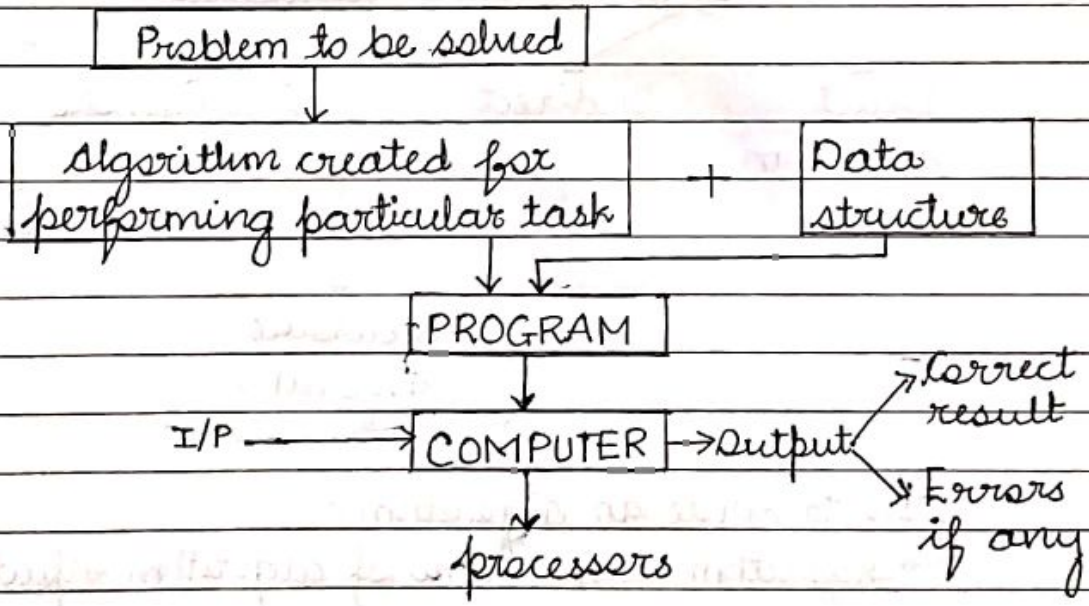
UNIT 1:

chapter 1

Introduction to Algorithms

Algorithm:
An algorithm is a logically arranged sequence of finite number of instructions which when implemented solves a given problem.

Problem → Algorithm → Efficient Algorithm.
(refining)
- → Less time taken
- → Less space taken
- → Accept all valid input



Criteria for an Algorithm/Algorithm characteristics steps

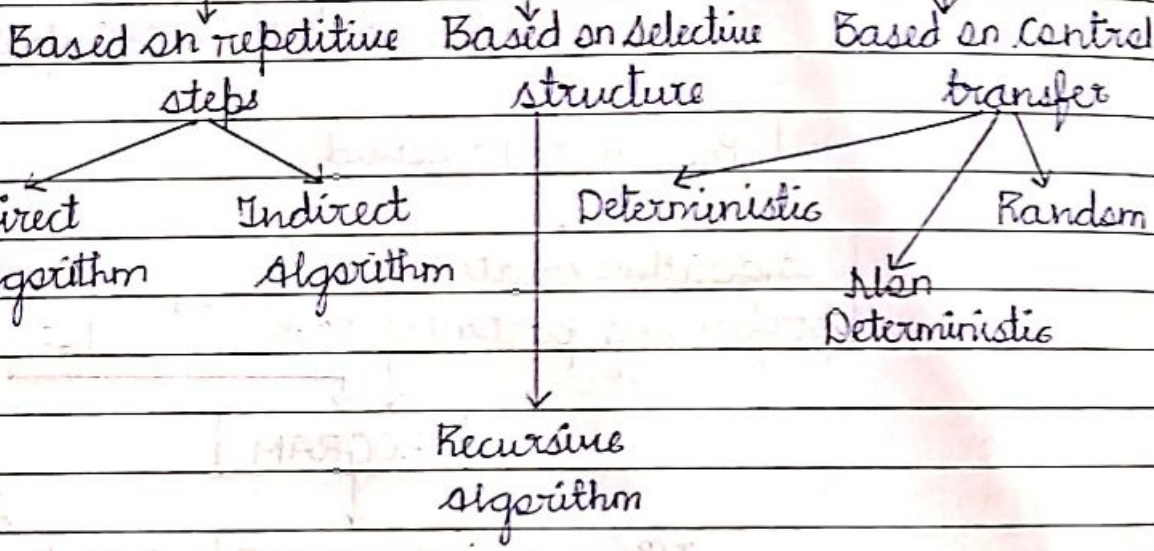
1) Finiteness: An algorithm should terminate in finite number of steps.
2) Definiteness: (No Ambiguity) It should be clearly and precisely defined.
3) Input: It should have zero or more but finite inputs. Hello world program is example of zero input program.
4) Output: Atleast one output should be present.

5) Effectiveness: Each step should be referred as principle and should be executing in finite time.

6) Feasibility: It should work effectively with available ressources.

7) Independence: An algorithm is independant of programming language.

Classification of Algorithms



How to write an algorithm?

↳ Algorithm head - name of algorithm input

Algorithm Analysis (Performance Analysis)

The efficiency of an algorithm can be decided by measuring performance of the algorithm.

Complexity is the function describing efficiency of the algorithm.

Complexity of an algorithm is the the measure of amount of time and memory space required by the algorithm for input of given size n.

Complexity can be classified as:

a) Space complexity

It is defined as amount of memory space required by an algorithm to run to completion.

- It is denoted by S(P)

$$S(P) = C + S_P$$

where C = constant/fixed part

S_P = instance char/ variable char

C denotes space of inputs and outputs.

S_P is the space dependent upon instant characteristic. This is a variable part whose space requirement depends on particular problem being solved, and the recursion stack space.

1) Algorithm abc(a,b,c)

{

return (a+b+c)/(a*b*c).

}

S(abc) = 3 words

S(abc) = sizeof(a) + sizeof(b) + sizeof(c)

Assume 1 word for a variable

O(1) is the efficiency since space remains the same

2) Algorithm sum(a, b)

{

```
    c = a+b+10;
}
```

S(sum) = 3 variables + 1 constant = 4

S(sum) = sizeof(a) + sizeof(b) + sizeof(c) + sizeof(10)

3) Algorithm sum(a,n)

```
{
    s=0;
    for i=1 to n do
        s = s+a[i];
    return s;
}
```

S(sum) = ?

Assume 1 word for one variable

i → 1 word

n → 1 word

s → 1 word

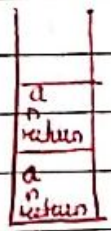
Array a → n

S(sum) = n+3

O(n).

4) Algorithm RSum(a,n)

```
{
    if (n<=0) then
        return 0;
    else
        return Rsum(a,n-1) + a[n];
}
```



Stack for AR of recursive function

space complexity, S(RSum) = 3(n+1)

= 3n+3

O(n) → always consider the higher component for big O notation.

b) Time Complexity

Amount of time required by the algorithm to run to its completion.

It is denoted by T(P)

T(P) = Sum of compile time + run time

1) step count Method

Introduce a new variable count initialized to zero. Statement to implement count by appropriate amount are introduced . This is done so that each time a statement in original program is executed, the count is incremented by the step count of that statement.

1. Algorithm Sum(a, n)

```
{  //count = 0
   S = 0 ------> 1
   //count = count + 1
   for (i = 1 to n) do ------> (n+1)
   {
      //count = count + 1
      S = S + a[i]; ------> n
      //count = count + 1
   }
   //count = count + 1
   return S; ------> 1
   //count = count + 1;
}
```

Total time complexity = 2n + 3

= O(n)

(OR)

Algo sum(a, n)

```
{
    for (i=1 to n) do count=count+2
    count = count +3
}
```

2. `for (i=0; i<n; i++)` → (n+1)

```
{
    cout<<i;  → n
    p=p+i;    → n
}
```

Total time complexity = 3n+1

2) Step Table Method:

1.

1	Algorithm Sum(a,n)	
2	{	
3	s = 0	
4	for (i=1 to n) do	
5	s=s+a[i]	
6	return s	
7	}	

The second method is the step table method used to determine step count of a program by using a step table in which total number of steps contributed by each statement is listed. The table is first obtained by determining number of steps per execution of statement and total number of store each statement is executing. By combining these two quantities the total contributions of each statement is obtained. By adding up contributions of all statement, the step count for entire program is obtained

line	s/e	frequency	total steps
1	0	–	0
2	0	–	0
3	1	1	1
4	1	n+1	n+1
5	1	n	n
6	1	1	1
7	0	–	0

Time complexity = 2n+3
⇒ O(n).

Find the time complexity of following algorithm.

```
1 Algorithm add(a[][],b[][],c[][],m,n)
2 {
3   for(i=1 to m) do
4     for(j=1 to n) do
5       c[i][j]= a[i][j] + b[i][j];
6 }
```

line	s/e	frequency	total steps
1	0	–	0
2	0	–	0
3	1	m+1	m+1
4	1	m(n+1)	m(n+1)
5	1	mn	mn
6	0	–	–

Total time complexity = 2mn + 2m + 1
O(mn)

Find

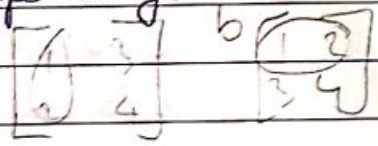
```
while(n>0)
{
  j=j+n;
```

```
    n = n+1;
}
```

line	s/e	frequency	total steps
1	1	n+1	n+1
2	0	—	0
3	1	n	n
4	1	n	n
5	0	—	0

Total time complexity = 3n+1
O(n)

write algorithm to multiply two matrices and find the time complexity and space complexity.




```
for (i=1; i<=n; i++)
{
    for (j=1; j<=n); j++)
    {
        c[i][j] = 0;
        for(k=1; k<n); k++)
        {
            c[i][j] = c[i][j] + (a[i][k] * b[k][j] + b[i][j]);
        }
    }
}
```

To find time complexity

line	s/e	frequency	total steps
1	0	—	0
2	0	—	0
3	1	n+1	n+1
4	0	—	0
5	1	n(n+1)	n(n+1) = n²+n
6	0	—	0
7	1	n²	n²

8	1	$n^2(n+1)$	n^3+n^2
9	0	–	0
10	1	n^3	n^3
11	0	–	0
12	0	–	0
13	0	–	0
14	0	–	0

Total time complexity $= 2n^3+3n^2+2n+1$

$O(n^3)$.

To find space complexity:

Since 2 D use n^2 for a, b, c

a → n^2

b → n^2

c → n^2

i → 1

j → 1

k → 1

n → 1

Total space complexity $= 3n^2+4$

```
1  Algorithm RSum(a,n)
2  {
3     if (n ≤ 0) then
4        return 0 ;
5     else
6        return [R Sum(a,n-1) + a [n]] ;
7  }
```

line	s/e	freq		total steps	
		n ≤ 0	n > 0	n ≤ 0	n > 0
1	0	–	–	0	0
2	0	–	–	0	0	

3	1	1	1	1	1
4	1	1	—	1	—
5	0	—	—	0	0
6	1+#(Rsum(n-1))	0	1	0	1+#(Rsum(n-1))
7	0	—	—	0	0
	Total time complexity			2	2+#(Rsum(n-1))

$$T_{Rsum}(n) = \begin{cases} 2 & n \le 0 \\ 2+\#(Rsum(n-1)) & n > 0 \end{cases}$$

write the algorithm to find factorial of number and find time complexity using recursion.

```
Algorithm fact(n)
{
    // Description
    // Input : n is +ve integer
    // Output : factorial of n
    if (n == 0)
        return 1;
    else
        return [n * fact(n-1)];
}.
```

line	s/e	freq n=0	freq n>0	total steps n=1	total steps n>1
1	0	—	—	—	—
2	0	—	—	—	—
3	1	1	1	1	1
4	1	1	—	1	—
5	0	—	—	0	0
6	1+#(fact(n-1))	0	1	0	1+#(fact(n-1))
7	0	—	—	0	0.
	Total time complexity=			2	2+#(fact(n-1))

Mathematical Analysis of Non-Recursive Algorithm

steps:

1) Decide on parameter indicating input size.
2) Identify the basic operation to complete the task.
3) Count the number of times the basic operation is executed.
4) Set up a sum reflecting a number of time a basic operation is executed.
5) Solve the sum and define the order of growth.

Time complexity of algorithm using operation Count

1. Write an algorithm to find the value of the largest element in a list of n numbers and analyse it.

```
ALGORITHM Max(A[0,---,n-1])
// Determines the largest element in the list
// Input: Array A [0,...,n-1]
// Output: The value of largest element in A, max
{
    max ← A[0]
    for(i from 1 to n-1)
        if(A[i] > max)
            max ← A[i]
    return max;
}
```

Soln:

step1: Parameters identifying size of the problem
The number of elements, n determines the input size.

step2: Basic operation → Basic operations are the operations in the innermost loop.
key comparison that is A[i] > max

step3: Count the number of times
Number of key comparisons made } = n-1.

Let C(n) denote the number of times the comparison is executed.

Step 1:

$$C(n) = \sum_{i=1}^{n-1} 1$$

$$= \underset{UL}{(n-1)} - \underset{LW}{(1)} + 1$$

$$= n-1$$

∴ Time complexity = n−1

O(n) = n.

2. write an algorithm to multiply two matrices and ~~analyze~~ multiply two matrices using operation count.

Soln:

```
ALGORITHM mult(a[][],b[][],n)
{
                                    //purpose
                                    //Input: n-size of array
                                      a- first array of size nxn
                                      b- second.
                                    //output: resultant matrix
  for(i=1 to n) do
    for(j=1 to n) do
    {
      c[i][j]=0;
      for(k=1 to n) do
      {
        c[i][j] = c[i][j]+(a[i][k]*b[k][j]);
      }
    }
}
```

step1: Parameters identifying size of the problem.
The number of elements 'n' determines input size.

step2: Basic operation
c[i][j]=c[i][j]+(a[i][j]*b[i][j]).
multiplication operation in the innermost loop.

Step3: count the number of times
Number of times multiplication statement executed } = n*n*(n) = n^3
It depends on value n.

Step4:

$$T(n) = \sum_{i=0}^{n-1} \sum_{i=0}^{n-1} \sum_{i=0}^{n-1} 1$$

$$= \sum_{i=0}^{n-1} \sum_{i=0}^{n-1} (n-1-0+1)$$

$$= \sum_{i=0}^{n-1} \sum_{i=0}^{n-1} n$$

$$= \sum_{i=0}^{n-1} n(n-1-0+1)$$

$$= \sum_{i=0}^{n-1} n^2$$

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

$$= n^3$$

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

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

Mathematical Analysis of Recursive Algorithm.

steps:
1) Based on the input size decide the various parameters to be considered.
2) Identify the basic operation in the algorithm.
3) Obtain the number of time basic operation is executed on different inputs of same size.
4) Obtain a recurrence relation with an appropriate initial condition

5) Solve the recurrence relation and obtain the order of growth ($O(n), \Omega(n)$)

1 Write an algorithm to find factorial of number and find its time complexity.

Soln: Algorithm fact(n)
//Purpose: To find factorial of a number
//Input: n: positive Integer
//Output: Factorial of number n.

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

Analysis:

step1: Identify parameter describing input size n determine size of the program.

step2: Basic operation.
Multiplication statement during recursive call.

Number of executed statements depends on n.

step3: Total number of multiplications can be obtained using recurrence relation as follows:-

$$T(n) = \begin{cases} 0 & n=0 \\ 1+T(n-1) & n>0 \end{cases}$$

The above recurrence relation can be solved using repeated or backward substitution.

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

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

$= 2 + T(n-2)$

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

$= 3 + T(n-3)$

$\vdots$

$= i + T(n-i)$

Let $i = n$,

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

$= n + T(0)$

$= n + 0$

$= n$

$T(n) = O(n)$

2 write an algorithm for tower of hansi problem

Soln: Algorithm TowerOfHansi(n, src, temp, dest)

//Purpose: To move n disc from source to destination and see that only one disc is moved and always the smaller disc should be placed above larger disc. using one of the needle and temperary holder as the disc being moved.

//Inputs: n = total number of disc to be moved.

//Output: All n disc are available on the destination need.

```
{
  if (n==1)
    Move the disk from source to dest
  else
    TowerOfHansi (n-1, src, dest, temp)
    Move the disk from source to dest
    TowerOfHansi (n-1, temp, src, dest)
}
```

Analysis :

step 1: Identify the parameters describing the size of algorithm.

step 2: Basic operation
Here the disc from source to destination.

step 3: Total number of disk moments can be obtained by following the recurrence relation.

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

(1 → Source → dest; T(n-1) → Number of movements from temp to destination; T(n-1) → Number of movements from source to temp)

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

$$T(n) = 2^i \cdot T(n-i) + 2^{i-1} + 2^{i-2} + \dots + 2^2 + 2 + 1 \quad \text{— (1)}$$

Sum of Geometric series $S = \frac{a(r^n - 1)}{r - 1}$

Here $a = 1 \quad r = 2 \quad n = i$

$$S = 1\left(\frac{2^i - 1}{2 - 1}\right)$$
$$= 2^i - 1$$

Substitute $2^i - 1$ in eqn (1)

$$T(n) = 2^i \cdot T(n-i) + 2^i - 1$$

Let $i = n - 1$

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

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

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

Differentiate between Priori Analysis and Posteriori Analysis (Before implementation / After implementation)

Priori Analysis	Posteriori Analysis
1) Analysis of an algorithm prior to running on any system.	1) Analysis of algorithm only after running a specific system.
2) Independent of language of compiler and type of hardware.	2) Depends on language of compiler and type of hardware.
3) It gives approximate value.	3) It gives exact answer.
4) Time complexity for an algorithm is same for every system.	4) Time complexity for algorithm differ from system to system.
5) It use asymptotic notation to represent how much time algorithm will take to complete its execution	5) It does not use asymptotic notation to represent time complexity of the algorithm

Order of Growth

Order of growth means how an algorithms time and space is going to increase or decrease when the size of the input is increased or decreased. This change in behaviour as the value of n increases is called order of growth.

Input size ↓ | Time ↓

N	logN	N	2^N	N!
1	0	1	2	1
2	1	2	4	2
~~4~~	2	4	16	24
~~8~~	3	8	256	40320
16	4	16	6553	high
32	5	32	4294967296	very high

↳ Fastest variable

1: Indicates the running time of the algorithm is constant

logN: Indicates that running time of the program is logarithm. Ex: Binary search

N: Indicates that the running time of the algorithm is linear. Ex: Linear Search

NlogN: Indicates that
Ex: quicksort, mergesort

N^2: Indicates running time is quadratic.
Ex: Bubble sort, Selection sort, addition of two matrices.

N^3: Indicates that the running time is cubic.
Ex: Multiplication of two matrices.

2^N: Indicates that the running time is exponential.
Ex: Subsets of given sets.

N!: Indicates running time is factorial.
Eg: algorithms that generate permutation of sets like caeser cipher, Brute force techniques, etc...

$1 < \log N < N < N\log N < N^2 < N^3 < 2^N < N!$

lowest to highest

Efficiency of algorithm

Worst case: The efficiency of an algorithm for an input size n for which algorithm takes longest time to execute among all inputs is called worst case efficiency.
Ex: Linear search wherein item is not present
O(n).

Best case: The efficiency of an algorithm for the input of size n for which the algorithm takes least time during execution among all possible inputs is called best case efficiency.
Ex: Linear Search wherein item is present at the beginning. O(1).

Average case: In average case efficiency the average number of basic operations are executed or taken on any instance of size n.
Ex: Linear search wherein item is present somewhere in between.

$$\frac{\text{All possible case time}}{\text{no. of cases}}$$

1, 2, 3, 4, 5, 6 $= \frac{n(n+1)}{2} = \frac{(n+1)}{2} = O(n)$.

n

Asymptotic Notations

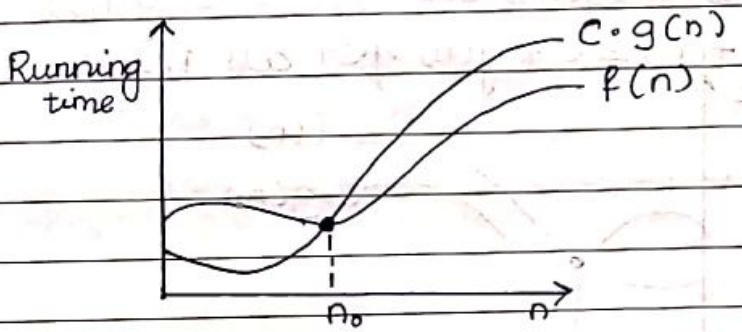
- Asymptotic behaviour of a function is the study of how the value of the function varies for large value of n where n is the size of the input.
- They are mathematical tools to represent the

time efficiency of algorithm for asymptotic analysis.

1) Big Oh Notation (O)
- It represents upper bound of the algorithm.
- longest time taken by an algorithm to complete
- It always give the worst case.

$f(n) \in O(g(n))$ or $f(n) = O(g(n))$

iff $f(n) \leq c * g(n)$ $\forall n \geq n_0$
(c ← constant)



c·g(n) always comes above f(n) after exceeding certain point n_0 such that $n > n_0$

1. Prove that $3n+2 = O(n)$.

Proof:
We know that

$f(n) \in O(g(n))$

iff $f(n) \leq c \cdot g(n)$ $\forall n \geq n_0$

Given $f(n) = 3n+2$,

Replace 2 with n (next higher order of number 2)

$f(n) = 3n+n$

$= \frac{4}{c} \cdot \frac{n}{g(n)}$

To prove: $f(n) \leq c \cdot g(n)$

$3n+2 \leq 4n$ Here, $c = 4$

$n=1$: $3(1)+2 \leq 4(1)$, $5 \leq 4$ ✗

$n=2$: $3(2)+2 \leq 4(2)$, $8 \leq 8$ ✓

$n=3$: $3(3)+2 \leq 4(3)$, $11 \leq 12$ ✓

$n=4$: $3(4)+2 \leq 4(4)$, $14 \leq 16$ ✓

It is clear from above relation that $c=4$, $n_0=2$ and

$g(n) = n$. So by definition $f(n) \in O(n)$

i.e: $3n+2 \in O(n)$.

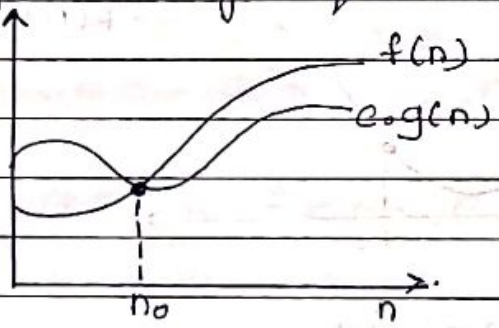
Hence proved.

2) Big Omega Notation (Ω)

- To express the best case and lower bound of the algorithm.
- It gives least amount of time the program takes.

$f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$

iff $f(n) \geq c * g(n)$ for all $n \geq n_0$.



Prove that $10n^2+3n+3 = \Omega(n^2)$

Proof:

The constraint to be satisfied is

$f(n) \geq c * g(n)$ for $n \geq n_0$

$10n^2+3n+3 \geq n^2$

Here we consider $c=1$

i.e $10n^2+3n+3 \geq 1 \cdot n^2$

$n=1$: $16 \geq 1$ ✓

$n=2$: $49 \geq 4$ ✓

It is clear from above relation that $c=1$, $n_0=1$, and $g(n) = n^2$.

3 Theta Notation (θ Notation)

→ Average

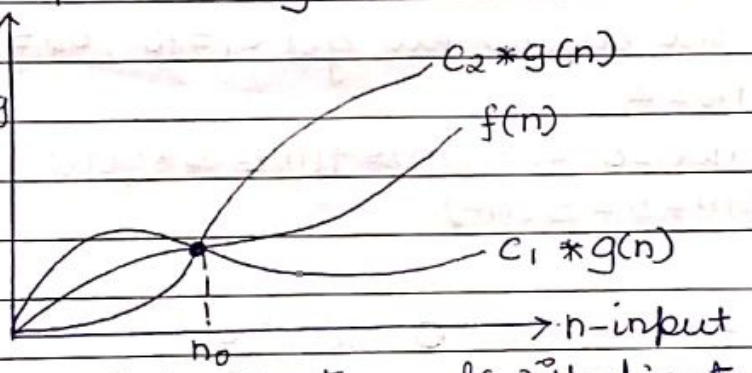
→ Let $f(n)$ be the time complexity of algorithm. The function $f(n)$ is said to be big theta of $g(n)$ denoted as $f(n) = \theta(g(n))$

or

$f(n) \in \theta(g(n))$

such that there exists some positive constant c_1, c_2 and a non-negative integer n_0 satisfying the constraint

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall n \geq n_0$$



The upper bound indicates on $f(n)$ indicates that function $f(n)$ will not consume more than the specified time $c_2 * g(n)$.

The lower bound indicates on $f(n)$ that the function $f(n)$ is the best case will consume atleast the specified time $c_1 * g(n)$

Let $f(n) = 10n^3 + 5$. Express the function using θ-notation

Soln: The constraint to be specified is $c * g(n) \leq f(n) \leq c_2 * g(n)$

$\forall n \geq n_0$

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

$10n^3 \leq 10n^3 + 5$

Here $c_1 = 10$ $g(n) = n^3$

$n = 0, 0 \leq 5$ ✓ But don't consider n_0 as 0.

$n = 1, 10 \leq 15$ ✓

$n = 2, 80 \leq 85$ ✓

$c_1 = 10, g(n) = n^3, n_0 = 1$

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

Consider $f(n) = 10n^2 + 5$

$= 10n^3 + n^3$ (Replace 5 with n^3)

$= 11n^3$

$= c_2 * g(n)$

Here $c_2 = 11$ and $g(n) = n^3$.

$n = 0, 10(0)^3 + 5 \leq 11(0)^3 ? \; 5 \leq 0$ ✗

$n = 1, 10(1)^3 + 5 \leq 11(1)^3 ? \; 15 \leq 11$ ✗

$n = 2, 10(2)^3 + 5 \leq 11(2)^3 ? \; 85 \leq 88$ ✓

$n = 3, 10(3)^3 + 5 \leq 11(3)^3 ? \; 275 \leq 297$ ✓

From the above we get $c_1 = 10$, $c_2 = 11$, $g(n) = n^3$ and $n_0 = 2$.

and hence $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

i.e. $10n^3 + 5 = \Theta(n^3)$

4 Little Oh Notation (o Notation)

- Denotes upper bound asymptotically tight.

$f(n) = o(g(n))$ for any positive constant $c > 0$ there exists a constant $n_0 > 0$ such that

$0 \leq f(n) < c \cdot g(n)$

Mathematically,

In the little oh notation the function $f(n)$ becomes insignificantly relative to $g(n)$ as $n \to \infty$ i.e.

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

E.g: $f(n) = n \quad g(n) = n^2$

$$\lim_{n \to \infty} \frac{n}{n^2} = \lim_{n \to \infty} \frac{1}{n} = \frac{1}{\infty} = 0$$

$f(n) = o(g(n))$

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

$$\lim_{n\to\infty} \frac{\log n}{n} = \frac{\log \infty}{\infty}$$

$$= \lim_{n\to\infty} \frac{\frac{d}{dn}\log n}{\frac{d}{dn} n} \quad \text{(Using L'Hospitals Rule)}$$

$$= \lim_{n\to 0} \frac{1}{n}$$

$$= \frac{1}{\infty} = 0$$

$\therefore f(n) = o(g(n))$

Differentiate between Little oh and Big O Notation

The difference between little oh and Big O is as follows.

If $f(n) = O(g(n))$, the bound $0 \le f(n) \le c * g(n)$ holds for some constant c. whereas

$f(n) = o(g(n))$, the bound $0 \le f(n) \le c * g(n)$ holds for all constant c.

Example 1:

1. $2n = O(n)$

$2n < c \cdot n$

$2n \le 3 \cdot n$

$c = 3$ &

$c > 0$

some

$2n = O(n)$

$2n = o(n)$

for all $c > 0$

1, 2, 3

$2n < c \cdot g(n)$

$2n < 1 \cdot n$ x

$2n < 2 \cdot n$

$2n \neq o(g(n))$

Example 2:

$2n = o(n^2)$

for all $c > 0$

1, 2, 3

$2 \cdot n < c \cdot g(n)$

$2n < 2n^2$

$2 < 2x$

$3n+5 = O(n^2)$ then prove than prove that $3n+5 = o(n^2)$

Proof:

Given $f(n) = 3n+5$

$g(n) = n^2$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{3n+5}{n^2} = \frac{3n}{n^2} + \frac{5}{n^2} = \frac{3}{n} + \frac{5}{n^2}$$

$$= \lim_{n\to\infty} \frac{3}{n} + \frac{5}{n^2}$$

$$= \frac{1}{\infty}$$

$$= 0$$

$3n+5 = o(n^2)$.

Proof:

$f(n) = 3n+5$

$g(n) = n$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n+5}{n}$$

$$= \lim_{n\to\infty} 3 + \frac{5}{n}$$

$$= 3 \neq 0$$

$3n+5 \neq o(g(n))$

5 Little omega Notation (ω notation).

$f(n) = \omega(g(n))$ iff $\forall c > 0, \exists n_0 > 0$ such that $0 \le c * g(n) < f(n) \ \forall n \ge n_0$

Mathematically,

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

(OR)

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

1. Find the little omega notation for $f(n) = 10n^2 + 7$

Given

$f(n) = 10n^2 + 7$

$g(n) = n$

$f(n) = 10n^2 + 7$

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = \frac{n}{10n^2+7} = \lim_{n \to \infty} \frac{1}{20n} = 0.$$

$\therefore f(n) = \omega(g(n)) n^2 + n^2 = 11n^2$

$\omega = 11.$

2 Is $3^n = \omega(2^n)$?

$$\lim_{n \to \infty} \frac{3^n}{2^n} = \left(\frac{3}{2}\right)^n = \lim_{n \to \infty} (1.5)^{\infty}$$

$= 1.5^{\infty}$

$= \infty.$

Yes.

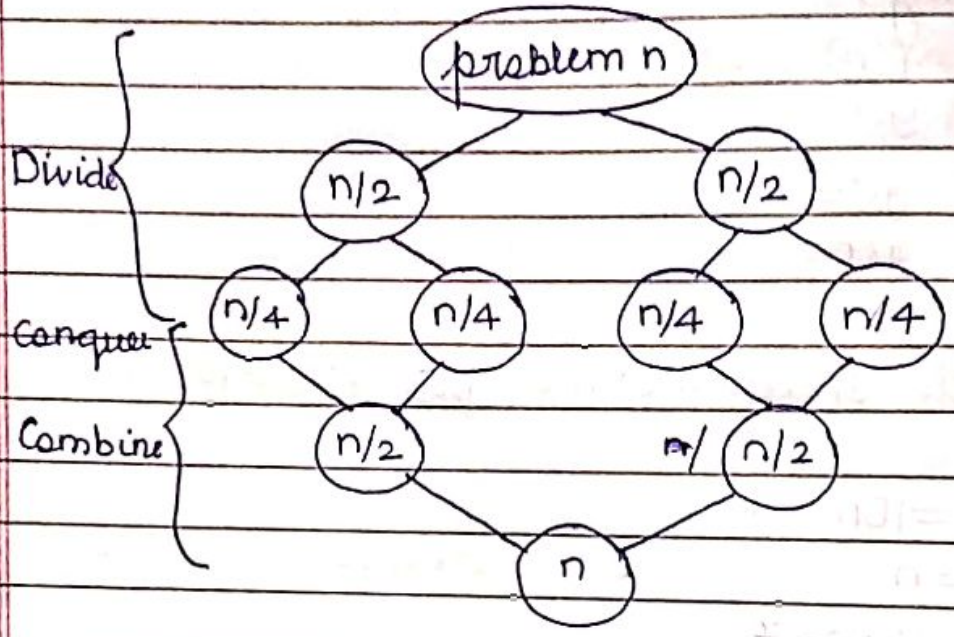
3 Is $n^2 = \omega(\log n)$?

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

$\therefore n^2 = \omega(\log n)$

Divide and Conquer

-top down technique



Control Abstraction for Divide and Conquer:
Algorithm D.

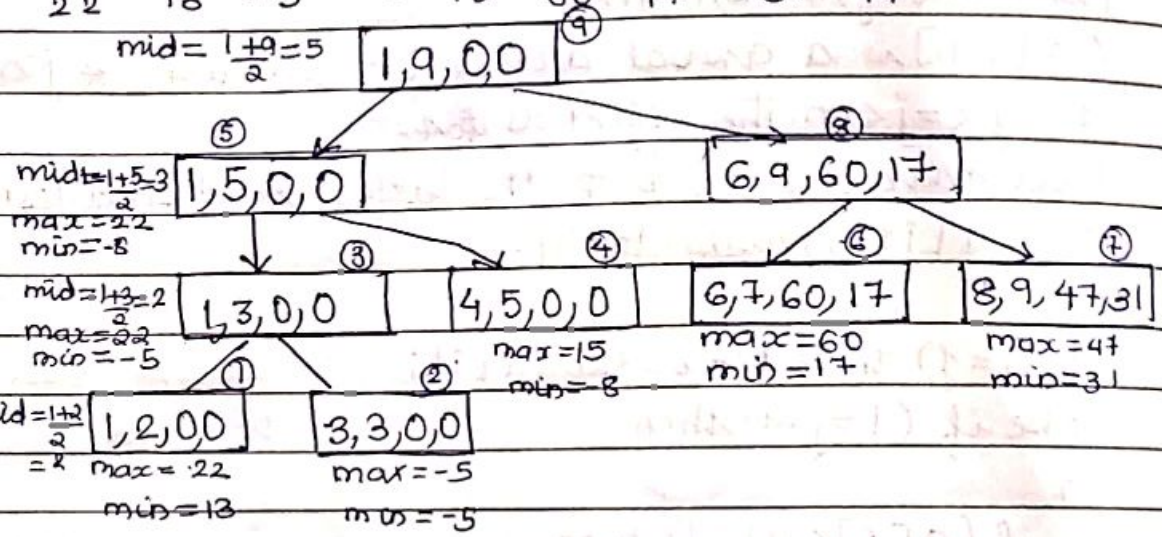
Finding Min and Max

```
MaxMin (i, j, max, min)
// a[1:n] is a global array, parameters i & j are integers,
1<=i<=j<=n. The effect is to4.
// set max and max to the largest and smallest value
5 in a[i:j], respectively.
{
 If (i=j) then Max=Min=a[i];        Consider floor value
 Else if (i=j-1) then               for mid.
 {
   if (a[i]<a[j]) then
   {
      max =a[j];
      min =a[i];
   }
   else
   {
      max =a[i];
      min =a[j];
   }
 }
 Else
 {
   Mid = (i+j)/2;
   MaxMin (i, Mid, Max, Min);
   MaxMin (Mid+1, Max1, Min1);
   If (Max<Max1) then Max = Max1;
   If (Min > Min1) then Min = Min1;
 }
}
```

Example:

A: 1 2 3 4 5 6 7 8 9

22 13 -5 -8 15 60 14 31 47



Analysis:

1) Parameter indicating size: n - number of elements in array.

2) Basic operations - key comparisons

3) Number of comparison

$$T(n) = \begin{cases} 0 & ; n = 1 \\ 1 & ; n = 2 \\ T(n/2) + T(n/2) + 2 & ; n > 2 \end{cases}$$

4) Solve recurrence relation.

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

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

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

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right) + 2\right] + 2^2 + 2$$

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

$$\vdots$$

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

Let $2^i = n$

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

$$= 2^{i-1} \cdot T(2) + 2^{i-1} + \ldots + 2^2 + 2^1$$

$$= 2^{i-1} + 2^{i-2} + \ldots + 2^2 + 2^1$$

$$= 2^{i-1} + \sum_{i=1}^{i-1} 2^i$$

$$= \frac{2^i}{2} + 2^i - 2$$

$$= \frac{3n}{2} - n - 2$$

$$= \frac{3n - 2}{2}$$

Binary Search

Consider a list of elements

10 20 30 40 50 60 70

Key element to be searched = 60.

(1) low = 0, high = 6, mid = $\lfloor \frac{0+6}{2} \rfloor = 3$

check if (a[3] = 60)

(40 ≠ 60) i.e 40 < 60

Search the key element in the right side of the array (50, 60, 70).

(2) 50 60 70

low = 4, high = 6, mid = $\lfloor \frac{4+6}{2} \rfloor = 5$.

check if (a[5] = key)

60 = 60 — True

∴ key element 60, is present in array at location 5.

Consider a list of elements

1 2 3 4

key element to be searched is 2

(1) low = 0, high = 3 ; mid = $\frac{\lfloor 0+3 \rfloor}{2} = 1$

check if (a[mid] = 2)

(a[1] = 2) ? YES ∴ 2 = 2. True

∴ key element 2 is present in array at location 1.

Analysis of Binary Search

(1) Best case Analysis

$T(n) = \Omega(1)$

(2) worst case Analysis

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

(1 ← No of comp, if n = 1; T(n/2) + 1 ← Time required to compare the middle element; n > 1 ← Time required to search either left or right part)

In worst case maximum number of element comparisons are required and time complexity is given by above recurrence relation.

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

$= T\left(\frac{n}{4}\right) + 1 + 1$

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

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

⋮

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

$n = 2^i$

Let $2^i = n$

$\log_2 2^i = \log_2 n$

$i = \log_2 n$.

Put i in equation ① that is $2^i = n$ and $i = \log_2 n$.

$$T(n) = T\left(\frac{n}{n}\right) + \log_2 n$$

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

$$= 1 + \log_2 n$$

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

(3) Average case Analysis.

sampled n elements

1) when $n = 1$

$a[0] = 10$

Number of comparison = 01

2) when $n = 2$

$a[0] = 10$ $a[1] = 20$

Number of comparisons = 2

$n = 2 = 2^1$

3) when $n = 4$, $n = 2^2$

No of comparisons = 3

4) when $n = 8$

No of comparisons = 4

N	Total number of comparisons
2^0	1
2^1	2
2^2	3
:	:
2^{k-1}	k

The average case of binary search is given by.

$$T(n) = \frac{1}{n} \sum_{i=1}^{k} i \cdot 2^{i-1} \quad \text{——— ①}$$

Now $2^{i-1} = 2^i \cdot 2^{-1}$

$$= \frac{2^i}{2}$$

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

$$= 2^i - 2^{i-1} \quad \text{— (2)}$$

Substitute (2) in (1).

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

$$= \frac{1}{n}\left(\sum_{i=1}^{k} i \cdot 2^i - \sum_{i=1}^{k} i \cdot 2^{i-1}\right)$$

$$= \frac{1}{n} \sum_{i=1}^{k} i \cdot 2^i - \frac{1}{n} \sum_{i=0}^{k-1} (i+1) 2^{i-1}$$

$$= \frac{1}{n}\left(\sum_{i=1}^{k} i \cdot 2^i - \frac{1}{n} \sum_{i=0}^{k-1} i \cdot 2^i - \sum_{i=0}^{k-1} 2^i\right)$$

$$= \frac{1}{n}\left(\sum_{i=1}^{k} i \cdot 2^i - \sum_{i=1}^{k-1} i 2^i - \sum_{i=0}^{k-1} 2^i\right)$$

$$= \frac{1}{n}\left(k \cdot 2^k - \sum_{i=0}^{k-1} 2^i\right)$$

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

$$= \frac{1}{n}\left\{k 2^k - 1\left(\frac{2^k - 1}{2 - 1}\right)\right\} \quad \text{using } S = a\left(\frac{r^n - 1}{r - 1}\right)$$

$a = 1, r = 2, n = k$

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

$$= \frac{1}{n}\{kn - n + 1\}$$

$$= k - 1 + \frac{1}{n}$$

As n increases to approximately at $k-1$

$$T(n) = k - 1 \quad \text{— (3)}$$

From eqn (1) we have $n = 2^{k-1}$. Taking log

$$\log_2 n = (k-1)\log_2 2$$

$\log_2 n = k-1$

Substitute $k-1 = \log_2 n$ eqn ③

$T(n) = \log_2 n$

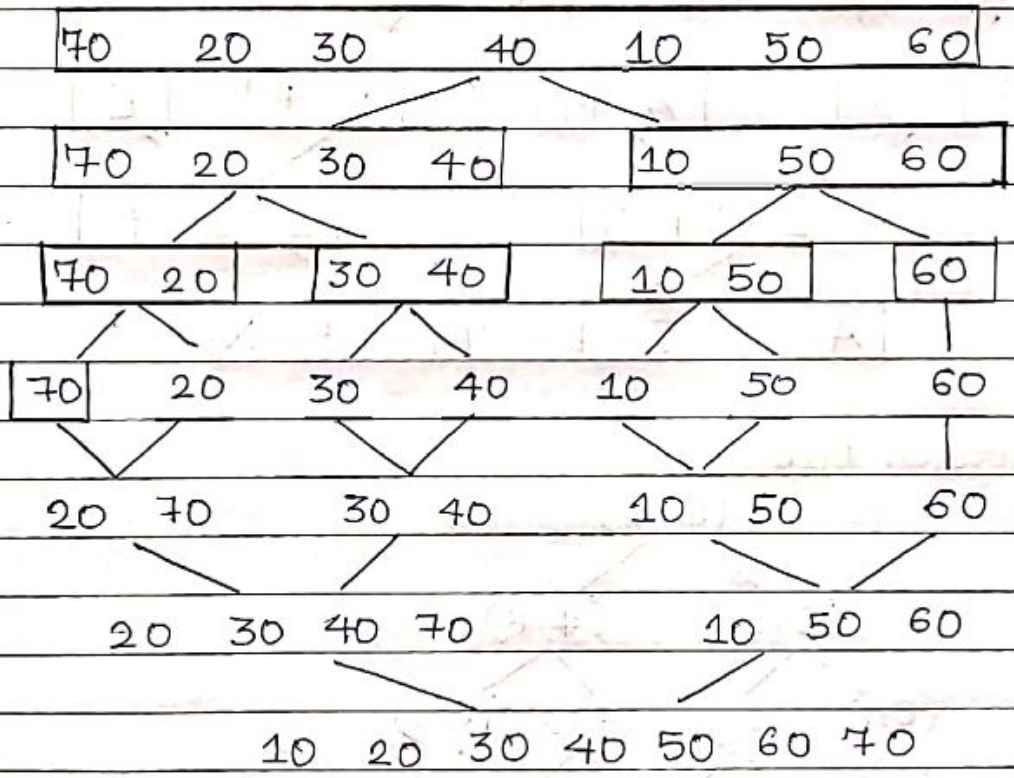
$T(n) = \theta(\log_2 n)$

Merge sort

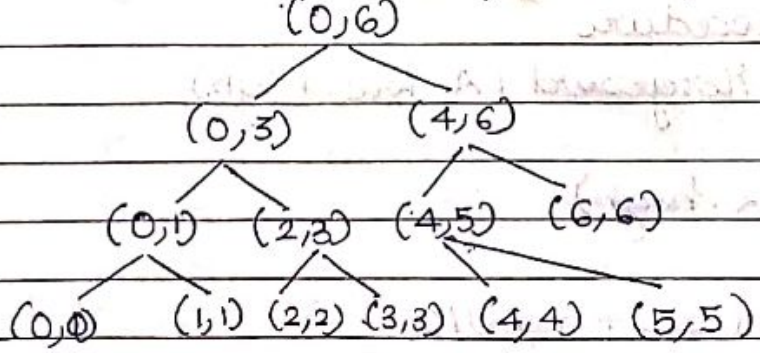
Consider an array

70 20 30 40 10 50 60.

mid $= \frac{0+6}{2} = 3$

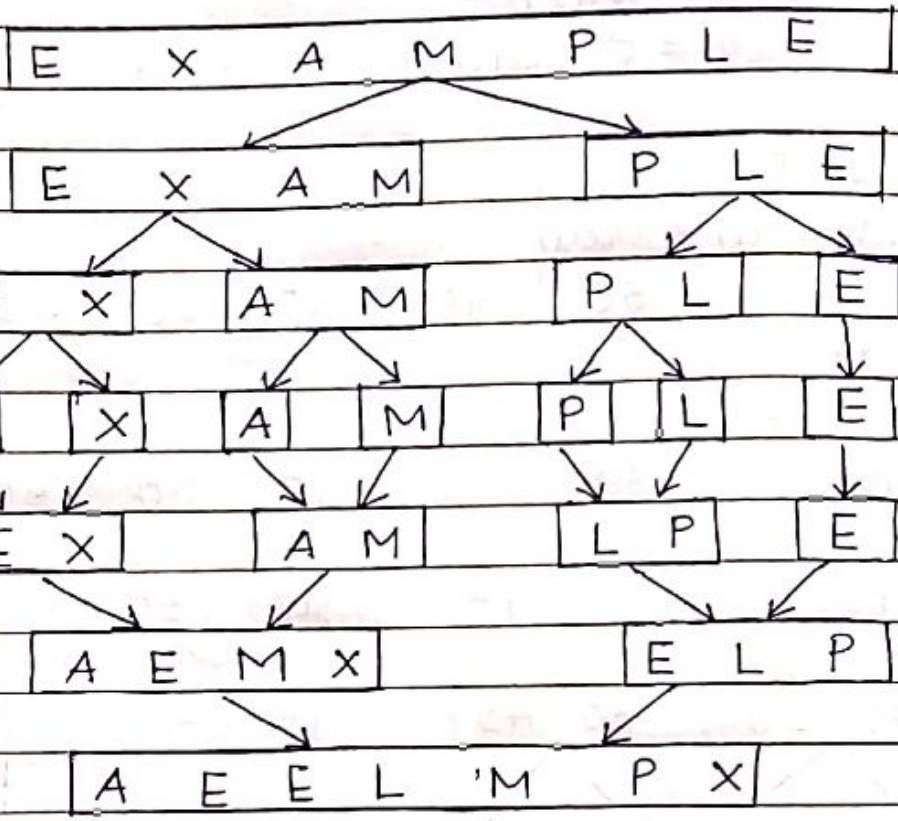


Recursion Tree (Tree calls for merge sort.

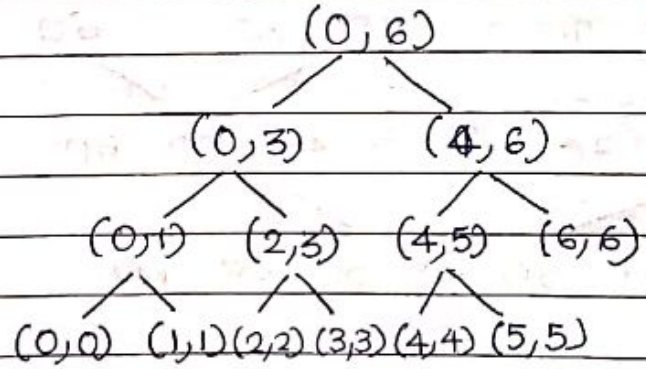


Consider an array

0	1	2	3	4	5	6
E	X	A	M	P	L	E



Recursion tree



General procedure

```
Algorithm MergeSort (A, low, high)
{
  if (low < high)
  {
    mid = (low + high)/2
    MergeSort (A, low, mid)
```

MergeSort (A, mid+1, high)
Merge (A, low, mid, high)
}
}

Analysis
1) Determining parameter size
Number of elements = n
2) key comparisons
3) Number of comparisons

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

→ Time required to sort left part of array
→ Time required to divide and merge one instance of solution with n elements.
→ Time required to sort right part.

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

Solving using master theorem,

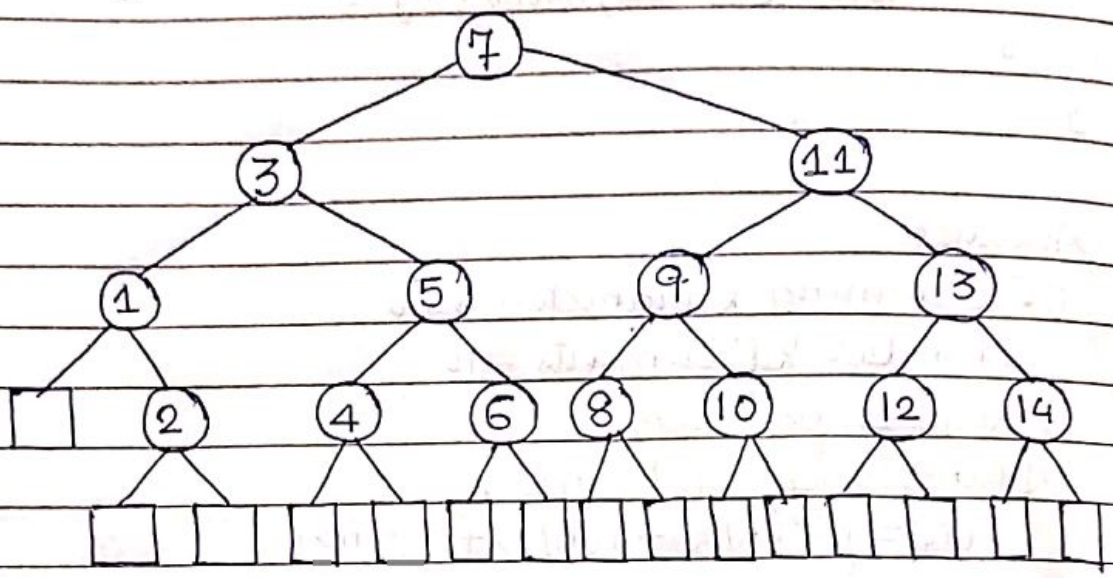
$a = 2$
$b = 2$
$f(n) = n \quad \therefore n^d = n \quad \therefore d = 1.$

Study merge sort algorithm.

To check: $a < b^d$, $2 < 2^1$ False
$a = b^d$, $2 = 2^1$ True
$a > b^d$, $2 > 2^1$ False.

$$T(n) = \Theta(n^d \log_b n)$$
$$= \Theta(n^1 \cdot \log_2 n)$$
$$= \Theta(n \log_2 n)$$

Binary Decision tree for binary search, n = 14



Quick Sort

General procedure:

1) i → initial value, i = 0
   j = n - 1
   pivot = a[i]
2) Keep Increementing index i as long as pivot ≥ a[i].
3) Once the above condition fails, keep decreementing j as long as pivot < a[j].
4) Once the above condition fails, if (i < j) exchange a[i] and a[j] and repeat all the above process as long as i ≤ j. Finally exchange the pivot or a[0] with a[j] and final value of j gives the position of the pivot element.

Consider the following array:

(25)	10	72	18	40	11	32	9	
25	10	72	18	40	11	32	9	
25	10	72	18	40	11	32	9	72 ≯ 25
25	10	72	18	40	11	32	9	9 ≯ 25
25	10	9	18	40	11	32	72	Exchange
		i					j	

25 10 9 18 40 11 32 72 i j

25 10 9 18 40 11 32 72 40 ≯ 25

25 10 9 18 40 11 32 72

25 10 9 18 40 11 32 72 11 ≯ 25

(Exchange a[i] & a[j])

25 10 9 18 11 40 32 72 i

25 10 9 18 11 40 32 72

25 10 9 18 11 40 32 72 25 ≯ 4, j--

Since i > j, swap a[j] and pivot.

11 10 9 18 25 40 32 72

11 10 9 18 i j

10 < 11, i++

11 10 9 i 18 j

9 < 11, i++

11 10 9 18

18 > 11, j--

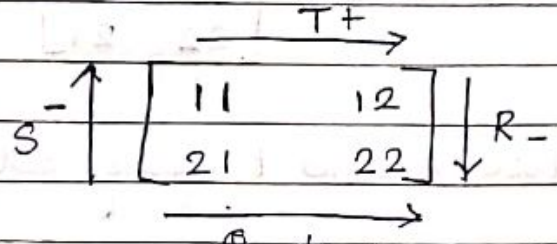
11 10 9 18

Exchange pivot with 18

18 10 9 18

i j

Problem statement



$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$

$Q = B_{11} (A_{21} + A_{22})$

$R = A_{11} (B_{12} - B_{22})$

$S = A_{22} (B_{21} - B_{11})$

$T = B_{22} (A_{11} + A_{12})$

$U = (A_{21} - A_{11}) (B_{11} + B_{12})$

$V = (B_{21} + B_{22}) (A_{12} - A_{22})$

$C_{11} = P + S - T + V$

$C_{12} = R + T$ RAT

$C_{21} = Q + S$

$C_{22} = P + R - Q + U$

Multiply the two matrices using Strassen's

$A = \begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$ $B = \begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix}$

Soln:

$P = (A_{11} + A_{22})(B_{11} + B_{22}) = (2+5)(3+1) = 28$

$Q = B_{11}(A_{21} + A_{22}) = 3(4+5) = 27$

$R = A_{11}(B_{12} - B_{22}) = 2(4-1) = 6$

$S = A_{22}(B_{21} - B_{11}) = 5(2-3) = -5$

$T = B_{22}(A_{11} + A_{12}) = 1(2+3) = 5$

$U = (A_{21} - A_{11})(B_{11} + B_{12}) = (4-2)(3+4) = 14$

$V = (B_{21} + B_{22})(A_{12} - A_{22}) = (2+1)(3-5) = -6$

$C_{11} = P + S - T + V = 28 + (-5) + 5 + (-6) = 12$

$C_{12} = R + T = 6 + 5 = 11$

$C_{21} = Q + S = 27 + (-5) = 22$

$C_{22} = P + R - Q + U = 28 + 6 - 27 + 14 = 21$

The result of matrix A×B is $C = \begin{bmatrix} 12 & 11 \\ 22 & 21 \end{bmatrix}$

Multiply the two matrices A and B using Strassen's method

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 6 & 0 & 3 \\ 4 & 1 & 1 & 2 \\ 0 & 3 & 5 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 4 & 2 & 7 \\ 3 & 1 & 3 & 5 \\ 2 & 0 & 1 & 3 \\ 1 & 4 & 5 & 1 \end{bmatrix}$$

Soln: The following are the 8 n/2 by n/2 matrices.

$$A_{11} = \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} \qquad A_{12} = \begin{bmatrix} 3 & 4 \\ 0 & 3 \end{bmatrix}$$

$$A_{21} = \begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} \qquad A_{22} = \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix}$$

using strassen's Method

$$P = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$= \left(\begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \right) \left(\begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix} \right)$$

$$= \left(\begin{bmatrix} 2 & 4 \\ 5 & 6 \end{bmatrix} \right) \left(\begin{bmatrix} 2 & 7 \\ 8 & 2 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 36 & 22 \\ 58 & 47 \end{bmatrix}$$

$$Q = B_{11}(A_{21} + A_{22})$$

$$= \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} \left(\begin{bmatrix} 4 & 1 \\ 0 & 3 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \right)$$

$$= \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 5 & 3 \\ 5 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} 241 & 15 \\ 20 & 12 \end{bmatrix}$$

$R = A_{11}(B_{12} - B_{22})$

$= \left(\begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix}\right)\left(\begin{bmatrix} 2 & 7 \\ 3 & 5 \end{bmatrix} - \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix}\right)$

$= \begin{bmatrix} 1 & 2 \\ 0 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ -2 & 4 \end{bmatrix}$

$= \begin{bmatrix} -3 & 12 \\ -12 & 24 \end{bmatrix}$

$S = A_{22}(B_{21} - B_{11})$

$= \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix}\left(\begin{bmatrix} 3 & 0 \\ 1 & 4 \end{bmatrix} - \begin{bmatrix} 1 & 4 \\ 3 & 1 \end{bmatrix}\right)$

$= \begin{bmatrix} 1 & 2 \\ 5 & 0 \end{bmatrix} \begin{bmatrix} 1 & -4 \\ -2 & 4 \end{bmatrix}$

$= \begin{bmatrix} -3 & 4 \\ 5 & -20 \end{bmatrix}$

$T = B_{22}(A_{11} + A_{12})$

$= \begin{bmatrix} 1 & 3 \\ 5 & 1 \end{bmatrix} \begin{bmatrix} 4 & 7 \\ 0 & 9 \end{bmatrix}$

$= \begin{bmatrix} 4 & \\ & \end{bmatrix}$

TUTORIALS:

Recurrences

A recurrence relation is an equation or an inequality that describes the function in terms of its value on smaller inputs.

Example:

```
void fun(int n)
{
  if(n>0) — 1
  {
    print(n) — 1
    fun(n-1) — T(n-1)
  }
}
```

$$T(n) = \begin{cases} T(n-1)+c & ; n>0 \\ 1 & ; n=0 \end{cases}$$

1.

```
fib(n)
{
  if(n≤1) — 1
    return 1
  else
    return fib(n-1) + fib(n-2)
}
```

(fib(n-1) → T(n-1), fib(n-2) → T(n-2))

$$T(n) = \begin{cases} 1 & ; n \le 1 \\ T(n-1)+T(n-2)+c & ; n>1 \end{cases}$$

* Solution of Recurrence Relations Methods

(a) Substitution Method (Backward Substitution)

1. $T(n) = \begin{cases} 1 & ; n=0 \\ T(n-1)+1 & ; n>0 \end{cases}$

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

$= [(T(n-1)-1)+1]+1$ $\longrightarrow T(n-1) = T(n-2)+1$

$= T(n-2)+2$

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

$\vdots$

$= T(n-i)+i$

Substituting $i = n$;

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

$T(n) = T(0)+n$

$= 1+n$.

$O(n) = n$

2. $T(n) = \begin{cases} 1 & ; n=0 \\ T(n-1)+n & ; n>0 \end{cases}$

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

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

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

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

$\vdots$

$= T(n-i)+(n-(i-1))+(n-(i-2))+\ldots+(n-1)+n$

Substituting $i = n$

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

$= T(0)+1+2+3+\ldots+(n-1)+n$

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

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

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

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

next term will be $T(n/2)$

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

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

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

$$= 2^3 \cdot T\left(\frac{n/2}{2^2}\right) + 2 \cdot \frac{n}{2} + 2n$$

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

$$\vdots$$

$$= 2^i \cdot T\left(\frac{n}{2^i}\right) + in \quad \text{---(1)}$$

Putting $2^i = n$

$$\log_2 2^i = \log_2 n$$

$$i \log_2 2 = \log_2 n$$

$$i \frac{\log 2}{\log 2} = \log_2 n$$

$$i = \log_2 n.$$

Substituting i in (1)

$$T(n) = n \cdot T(0) + \log_2 n \cdot n.$$

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

$$= n \cdot \log_2 n + n$$

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

(b) Master Theorem

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$

$b > 1$

$f(n)$ = asymptotically +ve function

if $f(n) = \Theta(n^d)$ $d \geq 0$.

then $T(n) = \begin{cases} \theta(n^d) & \text{if } a < b^d \\ \theta(n^d \log_b n) & \text{if } a = b^d \\ \theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$

Where θ, b = Constant.
d power of n in $f(n)$

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

Soln: Given, $T(n) = 4T\left(\frac{n}{2}\right) + n$

$a = 4$
$b = 2$
$f(n) = n^1 = n^d$
$d = 1$

check $a < b^d$, $4 < 2^1$ False
$a = b^d$, $4 = 2$ False
$a > b^d$, $4 > 2^1$ true

$T(n) = \theta\left(n^{\log_2 4}\right)$

$= \theta\left(n^{\frac{\log 4}{\log 2}}\right)$

$= \theta(n^2)$.

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

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

$a = 1$
$b = 2$
$f(n) = 1 \therefore d = 0$.
check $a < b^d$, $1 < 2^0$ False
$a = b^d$, $1 = 2^0$ True

$$T(n) = \theta(n^d \log_b n)$$
$$= \theta(1 \cdot \log_2 n)$$
$$= \theta(\log_2 n)$$

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

Soln: Given $T(n) = 8T\left(\frac{n}{2}\right) + n^2$.

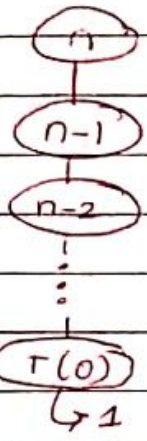
$a = 8$
$b = 2$
$f(n) = n^d = n^2$
$d = 2$

Check $a < b^d = 8 < 2^2$ False
$a = b^d = 8 \neq 2^2$ False
$a > b^d = 8 > 4$ True

$T(n) = \theta(n^{\log_2 8})$
$= \theta(n^{\frac{\log 8}{\log 2}})$
$T(n) = \theta(n^3)$

(c) Recursion Tree Method
- pictorial representation of relation
- In this form of tree
at each level the tree node is expanded

$$T(n) = \begin{cases} 1 & n \neq 0 \\ T(n-1) + n & n \geq 0 \end{cases}$$

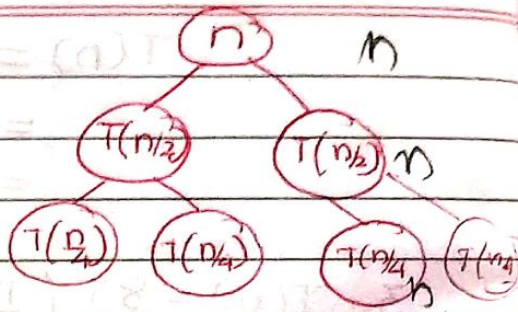


$T(n) = 1 + 2 + \ldots + (n-2) + (n-1) + n$
$= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} = O(n^2)$

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

classmate
Date
Page

soln: $T(n) = 1 + 2 +$



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