Algorithm : It is a step by step procedure for performing some task in a finite amount of time.

Characteristics of an algorithms –

(i) Input : must contain zero or more inputs or other quantities

(ii) Output : at least one output must be processed

(iii) finiteness : every algo must complete or terminated after some finite number of steps.

(iv) Definiteness

The steps of the algorithm

i.e. each statement must be out of ambiguity

(v) Effectiveness

The steps of the algorithm must be effective i.e. can be executed successfully.

Steps for solving any problem –

S1 : Identifying the problem statement. i.e. Arranging N-queens in N x N chess board.

S2 : Identifying the constraints or limitations.

Here, no two queens should attack each other, i.e. no two queens can be on same, row, column or diagonal

S3. Design logic

* Divide & Conquer
* Greedy Method
* Dynamic Programming
* Branch
* Back Tracking
* Bound

Depending on the characteristic of problem, we can choose any design strategy mentioned above.

S4. Validation

Most of the algorithm is validated by using any mathematical model.

S5. Analysis

It is a process of comparing two algo wrt time, space, network bandwidth etc.

Prior Analysis	Post Analysis
• Analysis is done before execution of the algo	• Analysis is done after executing the algo.
• Eg - $x \leftarrow x+1$ ↙ Frequency count	• Eg- $x = x+1$ System Time
• It provides estimated value	• It provides exact values
• It provides uniform value	• It provides non-uniform value.
• It is independent of CPU, System Architecture, OS	• It is dependent on various factors

S6. Implementation

S7. Testing & Debugging

Lecture - 3
13/01/23

Time Complexity

is the order of growth of an algorithm wrt different i/p data size.

i/p Size	Algo 1 → 2^n	Algo 2 → $n!$
n = 1	2	1
n = 2	4	2
n = 3	8	6
n = 4	16	24
n = 5	32	120
n = 6	64	720

$n! > 2^n \quad \forall \; n \geq 4$

slower Faster

time complexity

eg. Arrange the following in ascending

$n,\ 2^n,\ n\log n,\ n!,\ n^2,\ n^3,\ n^5,\ 1,\ \log n$

Sol: $1,\ \log n,\ n,\ n\log n,\ n^2,\ n^3,\ n^5,\ 2^n,\ n!$

Frequency Count

```
Algo 1 ( ){
    s1;                          → 1
    for i ← 0 to n do            → n+1   (for last one false, that's why +1)
        s2;                      → n
    s3;                          → 1
}
```

$2n+3$

```
Algo 2 ( ){
    s1;                          → 1
    s2;                          → 1
    for i←0 to n do              → n+1
        for(i←0 to n do          → sum n(n+1)
            s3;                  → n(n+1) - 1
    s4;                          → 1
```

$2n^2 + 3n + 3 \rightarrow O(n^2)$

```
Algo 3 (){
    s1;                          → 1
    for i←0 to n do              → n+1
        for j=i to n do          → (n+1) + n + (n-1) + ... + 1   = n(n+1)/2 + (n
            s2;                  → (n) + (n-1) + (n-2) + .. + 0  = n(n+1)/2
    s3;                          → 1
}
```

```
Algo 4 (){
    s1;                          → 1
    for i←0 to n do              → n+1
        for j←0 to i do          → 1 + 2 + 3 + ..... + n         = n(n+1)/2
            s2;                  → 0 + 1 + 2 + ... + (n-1)       = n(n+1)/2
    s3;                          → 1
}
```

```
Algo 5 (){
    if (x==1)
        s1;
    else if (x==2)
        for (i←0 to n do)
            s2;
    else
        for i←0 to n² do
            s3;
}
```

$$T(n) = \begin{cases} 2 & x = 1 \\ 2n+3 & x = 2 \\ 2n^2+3 & x \neq 1,2 \end{cases}$$

Eg - Linear Search

Best case - $O(1)$

Worst case - $O(n)$

Avg case - $\frac{1+2+3+\ldots+(n-1)+n}{n} = O(n)$

Lecture - 4

16/01/23

Best case - Algorithm that takes minimum no: of steps.

Worst case - Algorithm that takes maximum no: of steps.

Big-Oh (O) (Worst case / Upper bound)

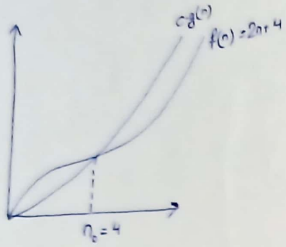
Let $f(n)$ & $g(n)$ are two non-negative function. The function $f(n)$ is equal to $O(g(n))$ if and only if there exist two +ve constant c & n_0 such that $f(n) <= c\,g(n)$ for all values of $n >= n_0$ & $c > 0$.

In other words $f(n) = O(g(n))$ means $g(n)$ is having higher growth rate than $f(n)$ for large value of n.

Exc - Let $f(n) = 2n+4$, then prove that $f(n) = O(n)$

Proof: Let $c = 3$, $f(n) = 2n+4$, $g(n) = n$

$n = 1$,	$f(n) = 2\times1+4 = 6$,	$g(n) = 3\times1 = 3$,	$6 <= 3$ false
$n = 2$,	$f(n) = 2\times2+4 = 8$,	$g(n) = 3\times2 = 6$,	$8 <= 6$ false
$n = 3$,	$f(n) = 2\times3+4 = 10$,	$g(n) = 3\times3 = 9$,	$10 <= 9$ false
$n = 4$,	$f(n) = 2\times4+4 = 12$,	$g(n) = 3\times4 = 12$,	$12 <= 12$ true
$n = 5$,	$f(n) = 2\times5+4 = 14$,	$g(n) = 3\times5 = 15$	$14 <= 15$ true



Q1 $f(n)=5n+3$, then prove $f(n)=O(n)$

Q2 $f(n)=6n^2+2n+3$, then prove $f(n)=O(n^2)$

Sol: Let $c=6$, $f(n)=5n+3$, $g(n)=n$

$n=1$,	$f(n)=8$, $g(n)=6$	$8<=6$ false
$n=2$,	$f(n)=13$, $g(n)=12$	$13<=12$ false
$n=3$,	$f(n)=18$, $g(n)=18$	$18<=18$ true
$n=4$,	$f(n)=23$, $g(n)=24$	$23<=24$ true

$n_0=3$, $c=6$

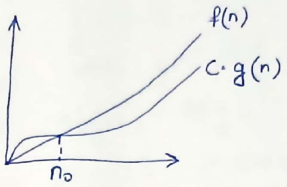
Sol: Let $c=7$, $f(n)=6n^2+2n+3$, $g(n)=n^2$

$n=1$,	$f(n)=11$, $g(n)=7$	$11<=7$ false
$n=2$,	$f(n)=31$, $g(n)=28$	$31<=28$ false
$n=3$,	$f(n)=63$, $g(n)=63$	$63<=63$ true
$n=4$,	$f(n)=107$, $g(n)=112$	$107<=112$ true

$n_0=3$, $c=7$

Omega Notation (Ω) / Best Case / Lower Bound

Let $f(n)$ & $g(n)$ are two non negative functions. The function $f(n)=\Omega(g(n))$, iff their exists two constants c & n_0 such that $f(n)>=c\cdot g(n)$, $\forall n>=n_0$ & $c>0$



Lecture -5
17/1/23

```
Algm One ( ){
    if (x==1){
        s1;
    }
    else {
        for i ← 1 to n do
            s2;
    }
}
```

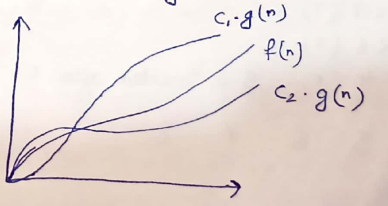
$$T(n)=\begin{cases}2 & x=1\\ 2n+2 & x>1\end{cases}$$

Worst Case: $O(n)$

Best Case: $O(1)$ or $\Omega(1)$

Theta Notation (Θ) / Average Case

Let $f(n)$ & $g(n)$ are two non negative functions, Then $f(n)=\Theta(g(n))$, iff there exits two positive constants c_1 & c_2 & n_0, such that $c_2\,g(n)<=f(n)<=c_1\cdot g(n)$, $\forall\ n\geq n_0$, c_1 & $c_2>0$



If an algorithm is having a single case / ... ame complexity is represented in θ notation. If an algorithm has multiple cases, but all cases are same, then also it is represented in θ notation.

Insertion Sort

eg. 7 3 8 4 6 5

pass 1. Pick 3, $3 < 7$ (T), 3 is inserted before 7

3 | 7 8 4 6 5

pass 2. Pick 8, $8 < 7$ (F)

3 7 8 | 4 6 5

pass 3. Pick 4, $4 < 8$ (T)
$4 < 7$ (T)
$4 < 3$ (F), 4 is inserted after 3

3 4 7 8 | 6 5

pass 4 Pick 6 ~~Pick 6 (F)~~ $6 < 8$ (T)
$6 < 7$ (T)
$6 < 4$ (F), 6 is inserted after 4

3 4 6 7 8 | 5

pass 5 Pick 5 $5 < 8$ (T)
$5 < 7$ (T)
$5 < 6$ (T)
$5 < 4$ (F), 5 is inserted after 4

3 4 5 6 7 8

Alg^m insertion_sort ()

1. for i ← 2 to n do { → n
2. temp = A[i] → n − 1
3. j = i − 1 → n − 1
4. while (j > 0 && A[j] > temp) { → $\sum_{i=2}^{n} t_i$
5. A[j+1] = A[j] → $\sum_{i=2}^{n} (t_i - 1)$
6. j = j − 1 } // End of while loop
7. A[j+1] = temp } // End of for loop → n − 1

Best Case

When Array is already sorted.

Step − 1, 2, 3 & 7 ⇒ No impact

Step − 4 ⇒ $\sum_{i=1}^{n+1} t_i = \sum_{i=1}^{n-1} 1 = n-1$

Step − 5 & 6 ⇒ $\sum_{i=1}^{n-1} (t_i - 1) = \sum_{i=1}^{n-1} 0 = 0$

$$f(n) = n + (n-1) + (n-1) + (n-1) + 0 + 0 + (n-1)$$
$$= 5n - 4$$
$$= O(n)$$

Lecture − 6
18/1/23

Worst Case (i/p is given in reverse order)

Step − 1, 2, 3 & 7 ⇒ No impact

Step 4 ⇒ $\sum_{i=2}^{n} t_i = 2 + 3 + 4 + \ldots + n = \frac{n(n+1)}{2} - 1$

Step 5 & 6 ⇒ $\sum_{i=2}^{n} (t_i - 1) = 1 + 2 + \ldots + n - 1 = \frac{(n-1)n}{2}$

$$f(n) = n + (n-1) + (n-1) + \frac{n(n+1)}{2} - 1 + \frac{(n-1)n}{2} + \frac{(n-1)n}{2} + (n-1)$$
$$= 3n - 4 + \frac{n(n+1)}{2} + n^2 = \frac{6n - 8 + n^2 + n + 2n^2}{2} = \frac{3n^2 + 7n - 8}{2}$$
$$= O(n^2)$$

Average Case

Step 1, 2, 3 & 7 ⇒ No impact

Step 4 ⇒ eg. 8, 9, 10 (1 hit)

or

9, 8, 10 (2 hits)

$i = 2 \Rightarrow \frac{1+2}{2}$

$i = 3 \Rightarrow \frac{1+2+3}{3}$

$i = n \Rightarrow \frac{1+2+3+\cdots+n}{n}$

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

$\frac{1+2}{2} + \frac{1+2+3}{3} + \cdots + \frac{1+2+3+\cdots+n}{n}$

$\frac{3}{2} + \frac{6}{3} + \frac{10}{4} + \cdots + \frac{n(n+1)}{2n}$

$\frac{2\times3}{2\times2} + \frac{2\times6}{2\times3} + \cdots + \frac{2n(n+1)}{2\times2n}$

$\frac{2\times3}{2\times2} + \frac{3\times4}{2\times3} + \cdots + \frac{n(n+1)}{2n}$

$\frac{3}{2} + \frac{4}{2} + \frac{5}{2} + \cdots \frac{n+1}{2}$

$= \frac{1}{2}\left(3+4+5+\cdots+(n+1)\right) = \frac{1}{2}\left(\frac{(n+1)(n+2)}{2} - 3\right)$

Step 5 & 6 ⇒ $i = 2 \Rightarrow \frac{0+1}{2}$

$i = 3 \Rightarrow \frac{0+1+2}{3}$

$i = n \Rightarrow \frac{0+1+2+\cdots+n-1}{n}$

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

$\frac{0+1}{2} + \frac{0+1+2}{3} + \cdots + \frac{0+1+2+\cdots+n-1}{n}$

$\Rightarrow \frac{1}{2}\left(\frac{n(n-1)}{2}\right)$

$\therefore f(n) = n + (n-1) + (n-1) + \frac{1}{2}\left(\frac{(n+1)(n+2)}{2} - 3\right) + \frac{1}{2}\left(\frac{n(n-1)}{2}\right) + n - 1$

$O(n^2)$

Q. Bubble Sort & Selection Sort

1. for (i=1; i<n; i=i+2) → $\frac{n}{2} + 1$

S1, → $\frac{n}{2}$

$n + 1$

2. for (i=1; i<n; i=i*2) → $\log_2 n$

S1; → $\log_2 n - 1$

$1 \to 2 \to 4 \to 8 \to \cdots \to n$

$2^0 \to 2^1 \to 2^2 \cdots 2^k$

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

$1 < 10$

$2 < 10 \quad 10 < 2$

$4 < 10$

$8 < 10 \quad 4 > 1$

$10 \quad 4$

$\log_2 10 =$

3. p = 0;

for (i=1; p<=n; i++)

p = p+i;

i = 1	2	...	k
p = 0	0		0
1	1		1
	2		2
			...
			k

$\frac{k(k+1)}{2}$

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

$\frac{k^2+k}{2} > n$

$k^2 > n$

$k > \sqrt{n}$

$O(\sqrt{n})$

1 0

0 < 10

1 < 10

2 < 10

3 < 10 } 4

4 < 10

5 < 10

6 < 10

7 < 10

4. A(){

i = 1; s = 1

while (s <= n){

i++;

s = s + i;

}

1 <= 11

3 <= 11

6 <= 11

10 <= 11

15 <= 11 ×

s → 1, 3, 6, 10 ..... n

i → 1, 2, 3, 4, ... k

$\frac{k(k+1)}{2}$

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

$O(\sqrt{n})$

23/1/23

Recurrence

The word recurrence is derived from recursion or repetition

Recursion is a technique which calls the same function again & again

eg. Factorial of no. can be solved by Recursion

fact(n) = n * fact(n-1)

fact(5) = 5 × fact(4)

↳ 4 × fact(3)

↳ 3 × fact(2)

↳ 2 × fact(1)

- Algorithm which use recursion is called recursive algorithm.
- Time complexity of recursive algorithm is given by formula or equation called recurrence.

A recursive method has two cases –

$$T(n) = \begin{cases} 1, & n <= 1 \quad \longrightarrow \text{Exit condition} \\ aT(n/b) + f(n), & n > 1 \quad \longrightarrow \text{Recursive case} \end{cases}$$

Time required to solve a problem of size n.

a → no: of times a function is called

b → is the factor by which the i/p size is reduced.

f(n) → It is the sum time of each function i.e. cost happening at each recurrence step.

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

$$T(n) = \begin{cases} 1, & n<=1 \\ 1 \cdot T(n-1) + 1, & n>1 \end{cases}$$

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

Merge Sort
$T(n) = 2\,T(n/2) + n$

Fibonacci

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

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

```
int doSomething (int n) {
    if ( n<=1 ) {
        return 1;
    }
    else {
        return doSomething (floor( sqrt(n)) + n );
    }
```

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

Solving the recurrence

1) Master Method 2) Tree Method 3) Substitution Method

Substitution Method

eg. $T(n) = \begin{cases} 1, & n<=2 \\ T(\sqrt{n}) + 1, & n>2 \end{cases}$

$$T(n) = T(\sqrt{n}) + 1 \quad \text{——— (1)}$$

Find $T(\sqrt{n})$

$$T(\sqrt{n}) = T(\sqrt{\sqrt{n}}) + 1 \quad \text{——— (2)}$$

Replace eq^n (1) with eq^n (2)

$$T(n) = (T(\sqrt{\sqrt{n}}) + 1) + 1$$
$$= T(n^{1/2^2}) + 2$$
$$\vdots$$
$$= T(n^{1/2^3}) + 3$$
$$= T(n^{1/2^4}) + 4$$
$$\vdots$$
$$T(n^{1/2^k}) + k \quad \text{— (3)}$$

The recurrence will stop when $n^{1/2^k} = 2$

$$\frac{1}{2^k}\log n = \log 2 \qquad 2^k = \log_2 n$$
$$\frac{1}{2^k} = \log_n 2 \qquad k = \log_2 \log_2 n \quad \text{— (4)}$$

$T(n) = T\left(n^{1/2^k}\right) + k$

$T(n) = T(n) + \log_2 \log_2 n$

$T(n) = 1 + \log_2 \log_2 n$

$T(n) = O(\log_2 \log_2 n)$

Q.

eg
```
int ABC (n){
  if(n<2){
    return 1
  else{
    return (ABC(n-1)+ABC(n-1));
  }
```

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

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

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

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

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

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

$\vdots$

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

Let

$n - k = 1$

$k = n - 1$

$n-(n-1)$
$n-n+1$

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

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

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

$= 2^{n-1}$

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

24/01/23

Q. Finding Max & Min from an Array using Divide & Conquer method.

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

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

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

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

$= 4T(n/4) + 3$

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

$T(n/4) = 4T(n/16) + 3$

$T(n) = 4[4T(n/16) + 3] + 3$

$= 16T(n/16) + 15$

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

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

$T(n) = 4[2T(n/8) + 1] + 3$

$= 8T(n/8) + 7$

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

$\frac{n}{2^k}$

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

$= 4T\left(\frac{n}{4}\right) + 3 = 4T\left(\frac{n}{4}\right) + 1 + 2 = 2^2 T\left(\frac{n}{2^2}\right) + 2^0 + 2^1$

$= 8T\left(\frac{n}{8}\right) + 7 = 8T\left(\frac{n}{8}\right) + 1 + 2 + 4 = 2^3 T\left(\frac{n}{2^3}\right) + 2^0 + 2^1 + 2^2$

$= 2^k T\left(\frac{n}{2^k}\right) + \underbrace{2^0 + 2^1 + \ldots + 2^{k-1}}_{2^k - 1}$

At k it stops

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

$n = 2^k$

$\log n = k \log 2$

$\log_2 n = k$

$n = 2^k$

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

$= n + n - 1$

$O(n)$

Q. $T(n) = \begin{cases} 1, & n = 1 \\ 2T\left(\frac{n}{2}\right) + n, & n > 1 \end{cases}$

$T(n) = 2T\left(\frac{n}{2}\right) + n$

$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$

$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n$

$= 4T\left(\frac{n}{4}\right) + n + n$

$= 4T\left(\frac{n}{4}\right) + 2n$

$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{n}{4}$

$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + 2n$

$= 8T\left(\frac{n}{8}\right) + n + 2n$

$= 8T\left(\frac{n}{8}\right) + 3n$

$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$

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

$n = 2^k$ $\quad k = \log_2 n$

$= nT(1) + n \cdot \log_2 n$

$= n + n \log_2 n$

$= O(n \log_2 n)$

×

$$T(n) = \begin{cases} 2, & n=1 \\ 2T(n/2)+7, & n>1 \end{cases}$$

$T(n) = 2T(n/2) + 7$

$T(n/2) = 2T(n/4) + 7$

$T(n) = 2\left[2T(n/4)+7\right] + 7 \quad = 7$

$= 4T(n/4) + 21$

$T(n/4) = 2T(n/8) + 7$

$T(n) = 4\left[2T(n/8)+7\right] + 21 \quad = 14+7$

$= 8T(n/8) + 49 \rightarrow 28+21$

28
21
49

$1 \cdot 7$
$3 \cdot 7$
$7 \cdot 7$

$T(n/8) = 2T(n/16) + 7$

$T(n) = 8\left[2T(n/16)+7\right] + 49$

$2^k T(n/2^k) + \left((2^k - 1)(7)\right)$

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

$n = 2^k$

$n = 2^{k+1}$

~~$k = \log_2 n$~~

$k = \log_2 n - 1$

$n + ((n-1) \cdot 7)$

$n + 7n - 7$

$O(n)$

Q. $T(n) = \begin{cases} 1, & n=1 \\ 2T(n/2) + 4n, & n>1 \end{cases}$

Q. $T(n) = 5T(n/5) + \frac{n}{\log n}$

25/1/23

Recursive Tree Method

$T(n) = aT(n/b) + f(n)$

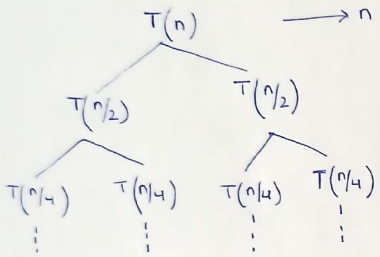
i.e. the i/p size is divided into each of size (n/b) and the cost of division is $f(n)$.

$a \rightarrow$ No of children

$n/b \rightarrow$ Size of each child

$f(n) \rightarrow$ Cost of division

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



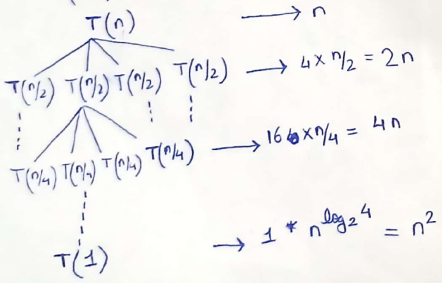
Theorem - 1

If the recurrence is in the form of $T(n) = aT(n/b) + f(n)$ then the height of the tree is $h = \log_b n$.

Theorem - 2

If the recurrence is in the form of $T(n) = aT(n/b) + f(n)$ then the no: of nodes at the last level is equal to $n^{\log_b a}$.

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



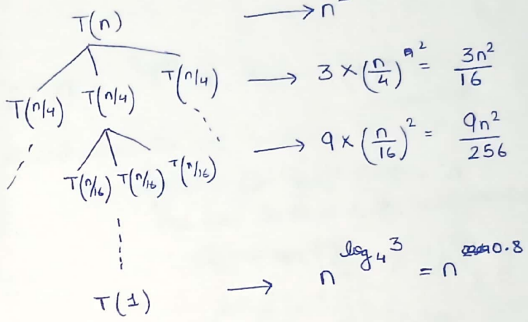
$T(n) = n + 2n + 4n + 8n + \ldots + n^2$

$= n(1 + 2 + 4 + 8 + \ldots) + n^2$

$= n + n^2$

$= O(n^2)$

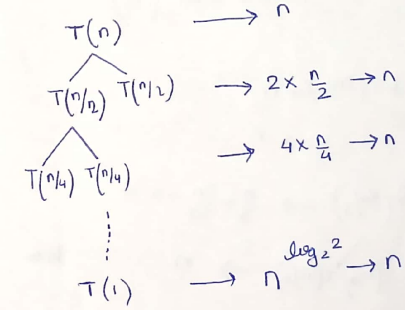
eg. $T(n) = 3T(n/4) + n^2$



$T(n) = n^2 + n^{0.8}$

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

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



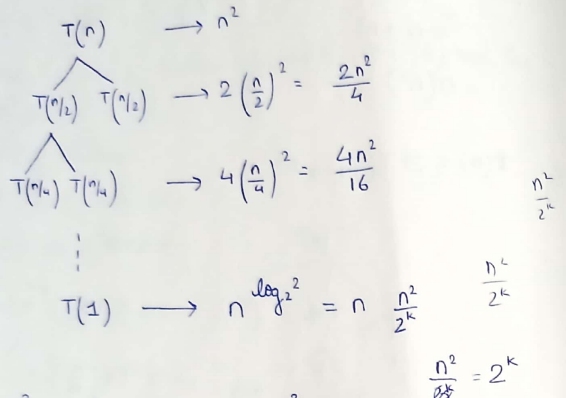
$T(n) = n + n + n + \ldots + n$

$= n(\log_2 n)$

$= O(n \log n)$

log n

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



$n^2 + \frac{2n^2}{4} + \frac{4n^2}{16} + \cdots \frac{n^2}{2^k}$

$n^2\left(1 + \frac{2}{4} + \frac{1}{2^k}\right)$

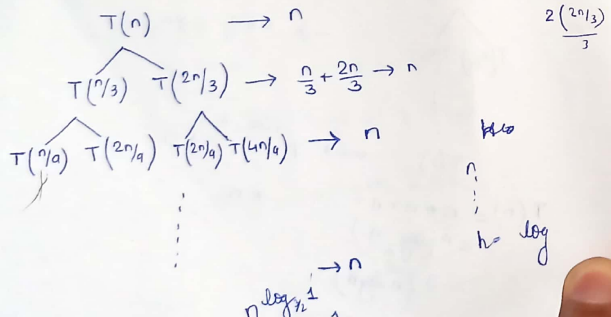
$n^2\left(\frac{(1/2)^{k+1} - 1}{1/2 - 1}\right) = 2n^2\left[1 - (1/2)(1/2)^k\right]$

$= 2^k n^2 - 2n^2 (1/2)(1/2)^k$

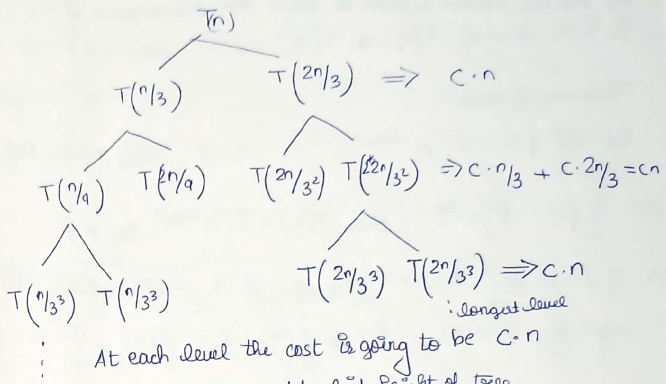
$= 2n^2 - n^2 (1/2)^{\log_2 n}$

$S_{GP} = \frac{a(r^n - 1)}{r - 1}$

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



$= T(n/3) + T(2n/3) + c \cdot n$



At each level the cost is going to be $c \cdot n$

Now we need to find height of tree

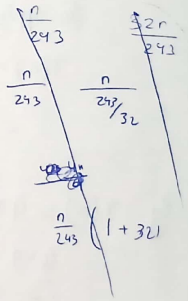
But here the level height is different at different levels

$n \Rightarrow \frac{n}{3/2} \Rightarrow \frac{n}{(3/2)^2} \Rightarrow \frac{n}{(3/2)^3} \cdots 1$

$\frac{n}{(3/2)^k} = 1$

$\left(\frac{3}{2}\right)^k = n$

$k = \log_{3/2} n$



Master Method

The master method is used to solve the recurrence if the recurrence is in the form of $T(n) = a \cdot T(n/b) + f(n)$ where $a \& b > 1$ $f(n) > 0$

There are 3 cases -

C1: If $f(n) = O(n^{\log_b a - t})$, $t > 0$, then $T(n) = \Theta(n^{\log_b a})$

C2: If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \cdot \log n)$

C3: If $f(n) = \Omega(n^{\log_b a + t})$, $t > 0$, & it should satisfy the regularity condition, i.e. $a\, f(n/b) \leq c \cdot f(n)$, $0 < c < 1$

then $T(n) = \Theta(f(n))$

$n^{\log_b a}$	$f(n)$
>	①
=	②
<	③

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

$a = 4 \quad b = 2 \quad f(n) = n$

Find $n^{\log_b a} = n^{\log_2 4} = n^2$

So, $f(n) = n = O(n^{2-t})$, for some $t > 0$

So, apply C1

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

$= \Theta(n^{\log_2 4}) = \Theta(n^2)$

eg. $T(n) = 9T(n/3) + n$

$a = 9 \quad b = 3$

$n^{\log_b a} = n^{\log_3 9} = n^2$

$n = O(n^{2-t})$

$T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$

eg. $T(n) = 8T(n/2) + n \log n$

n^3

$n \log n \quad n = O(n^{3-t})$

$\Theta(n^3)$

$\log_b a = \frac{\log_2 a}{\log_2 b}$

$\log 7 = \log 2$

* $T(n) = 7T(n/2) + n^2$

$a = 7 \quad b = 2 \quad f(n) = n^2$

$n^{\log_b a} = n^{\log_2 7} = n^{2.8}$

$f(n) = n^2 = O(n^{2.8-t})$

$T(n) = \Theta(n^{2.8})$

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

$a = 2 \quad b = 2 \quad f(n) = n$

$n^{\log_2 2} = n$

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

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

$a = 1 \quad b = 2 \quad f(n) = 1$

$n^{\log_2 1} = 1$

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

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

n^2

$T(n) = \theta(n^2 \log n)$

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

$T(n) = \theta(n^3)$

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

$n^{\log_2 2} = n$

CB:

$f(n) = n^4 = \Omega\left(n^{\log_2 2 + \epsilon}\right)$

Regularity condition

$a \cdot f(n/b) <= c \cdot f(n)$

$f(n) = n^4$

$f(n/2) = \left(\frac{n}{2}\right)^4 = \frac{n^4}{16}$

$2 \cdot \frac{n^4}{16} <= c \cdot n^4$

$\frac{n^4}{8} <= c \cdot n^4$

$\frac{1}{8} \leq c < 1$

Hence, the regularity condition is satisfied

$T(n) = \theta(n^4)$

* $T(n) = 4T(n/2) + n^3$

$\frac{n}{2}$

Regularity condition

$a \cdot f(n/b) <= c \cdot f(n)$

$4 \cdot \frac{n^3}{8} <= c \cdot n^3$

$\frac{n^3}{2} <= c \cdot n^3$

$\frac{1}{2} <= c < 1$

$T(n) = \theta(n^3)$

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

$a \cdot f(n/b) <= c \cdot f(n)$

$1 \cdot \frac{n^2}{4} <= c \cdot n^2$

$\frac{1}{4} <= c < 1$

$\theta(n^2)$

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

$2 \cdot \frac{n^2}{4} <= c \cdot n^2$

$\frac{1}{2} n^2 <= c \cdot n^2$

$\theta(n^2)$

* $T(n) = 3T(n/4) + n \log n$
* $T(n) = 2T(n/2) + n^3$
* $T(n) = 2T(n/2) + 2^n$

* $T(n) = T\left(\frac{9n}{10}\right) + n$

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

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

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

$n^{\log_4 3}$

$n^{0.8} < n \log n$

$3 \cdot \frac{n}{4} \log \frac{n}{4} <= c \cdot n \log n$

$3 \frac{n}{4} \log \frac{n}{4}$ / $n \log n$

$\frac{3}{4} \cdot \frac{\log \frac{n}{4}}{\log n}$

$\frac{3}{4} \log_n \frac{n}{4}$

251.10

* $T(n) = 2T(n/2) + n^3$

$2 \cdot \frac{n^3}{8} <= c \cdot n^3$

$\frac{1}{4} <= c < 1$

$\theta(n^3)$

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

$n < 2^n$

$2 \cdot 2^{n/2} <= c \cdot 2^n$

$2^{n/2+1} <= c \cdot 2^n$

$c = 2^{1-n/2}$

$\theta(2^n)$

$a \cdot f(n/b) <= c\, f(n)$

$2 \cdot 2^{n/2} <= c\, 2^n$

$2 \cdot 2^{n/2} <= 2^{-n/2} \cdot 2^{n/2+1-n}$

$2 \cdot 2^{n/2} <= 2^{n-1} \cdot 2^{1-n/2}$

$f(n) = n$

$f(n/b)$

$\frac{n}{10/a}$

* $T(n) = T\left(\frac{9n}{10}\right) + n$

$n^{\log_{10/9} 1}$

$1 < n$

$1 \cdot \frac{9n}{10} <= c \cdot n$

$\frac{9n}{10} <= c \cdot n$

$\theta(n)$

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

$T(n) = \theta(n)$

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

$n^{\log_3 7} = n^{1.77}$

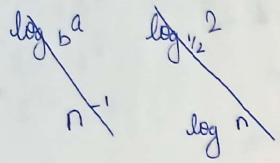
$7 \cdot \frac{n^2}{9} <= c \cdot n^2$

$c = 7/9$

$\theta(n^2)$

Master method by changing variables

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



$T(n) = 2T(\sqrt{n}) + \log n \quad -(1)$

Let $n = 2^m$

$T(2^m) = 2T(\sqrt{2^m}) + \log 2^m$

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

Let $T(2^m) = S(m)$

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

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

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

$\theta(m \log m)$

$n = 2^{\theta(m \cdot \log \cdot}$

$\log n = \theta(m \cdot \log m)$

Find $m^{\log_b a} = m^{\log_2 2} = m$, which is equal to $f(m)$.

$f(m) = \theta(m \log m) \quad \because m = \log_2 n$

$= \theta(\log n \log \log n)$

$f(x) = n^5 \qquad g(x) = n^3 \log n$

$= n^2 \qquad = \log n$

$f(x) = n \qquad g(x) = n \log n$

$1 \qquad \log n$

} They are asymptotically diff.

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

Here $f(n)$ & $n^{\log_b a}$ are not non-polynomial different i.e why master method can not be applied. $n^{\varepsilon}, \varepsilon > 0$

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

Extended Master Method

1. $T(n) = aT(n/b) + \theta(n^k * \log^p n), \quad a \geq 1, b > 1, k \geq 0$

p is real no.

C1: If $(a > b^k)$, $T(n) = \theta(n^{\log_b a})$

C2: If $(a == b^k)$, (i) If $(p > -1)$, $T(n) = \theta(n^{\log_b a} \cdot \log^{p+1} n)$

(ii) If $(p == -1)$, $T(n) = \theta(n^{\log_b a} \log \log n)$

(iii) If $(p < -1)$, $T(n) = \theta(n^{\log_b a})$

C3: If $(a < b^k)$, (i) If $(p \geq 0)$, $T(n) = \theta(n^k * \log^p * n)$

(ii) If $(p \leq 0)$, $T(n) = \theta(n^k)$

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

$a = 2 \quad b = 2 \quad k = 1 \quad p = 1$

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

$a = 6 \quad b = 3 \quad k = 2 \quad p = 1$

$6 > 3^2$

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

Q. $T(n) = 2T(n/2) + \frac{n}{\log n}$

$a=2 \quad b=2 \quad k=1 \quad p=-1$ $\quad 2 \quad 2^1$

$\theta(n \log\log n)$

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

$a=3 \quad b=2 \quad k=2 \quad p=0 \quad 3 < 2^2$

$\theta(n^2)$

Master Theorem for Decreasing Recurrence

In dec f^n, we have the recurrence in the form of

$T(n) = aT(n-b) + f(n)$, where $f(n) = \theta(n^k)$

C1: If $a < 1$, then $T(n) = \theta(n^k)$

C2: If $a == 1$, then $T(n) = \theta(n * n^k)$

C3: If $a > 1$, then $T(n) = \theta(a^{n/b} \cdot n^k)$

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

$\theta(2^{n/2} \cdot n)$

Q. $T(n) = \frac{1}{2} T(n-1) + n^2$

$\theta(n^2)$

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

$\theta(n^3)$

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

$\theta(n * f(n))$

$\theta(n \log n)$

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

$\theta(3^{n/2} \cdot n)$

Scanned by Narsimha Rao

Divide & Conquer

Divide

~~The prob~~ Divide the problem into number of sub-problem.

Conquer

Solve the ~~solve~~ subproblem recursively.

Combine -

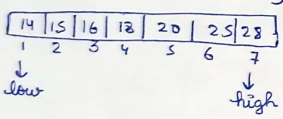
If require combine the solutions to get the final solutions

Eg - Binary Search, Quick Sort & Merge Sort

Binary Search

* Before applying BS, the elements should be in a sorted order.

eg. $A = \{14, 15, 16, 18, 20, 25, 28\}$



$Mid = (1+7)/2 = 4$

Compare (Key, A[4])

low = 1 , high = mid - 1

Recursive Algo

```
binary_Search (low, high, key){
    if (low == high){
        if (A[low] == key){
            return low;
        }
        else{
            return 0;
        }
    }
    else{
        mid = (low + high)/2;
        if (key == A[mid]{
            return mid; }
        else if (key < A[mid]){
            return binary_Search(low, mid-1, key);
        }
        else{
            return binary_Search(mid+1, high, key);
    }
}
```

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

$$T(n) = \begin{cases} 1, & n == 1 \quad // \text{Best Case} \\ T(n/2)+1 & n > 1 \quad // \text{Worst Case} \end{cases}$$

Quick Sort

Divide - Split the array into two sub-arrays such that each element in the left sub-array is less than or equal to middle element & each element in right sub-array is greater than the middle element.

Conquer - Recursively sort the two sub arrays.

Combine - Combine all the sorted elements to form the list of sorted elements.

eg. 50, 30, 10, 90, 80, 20, 40, 70 (0 1 2 3 4 5 6 7)

pivot, i, j

i = 1, j = 7, pivot = 50

i: 30 < 50 ⇒ i++ = 2; 10 < 50 ⇒ i++ = 3; 90 < 50 (False) ⇒ i = 3

j: 70 > 50 ⇒ j-- = 6; 40 > 50 (False) ⇒ j = 6

i with j: 3 < 6

Swap A[3] & A[6]

50 30 10 40 80 20 90 70

pivot, i = 3, j = 6

i: 40 < 50 ⇒ i++; 80 < 50 (F) ⇒ i = 4

j: 90 > 50 ⇒ j--; 20 > 50 (F) ⇒ j = 5

i < j

Swap A[4] with A[5]

50 30 10 40 20 80 90 70

pivot, i, j

i: 20 < 50 ⇒ i++; 80 < 80 (F) ⇒ i = 5

j: 80 > 50 ⇒ j--; 20 > 50 (F) ⇒ j = 4

As i < j is False

Swap pivot with A[j]

20 30 10 40 | 50 | 80 90 70

pivot

Now, the pivot element has been placed in its proper posⁿ.

Algm Quick Sort

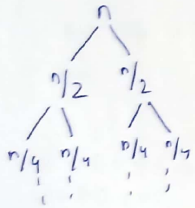
```
quick_Sort (A, low, high) // initially low = 0, high = n-1, n → no. of elements
{
    if (low < high) then
        m ← partition (A, low, high)
    quick_sort (A, low, m-1)
    quick_sort (A, m+1, high)
}
```

```
Algo Partition (A, low, high){
    pivot = A[low] , i ← low+1 , j ← high
    while (i <= j){
        while (A[i] <= pivot)
            i++
        while (A[j] > pivot)
            j--
        if (i <= j) then
            swap(A[i], A[j]);
    }
    swap ( pivot, A[j] );
    return j
}
```

Analysis of Quick Sort

(Time Complexity of Quick Sort depends upon the pivot element)

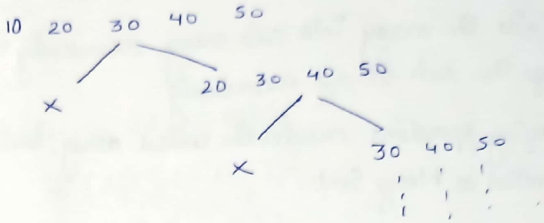
Best Case: (Middle)



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

$\theta(n \log n)$

Worst Case: (1st, last)

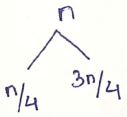


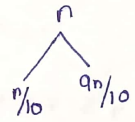
$T(n) = T(n-1) + n$

$\theta(n^2)$

Avg Case:

* pivot element not placed at middle, 1st or last index





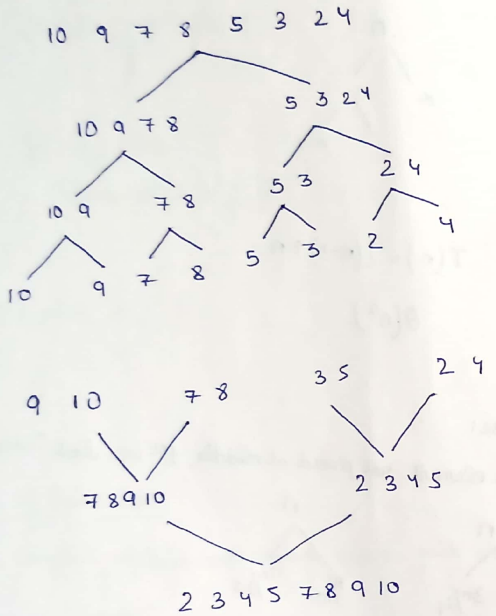
$T(n) = T(n/4) + T(3n/4) + n$

$T(n) = T(n/10) + T(9n/10) + n$

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

Merge Sort

* Divide the array into sub-array recursively
* Merge the sub-arrays recursively
* Merging operations creates the sorted array, hence it is called as Merge Sort.




```
Divide (low, high) {
    low = 0, high = n-1
    if (low < high) {
        mid = (low + high) / 2
        Divide (low, mid)
        Divide (mid+1, high)
        Merge ( low, mid, high)
    }
}
```

```
Merge (low, mid, high) {
    i = low;
    j = mid + 1;
    k = 0;
    while (i <= mid && j <= high) {
        if (A[i] <= A[j]) {
            B[k] = A[i];
            i++;
        }
        else {
            B[k] = A[j]
            j++;
        }
    }
    if (i > mid) {
        while (j >= high) {
            B[k] = A[j]
            j++;
            k++;
        }
    }
    else {
        while (i <= mid) {
            B[k] = A[i]
            i++; k++
        }
    }
}
```

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

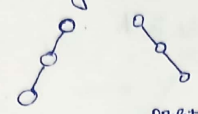
$\theta(n \log n)$

No, Best Case, Worst Case & Avg Case

Quick Sort is preffered over Merge Sort coz of Space Complexity

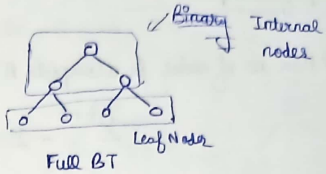
Heap

Binary Tree

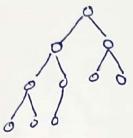


Left Skew Tree

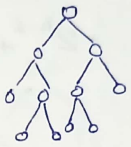
Right Skewed Tree



Full BT



Complete BT



Not a complete BT

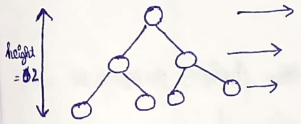
$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

$\log_2 n+1 = h+1 \log_2$

$\log_2 n+1 - 1$

Proof of Full Binary Tree –



Height of leaf node = 0

Height of any internal node = Longest path from that node to leaf node

Height of a FB tree = height of the root = $\log_2$ (no: of leaf nodes)

= $\log_2$ (no: of internal nodes + 1)

Max No: of nodes in a FBT = $2^0 + 2^1 + 2^2 + \dots + 2^h$

$= 2^{h+1} - 1$

Prop of Complete BT

* Max no: of nodes in a Complete BT whose height is 'h'

$$= 2^{h+1} - 1$$

* Min no: of nodes of a ~~co~~ CBT whose height is 'h'

$$2^0 + 2^1 + 2^2 \ldots + 2^{h-1} + 1$$

$$2^h$$

* If 'n' represents total no: of nodes of a CBT whose height is h then

$$2^h <= n <= 2^{h+1} - 1$$

* If there are n no of nodes in a CBT, then height is equal to

$$\log_2 n$$

Binary Heap

Heap is a CBT, where every parent is greater than or smaller than its children.

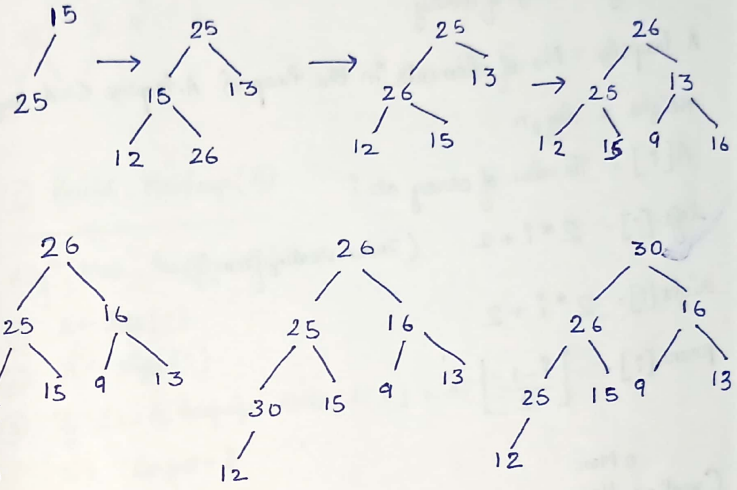
Heap is of 2 types - Max & Min Heap

In Max Heap, every parent is greater than its children & largest element among the all nodes is present at the root.

In Min Heap, every parent is smaller than its children & smallest element among the all nodes is present at the root.

eg Create a Max Heap

15, 25, 13, 12, 26, 9, 16, 30



The height of the heap with n no: of elements is equal to ~~log~~

$$h = \log_2 n$$

The minimum no: of nodes possible in a heap with height h ~~=~~

$$n = 2^h.$$

The max no: of nodes is $n = 2^{h+1} - 1$

If there are n no: of elements in a tree, then the children of sub-tree has the size at most $\lfloor 2n/3 \rfloor$

Heap Sort

A.length = size of array

A.heapsize = No: of elements in the heap & A.heapsize <= A.length

Height = $\log_2 n$

A[i] = The value of array at i

left[i] = 2*i + 1 (Index starting from 0)

right[i] = 2*i + 2

parent[i] = $\left\lfloor \frac{i-1}{2} \right\rfloor$

Creating a Max Heap

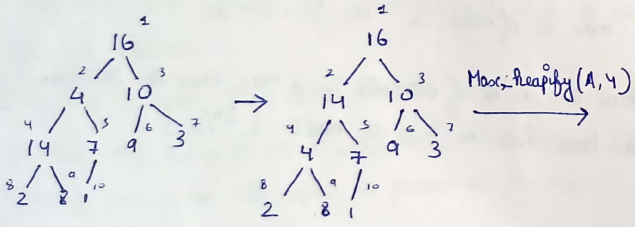
(i) Max_heapify (A, i)

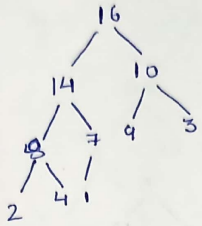
(ii) Build_Maxheap (A)

Max_heapify (A, i)

(i) Apply Max Heapify at (A, 2), if the Array is

<16, 4, 10, 14, 7, 9, 3, 2, 8, 1>





(2) Build_Maxheap (A)

Algm Max_Heapify (A, i)

(1) l ← left[i]

(2) r ← right[i]

(3) if l <= A.heapsize AND A[l] > A[i]

3.1 largest = l

(4) else

4.1 largest = i

(5) if r <= A.heapsize AND A[r] > A[largest]

5.1 largest = r

(6) if largest ≠ i

6.1 Exchange (A[i], A[largest])

6.2 Max_heapify (A, largest)

(7) Exit

Line (1), (2), (6.1) it takes a constant time i.e. O(1). So, the worst case run time will be when the recurrsion occurs maximal no: of times. As max no: of nodes in either side of the tree is $2n/3$, so the recurrence relation will be

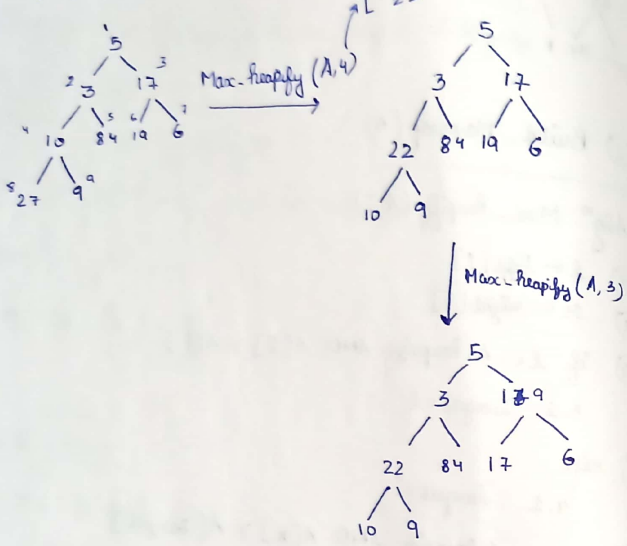
$$T(n) = T(2n/3) + 1$$

$$O(\log n)$$

Build_Maxheap(A)

eg. Apply Build_Maxheap( ) on A

⟨5, 3, 17, 10, 84, 19, 6, 27, 9⟩



Algm Build_Maxheap (A)

① for i ← ⌊A.heapsize/2⌋ to 1
  1.1 Max_heapify (A, i)

② Stop & Exit

} O(n log n)

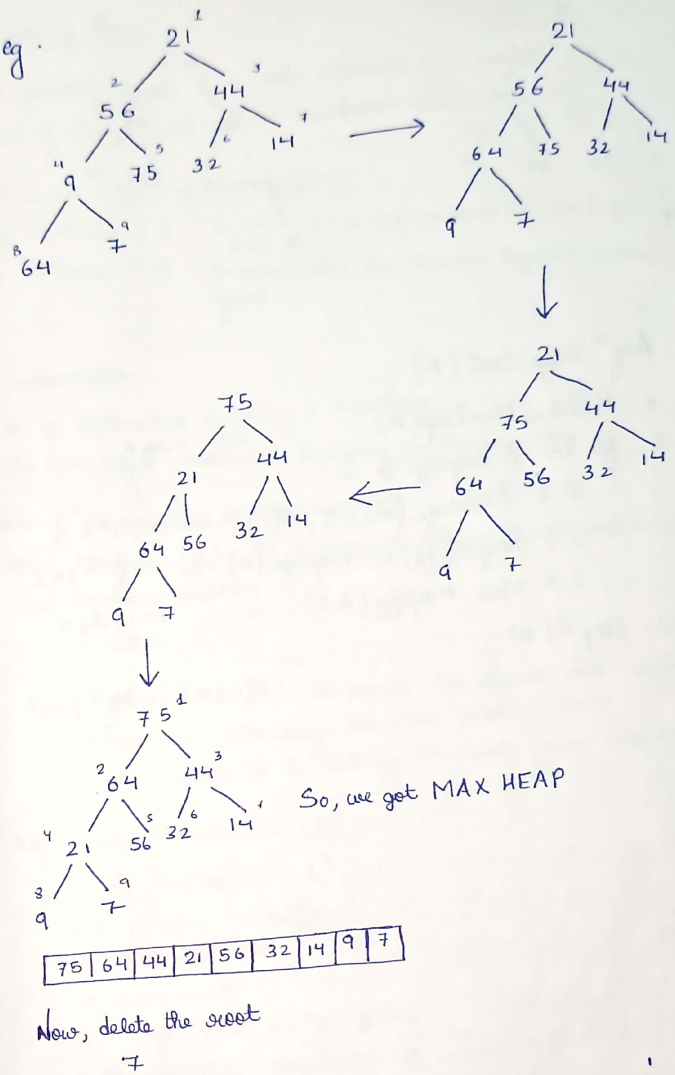
Deletion Operation of Heap

It consists of two steps -

(i) Swap the root node with last node

(ii) Apply the Heapify as needed for Max/Min Heap.

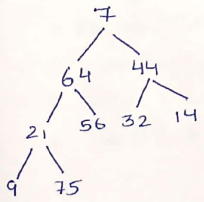
eg.



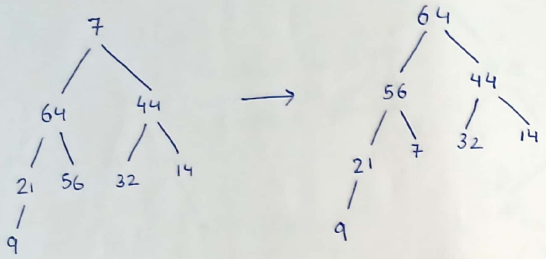
So, we get MAX HEAP

75	64	44	21	56	32	14	9	7

Now, delete the root



7	64	44	21	56	32	14	9	75



Algm Heap_Sort (A)

1. Build_Maxheap(A) — n log n
2. for i ← A.heapsize to 2 — (n-1)
   - 2.1 Exchange (A[1], A[i]) — (n-1) * 1
   - 2.2 HeapSize (A) ← HeapSize (A) - 1 — (n-1) * 1
   - 2.3 Max_Heapify (A, 1) — (n-1) log n
3. Stop & Exit

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

Priority Queue

A priority queue is a data structure for maintaining a set of element each with an associated value called key.

There are 2 types of priority queue -

1. Maximum P Q - The element which has maximum key value is processed first
2. Minimum P Q - The element which has minimum key value is processed first.

Implementation-

1. It is implemented by using a max heap.
2. The max heap is constructed depending on the key value.

Some fn or operation on Maximum PQ -

1. Heap_Maximum (A): It displays or returns the element which has maximum priority.
   return (A[1]).
2. Heap_Extract_Max (A): It process the element with maximum priority i.e. extracting the max priority element. In other words, it is deleting the root from max heap.

Algm Heap_Extract_Max (A)

1. If (heapSize[A] < 1)
   - 1.1. Display "Underflow"
   - 1.2 Return & Exit
2. Max ← A[1]
3. Swap (A[1], A[heapsize[A]])
4. Heapsize (A) ← Heapsize (A) - 1
5. Max_heapify (A, 1) → O(log n)
6. Return Max
7. Stop & Exit

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

Greedy Algorithm

- A greedy alg^m always makes the choice that looks best at that moment.
- Greedy Alg^m doesn't always give optimal sol^n, but for some problems they do.
- A Greedy Alg^m gets optimal sol^n to a problem by making sequence of greedy choices which seems to be best at that present state.
- The greedy approach may not guarantee or give optimal solution for all the problems.
- The problems which can be solved using greedy approach are -

→ Fractional Knapsack Problem
→ Activity Selection Problem
→ Huffman Code Problem.

Knapsack Problem

In the Knapsack problem, a person wants to choose some item in a bag (Knapsack) from the store, such that the cost of the bag is maximum.

- It is of two types -

0/1 knapsack - Here each item must be either taken or left behind. The optimal sol^n is achieved by Dynamic Programming.

Fractional Knapsack - Here the person can take fractions of item, rather than having to make a binary choice for any item. This can be solved using Greedy Alg^m

Fraction KnapSack Problem

Let $i = 1, 2, 3 \ldots n$ be the n items present in the store.

$V_i \leftarrow$ Value of the i^{th} item
$W_i \leftarrow$ Weight of the i^{th} item
$x_i \leftarrow$ Fractional amount of the i^{th} item where $0 \le x_i \le 1$
$k \leftarrow$ Maximum capacity of the KnapSack

So, we need Maximise Profit $= \sum_{i=1}^{n} V_i x_i$, which is subject to $\sum_{i=1}^{n} W_i x_i \le k$, $0 \le x_i \le 1$, $1 \le i \le n$

Eg. Let $W = \{40, 10, 30, 10, 40\}$, & the corresponding cost $V = \{200, 60, 120, 10, 80\}$, k is given as 5

Item	1	2	3	4	5
Weight (W_i)	40	10	30	10	40
Cost (V_i)	200	60	120	10	80
Cost/Unit (V_i/W_i)	5	6	4	1	2

Then, arrange (V_i/W_i) in descending order

Item	2	1	3	5	4
(W_i)	10	40	30	40	10
(V_i)	60	200	120	80	10
(V_i/W_i)	6	5	4	2	1

Item No	$W_i <= k$	x_i	k	Total cost
2	10 <= 85	1	85-10 = 75	1 * 60 = 60
1	40 <= 75	1	75-40 = 35	1 * 200 = 200
3	30 <= 35	1	35-30 = 5	1 * 120 = 120
5	40 <= 5 (False)	5/40 = 1/8	5-5=0	1/8 * 80 = 10
				390

Eg- $V = \{25, 14, 15\}$ $W = \{18, 15, 10\}$, k=20

Item	1	2	3
W_i	18	15	10
V_i	25	14	15
V_i/W_i	1.38	0.93	1.50

Item	3	1	2
W_i	10	18	15
V_i	15	25	14
V_i/W_i	1.50	1.38	0.93

Item No	$W_i <= k$	x_i	k	Total Cost
3	10 <= 20	1	20-10=10	1 * 15 = 15
1	18 <= 10	10/18	10-10=0	10/18 x 25 = 13.8
				28.8

Algm Knapsack_Fractional (W, V, n, k)

W → Array representing weight V → capacity of knapsack

n → no: of items k → capacity of knapsack

1. Compute V_i/W_i
   Arrange in descending order
2. for i ← 1 to n
   2.1 x_i ← 0
3. weight ← 0 , i ← 1 , cost ← 0
4. while (weight + w_i <= k)
   4.1. do x_i ← 1
   4.2 weight ← weight + w_i
   4.3 i ← i + 1
5. x_i ← (k - weight) / w_i
6. for i ← 1 to n
   6.1 cost ← cost + (x_i * V_i)
7. Return cost
8. Stop & Exit.

$O(n \log n)$

Activity Selection Problem

Let $A=\{a_1, a_2 \ldots a_n\}$ is a set of activities.

Let $S=\{s_1, s_2 \ldots, s_n\}$ is the starting time of activities & $F=\{f_1, f_2 \ldots f_n\}$ is the finishing time of activities

Activities a_i & a_j are compatible, if the time interval are not overlapped i.e. $s_i < f_i$ & $s_j < f_j$ & $s_j \geq f_i$

The activity selection problem is a problem to select maximum no. of compatible activities.

Steps to solve the problem:

S1: Arrange the activities in increasing order of finish time.

S2: Select the first activity.

S3: Select next activity whose start time ≥ finish time of previously selected activity

S4. Repeat S3 until all activities are checked.

Eg. $S_i = \{1, 0, 3, 3, 5, 5, 6, 8, 8, 2, 12\}$

$F_i = \{4, 6, 5, 8, 7, 9, 10, 11, 12, 13, 14\}$

Sol: Arrange all the activities in increasing order of finish time

A_i	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}
S_i	1	3	0	5	3	5	6	8	8	2	12
F_i	4	5	6	7	8	9	10	11	12	13	14
Selection	✓	×	×	✓	×	×	×	✓	×	×	✓

$\langle a_1, a_4, a_8, a_{11} \rangle$

Minimum No. of platform problem

Arrival & Depodion times of all trains are given to reach a railway station. The task is to find the minimum no. of platforms required so that no trains needs to wait.

Eg.

Arrival Time	900	940	950	1100	1500	1800
Departure Time	910	1200	1120	1130	1900	2000

900	A	1
910	D	0
940	A	1
950	A	2
1100	A	3
1120	D	2
1130	D	1
1200	D	0
1500	A	1
1800	A	2
1900	D	1
2000	D	0

3 is highest

No: of platforms needed = 3

- Sort all the times in increasing order
- If Arrival +1 else -1

Huffmann Code (Encoding technique)

There are two encoding techniques -

Fixed length code

Variable length code

Fixed Length Code:-

In a string for eg. we have abaab....., a is present 800 times & b is present 200 times. If it is represented in bits then we need (8x1000) bits of space. It can be reduced if a & b can be represented as 1 & 0. Then instead of 8000 bits, now we need only 1000 bits.

So, if there are n different characters, then no: of bits required to represent 1 character is $\lceil \log_2 n \rceil$.

Variable Length Code:-

1. Here we assign different no: of bits to each character.
2. But there should not be any ambiguity i.e. let the string is

   a b a a b c c

   a=0, b=01, c=10

   0 01 0 0 01 10 10

   then the encoded data may have lot of ambiguity while decoding.
3. The code word where no code word is a prefix of some other ~~word~~ code is called prefix code.

   0, 101, 100, 111, 1101, 1100

Huffmann Code follows prefix code properties.

Huffmann Code is generated from Huffmann tree.

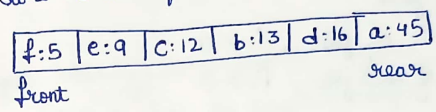
Huffmann tree is a binary tree in which 0 represents go to left & 1 represents go to right.

Eg.

a=45, b=13, c=12, d=16, e=9, f=5

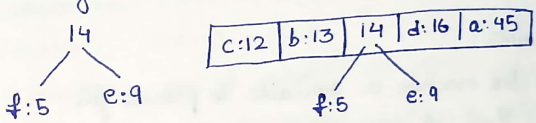
Arrange all characters in the ascending order of frequency & store it in a queue

f(n)-

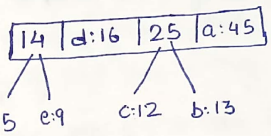


S1: Select the two smallest frequencies & combine it to one.

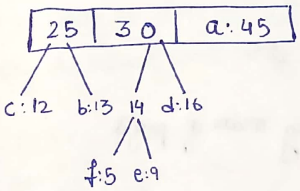
After combining



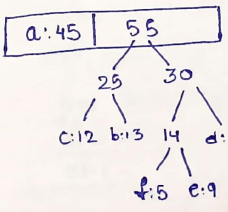
S2:



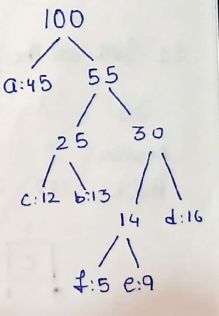
S3:



S4:



S5:



a=0

b=101

c=100

d=111

e=1101

f=1100

Job Sequencing Problem:

Given an array of jobs, where every job has a deadline & an associated profit if job is finished by the deadline

Every job takes a single unit of time i.e. maximum possible deadline for any job is 1 time unit.

Our objective is to maximise profit.

n: set of jobs
d_i: deadline of jobs, $d_i \geq 0$
P_i: Profit of each job, $P_i > 0$

For any job i, profit P_i is earned if and only if, the job is completed by deadline.

Constraints:

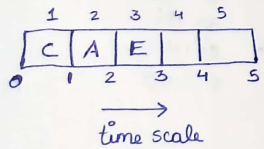
(i) One machine is available to process job.
(ii) Each job takes single unit of time to complete

Eg.

Job	A	B	C	D	E
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

S1. Sort all the jobs in descending order of profit

Job :	A	C	D	B	E
Deadline :	2	2	1	1	3
Profit :	100	27	25	19	15



100
+ 27
+ 15
142

S1. Sort the jobs in decreasing order of their profits
S2: Select the job with the highest profit
S3: Find the deadline of highest profit & put it in the index equivalent to deadline, if the slot is empty.
eg.
S4. If it is not empty, then put it in the previous one slot. If the index is greater than 0.
S5. Repeat from S2, until all the jobs have been processed.

*Longest Substring without repeating character

Given a string S, find the length of longest substring without repeating characters.

eg. S = <abcabcbb> , ans = 3, i.e. abc
S = <bbbbb> , ans = 1 , ie. b
S = <pwwkew> ans = 3 i.e. wke or kew

We will be using two pointers i & j which are initially at index 0.
We will be using frequency table or array which is used to store frequency of each char in string
As there are a total of 256 char in extended ASCII codes, so we will be using an array count 256. COUNT[256]
Initially the value of count array is initialise to zero.

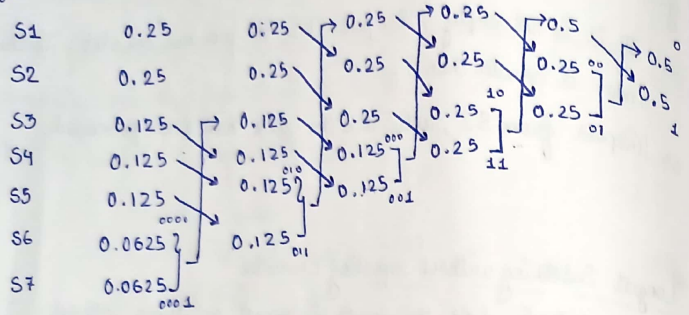
```
i=0 ; j=0 ; ans=0
for (j=0 ; j< s.size() ; ++j){
        count[s[j]]++;
        while(count[s[j]]>1){
                count[s[i]]--;
                i++;
        }
        ans = max(ans, j-i+1);
}
```

Q) S1 S2 S3 S4 S5 S6 S7

0.25 0.25 0.125 0.125 0.125 0.0625 0.0625

↳ Probabilities

Descending order ↑



S1 - 10

S2 - 11

S3 - 001

S4 - 010

S5 - 011

S6 - 0000

S7 - 0001

			Length
S1	0.25	10	2
S2	0.25	11	2
S3	0.125	001	3
S4	0.125	010	3
S5	0.125	011	3
S6	0.0625	0000	4
S7	0.0625	0001	4

L (Avg code length) $= \sum_{1}^{k} P_k L_k = (0.25 \times 2) + (0.25 \times 2) + (0.125 \times 3) + (0.125 \times 3) + (0.125 \times 3) + (0.0625 \times 4) + (0.0625 \times 4)$

$= 2.625$ bits/symbol

$H(x) = -\left(\sum_{1}^{k} P_k \log_2 P_k\right)$ // -ve has no value

$= P_1 \log_2 P_1 + P_2 \log_2 P_2 + \ldots + (P_7 \log_2 P_7)$

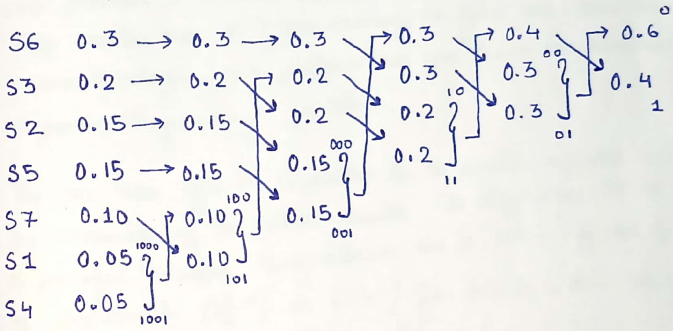
$= (0.25 \log_2 0.25) + \ldots + (0.0625 \log_2 0.0625)$

$= 2.625$ bits/symbol

Efficiency $= \frac{H}{L} \times 100 = \frac{2.625}{2.625} \times 100 = 100\%$

Q. Apply Huffmann code procedure for the following symbols & find the efficiency

$P = \langle 0.05, 0.15, 0.2, 0.05, 0.15, 0.3, 0.1 \rangle$



S1 - 1000

S2 - 000

S3 - 11

S4 - 1001

S5 - 001

S6 - 01

S7 - 1001

$H(x) = 2.571$

$L = 2.6$

Efficiency $= \frac{H(x)}{L} \times 100$

$= \frac{2.571}{2.6} \times 100$

$= 98.85\%$

Huffman (C)

1. n = |C| // no of characters
2. Build MinHeap Q with C
3. for i ← 1 to (n-1)
   - 3.1 Allocate a new node z
   - 3.2 z→left = x = Extract_min(Q)
   - 3.3 z→right = y = Extract_min(Q)
   - 3.4 z→freq = x→freq + y→freq
   - 3.5 Insert(Q, z)
4. Return Extract_min(Q)

DYNAMIC PROGRAMMING

"Those who can not remember the past are condemned to repeat it"

"Is repeating the things for which you already have the answer, A Good thing?"

A programmer would disagree. That's what Dynamic Programming is about to always remember answers to the subproblem that you have already solved.

Given a problem which can be broken down into smaller sub-problem & smaller sub-problems can still be broken into smaller ones & if you manage to find out there are some overlapping sub-problem then you have encountered a Dynamic Programming problem.

The core idea of a DP problem is to avoid repeated work by remembering the partial results. In algorithm, DP is a powerful technique that allows one to solve different types of problem in $O(n^2)$ or $O(n^3)$ for which a naive approach would take exponential time.

eg. 1+1+1+1+1 = on a paper, the answer is 5 here. Now write +1 in the left. So the new answer is now 6. Here we do not require recount because we remembered their were 5 no of ones.

eg. Fibonanci series

fibo(n) = 1, if n=0
fibo(n) = 1, if n=1
fibo(n) = fibo(n-1) + fibo(n-2)