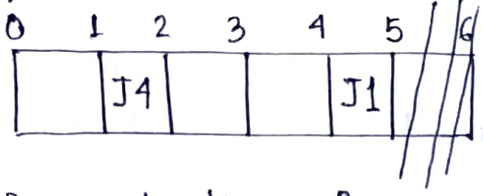
2. Gantt chart of size 5, as the max. deadline is 5.

J4	>	J1	>	J3	>	J2	>	J5	>	J6
2		5		3		3		4		2

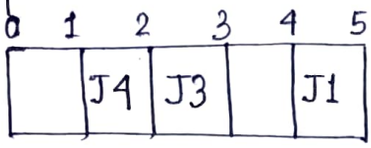
3. Place J4 in the 1st empty cell before deadline 2.



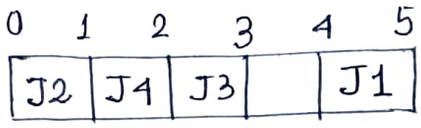
J1 before deadline 5



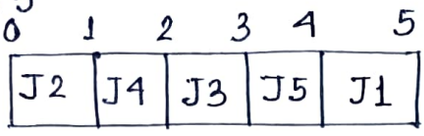
J3 before deadline 3



→ J2 before deadline 3



J5



J6 can't be completed, if we optimise the profit.

∴ Optimal schedule is

J2, J4, J3, J5, J1.

Profit = 180 + 300 + 190 + 120 + 200 = 990 units

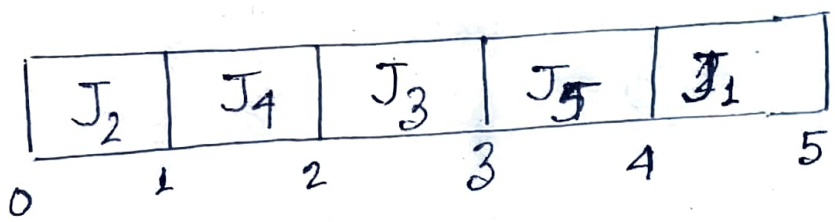
→ We use priority queue to extract the job with minimum runtime.

→ Algorithm_Job_sch_w_deadline TC $\boxed{O(n \log n)}$

1. Sort all jobs in decreasing order of profit.
2. Iterate on jobs in decreasing order of profit. For each job, do the following:
   a) find a time slot i, such that slot is empty & i < deadline & i is the greatest. Put the job in this slot & mark it filled.
   b) If no such i exists, ignore the job.

eg

	J_1	J_2	J_3	J_4	J_5	J_6
D	5	3	3	2	4	2
P	200	180	190	300	120	100

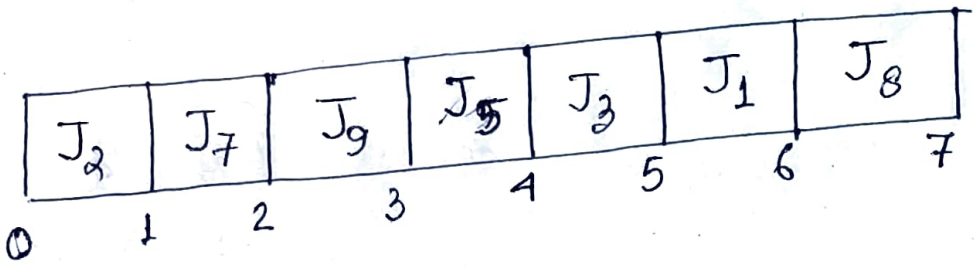


Profit = 200 + 180 + 190 + 300 + 120 + 100

→ Time complexity $O(n^2)$.

eg.

J	1	2	3	4	5	6	7	8	9
P	15	20	30	18	18	10	23	16	25
D	7	2	5	3	4	5	2	7	3



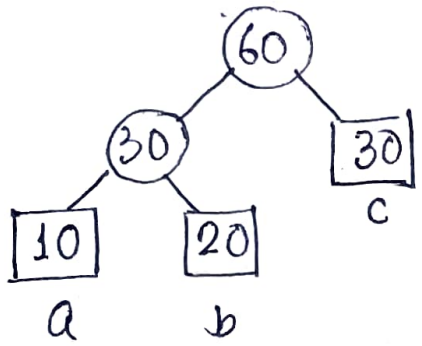
P ↑ $J_3 \geq J_9 \geq J_7 \geq J_2 \geq J_4 \geq J_5 \geq J_8 \geq J_1 \geq J_6$

P = 15 + 20 + 30 + 18 + 23 + 16 + 25

* Optimal Merge Patterns (Huffman code etc.)

eg. a, b, c are 3 files containing 10, 20, 30 records respectively in sorted order. Merge them such that final file is also sorted.

Minimise no. of record movements.



External weighted path length =

$10 \times 2 + 20 \times 2 + 30 \times 1$

$= 90$

→ Time complexity - $O(n \log n)$

* Kirchoff's Matrix Tree Theorem.

It's a formula for the number of spanning trees of a graph in terms of the determinant of a certain matrix.

A subgraph $T(V, E')$ of a graph $G(V, E)$ is a spanning tree if it is a tree that contains every vertex in V, with minimum #edges.

→ If a graph is a complete graph with n vertices, then total number of spanning trees is n^{n-2} (n is #nodes in the graph).

→ for a graph, that is not complete, we have Kirchoff's Matrix Tree theorem.

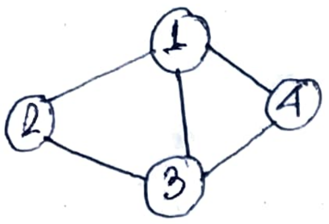
- Theorem We first construct the Laplacian matrix Q for the graph G which can be shown to be such that:

1. for $i \neq j$
   - if vertex i and j are adjacent on G, then $q_{ij} = -1$
   - otherwise, $q_{ij} = 0$
2. for $i = j$, q_{ij} equals the degree of vertex i in G.

We now construct matrix Q' by deleting any row r & column c (where $r = c$ or $r \neq c$) from Q. (Cofactor)

Finally we take the determinant of Q' to obtain the no. of spanning trees of the G.

eg.



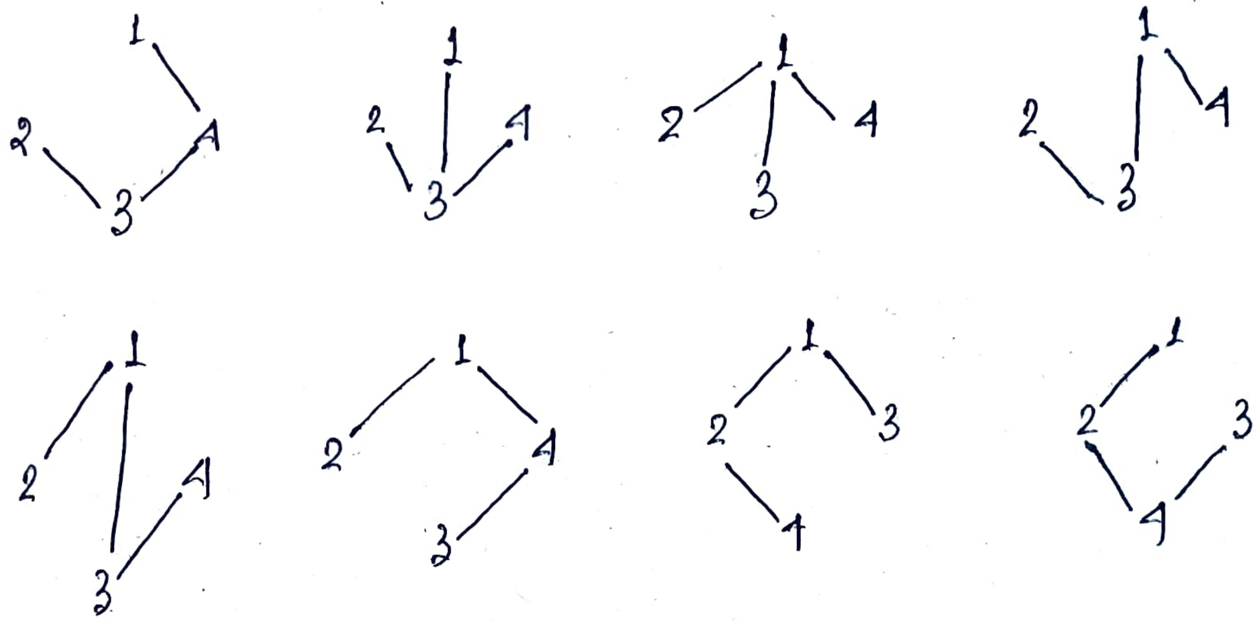
$$Q = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix} \end{array}$$

$$Q' = \begin{pmatrix} 2 & -1 & 0 \\ -1 & 3 & -1 \\ 0 & -1 & 2 \end{pmatrix}$$

- Adjacency matrix.
- Diagonal elems → degree of nodes
- Non-diagonal 1's → -1
- Non-diagonal 0's → 0
- Determinant of cofactor of any elem.

$Det(Q') = 8$

#spanning trees = 8.



Weighted Graphs.

* Prim's Algorithm. (MST)

finding minimum cost spanning tree.

- Idea is to maintain 2 sets of vertices:

a) Contain vertices already included in MST (min. spanning tree)

b) Contain vertices not yet included.

At every step it consideres all the edges & picks the minimum weight edge (Greedy). After picking the edge, it moves the other endpoint of edge to set containing MST.

- To apply Prim's algorithm, the graph must be weighted, connected and undirected.

- Algorithm

1. Randomly choose any vertex. The vertex connecting to the edge having least weight is usually selected.

2. Find all the edges that connect the tree to new vertices. Find the least weight edge among those edges and include it in existing tree. If including that edge creates a cycle, then reject that edge & look for the next least weight edge.

3. Keep repeating 2 until all the vertices are included and MST is obtained.

→ Time complexity

If adjacency list is used to represent the graph then using BFS, all the vertices can be traversed in $O(V+E)$ time. We traverse all the vertices of graph using BFS & use a min heap for storing the vertices not yet included in the MST. To get the minimum weight edge, we use a min heap as priority queue. Min heap operations like extracting minimum element & decreasing key value takes $O(\log V)$ time.

Overall time complexity =

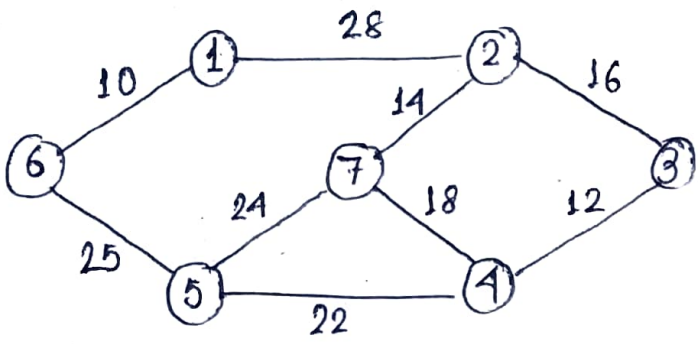
$$O(E+V) \times O(\log V)$$

$$= O((E+V)\log V)$$

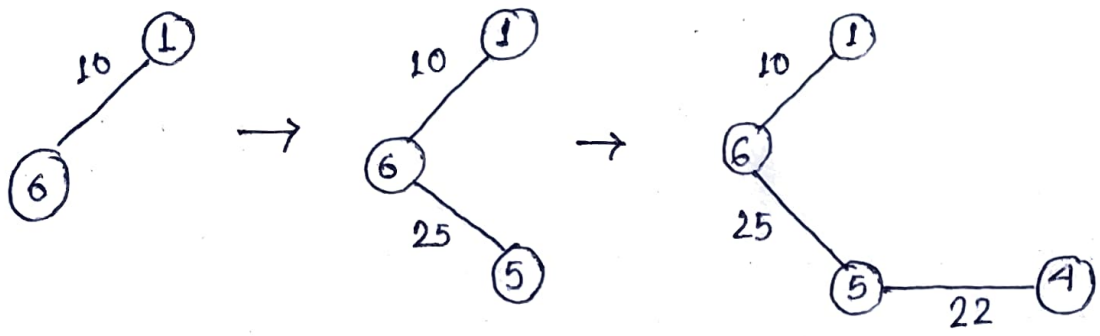
$$= O(E \log V)$$

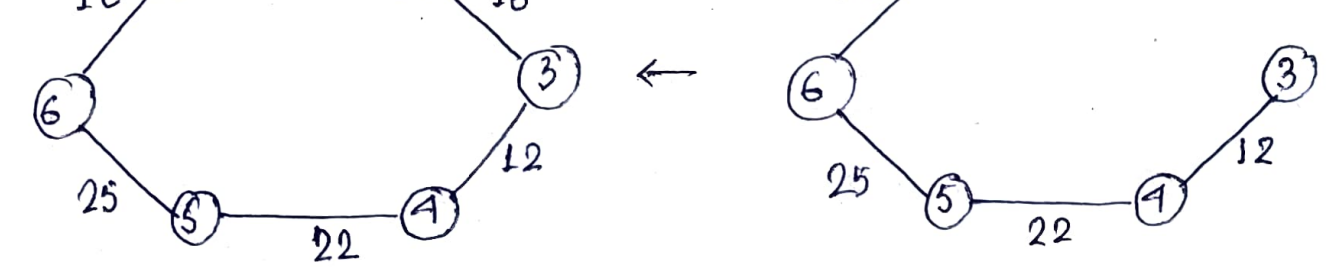
TC can be improved and reduced to $O(E + V\log V)$ using Fibonacci heap.

Eg. MST for given graph using Prim's.



→





Cost of MST =

$10 + 25 + 22 + 12 + 16 + 14$

= 99 units

Q. G'10 Consider a complete undirected graph with vertex set $\{0,1,2,3,4\}$. Entry W_{ij} in the matrix W below is the weight of the edge $\{i,j\}$

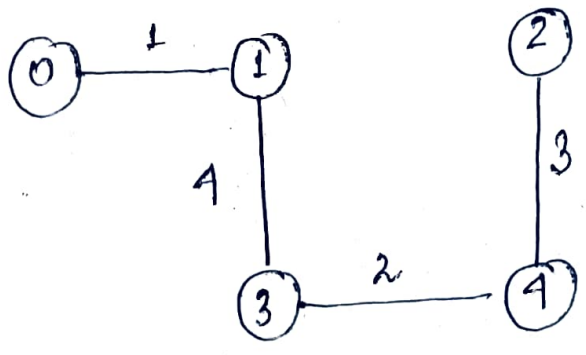
$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T?

→ For finding MST with vertex 0 as a leaf, first of all remove 0th row and 0th column & then get the MST of remaining graph & then connect the vertex 0 with the edge with minimum weight (we have two options as there are 2 1's in 0th row).

Look cost matrix, pick min value (which is not forming cycle with already included vertices).

$$W = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix} \end{array}$$



Cost =

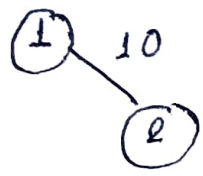
10

→ Building MST from cost matrix :

	1	2	3	4
1	0	10	10	50
2	10	0	40	30
3	10	40	0	20
4	50	30	20	0

(1) Start with vertex 1.

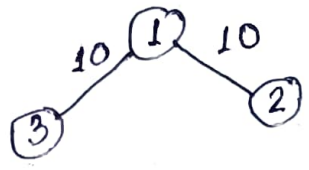
Least cost from 1 is the edge 1-2 (tie with 1-3).



(2) We have chosen 1, 2 now.

These are the parts of MST.

Among the rows 1 and 2, find least number.



	1	2	3	4
1	0	~~10~~	(10)	50
2	~~10~~	0	40	30

1-2, 2-1 entries ✗

1-2 chosen. So, 1-2, 2-1 entries from matrix are overlooked.

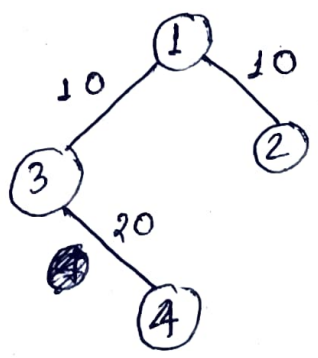
(1-3) is chosen having least cost. 10.

(3) Now, 1, 2, 3 have been chosen & they are part of MST.

Find smallest cost among the rows of 1, 2 & 3.

	1	2	3	4
1	0	~~10~~	~~10~~	50
2	~~10~~	0	40	30
3	~~10~~	40	0	(20)

1-3, 3-1 entries ✗



MST

(3-4) chosen having least cost 20.

(Also, be alert whether any cycle is formed or not.)

- Prim's algorithm without using Min-heap.

PrimAlgo (E, cost, n, t) * → next pg bottom.

{

// E is the set of edges. Cost is nxn adjacency matrix

// MST is computed & stored in array t[1:n-1, 1:2]

let (k, l) be an edge of min cost of E; // O(e)

mincost = cost [k, l]; // O(1) | e = |E|

t[1,1] = k ; t[1,2] = l;

O(n) { for (i = 1 to n) // initialise near
    if (cost[i, l] < cost[i, k]) then near[i] = l ;
    else near[i] = k ;

near[k] = near[l] = 0 ; Watch vbr

O(n²) { for (i = 2 to n-1) // find n-2 additional edges for t.
    { let j be an index s.t. near[j] ≠ 0 and
      cost[j, near[j]] is minimum ;
    t[i,1] = j ; t[i,2] = near[j];
    mincost = mincost + cost[j, near[j]];
    near[j] = 0.
    O(n) { for k ← 1 to n do
        if (near[k] ≠ 0 and cost[k, near[k]] > cost[k, j])
            near[k] = j; }

if mincost ≥ ∞ print ('no spanning tree')

→ Time complexity $\underline{\theta(n^2)}$.

• Prim's algorithm with min heap.

All vertices that're not in the tree reside in a min-priority queue Q based on a key field. For each vertex v, key[v] is the minimum weight of any edge connecting v to a vertex in the tree; by convention, key[v] = ∞ if there's no such edge. The field π[v] names the parent of v in the tree.

r root
w weights

```
MST_Prim (G, w, r)
    for each u ∈ V[G]                          ┐
        do key[u] ← ∞                          │ Setting key of each vertex
           π[u] ← NIL                          │ to ∞, except for r, so that
    key[r] ← 0                                 │ it will be the first vertex
    Q ← V[G]                                   │ processed.
    While Q ≠ ϕ                                │ Initializing the min-priority
        do u ← Extract-min (Q)                 │ queue Q to contain all
           for each v ∈ Adj[u]                 │ the vertices.
               do if v ∈ Q and w(u,v) < key[v] ┘
                  then π[v] ← u
                       key[v] ← w(u,v)
```

Watch vbr

TC - $O(V \log V + E \log V) = O(E \log V)$

---

* t is set of edges storing MST.

t(i,1), t(i,2) is an edge in the MST.

* Kruskal's Algorithm. (MST)

Finding MST for a given graph that is weighted, connected & undirected.

Algo

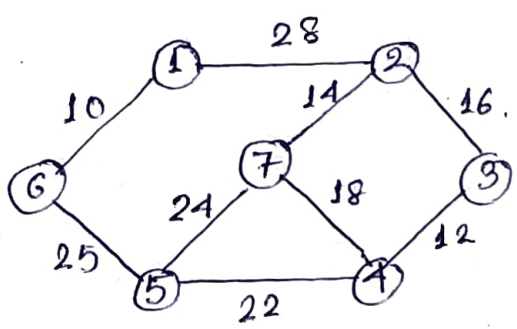
- Sort the graph edges wrt their weights.
- Start adding edges to the minimum spanning tree from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which don't form a cycle - edges which connect only disconnected components.

* Kruskal_Algo (E, cost, n, T, mincost)

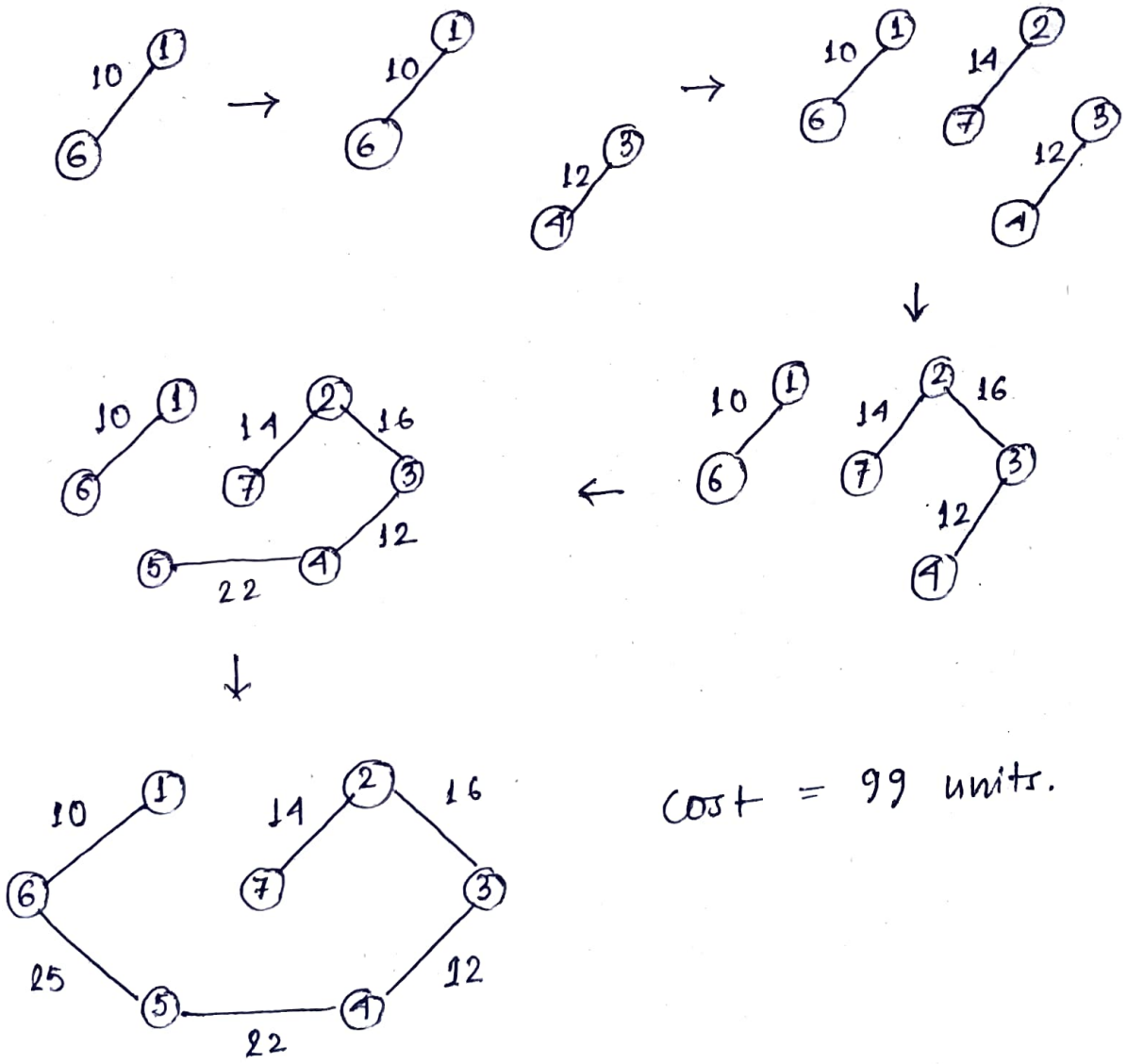
```
//E is the set of edges in G. G has n vertices.
// Cost(u,v) is the cost of edge (u,v). T is the set
//of edges in the MST & mincost is its cost.
{
    real mincost, cost (1:n, 1:n)
    integer parent (1:n), T(1:n-1, 2), n
    construct a heap out of the edge costs using Heapify;
    parent ← -1  // each vertex in different set
    i ← mincost ← 0
    while i < n-1 and heap not empty do
        delete a min. cost edge (u,v) from heap &
                                reheapify using Adjust;
        j ← find (u); k ← find (v);
        if j ≠ k, then  i ← i + 1
```

```
        T(i,1) ← u; T(i,2) ← v
        mincost ← mincost + cost(u,v)
        call Union(j,k)
    endif
repeat
if i ≠ n-1 then print (no spanning tree) endif
return
end Kruskal_Algo
```

eg.



MST using Kruskal.



Cost = 99 units.

Q. G'00. Let G be an undirected connected graph with distinct edge weight. Let e_{max} be the edge with max weight & e_{min}, the min. weight. Which's false?

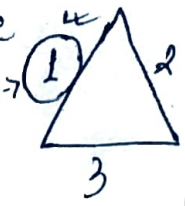
a) Every minimum spanning tree of G must contain e_{min}

b) If e_{max} is in a MST, then its removal must disconnect G.

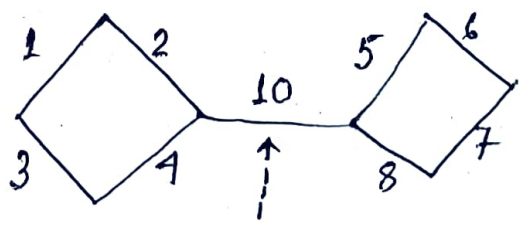
c) No MST contains e_{max}.

d) G has a unique MST.

→ (a) Kruskal's always picks the edges in ascending order of their weights while constructing MST of G. ⇒ True



(b) e_{max} would be included in MST iff e_{max} is a bridge between 2 connected components, removal of which will surely disconnect the graph. ⇒ True



If e_{max} was included in the MST, the reason must have been that there was no other less weighted edge that can connect the MST (true).

(c) Already proved in (b) that e_{max} can be in MST. ⇒ False

(d) G has unique edge weights, so MST will be unique. In case if weights were repeating there could've been a possibility of non-unique MSTs. ⇒ True

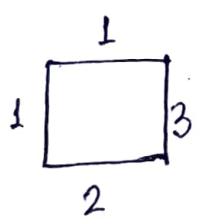
- In case, in the question, the constraint distinct was not there —

(a) There could be a no. of edges having min. weight. Then at least one minimum weight edge should be included in the MST.

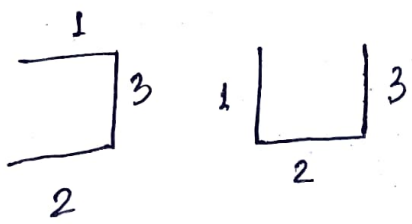
(b) True.

(c) False

(d) G may not have unique MST.



MSTs



- For both same & distinct edge weights, if every edge weight is increased by same value then also MST of graph will not change.

Q. GOT Let w be the min weight among all edge weights in an undirected connected graph. Let e be a specific edge of weight w. Which's false?

a) There's an MST containing e.

b) If e isn't in an MST, then in the cycle formed by adding e to T, all edges have the same weight. (as w e)

c) Every MST has an edge of weight w.

d) e is present in every MST.

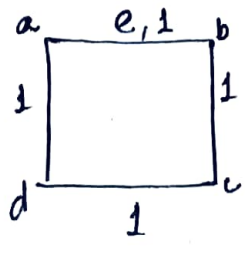
→ a) There is an MST containing e that has the minimum weight. ⇒ True

b) If e is not part of MST, then all edges which are part of a cycle with e, must have weight ≤ e, as otherwise we can interchange that edge with e & get another MST of ≤ weight. ⇒ True

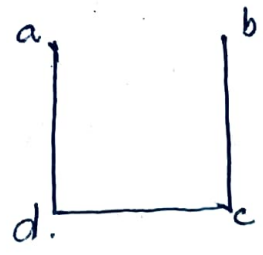
c) An MST must have the edge with the smallest weight. ⇒ True

d) Suppose, a cycle is there with all edges having the same min weight w. Now, any of them can be avoided in any MST.

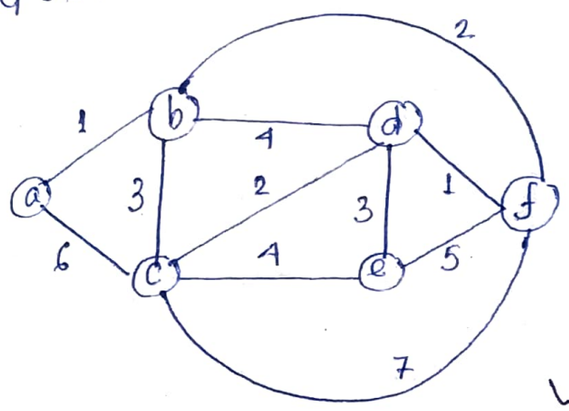
⇒ False.



→ MST not having e



Q, G'06.



Which can't be the seq. of edges added in that order to an MST, using Kruskal's ?

a) ab, df, bf, dc, de
b) ab, df, dc, bf, de
c) df, ab, dc, bf, de
~~d)~~ df, ab, bf, de, dc.

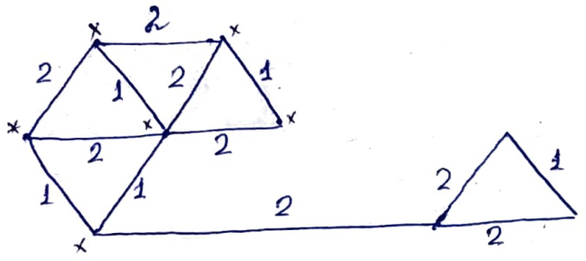
d-e weight 3
d-c weight 2

All possible ways -

ab, df taken in 2 ways
dc, bf taken in 2 ways.

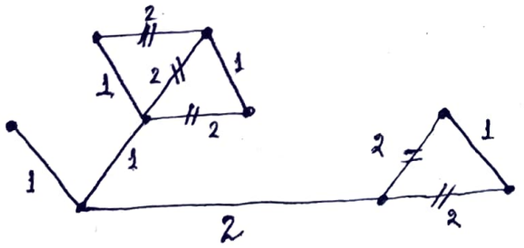
ab, df → dc, bf, de
df, ab → bf, dc, de
} de always at the end.

Q. G'11 # distinct MSTs for the weighted graph -.

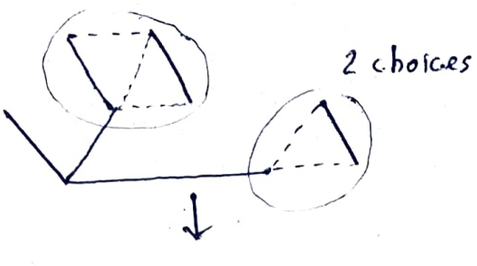


→ Selection of edges of cost 1 will not form a cycle.

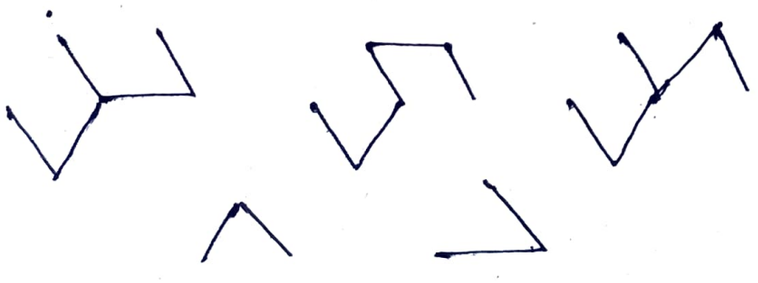
Choices $3 \times 2 = 6$

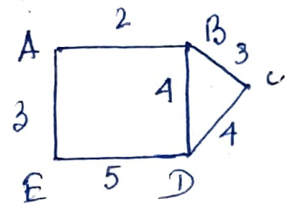


3 choices

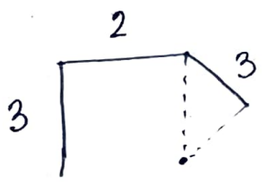


$3 \times 2 = 6$



Q 

possible MSTs.

→ 

cost 3 edges don't form cycle. So, we can use them both.

Connecting cost-4 edges we have 2 choices - BD or CD

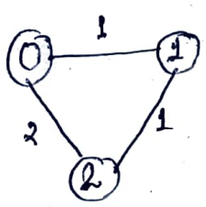
Answer = 2.

Q. A complete, undirected, weighted graph G is given on the vertex $\{0, 1, \ldots, n-1\}$ for any fixed n. Draw the MST of G if

a) Weight of the edge (u, v) is $|u-v|$

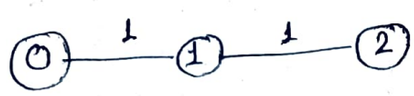
b) Weight of the edge (u, v) is $|u+v|$

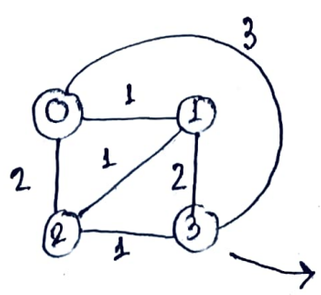
→ n = 3.

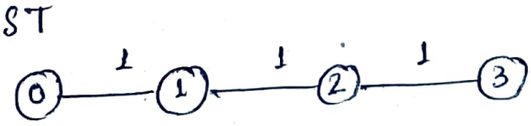
a) 

Between every increasing ordered vertices, there is an edge of weight 1.

↓

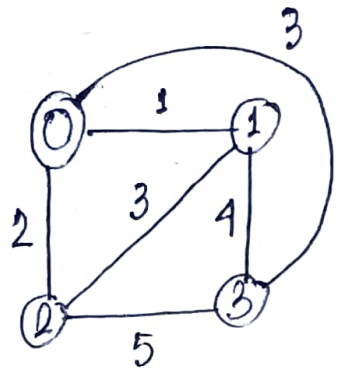
MST 



MST 

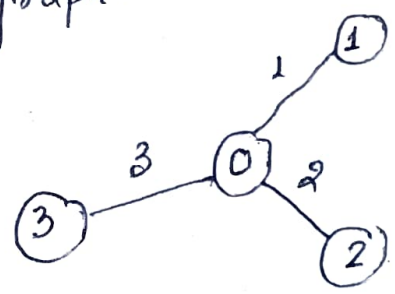
We get a line graph. 

b) $n = 4$.



→

We always get a star graph.



As the edges connecting ⓪ and (n) have the least weight for corresponding (n).

• Disjoint-Set Data Structure

Data structure representing a dynamic collection of sets $S = \{S_1, \ldots, S_n\}$. Given an element u, we denote by S_u the set containing u. Each set S_i has a representative element $rep[S_i]$.

- Checking u & v in same set or not - whether $rep[S_u] = rep[S_v]$ or not.

- Disjoint-set operations. :

O(1) LL rep^n a) Make_Set (u) : Creates new set containing the single element u.

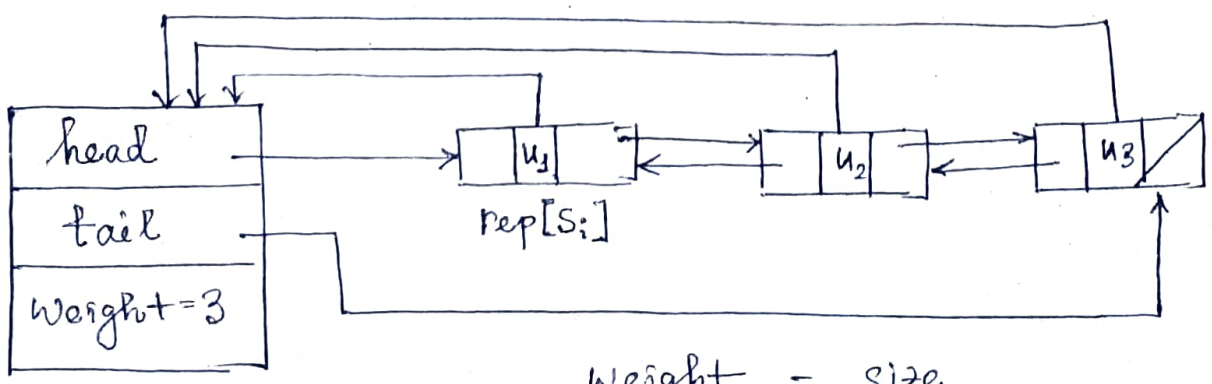
u must not belong to any already existing set.

u will be the representative element initially.

O(1) LL rep^n b) Find_Set (u) : Returns the representative element $rep[S_u]$.

O(n) LL rep^n c) Union (u, v) : Replaces S_u and S_v with $S_u \cup S_v$ in S. Updates rep.

S_i :



Weight = size.

- When S_u merges with S_v,

~ $S_u.weight > S_v.weight$ ⇒ No update to u's head pointer is needed.

~ $S_v.weight \geq S_u.weight$ ⇒ We update u's head ptr.

Value of $S_u.weight$ at least doubles.

Because $S_u.weight$ at least doubles every time we update u's head pointer & because $S_u.weight$ can only be at most n (n = # of elements = # of Make-set operations), it follows that the total number of times we update u's head ptr is at most $\log n$.

CLRS

Thus, total cost of all UNION operations is $O(n \lg n)$ and the total cost of any sequence of m operations is $O(m + n \log n)$. [LL rep^n, weighted union - heuristic]

~~Forest-of-Trees Implementation~~.

- Linked List operations.

```
Make_set (x)        // O(1)
    Set ← new set
    Set.head ← x
    Set.tail ← x
    Set.weight ← 1
    x.set ← Set
    x.next ← NIL

Find_set (x)        // O(1)
    return x.set
```

```
Union (x, y)
    Sx ← Find-set (x)
    Sy ← Find_set (y)
    if Sx = Sy return
    if Sx.size < Sy.size
        then exchange Sx ↔ Sy
    Sx.tail.next ← Sy.head
    Sx.tail ← Sy.tail
    Sx.weight ← Sx.weight + Sy.weight
    While y ≠ NIL
        y.set ← Sx
        y ← y.next.
```

– Amortised analysis of LL implementation.

Assume m operations including n Make_set's.

There can be $O(m)$ Find_set operations,

total time $O(m+n\lg n)$. ☺

CLRS

Operation	# of objects updated.	
Make-set (x_1)	1	$\Theta(n)$
⋮	⋮	
Make-set (x_n)	1	
Union (x_1, x_2)	1	$\Theta(n^2)$
Union (x_2, x_3)	2	
Union (x_3, x_4)	3	
⋮		
Union (x_{n-1}, x_n)	$n-1$	

Sequence of $2n-1$ operations on n objects takes $\Theta(n^2)$ time or $\underline{\Theta(n)}$ time per operation, on the average.

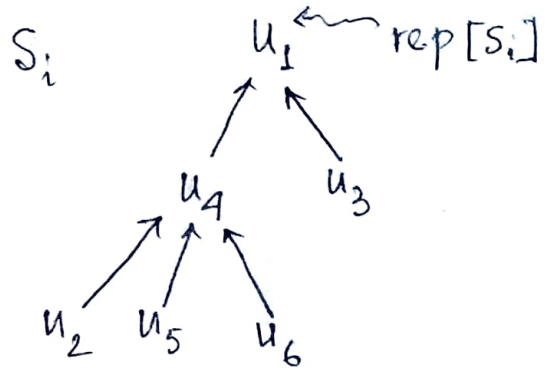
– Weighted union heuristic:

Always append the shortest list to the end of the longest while doing union.

This does not always save time. When both sets have $n/2$ elems, union still requires $O(n)$ time.

But, if there are 2 randomly generated lists with a total of n elements, then there's improvement.

– Forest-of-trees Implementation.



Make-set (u) : Initialize new tree with root node u — $\theta(1)$

Find-set (u) : Walk up tree from u to root — $\theta(\text{height}) = \theta(\lg n)$ best ca

Union (u, v) : Change rep[S_v]'s parent to rep[S_u] — $O(1) + 2T_{\text{Find-set}}$

Make_set (x)

```
x.parent ← x
x.rank ← 0
```

Find_set (x)

```
if x = x.parent
  then return x
else y ← Find_set(x.parent)
     x.parent ← y
     return y
```

Union (x, y)

```
x ← Find-set (x)
y ← Find-set (y)
if x = y return
if x.rank > y.rank
   then y.parent ← x
else x.parent ← y
     if x.rank = y.rank
     then y.rank ← y.rank + 1
```

• **Union by rank** : Heuristics

When we call Union (u, v), rep[S_v] becomes a child of rep[S_v]. Merging S_v into S_u results in a tree of height max {height[S_u], height[S_v] + 1}. Thus, the way to keep our trees short is to always merge the shorter tree into the taller tree. (analogous to smaller into larger in LL reph).

Rank of a node is the height of the subtree rooted at that node. MCake the root with the smaller rank into a child of the root with the larger rank. The combined tree, that is no higher than the taller of the 2 trees, produces a taller tree only when both original trees were the same size.

Union $(\tilde{u}, \tilde{v})$

```
u ← find-set (ũ)
v ← find-set (ṽ)
if u.rank = v.rank then
    u.rank ← u.rank + 1
    v.parent ← u
else if u.rank > v.rank then
    v.parent ← u
else
    u.parent ← v.
```

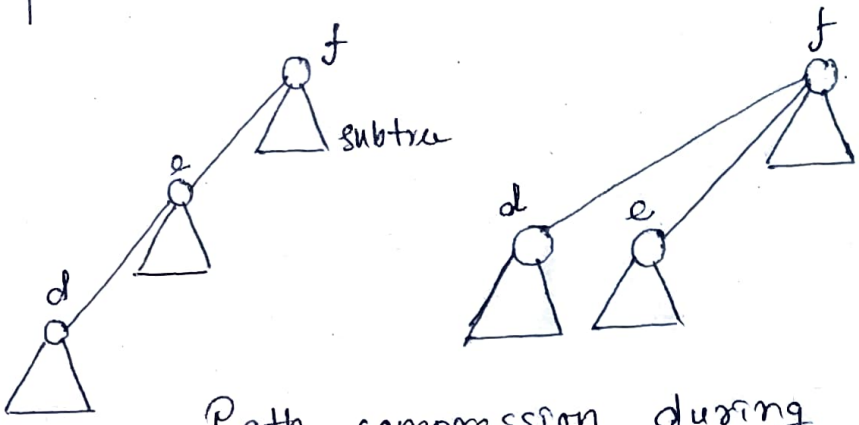
The worst case performance of a disjoint-set forest with union by rank having n elems is

MCake-set	$O(1)$
find-set	$\theta(\lg n)$
Union	$\theta(\lg n)$

* A root node of rank k is created by merging 2 trees of rank $k-1$. So, a root node of rank k has at least 2^k nodes in its tree. If we started off with n elems, then there are at most $n/2^k$ nodes of rank k. Consequently, the max. rank is $\log_2 n$. So, all trees have a height $\leq \log_2 n$. Therefore, find-set & Union have a running time of $O(\log n)$.

- Path compression.

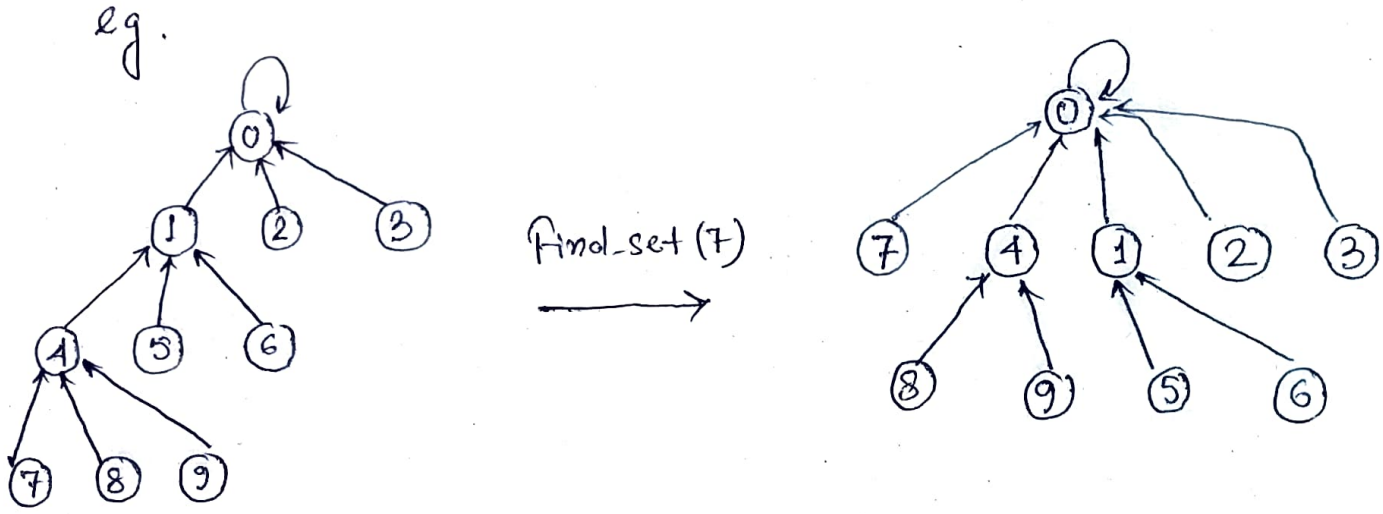
When executing find-set (a), make all nodes on the find path of a direct children of the root.



Path compression during find_set

So, every time we invoke find-set & walk up the tree, we should reassign parent pointers to make each node we pass a direct child of the root. This locally flattens the tree.

eg.



Find_set (n)

$A \leftarrow \phi$

$n \leftarrow u$

While n is not the root do

$A \leftarrow A \cup \{n\}$

$n \leftarrow n.parent$

for each $x \in A$ do

$x.parent \leftarrow n$

return n

It can be shown that, with path-compression (but not union by rank), the running time of any sequence of n Make_set operations, f Find_set operations & up to $n-1$ Union operations is

$$\Theta\left(n + f\left(1 + \log_{2+f/n} n\right)\right).$$

– Both Heuristics together –

Union by rank & Path compression.

On a disjoint-set-forest with these 2 heuristics to improve performance, any sequence of m operations, n of which are Make_set's, has worst case running time $\Theta(m\alpha(n))$, where α is the inverse Ackermann-fun^n. Thus, the amortized worst-case running time of each operation is $\Theta(\alpha(n))$. If one makes the approximation $\alpha(n) = O(1)$, which is valid for literally all conceivable purposes, then the operations on a disjoint-set-forest have $O(1)$ amortized running time.

$\alpha(n) = \min\{k : A_k(1) \geq n\}$, where $(k, j) \mapsto A_k(j)$ is the Ackermann fun^n.

$A_k(j)$ is rapidly growing fun^n $\longrightarrow$ inverse Ackermann f^n is slowly growing f^n.

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1 \end{cases}$$

($A_{k-1}^{(j+1)}$ that is A_{k-1} iterated $j+1$ times)

$A_1(1) = 3$

$A_2(1) = 7$

$A_3(1) = 2047$

$A_4(1) \gg 10^{80}$.

$A_1(j) = 2j + 1$

$A_2(j) = 2^{j+1}(j+1)$

– Application of Disjoint Set DS. – Determining the connected components in an undirected graph

When the edges of the graph are static - not changing over time - the connected components can be computed faster by using DFS. Sometimes, however, the edges are added dynamically & we need to maintain the connected components as each edge is added. In this case, using disjoint-set operations is more efficient than running a new DFS for each new edge.

Connected_components (G)

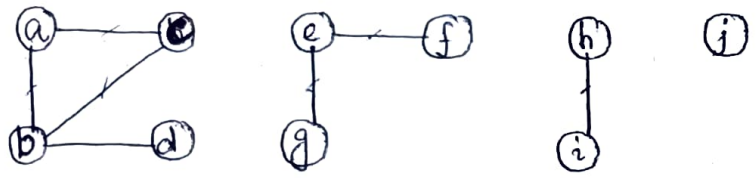
```
for each vertex v ∈ V[G]                          E O(1)
    do Make-set(v)   // O(V)                      O(E)
for each edge (u,v) ∈ E[G]
    do if Find_set(u) ≠ Find_set(v)               O(1)
       then Union(u,v)
```

TC O(E)
SC O(V)

Same_component (u,v)

```
// Whether u & v in same component
if Find_set(u) = Find_set(v)
    then return True
else return False.
```

eg.



Edge processed — Collection of disjoint sets

initial sets	a	b	c	d	e	f	g	h	i	j
(b,d)	a	b,d	c		e	f	g	h	i	j
(e,g)	a	b,d	c		e,g	f		h	i	j
(a,c)	a,c	b,d			e,g	f		h	i	j
(h,i)	a,c	b,d			e,g	f		h,i		j
(a,b)	a,b,c,d				e,g	f		h,i		j
(e,f)	a,b,c,d				e,g,f			h,i		j
(b,c)	a,b,c,d				e,f,g			h,i		j

Connected components

{a,b,c,d} {e,f,g} {h,i} {j}

- **Implementation of Kruskal's Algorithm using Disjoint-set operations.**

Kruskal_MCST (V, E, w)

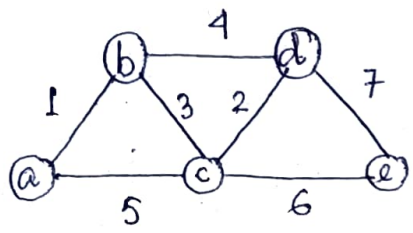
Using union by rank & path compression

```
// Initialization
1. T ← φ                                   // O(1)
2. for each vertex v ∈ V do                } O(V)
3.     Make-set (v)                        }
4. Sort the edges in E into non-decreasing order of weight // O(ElgE)
   // main loop.
5. for each edge (u,v) ∈ E in non decreasing of weight do
6.     if Find-set (u) ≠ Find-set(v) then   } EO(1)
7.         T ← T ∪ {(u,v)}                 }
8.         Union (u, v)                    }
9. return T.
```

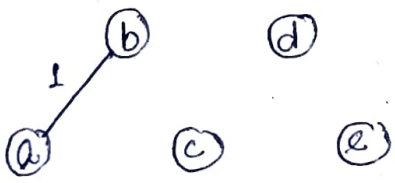
Running time $O(E \log E) = O(E \log V^2) = O(E \log V)$.

CLRS.

eg.

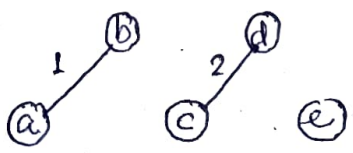


{a} {b} {c} {d} {e}



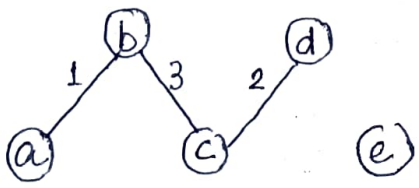
{a,b} {c} {d} {e}

→



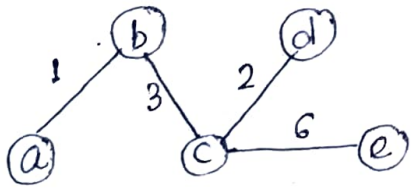
{a,b} {c,d} {e}

↓



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

←



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

- Properties.

1. Kruskal's can return different spanning trees for the same input graph G, depending on how it breaks ties when the edges are sorted into order (when graph has duplicate weight edges).

2. Time complexity $O(E \log V)$. When edges are already sorted, $O(E\alpha(V))$, which is almost linear.

- Time complexity analysis.

1. Initializing the set takes $O(1)$ time.

2. Time to sort the edges $O(E \lg E)$

3. 2-3 & 5-8 lines take a total of $O((V+E)\alpha(V))$ time.

4. Since, $\alpha(|V|) = O(\lg V) = O(\lg E)$,

Total running time $O(E \log E)$.

Now, $|E| < |V|^2$

So $O(E \log E) = \underline{O(E \log V)}$