What is algorithm?

The word algorithm comes from the name of Auther "Abu Jafar Mohammed ibn Musa al khowarizmi".

Algorithm:

An algorithm is a finite set of instructions that is followed, accomplishes a particular task.

Criteria for algorithm:

1) Input: 0 or more inputs are externally supplied.

2) Output: Atleast one quantity is needed one or more outputs.

3) Definiteness: Each instruction is clear & unambigious.
[Not more o/p]
eg: Adding any term to x is possible operations.
Number/0 not possible.

4) finiteness: If we trace out instructions of an algorithm then for all cases, the algorithm terminates after a finite number of steps.
The o/p should be in finite time not be too long for computation.

5) Effectiveness: Every instruction must be very basic so that it can be carries out a person by paper & pencil method.

4 area's of Research:

1) How to device a algorithm:
- Device New & useful algorithm.
- Dynamic programming - Develop the program / algorithm in fields like electrical engineering, operation Research.
- To study linear, nonlinear and integer programming.
- Creating a program is not a totally automated iso. create fully automated programs.

2) How to validate Algorithm:
- Once we device / create an algorithm it is neccessary that it should give correct / legal output for all legal inputs.
- The purpose of validation is that once the program runs independently & correctly we goes to programming prove / varification state.
- Program contains:
  1) Assertions
  2) Specifications
- Assertions are nothing but all inputs & outputs in program & specifications are the predicate ca

3) How to analyze Algorithms:
- Computer time required & space required for storage an algorithm.
- As algorithm is executed it uses CPU to perform operations & memory.

- It gives complexity like time & space with 3 conditions:
  1) Best
  2) Average
  3) Worst

4) How to Test a program:

- In testing there are two phases.
  1) Debugging
  2) Profileing (Performance Measure)

1) Debugging:

Process of executing a program & determine weather a error or not but according to "E Dijkstra" debugging can only point to presence of errors but not to their absence.

Let more than one programmer developer a program for same task. If their o/p's are same then good chance to correct the problem.

2) Profiling:

It execute a correct program & measure time & space required for it. i.e [performance measurment]

Algorithm Specification:

Pseudocode Conventions:

1) Comments begin with //[ ] & continue until end of line.

2) Statements are terminated / delimited by ; (semicolon). Blocks are indicated with matching braces: { and }
A compond statement can be represented as block.

3) Identifier begins with latter, The data types of variables are not explicitly declared.
The simple data types such as integer, float, char, boolean etc. can be assume.
The compound datatypes can be formed with records.

```
node = record
       {
          datatype-1 data 1;
          .
          .
          datatype-n data n;
          node *link;
       }
```

· Here link is pointer to node it will contain datatype.

If datatype-1 then data 1 & so on

4) Assignment of values to variables is done using the assignment statements.

<variable> := <expression> ;

5) There are two boolean values true & false. In order to procedure these values the logical operators AND, OR, NOT & Relational operators $<, \leq, =, \geq, >, \neq$ are provided.

6) The elements of multidimentional arrays can be accessed by [ ] (matrix form) for eg: in two dimensional array A(i, j) the element can be denoted as A[i, j].

7) Looping Statements :

1) for
2) While
3) Repeat-until

a) for loop:

```
for variable = value 1 to value2 step step do
{
  <statement 1>
     :
  <statement n>
}
```

value 1, value2, & step are arithmetic expressions.

2) While :-

```
While <condition> do
{
   <statement 1>
      :
   <statement n>
}
```

As long as condition is true the statement is executed when it false then terminate.

3) Repeat-until:

```
repeat
    <statement 1>
        :
    <statement n>
until <condition>
```

The statement are execute until condition is

8) Conditional statement:

1) if:

```
if <condition>
then <statement>
```

2) if -else:

```
if <condition>
then <statement 1>
else <statement 2>
```

Case:

Case
{
<Condition 1> <statement 1>
:
<Condition n>, <statement n>
else <statement n+1>
}

If condition 1 is true then statement 1 is executed upto n statement & statement n+1 is executed.
The else clause is optional.

9) The input & output are done by read & write instructions No. format is specify the size of input or output quantities.

10) The algorithm can be maintion in only following type:

Algorithm Name (<Parameters list>)
{
Name= Name of program

Recurrence Relations:

1) Iterative Algorithm
2) Recursive Algorithm

1) Recursive Algorithm:

A function defined itself called recursive like this if same algorithm invoked in body then called recursive algorithm. (direct recursive)

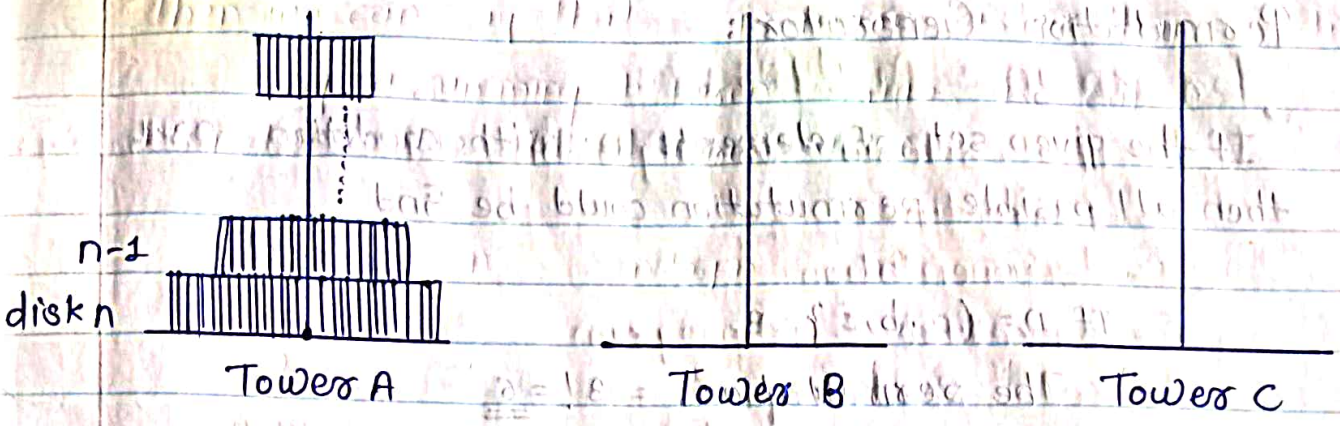
If algorithm calls another algorithm called (Indirect recursive algorithm)

Binomial Coefficients $= \binom{n}{m} + \binom{n-1}{m} + \binom{n-1}{m-1} = \frac{n!}{m!(n-m)!}$

Eg: 1) Towers of Hanoi
ii) Possible permutation.

i) Towers of Hanoi

When the world was created, the Brahma was created a tower of 64 dimond tower with golden disks. The disks were arranged in ~~accessing~~ assending order from top to bottom & he was challenged that more the tower from A to B in such a way that the sequence will same & disk can more only single then world will distory.



By recursive defination it is possible to move by using "Tower of Honai" method.

If tower A contain n disks then move (n-1) disks to tower C. Then remaining nth disks move to B & then disk from C will move to B (it will be complete the task

The soln 2(n-1) - disk problem

n = number of disks.

```
Algorithm tower of Hanoi (n, x, y, z)
{
    if (n>=1) then
    {
        Tower of Honai (n-1, x, z, y);
        write ("move top disk from tower", x,
               "top of tower", y);
        Tower of Hanai (n-1, z, y, x);
    }
}
```

Permutation Generator:

If the given set of elements n with condition $n \geq 1$ then all posible permutation could be find

If $n = \{a, b, c\} = 3$

The permutations $= 3! = 6$

$\{a,b,c\}$ $\{a,c,b\}$, $\{b,a,c\}$, $\{b,c,a\}$ $\{c,a,b\}$ $\{c,b,a\}$

- i.e permutations could be find out by formula n!

```
Algorithm perm (a, k, n)
{
  if (k=n) then write (a[i:n])
    else
      for i = k to n do
      {
        t: a[k]; a[k] = a[i]; a[i] = t;
        perm (a, k+1, n);
        t = a[k]; a[k] = a[i]; a[i] = t;
      }
}
```

Practical Complexities / Graphical Representation:

- Time complexity of algorithm is generally some function of instance characteristics.
- The complexity function used for compare complexity of function.
- Let us consider two algorithms P & Q having complexity

  $P \rightarrow O(n)$

  $Q \rightarrow O(n^2)$

Then we can say that ___ is more faster than ___ like ___ by assumming different values use can find a function values for different values

log n	n	n log n	n^2	n^3	2^3
0	1	0	1	1	2
0.30	2	0.60	4	8	4
0.60	4	2.40	16	64	16
0.90	8	7.20	64	512	256

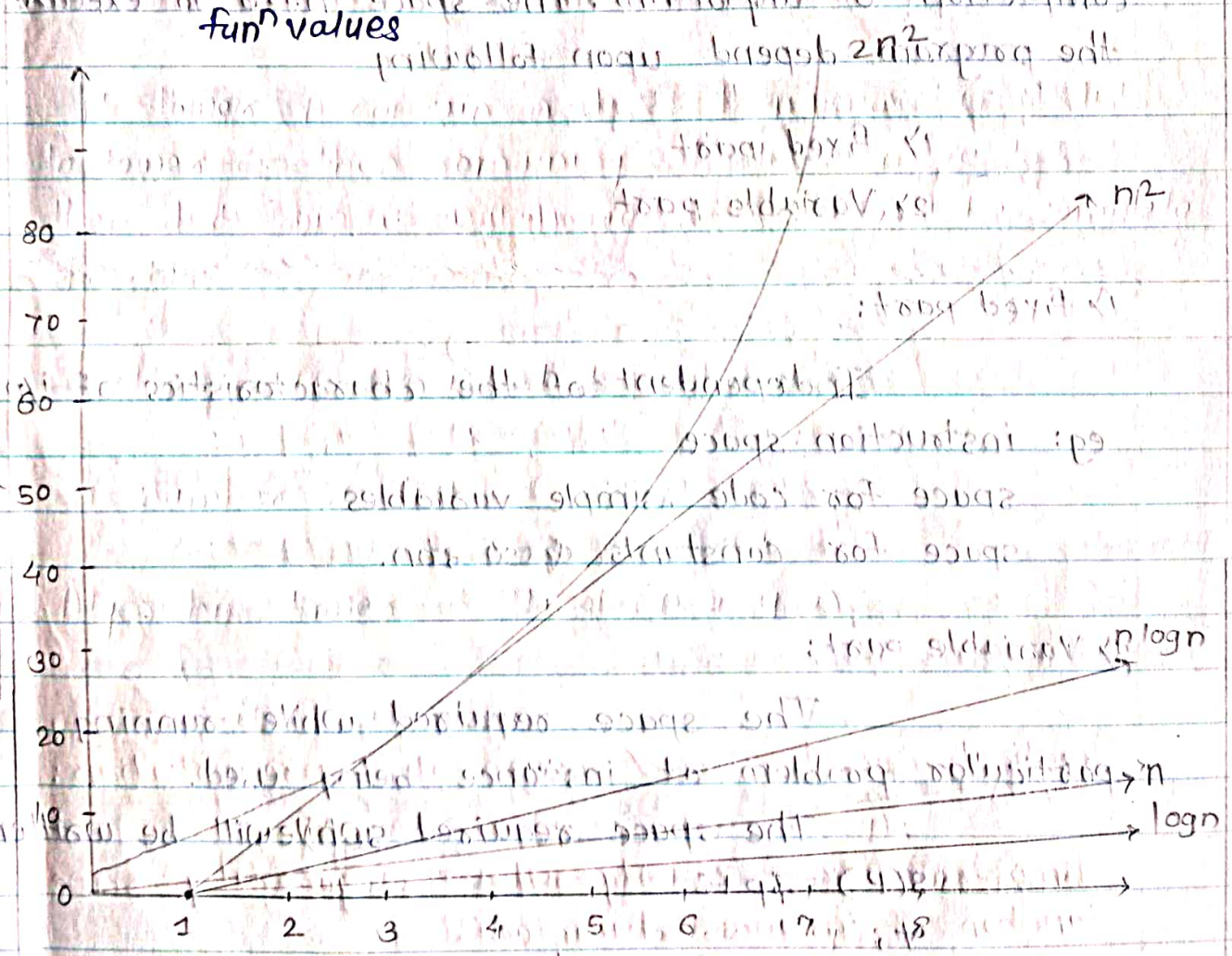


fig: Plot of fun^n values

Time complexity

log n
n
n log n
n^2
2^2

fast ↓ slow

less time

more time

Performance Analysis :

The criteria for algorithm depend on following :

- Correctiness
- documentation
- procedure & subprogram
- code Readiblity

space complexity :

The amount of memory needs to completion of algorithm. The space need to execute the program depend upon following

1) fixed part
2) Variable part

1) fixed part:

Independent of the charactoristics of i/p & o/p.

eg: instruction space
space for code, simple variables
space for constants & so on.

2) Variable part:

The space required while running particular problem at instance being used.

The space required S(P) will be written as,

$S(P) = Sp + C$

Where, Sp = instance charactoristics &
C = constant

Time complexity : time

The T(P) is taken by a program P is sum of run time & compile time.

T(P) = run time + compile time.

- The compile time is not an instance charactoristics because while running a single program we can't recompile a program for a many times.

- Instance character depend upon-

Not dependent	Dependent
[compile time]	[Run time]

- In case of time complexity we have consider only run time i.e t(P), run time t(P) for n numbers of (instance characteristics) operations could be,

$t(P) \Rightarrow tp(n)$.

- Ca, Cs, Cm, Cd is time needed for addition, substraction, multiplication & division, ADD, SUB, MUL, DIV are functions of addition, substraction, multiplication & division.

$tp(n) = Ca\ ADD(n) + Cs\ SUB(n) + Cm\ MUL(n) + Cd\ DIV(n) + \cdots\cdots$

Asymptotic Notation [O, Ω, Θ]

Let us consider f & g are non-negative functions.

1) Big 'oh' [O] : upperbound Notation

The fun^n $f(n) = O(g(n))$ [read that f of n big oh of g of n] iff there exist a positive constant c & n_0 such that,

$$f(n) \leq c \cdot g(n) \text{ for all } n, n \geq n_0$$

Big oh Notation is also called as upper bound notation.

eg:- 1) The fun^n $3n+2 = O(n)$

$3n+2 \leq 4n$ for all $n, n \geq 2$

2) $10n^2+4n+2 = O(n^2)$

$10n^2+4n+2 \leq 11n^2$ for all $n \geq 5$

3) $6 \cdot 2^n + n^2 = O(2^n)$ as

$6 \cdot 2^n + n^2 \leq * 2^n$ for all $n \geq 4$

Here,

$O(1)$ = constant
$O(n)$ = linear
$O(n^2)$ = quadratic
$O(n^3)$ = cube
$O(2^n)$ = exponential
$O(\log n)$ = logrethmic

$1 < \lg n < n < n \log n < n^2 < n^3 < 2^n < 2!$

2) Omega (Ω) : lower bound

The function $f(n) = \Omega(g(n))$ is read a f of n is omega of g of n iff there exists positive constant c & no.

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

$f(n) \geq c \cdot g(n)$ ∀, c is constant.

The omega Notation contain the lower bound.

if 1) $3n+2 = \Omega(n)$

$3n+2 \geq 3n$ for $n \geq 1$

2) $100n+6 = \Omega(n)$

$100n+6 \geq 100n$ for $n \geq 1$

3) Theata (θ): Both upper & lower bound.

The function $f(n) = \theta(g(n))$ could be read as f of n is theta g of n iff c_1, c_2, n are constants such a way that other than θ.

$c_1 \cdot g(n) \leq f(n) \leq c_2 g(n)$

eg. $3n+2 = \theta(n)$

as $3n+2 \geq 3n$ &

$3n+2 \leq 4n$

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

so here $c_1 = 3$, $c_2 = 4$ for all $n \geq 2$

The θ contain both upper as well as lower bound. so θ is more reliable than O & Ω

4) little o:

The function $f(n) = o(g(n))$ read as "f of n is little oh of g of n" iff,

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

eg:

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

$$\lim_{n\to\infty} = \frac{3n+2}{n^2} = 0$$

because n is ∞

5) little omega (ω):

The fun^n $f(n) = \omega(g(n))$ is read as f of n is little omega of g of n iff,

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

eg.

$3n+2 = \omega(n^2)$

$$\lim_{n\to\infty} \frac{n^2}{3n+2} = 0$$

Proove that,

if $f(n) = a_m n^m + \cdots + a_1 n^1 + a_0 n^0$ then, $f(n) = O(n^m)$

$$f(n) = a_m n^m + \cdots + a_2 n^2 + a_0 n^0$$

$$= \sum_{i=0}^{m} a_i n^i$$

$$= n^m \sum_{i=0}^{m} a_i n^{i-m}$$

$$\leq n^m \sum_{i=0}^{m} a_i n^{i-m} \quad \because n^{m-m} = n^0 = 1$$

$$\leq n^m \sum_{i=0}^{m} b_i n^{i-m}$$

assume $b_i = a_i$

$$\leq n^m \sum_{i=0}^{m} b_i$$

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

by assume $n \geq 1$ & upper bound

Randomized Algorithm :

The algorithm in which the resources are randomally selected called as randomized algorithm.

1) Basics of probability Theory.
2) Randomized Algorithm with informal description
3) Identifying the repeated element.
4) Primality Testing.

1) Basics of probability Theory :

In probablity the sample space (S) is drawn or selection of item randomly from given set of elements.

The single task or subset of sample called as event.

If there are n sample points then possible events $= 2^n$.

eg. tossing a coin ten times, playing a cards, playing a lottery, picking a ball from an urn containing white & red balls.

If we toss 3 coins then there will be 8 possiblities.

{HHH, HHT, HTH, HTT, THH, THT, TTH, TTT}

eg. if we draw set of atleast two heads $= 4/8$

$= 1/2$

2) Randomized Algorithms : An Informal description:

Randomizer : which generate random numbers. Decision mode in the algorithm depend upon Randomizer.

Randmized Algorithm classified into two types:

1) Las Vegas.

2) Monte Carlo

1) Las Vegas:

Always procede the same o/p for same input.

Execution time depend upon i/p of randomizer

2) Monte Carlo Algorithms-

output Might be differ from run to run.

3) Identifying the Repeated Element:

If an array a[n] containing 2 elements in such a way that

$\frac{n}{2}$ = distinct elements

$\frac{n}{2}$ = Repeated elements

By using Las vegas algorithm it required atleast $\frac{n}{2}+2$ steps for worst case.

4) Primality Testing:

Any integer greater than 1 is said to be prime if it is divisible by only one (1) & itself.

from 1 to 15, there are 2, 3, 5, 7, 11, 13 are first 6 prime numbers if we have given n integer then finding prime from n numbers called as primality testing.

Eg. If n is any number have to test weather prime or composite then,

Divide or take square root of n ie $\sqrt{n}$ & make interval of $[2, \sqrt{n}]$.

The divide the given number by any number if it divides then it is composite number otherwise it is prime number.

Divide & Conquer :

In divide & conquer method big problem is divided into small subproblems & slove them seperately & joint their o/p together.

If problem 'n' distinct subproblem with k steps

$$1 < k \leq n$$

1) If P is given problem then split it into small subproblems $P_1, P_2, \ldots\ldots P_n$.

2) If P is too small then solve it independently.

```
Algorithm D&C(P)
{
  if small(P) then return s(P).
  else
  {
    divide P into P1, P2, P3 ..... Pk.
        Apply D&C on each subproblem to get o/p.
  return combine  D & C(P1), .....
                        D & C(P2) ..... D & C(Pk)
  }
}
```

If P = given problem with size n

$P_1, P_2, P_3 \ldots\ldots P_k$ small subproblems $k \geq 1$

If P is very small then

$$T(n) = \begin{cases} g(n) & n \text{ is small} \\ T(n_1) + T(n_2) + T(n_3) + \ldots\ldots + T(n_k) + g(n) \end{cases}$$

∀, g(n) = time require to compute answer for small subproblem.

If the equation follows recurrence relation then,

$$T(n) = \begin{cases} T(1) & n=1 \\ a \cdot T(n/b) + f(n) & n>1 \end{cases}$$

∀, a, b = constants

To solve above relation we can use recurrence reln by using substitution

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

$$T(n) = a\,T(n/b) + f(n)$$

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

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

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

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

Again Divide method

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

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

$$= 8T(n/8) + 3n$$

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

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

$$\boxed{T(n) = 2^i T(n/2i) + in}$$

Binary Search :

Searching :

finding the location of given data elemen in data structure.

It having two types:

1) Linear Search (sequential)
2) Binary Search

1) Linear Search :

finding the location of given data elements by visiting each element of array or links until you get an element.

Time required it too high
Not feasible for big size arrays.

2) Binary Search :

Divide the array into equal two parts & solve it.

1) Compare middle elements with divided arrays.

2) If middle elements equal to output then return to index with middle value.

3) If search element smaller than middle element concentrate on 1st half & set end to (middle-1)

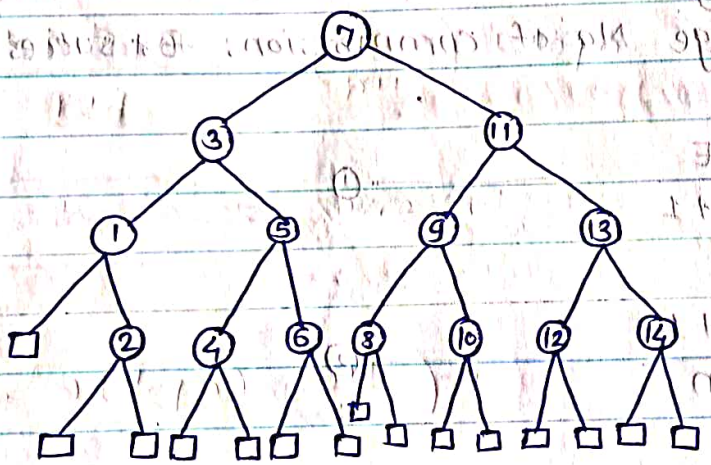
4) If search is greater than middle value in 2nd half set start at (middle+1)

$\frac{low + high}{2} = mid$ If mid is greater than element then search is first half by (mid-1)

eg: -15, -6, 0, 7, 9, 23, 54, 82, 101, 112, 125, 131, 142, 151

i.e element (total elements = 14)

$mid = \frac{1+14}{2} = 7$



□ => unsuccessful Node search.

O => successful search.

The binary elements array should contain elements in nondecreasing order.

To solve the given problem we are using divide & conquer technique.

If problem is too small then complexity = $\theta(1)$ only one search is require.

If element is found in given they end at circular node it is also called as internal node.

In unsuccessful the node is square & called unsuccessful node / external node.

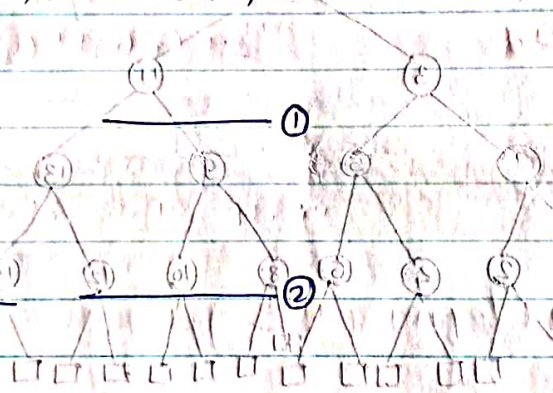
$E = I + 2n$

E = External path length E is the sum of the distance of all external nodes from the root.

I = Internal path length.

Au = Average No of comparisions for unsuccessful search

As = Average No of comparisions for successful search.



$$Au = \frac{E}{n+1} \quad \text{——— ①}$$

$$As = \frac{I+1}{n} \quad \text{——— ②}$$

Now

$$Au = \frac{E}{n+1}$$

$$Au = \frac{I+2n}{n+1} \quad \text{——— ③}$$

from ③

$$I = Au(n+1) - 2n \quad \text{——— ④}$$

from ② & ④

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

$$As = \frac{Au\,(n+1)}{n}$$

$$\boxed{As = \frac{Au\,(n+1)}{n} - 1}$$

We know that

$$E \propto n \log n$$

$$E = k n \log n$$

$$Au = \frac{k\,n \log n}{n+1} = \left(\frac{n}{n+1}\right) k \log n$$

$$\boxed{Au = \theta \log n}$$ for unsuccessful

$$As = \theta(\log n)\left(\frac{n+1}{n}\right) - 1$$

$$= \theta(\log n) - 1$$

$$\boxed{As = \theta \log n}$$ successful

successful search

Best $\theta(1)$

Average $\theta(\log n)$

Worst $\theta(\log n)$

unsuccessful search

$\theta(\log n)$

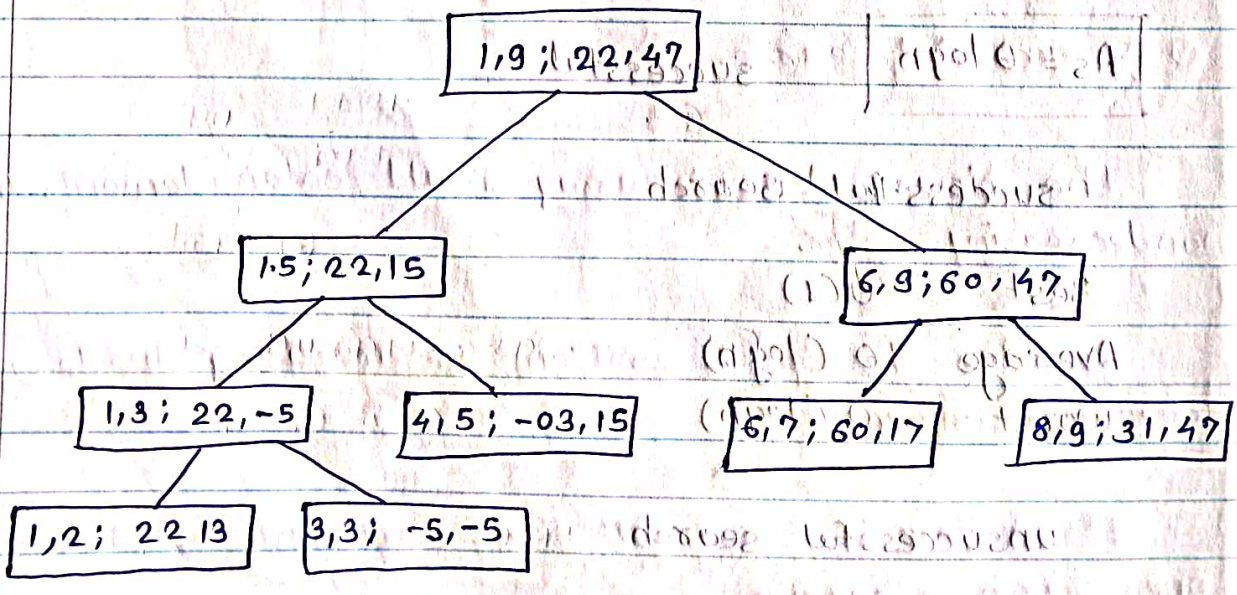
Finding the Maximum and minimum :

from given set of n elements we have to find maximum & minimize item. If an array contains element $a[1,n]$ then by finding mid we will find out minimum & maximum element.

If an array, vectors, very large numbers or string of charactor contain only single element then,

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

eg: 22, 13, -5, -3, 15, 60, 17, 31, 47



In straight forword method of comparision it requires 2(n-1) comparisions in all best, average & worst case.

When the elements are in increasing order the no of comparisions of elements = n-1

When it in non-decreasing order it requires 2(n-1) comparisions.

The No. of comparisions in Min-Max algorithm by using divide & conquer strategy reduce comparisions upto 25%.

If we uses T(n) = No of elements.

Originally,

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

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

Again by divide & conquer

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

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

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

Again

$$T(16) = 16T(n/16) + 16 + 8 + 4 + 2$$

Now put,

T(8) in T(16)

$$T(16) = 2[T(8) + 2$$

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

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

$$T(16) = 8T(2) + 8 + 4 + 2 \quad \text{---(1)}$$

The eqn ① will retain in 2's power

$$T(2^4) = 2^3 T(2) + 2^3 + 2^2 + 2$$

When $\boxed{n=2^4}$

Now put,

$n=2^k$ the eqn will be,

$$T(2^k) = 2^{k-1} T(2^{1+k-k}) + 2^{k-1} + 2^{k-2} + 2^{k-3} + \cdots + 2^1$$

We know $T(2)=1$

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

$$= 2^{k-1} + 1 - 1 + 2 + \cdots + 2^{k-2} + 2^{k-1}$$

$$= 2^{k-1} - 1 + 1 + 2 + \cdots + 2^{k-2} + 2^{k-1} \quad \text{---(2)}$$

Now the term

$$= 1 + 2 + 4 + 8 + \cdots + 2^{k-2} + 2^{k-1}$$

$$= 2^0 + 2^1 + 2^2 + \cdots + 2^{k-2} + 2^{k-1}$$

put $a=2$ $\quad = a^0 + a^1 + a^2 + \cdots + a^{k-2} + a^{k-1}$.

$$= \frac{a^{k-1+1} - 1}{a-1}$$

$$= \frac{2^{k-1+1} - 1}{2-1} = 2^k - 1$$

By putting in eqn (II)

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

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

$$= 2^k + 2^k \times 2^{-1} - 2$$

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

$= 2^k(3/2) - 2$

$= \frac{3}{2}2^k - 2$

$T(n) = \frac{3}{2}n - 2$

Now resubstitude $2^k = n$

Hence require 25% less comparisions in divide & conquer.

Merge Sort:

Complexity : $O(n \log n)$ [worst case]

- elements are sorted in nondecreasing order from n elements, sequence of elements are also called keys.

- If $a[1] \cdots a[n]$ is sequence of n elements then split it into

  $a[1], \cdots a[n/2]$ &

  $a[n/2+1] \cdots a[n]$

- Each set is individualy sort & resulting sorting sequence merged into produce single sorted array of n elements.

Eg: Consider an array of 10 elements.

$a[1:10] = \{310, 285, 179, 652, 351, 423, 861, 254, 450, 520\}$

By merge sort spliting given array into a[1:5] & a[6:10]

Again a[1:5] into a[1:3] & a[4:5]

a[1:3] into a[1:2] & a[3:3]

a[1:2] into a[1:2] may compared.

(310 | 285 | 179 | 652, 351 | 423, 861, 254, 450, 520)

∀,

Vertical bar represents boundaries of subarrays

Now element a[1] & a[2] merged together to get

(285, 510 | 179 | 652 | 351 | 423, 861, 254, 450, 520)

(179, 285, 510 | 652 | 351 | 423, 861, 254, 450, 520)

(179, 285, 510 | 351, 652 | 423, 861, 254, 450, 520)

In above case a[4] & a[5] are merged together

(179, 285, 351, 510, 652 | 423, 861, 254, 450, 520)

finally,

(179, 285, 352, 510, 652 | 254, 423, 450, 520, 861)

(179, 254, 285, 351, 423, 450, 510, 520, 652, 861)

Hence final Result

Algorithm:

```
Algorithm mergesort(low, high)
{
    if low < high then
    {
        mid = [low + high] / 2
        Mergesort(low, mid);
        mergesort(mid+1, high);
        merge(low, mid, high);
    }
}
```

find complexity for merge sort is

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

when n is power of 2 $n = 2^k$

$$T(n) = T(n)$$

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

if we take $n = 2^k$

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

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

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

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

$$= 8\left[2T(n/16) + \frac{cn}{8}\right] + 3cn$$

$$T(n) = 2^3\left[2T(n/2^4) + \frac{cn}{2^3}\right] + 3cn$$

if we put $n = 2^k$.

$$T(8) = 2^3 T(n/8) + 3cn$$

$$T(2^3) = 2^3 (T(n/8) + 3cn$$

$$T(2^k) = 2^k (T(8/8) + kcn$$

$$T(2^k) = 2^k (T(1) + kcn$$

$$= 2^k \cdot + kcn$$

Now

$$2^k = n$$

$$k = \log_2 n$$

$= 2^k \cdot c + kcn$

$= n \cdot a + cn \log n$

$=$ Now $\forall$, $n \cdot a \Rightarrow$ constant

$c \Rightarrow$ any Non-negative constant

$\leq n \cdot a + c \cdot n \log n$

$T(2^k) = O[n \log n]$

i.e Complexity $= O(n \log n)$

- Merge sort

  worst case $O(n \log n)$

- Selection sort

  Best, worst & average $O(n^2)$

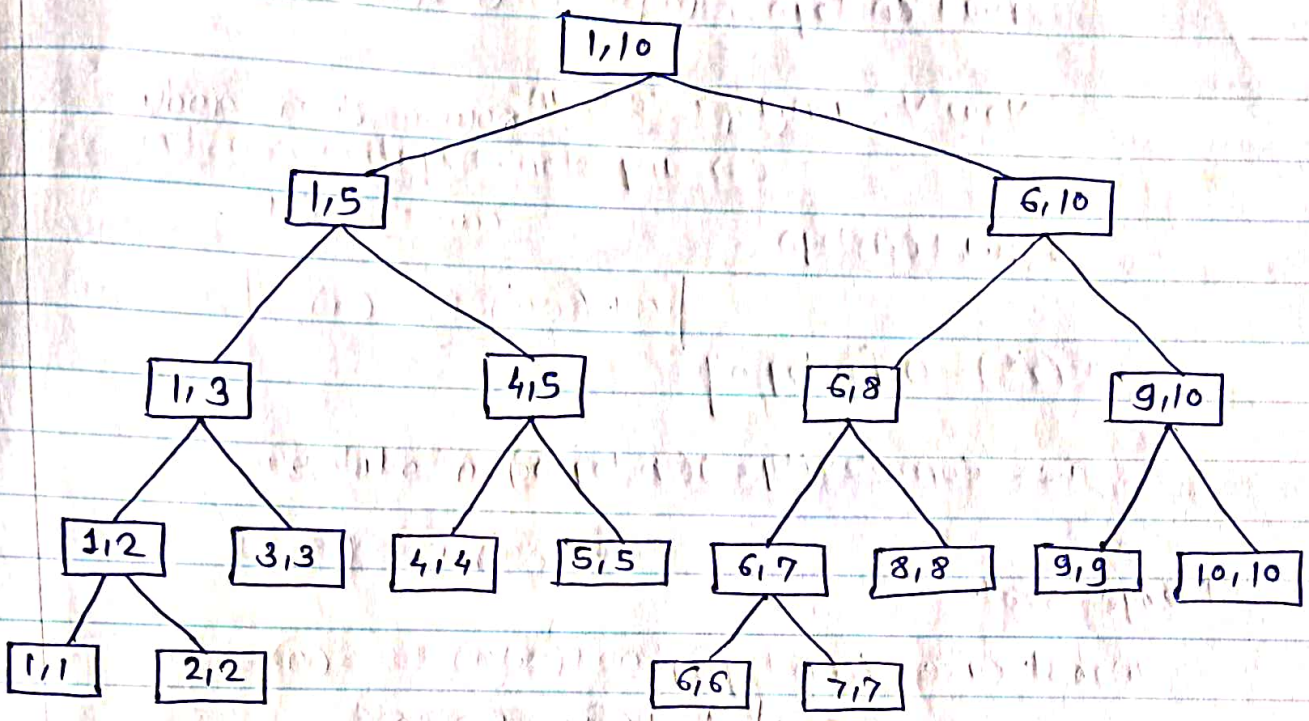
- Quick sort

  worst $O(n^2)$

  Best $O(n \log n)$

  average $O(n \log n)$

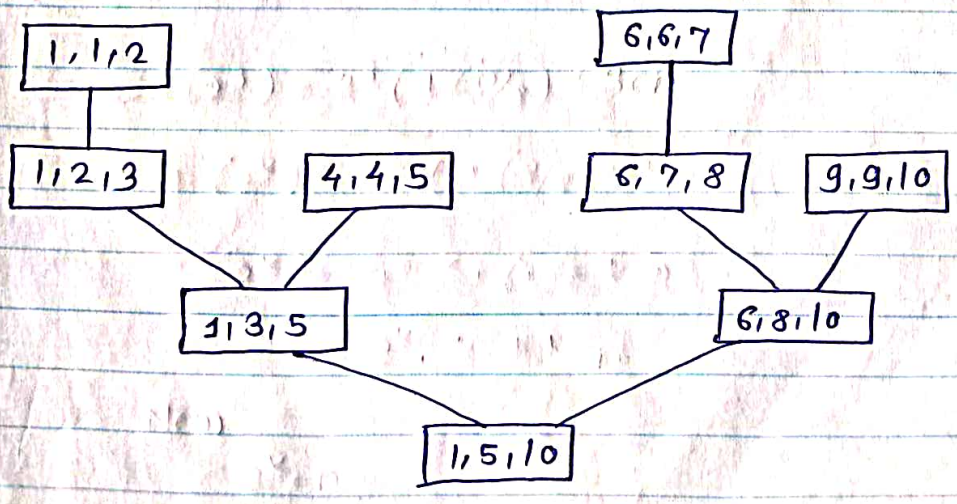
Tree calls of Merge Sort (1,10)



The time for merging operation is proportional to n. so computing the merging by,

$$T(n) = \begin{cases} a & n=1 \quad a \text{ is constant} \\ 2T(n/2) + cn & n>1 \quad c \text{ is constant} \end{cases}$$

Tree of calls of Merge:



Quick Sort:

Divide & conquer method is used but compared to merge there is no any merge of subarrays.

In quicksort the division into two subarrays is made so that the sorted array do not need to merged later this is done by rearranging the elements in a[1:n] such that a[i] ≤ a[j] for all [i between i & m &] j bet^n [m+1 & n]

$1 \leq m \leq n$

Elements in a[i:m] & a[m+1:n] are independently sorted No. Merge is needed.

Algorithm:

```
Algorithm quicksort (p,q)
{
  if (p<q) then,
  {
    j= partition (a, p, q+1);
    quicksort (p, j);
    quicksort (j+1, q);
  }
}
```

quick sort by partitioning

$$C(n) \leq \begin{cases} 2+s\left(\frac{n-1}{2}\right) & n>1 \\ 0 & n \leq 1 \end{cases}$$

Performance of Quicksort

The total no of count made in quick sort $C(n)$.

$\forall$, n is size of array

$$C(n) = n + C(k-1) + C(n-k) \quad \forall\ 1 < k < n$$

- n: No. of elements
- $C(k-1)$: sort left subarray
- $C(n-k)$: sort Right subarray by recursive call

$$\boxed{C(0) = C(1) = 0}$$

The sorting will possible if

$k=1$ or $k=n$

① Worst Case :

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

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

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

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

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

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

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

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

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

$$\approx \frac{n^2}{2} \times 2$$

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

$\frac{n^2}{2}$ calculated for either highest or lowest compasision it get twise

$\frac{n^2}{2} \times 2 = n^2$

When worst case &

When pivot no is too small or largest number.

2) Best case Complexity:

When the pivot is middle element,

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

$$= 2(2T(n/4) + \frac{cn}{2}) + cn$$

$$= 4T\frac{n}{4} + 2cn$$

$$= 8T\frac{n}{8} + 3cn$$

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

Now, $n = 2^k$

In above case $k=3$

$$T(n) = 2^k T(n/n) + kcn$$

$$= 2^k T(1) + kcn$$

$T(1) = a$

$$T(n) = 2^k \cdot a + kcn$$

put $n = 2^k$

$k = \log_2 n$

$$T(n) = 2^k \cdot a + \log_2 n \cdot cn$$

$$= a \cdot 2^k + c \cdot n \log n$$

Now by using big O Notation properly,

$$\boxed{T(n) = O(n \log n)}$$ best case complexity

3) Average Case Complexity :

The recurrence relation could be write as,

$$C(n) = n + C(k-1) + C(n-k)$$

$$\forall,\ C(0) = C(1) = 0$$

equation indicates value for n elements so for single element it will become,

$$C(n) = n + \frac{1}{n} \cdot \sum_{k=1}^{n} C(k-1) + \frac{1}{n} \sum_{k=1}^{n} C(n-k).$$

Now solve like follow,

$$\sum_{k=1}^{n} C(k-1) = \sum_{i=0}^{n-1} C(i) \quad \text{by putting } i = k-1$$

$$\sum_{k=1}^{n} C(n-k) = \sum_{i=0}^{n-1} C(i) \quad \text{by putting } i = n-k$$

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

$$\boxed{n \cdot C(n) = n^2 + 2 \sum_{i=0}^{n-1} C(i)} \quad \text{---} (1)$$

If we replace same recurrence by $n-1$ instead of n,

$$\boxed{(n-1)\, C(n-1) = (n-1)^2 + 2 \sum_{i=0}^{n-2} C(i)} \quad \text{---} (2)$$

Now substracting (2) from (1) get

$$n \cdot C(n) - (n-1)\, C(n-1) = n^2 - (n-1)^2 + 2\, C(n-1)$$

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

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

$n \cdot c(n) = (2n-1) + 2c(n-1) + (n-1)\,c(n-1)$

$= (n+1)\,c(n-1) + (2n-1)$

$n \cdot c(n) = (n+1)\,c(n-1) + (2n-1)$

Now dividing $n(n+1)$ on both sides,

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

$$\frac{[c(n)]}{(n+1)} = \frac{c(n-1)}{n} + \frac{2}{n} \quad \text{---③}$$

Approximation to above case

Let,

$$D(n) = \frac{c(n)}{n+1}$$

equation ③ becomes,

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

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

$= D(n-3) + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n}$

$= D(1) + \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n-2} + \frac{n}{n+1} + \frac{2}{n}$

$= 2 \cdot cn(n) - 2$

$\approx 2 \ln(n)$

$= 1.39 \log n$

$C(n) = (n+1) \cdot D(n)$

$C(n) = (n+1)\ D(n)$

$\approx 1.39\ n \log n$

$C(n) \leq 1.39\ (n+1) \log n$

$\boxed{C(n) = O(n \log n)}$ Average case

As compare to Best case Average case is only 0.39 i.e 39% get more comparisions.

As compared to merge Average complexity it require only 39% extra time.

Selection Sort:

- Complexity at worst, best & average $O(n^2)$
- selection sort is known for it's simplicity.
- If a[1:n] is an array then we required to determined k^{th} smallest element* in it

eg:

64, 25, 12, 22, 11

Compare a[1] & a[5]

11, 25, 12, 22, 64

Compare a[2] & a[3]

11, 12, 25, 22, 64

Compare a[3] & a[4]

$\boxed{11, 12, 22, 25, 64}$ final Result.

As compaired to other elements it is woost in all type of selection.

finding k^{th} smallest Element:

```
Algorithm select 1 (a,n,k)
{
  low=1 ; up=n+1;
  a[n+1]= ∞
  repeat
  {
    j= partition (a, low, up);
      if (k=j) then return;
    else if (k<j) then up=j;
    else low=j+1;
  } until (false);
}
```

By strong principle of mathematical induction,

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

$= [(n+1)] + [(n-1)+2] + [(n-2)+3]$

By combining starting & Last condition.

$= [(n+1) + (n+1) + (n+1) + \cdots$

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

$$= \frac{n^2}{n} + \frac{n}{2}$$

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

Now by using big O Notation we get,

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

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