Design & Analysis of Algorithms

Analysis of algorithm is the determination of the amount of time and space resources required to execute it.

Usually the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as time complexity, or volume of memory, known as space complexity.

Complexity in DAA :-

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem.

It evaluates the order of count of operations executed by an algorithm as a function of i/p datasize.

DESIGN & ANALYSIS OF ALGORITHMS.

What is an algorithm?

Persian author: Abu Jafar Mohammed ibn Musa al Khowarizmi (c.825 A.D)
Textbook on mathematics

Algorithm:-

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria

characteristics of algorithm:

1. Input: Zero or more quantities are externally supplied.
2. Output: At least one quantity is produced. "1 or more o/p"
3. Definiteness: Each instruction is clear and unambiguous.
4. Finiteness: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.

5. Effectiveness :- Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite, it also must be feasible.

* must be able to solve by pen & paper.

Algorithms that are definite and effective are also called "computational procedures"

A program is the expression of an algorithm in a programming language.

Study of algorithms :-

There are four distinct area of study. They are how to

1. ↳ devise algorithm
2. ↳ validate algorithm
3. ↳ analyze algorithm
4. ↳ test to program.

(1.) Devise algorithm:-

Creating an algorithm is an art which may never to fully automated

(2.) Validate algorithms:-

Once an algorithm is devised, it is necessary to show that it computes the correct answer for all possible legal inputs. This process is referred as "algorithm validation".

Algorithm	program
↳ Design	↳ Implementation
↳ Domain knowledge	↳ programming skill
↳ Any language	↳ programming lang.
↳ H/w & OS - No dependency	↳ Dependent on H/W & OS
↳ Analyze	↳ Testing
↳ prior analysis	↳ posteriori Testing.
↳ Time & space Efficiency.	↳ watch time & Bytes.

③ Analyze algorithms :-

When an algorithm is executed, it uses the computer's central processing unit (CPU) to perform operations and its memory (both immediate and auxiliary) to hold the program and data.

'Analysis of algorithms or performance analysis' refers to the task of determining how much computing time and storage an algorithm requires

④ Test a program :-

Two phases : debugging & profiling. (or performance measurement).

@. Debugging :- It is the process of executing programs on sample data sets to determine whether faulty results occur

and if so, to correct them.

(5.) Profiling / Performance measurement :-

It is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.

These timings figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization.

— X —

Algorithm design:-

An algorithm design refers to a method or a mathematical process for problem-solving and engineering algorithms. The design of algorithms is part of many solutions theories of operation research, such as dynamic programming and divide-and conquer.

How do you design an algorithm?

An algorithm is a plan for solving a pblm.

Step 1: Obtain a description of the problem. This step is much more difficult than it appears.

Step 2: Analyze the problem.

Step 3: Develop a high-level algorithm.

Step 4: Refine the algorithm by adding more detail

Step 5: Review the algorithm.

Describing the Algorithm:-

The skills required to effectively design and analyze algorithms are entangled with the skills required to effectively describe algorithms. A complete description of any algorithm has four components.

* **What:** A precise specification of the problem that the algorithm solves.
* **How:** A precise description of the algorithm itself.
* **Why:** A proof that the algorithm solves the problem it is supposed to solve.
* **How fast:** An analysis of the running time of the algorithm.

It is not necessary (or even desirable) to develop these four components in this particular order. Problem specifications, algorithm

descriptions, correctness proofs and time analyses usually evolve simultaneously with the development of each component informing the development of the others.

An algorithm can be specified either using

* Natural language
* Pseudo code
* Flow chart.

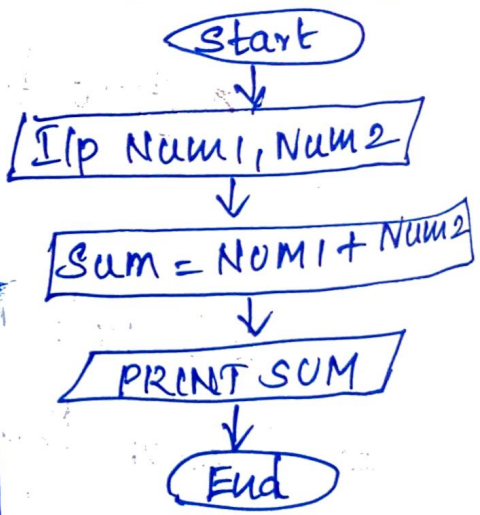
Natural language

Step1: Start
Step2: Declare variables num1, num2 and sum
Step3: Read values num1 & num2.
Step4: Add num1 & num2 & assign the result to sum.
Step5: Display sum
Step6: Stop.

Pseudo code:

```
BEGIN
NUMBER S1, S2, SUM
OUTPUT("Input No.1")
INPUT S1,
OUTPUT("I/P No.2:")
INPUT S2
SUM = S1+S2,
OUTPUT SUM.
END
```

Flowchart



@ NPSIN

★ Understanding Algorithms.

★ Correctness

★ Efficiency
↳ Asymptotic complexity, O() notation

★ Modelling
↳ Graphs, data structures, decomposing the problem.

★ Techniques.
↳ Divide and Conquer, greedy, dynamic programming.

↳ Asymptotic complexity:- (W1)

↳ Searching & sorting in arrays. (W2)
↳ Binary search, insertion sort, selection sort, merge sort, quick sort.

↳ Graphs and graph algorithms. (W3)
↳ Representations, reachability, connectedness
↳ Directed acylic graphs.
↳ Shortest paths, spanning trees (W4)

↳ Algorithm design techniques:

(W5) · Divide & conquer, greedy algorithms (W6)

Dynamic programming (W7)

↳ Data Structures:

(W6)

· Priority queues/heaps, Search trees, Union of disjoint Sets (union-find)

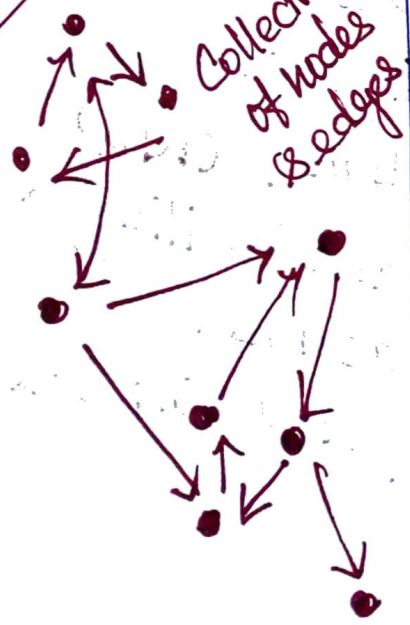
↳ Miscellaneous topics. (W8)

↳ Algorithms Design. Jon kleinberg

↳ Algorithms.

Ex: Air Travel :-

Graph:-



Xerox Shop:

Document Similarity

* Analysis of Algorithms :-

↳ Measuring efficiency of an algorithm.

- Time: How long the algorithm takes (running time)

Space: Memory requirement.

* Time and Space:

- Time depends on processing speed.

  Impossible to change for given H/w.

- Space is a function of available memory.

  Easier to reconfigure, augment.

⇒ focusing on time ≠ space.

* Measuring running time :-

- Analysis independent of underlying H/W.
  - ↳ Don't use actual time.
  - ↳ Measure in terms of "basic operations".
- Typical basic Operations.
  - ↳ Compare two values.
  - ↳ Assign a value to a variable.

- Other Operations may be basic, depending on context.
  - ↳ Exchange values of a pair of variables.

* Input Size :-

- ↳ Running time depends on i/p size.
  - Larger arrays will take longer to sort.
- ↳ Measure time efficiency as function of i/p size
  - Input size n
  - Running time t(n).
- ↳ Different i/p's of size n may each take a different amount of time.
- ↳ Typically t(n) is Worst case estimate

Ex: 1 Sorting

→ Sorting an array with n elements

- Native algorithms : time $\propto n^2$,
- Best algorithms : time $\propto n \log n$

→ How important is this distinction?

- Typical CPUs process up to 10^8 operations per second.
- Useful for approximate calculations.

* I/p size, Worst case, average case (a)

$\log_b n$. - we write n in base b.

$t(n)$ → focus on orders of magnitude
Ignore constants.

$f(n) = n^3$ eventually grows faster than
$g(n) = 5000\,n^2$.

∀ small values of n, $f(n)$ is smaller than $g(n)$

@ $n = 5000$, $f(n)$ overtakes $g(n)$.

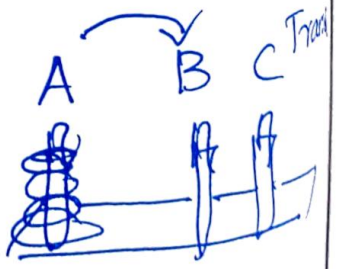
Examples :-

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

Linear
Quadratic

iterative & recursive.

Towers of Hanoi



Ex:

```
function matrixMultiply (A, B)
n  for i = 0 to n-1;
n  for j = 0 to n-1;
       c[i][j] = 0
n  for k = 0 to n-1
       c[i][j] = c[i][j] + A[i][k] * B[k][j]
   return c
```

$O(n^3)$ //.

Ex:

```
function numberofBits(n):
    count = 1
    while n > 1
        count = count + 1
        n = n div 2        integer division
    return count.
```

count = 1, 2, 3, 4

$n, \frac{n}{2}, \frac{n}{4}, \ldots 1$

$n = 2 \times 2 \times 1$

$\log_2 n$

18. (a)

2 steps.

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

$M(1) = 1$

Recursive expression for M(n).

WI:3 Quantifying efficiency:- O(), omega(), Theta()

Comparing time efficiency:-

- ✓ We measure time efficiency only upto an order of magnitude.
  - Ignore constants.
- ✓ How do we compare functions with respect to orders of magnitude?

pseudocode :-

Pseudocode is an informal way of programming description that does not require any strict programming language syntax or underlying technology considerations.

It is used for creating an outline/ rough draft of a program.

Pseudocode summarizes a programs flow, but excludes underlying details.

Pseudo: • being apparent rather than actually stated.

- defn. of pseudo is someone or something fake, false or pretend.
- close resemblance to.

Pseudo conventions :-

- // comments
- { } blocks.
- Identifier → lv..

- ; ( ), → oprtr..

node=record
{ datatype1. data1
datatype n. data n

node *link.

- Assign := (variable) := (expry.);
- Boolean values < , ≤ , ≥ & >
- [ ] - xtfy dim/. array access.
  Ex: A[i,j].
  [Arr-ind:o]
- looping statements:

```
While (cond) do
{ (st.1)
  :
} (st.n)
```

```
for var := val1 to val2 step step do.
{
  (Stat.1)
    :
} (Stat.2)
```

- repeat-until

```
repeat
  (st.1)
    :
  (st.n)
until (condtn/.)
```

- break
- return.

- Conditional statement:

```
if (cond) then (stat.)
if (cond) then (stat.1) else (stat.2)
```

- xle decision

```
case
{   :Cond.1): (st-1)
    :
}   :cond.n):(st.n)
    else (stat. n+1).
```

- Ip, O/p : rd, wr.
- procedure:

```
Algo. (head is body)
Algo. Name{(parameter list)}.
```

Example of pseudo code:-

```
Algorithm Max(A,n)
// A is an array of size n
Result := A[i]
for i:=2 to n do
    if A[i] > Result
Then Result := A[i];
return Result;
}.
```

```
Alg.nm: Max.
A
n }->proc.para {
Result, i - loc. var.
```

Analysing Algorithm:-

- **Correctness:-** ∀ reasonable i/p's, ∀ possible i/p's.
  - usually involves induction
  - ∀ correctness proof.
  - # by trusting instincts / by trying few testcases, ≠ good enough.
- **Running time:**
  - Common way of ranking diff. algo's for the same pblm.
  - Need fatest possible algo ∀ any particular pblm.

Summary:-

pblm → algo/ → analysed → C++
pseudo code.

Correctness of algorithm:-

Methods:- @ ↓pg.27.

proof by
- Counter example
- Induction
- Loop invariants

other approaches
proof by
cases/enumeration.
chain of iff's.
contradiction.
contrapositive.

Loop invariants:-

Invariants: assertions that are valid any time they are reached (many times during the execution of an algorithm, eg. in loops)

3 things to show about loop invariants.

- **Initialization:** It is true prior to the first iteration
- **Maintenance:** If it is true b4 an iteration, it remains true b4 the next iteration.
- **Termination:** When loop terminates the invariant gives a useful property to show the correctness of algorithm.

Methods of proving correctness.

2 prove an algo is correct by i.e Proof by:

- Counter example [indirect proof]
- Induction [direct proof]
- Loop invariant.

other approaches:

- proof by cases/enumeration
- proof by chain of iffs.
- proof by contradiction
- proof by contrapositive.

Assertions:-

To prove correctness we associate a no. of assertions (statements about the state of the execution) with specific checkpoints in the algorithm.

Eg. A[1], ... A[k] form an ↑sing seq.

- **preconditions:-**

. postconditions - assertions that must be valid after the execution of a algorithm or a subroutine.

Correctness of algorithms:-

Loop Invariants (Ref. Pg. 25)

A proof of correctness requires that the solution be stated in two forms.

. One form is usually as a program which is annotated by set of assertions about the i/p and o/p variables of the program. These assertions are often expressed in the predicate calculus.

. The second form is called a specification, & this may also be expressed in the predicate calculus.

A proof consists of showing that these two forms are equivalent in that for every given legal i/p, they describe the same o/p.

* More valuable than a thousand times.
∴ it guarantees that the pgm will work correctly ∀ possible i/p's.

a. Pg25

Proof by Mathematical Induction:-

Mathematical Induction (MI):-

It is an essential tool for proving the statement that proves an algorithms correctness.

The general idea of MI is to prove that a statement is true for every natural number n.

It contains 3 steps:

1. **Induction Hypothesis:**

Define the rule we want to prove for every n, let's call the. f(n)

2. **Induction Base:** - 2 prove that a statement is true for initial value.

Proving ~~that~~ the rule is valid for an initial value, or rather a starting point that is often proven by solving the Induction Hypothesis f(n) or f(n)=1 or whatever initial value is appropriate.

3. Induction step:-

Proving that if we know that f(n) is true, we can step one step forward and assume f(n+1) is correct.

Ex:-

Let us define S(n) → as the Σ [first n natural no's]

∨ ex:- S[3] = 3+2+1.

To prove that the following formula can be applied to any n.

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

Proof:-

1. Induction Hypothesis.

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

2. Induction base: S(1)

$$S(1) = \frac{(1+1)*1}{2} = \frac{2}{2} = 1$$

$$(1) = 1.$$

Induction step:

if factor $\Rightarrow S(n)$

then it should also applies to $S(n+1)$

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

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

This is known as implication $(a \Rightarrow b)$, which just means that we have to prove b is correct providing we known a is correct

Hence.

$$S(n+1) = S(n) + (n+1) \longrightarrow (a)$$

$a \Rightarrow b$ $\rightarrow S(n+1) = S(n) + (n+1)$

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

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

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

$$= \frac{n2+n+2n+2}{2}$$

$$= \frac{n2+3n+2}{2} = \frac{(n+2)(n+1)}{2}$$

Note that $S(n+1) = S(n) + (n+1)$ means we just recursively calculating the sum.

$\therefore S(3) = S(2) + 3$

$= S(1) + 2 + 3$

$= 1 + 2 + 3 = 6$

Hence proved //:

Sum up:-

For every algorithm proof of correctness is very important to show the program is correct in all cases.

1.
```
for(i=0; i<n; n++)   -n+1
{
  stmnt;   ——— n
}
```
$2n+1$

$O(n)$

2.
```
for (i=n; i>0; i--)
{
  stmnt;   — n
}
```
$O(n)$

3.
```
for (i=1; i<n; i=i+2)   (20)
{
  stmnt;   — n/2
}
```
$f(n) = n/2$

$O(n)$

4.
```
for(i=0; i<n; i++)  → n+1
{ for(j=0; j<n; j++)  → n x (n+1)
  { stmnt;  (n x n)
```
$O(n^2)$

(3a)

★ Performance Analysis :-

- Time Complexity
- Space Complexity
- Communication Bandwidth

Time Complexity analysis

The time complexity $T(P)$ taken by a program P is

$$T(P) = \text{Compile time} + \text{run (exec) time}$$

- Compile time does not depend on the instance characteristics.
- Also, we may assume that a compiled program will be run several times without recompilation.
- Consequently we concern ourselves with just the run time of a program.
- The run time is denoted by t_p. (instance c...

(5)

```
for(i=0; i<n; i++)
{
  for(j=0; j<i; j++)
  { stmt; }
}
```

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

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

i	j	No. of times
0	0	1
1	0	1
	1	x
2	0	2
	1	
	2	x
3	0	3
	1	
	2	
	3	x
n		

P.T.O.

* Bcoz many of the time we did no know, it is difficult to arrive with one such following eqn.

$$t_p(n) = c_n ADD(n) + c_s SUB(n) + c_m Mul(n) + c_d DIV(n) + \cdots$$

If compiler chs/. & time are known, we can calculate $t_p(n)$ *↑

* Expmntl approach → depends on other pgms/ running on the computer

Solution:-

Obtain a count for the total no/. of oprtns/ in the algorithm. (count 1 by the no/. of pgm steps,

The number of steps any program statement is assigned depends on the kind of the statement.

~~Ex:~~-

- Comments → Zero steps.
- assignment statement → one step. (which doesn't invoke any calls to other algo's.
- Iterative statement → control part of stmt. (for, while and repeat-until statements - considering only the control parts of the stmts.

The control parts for 'for' & 'while' statements have the following forms.

for i = <expr> to <expr1> do

while {<expr>} do.

One can determine the number of steps needed by a program to solve a particular problem instance in two ways.

In the first method we introduce a

global variable with initial value 0. Statements to increement count by the appropriate amount are introduced into the program. This is done each time a statement in the original program is executed, count is incremented by the step count of that statement.

Ex:-

```
Algorithm Sum(a, n)
{   s:0.0;      count:=count+1
    for i:=1 to n do    count:=count+1
    {   s:=s+a[i];      count:=count+1
        count:=count+1
    }
    count:=count+1
    (V last time for)
    return s;
    count:=count+1
}
```

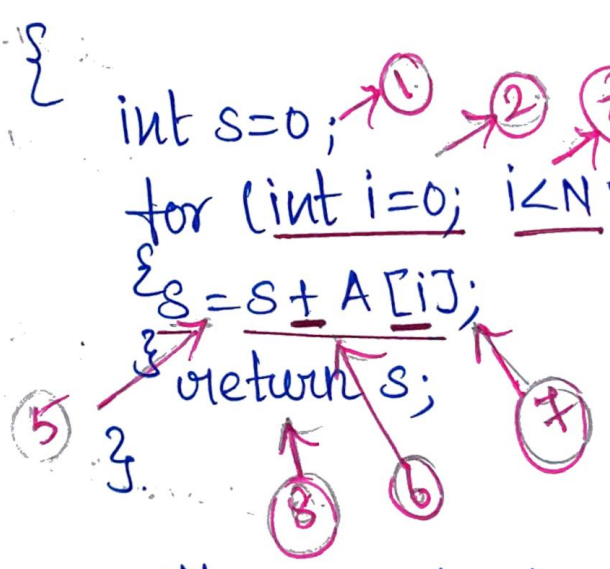
Two methods to find the time complexity for an algorithm

1. Count Variable method.
2. Table method.

Using Count method:-

```
// Input : int A[N], array of N integers.
// output : Sum of all no's in array A.

int Sum (int A[], int N)
{
    int s=0;                    — 1
    for (int i=0; i<N; i++)     — 1+n+1
    {s=s+A[i];                  — n
    }
    return s;                   — 1
}                               5n+3
```



∴ The complexity function of the algorithm f(n) is 5N+4

1, 2, 8 → 1.

3, 4, 5, 6, 7 → N (N iteration)

∴ Total → 5N+3

Table method :-

The table contain s/e and frequency.

s/e → s/e of a statement is the amount by which the count changes as a result of the execution of that statement.

frequency → It is defined as the total no. of times each statement is executed.

By combining s/e & frequency, the step count for the entire algorithm is obtained.

Ex:1

	Statement:	s/e	freq	Total steps
1	Algorithm: Sum (a, n)	0	—	0
2	{	0	—	0
3	s := 0.0	1	1	1
4	for i := 1 to n do	1	n+1	n+1
5	s := s + a[i];	1	n	n
6	return s;	1	1	1
7	}	0	—	0
		Total		2n+3

Ex: 2

	Statement	S/e	Freq n=0	Freq n>0	Total Steps n=0	Total Steps n>0
1	Algorithm Rsum(a,n)	0	0	—	0	0
2	{	0	0	—	0	0
3	if (n<=0) then	1	1	1	1	1
4	return 0.0;	1	1	0	1	0
5	else return					
6	Rsum(a, n-1) + a[n];	1+x	0	1	0	1+x
7	}	0	—	—	0	0
	Total				2	2+x

$x = t_{Rsum}(n-1)$

Space Complexity:-

The space complexity of an algorithm or a computer program is the amount of memory space required to solve an instance of the computational problem as a function of characteristics of the i/p.

It is the memory required by an algorithm until it executes completely.

otherwise we can say.

"Space complexity of an algorithm quantifies the amount of space or memory taken by an algorithm to run as a function of the length of the i/p".

$$S(p) = C + Sp(I)$$ //.

Fixed Space Requirements (C)

- Independent of the char. of the i/p's and o/p's.
- instruction space.
- space for simple variables, fixed-size structure variables & constants.

* Variable space Requirements.

↳ depends on the instance characteristic

↳ no., size, value of i/p's & o/p's associa with I

↳ recursive stack space, formal parameters, local variables, return address.

Ex:-1

```
#include<stdio.h>
int main()
{ int a=5, b=5, c
  c=a+b;
  printf("%d", c)
}
```

space complexity :- size(a)+size(b)+ size(c)

Let size of (int) = 2 bytes.

= 2+2+2 = 6 bytes.

=> O(1) or constant. //

Ex:2 :-

```
#include<stdio.h>                          ~1
int main()
{
  int n, i, sum=0;                         -1+1+1
  scanf("%d",&n);                          -1
  int arr[n];                              -n
  for(i=0; i<n; i++)                       -n+1
  {
     scanf("%d", &arr[i]);                 -n
     sum = sum+ arr[i];                    -n+n
  }
  printf("%d", sum);                       -1
}
```

assume 4 bytes of each variable

n	- 4
i	- 4
sum	- 4
arr[n]	- 4n
	4n+12

The array consists of n integer elements. So, the space occupied by the array is 4*n. Also we have integer variables such as n, i and sum. Assuming 4 bytes for each variable,

the total space occupied by the program is $4n+12$ bytes.

∵ the highest order of n in the equation $4n+12$ is n, so the space complexity is $O(n)$ or linear.

(5.)

```
p=0;
for(i=1; p<=n; i++)
{
    p=p+i;
}
```

i	p
1	0+1=1
2	1+2=3
3	1+2+3
4	1+2+3+4
⋮	⋮
k	

1+2+3+4...+k

Assume $p > n$

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

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

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

$$= k^2 > n$$

$$k = \sqrt{n}$$

$O(\sqrt{n})$

Algorithm Design and Analysis & ADP

Algorithm Design Paradigms:

Efficiency of the algorithm's design is totally dependent on the understanding of the problem.

The important parameters to be considered in understanding the problem are

- Input
- Output
- Order of Instructions
- check for repetition of Instructions
- check for the decisions based on conditions.

Specify the pattern to write or design an algorithm

Algorithm Design Paradigms

Various algorithm paradigms are.

1. Divide and Conquer
2. Dynamic programming
3. Backtracking.

↳ 4 Greedy Approach

↳ 5 Branch and Bound.

Selection of the paradigm depends upon the problem to be addressed.

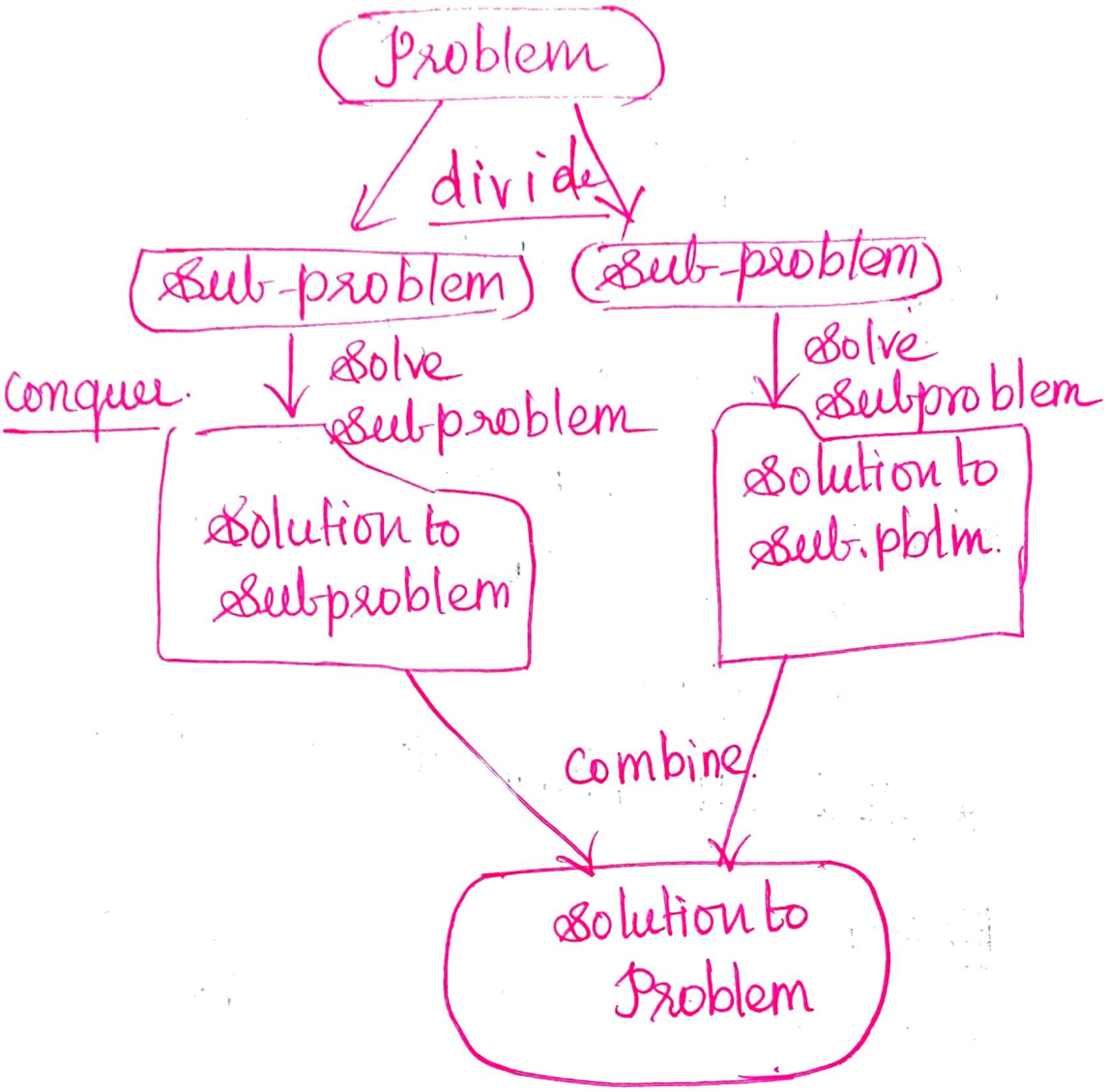
★ Divide and Conquer :-

The divide and Conquer paradigm is an algorithm design paradigm which uses this simple process. It divides the problem into smaller sub-parts until these sub-parts become simple enough to be solved, and then the sub parts are solved recursively, and then the solutions to these sub-parts can be combined to give a solution to the original problem.

Ex:- Binary Search

Merge Sort

Quick Sort.


Problem
divide
Sub-problem
Sub-problem
Solve
Sub-problem
Solve
Subproblem
Conquer
Solution to
Subproblem
Solution to
Sub. pblm.
Combine
Solution to
Problem

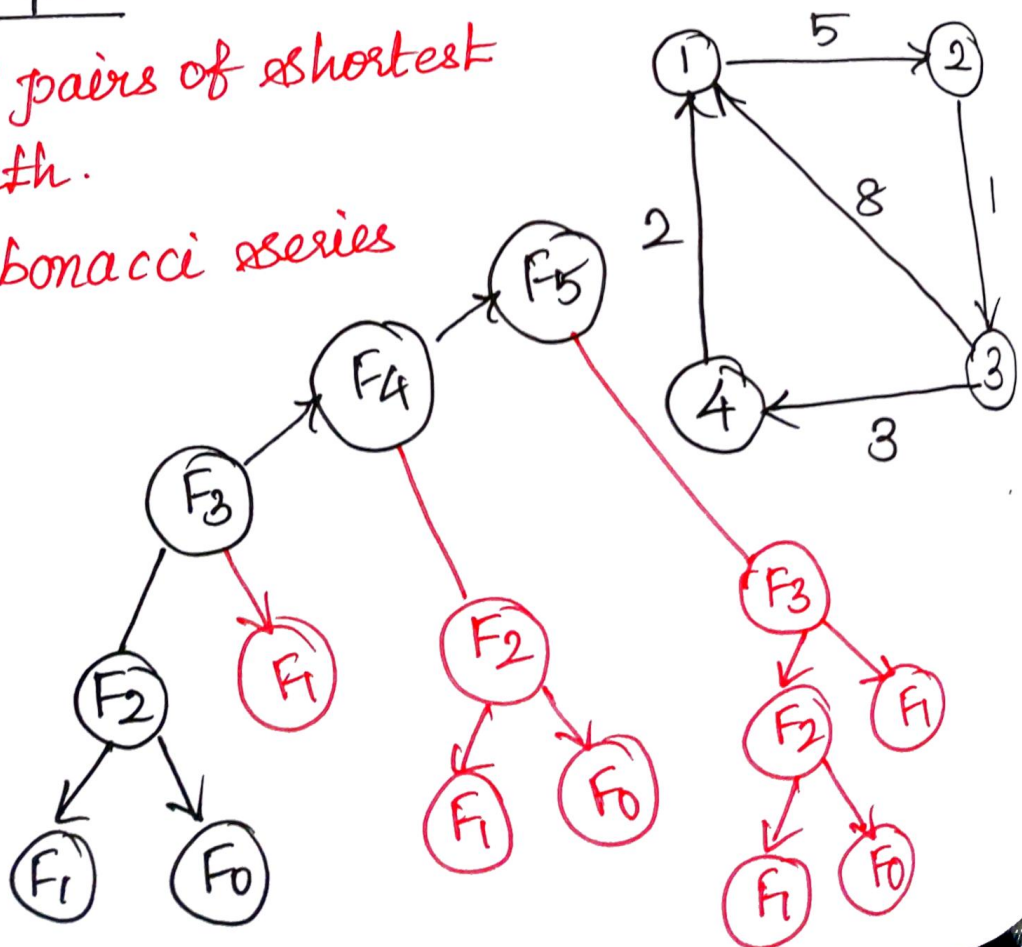
★ Dynamic Programming :-

Dynamic programming is an algorithm paradigm that solves a given complex problem by breaking it into subproblems and stores the result of sub-problems to avoid computing the same results again.

* It is an optimization technique.

Examples :-

- All pairs of shortest path.
- Fibonacci series



★ Backtracking:-

Backtracking is an algorithmic paradigm aimed at improving the time complexity of the exhaustive search technique if possible. Backtracking does not generate all possible solutions first and checks later.

It tries to generate a solution and as soon as even one constraints fails, the solution is rejected and the next solution is tried.

A backtracking algorithm tries to construct a solution incrementally, one small piece at a time. It's a systematic way of trying out different sequences of decisions until we find one that works.

Ex:-

- 8 Queens problems
- Sum of subsets.

Greedy Approach:-

Greedy algorithms build a solution part by part, choosing the next part in such a way, that it gives an immediate benefit.

This approach is mainly used to solve optimization problems.

Finding the shortest path b/w two vertices using Dijkstra algorithm.

Examples:-

- Coin exchange pblm.
- Prim's
- Kruskal's algorithm.
- Travelling salesman pblm.
- Graph-map coloring

Insertion Sort :-

* Consider first element gets sorted.

Sorted | Unsorted.

9	2	7	5	1	4	3	6

Max: 1 comp 1 swap.

Way of Working

Select - Compare - Shift - Insert.

I st pass

0 1 2 3 4 5 6 7

Insert 2.

9	7	5	1	4	3	6

2

2	9	7	5	1	4	3	6

II nd pass.

2	9	5	1	4	3	6

2 comp 2 swap.

7

2	7	9	5	1	4	3	6

Sorted | Unsorted.

III rd pass

2	7	9	1	4	3	6

5

3 comp 3 swap.

2	5	7	9	1	4	3	6

IV th pass.

2	5	7	9	4	3	6

4 comp 4 swap.

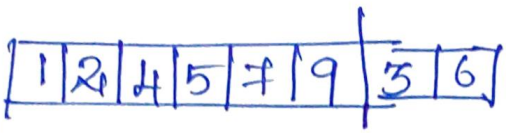
1

1	2	5	7	9	4	3	6

V th pass

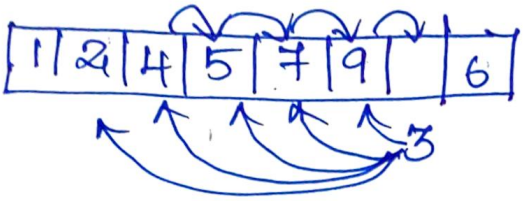
1	2	5	7	9	3	6

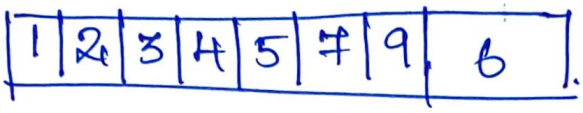
4



Max:
~~5~~ - comp.
~~5~~ - swap.

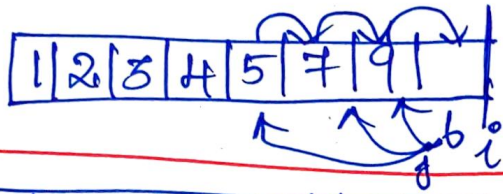
VI th pass





6 : comp
6 swap.

VII th pass



7 - comp.
7 - swap.

No. of ~~swaps~~: (n-1) pass.

No. of comp : $1+2+3+4+5+\cdots+n$

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

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

So. This polynomial of degree n^2.

As the no: of comparisons are taken as the ~~time~~ Complexity.

$\therefore O(n^2)$.

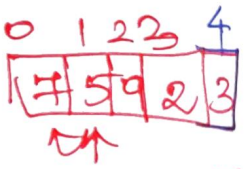
Swaps III^r No. of Comp $\Rightarrow n^2 \Rightarrow O(n^2)$

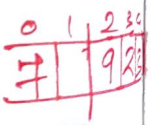
Insertion sort in c is the simple sorting algorithms that virtually splits the given array into sorted and unsorted parts, then the value from the unsorted parts are picked and placed at the correct position in a sorted part.

Algorithm: Insertion sort

```
Algorithm: Insertion sort (A)
{
// repeating passes.  for i = 1 to n do
                      {
                        key = A[i]
                        j = i-1.
/* Move elements of arr[0...i-1], that are greater than key to one position ahead of their cpt. position*/
                        while (j>=0 && A[j] > key)
                        {
                          A[j+1] = A[j];
                          j = j-1
                        }
                        A[j+1] = key.
                      }
}
```

n=5





Shifting the elements.

		Cost	Times
1.	for (i=1; i<n; i++)	c_1	n
2.	key = A[i]	c_2	$n-1$
3.	j = i-1	c_3	$n-1$
4.	While (j>=0 && A[i] > key)	c_4	$\sum_{j=1}^{n} t_j$
5.	A[j+1] = A[j]	c_5	$\sum_{j=1}^{n} (t_j - 1)$
6.	j = j-1	c_6	$\sum_{j=1}^{n} (t_j - 1)$
7.	A[j+1] = key.	c_7	$n-1$

Running time of the Inserting Sort:-

The running time of the algorithm is

$\sum$ (cost of statement) x (number of times statement is executed). all statements

Let T(n) = running time of Insertion Sort

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=1}^{n} t_j + c_5 \sum_{i}^{n} (t_i - 1) + c_6 \sum_{i}^{n} (t_i - 1) + n-1$$

Thus the running time depends on the values of t_i and these vary according to the i/p.

Insertion sort : Line count & Operations count.

```
Insertion_Sort(A)
1.  for j = 2 to A.length.
2.      key = A[j]
3.      Insert A[j] into sorted seq.
4.      i = j-1
5.      While i>0 && A[i]>key
6           A[i+1] = A[i]
7           i = i-1
8       A[i+1] = key.
```

Move elements as well as

Insert in Correct position.

A[i] A[j] = key.

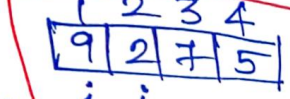
i = j-1.

A[i] > key.

A[i+1] = A[i]

i-1

A[i+1] = key.



Key = A[j] = 2

i = j-1

> 2

A[i+1] = A[i].

i = i-1

A[i+1] = key.

Algorithm: Insertion sort A

```
1. for j = 2 to A.length
2.     key = A[j]
3.     // Insert A[j] into the sorted sequence A[1...j-1].
4.     i = j-1
5.     while i>0 and A[i] > key.
6.         A[i+1] = A[i]
7.         i = i-1
8.     A[i+1] = key.
```

Average Case :- partially sorted $\{t_j = j/2\}$

$$T_n = c_1(n) + c_2(n-1) + c_4(n-1) + c_5\sum_{j=2}^{n} j/2 + c_6\sum_{j=2}^{n}(j/2-1) + c_7\sum_{j=2}^{n} c_{j/2-1} + c_8(n-1)$$

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

$$\sum_{j=2}^{n}(j/2-1) \Rightarrow \sum_{j=2}^{n} j/2 - \sum_{j=2}^{n} 1$$

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

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

$$= \frac{n^2+n-2}{4} - (n-1)$$

$$= \frac{n^2+n-2-4n-4}{4}$$

$$= \frac{n^2-3n-2}{4}$$

$$= \frac{n(n-3)}{4} - 1/2$$

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

Worst case: (Unsorted array) $t_j = j$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + \quad \rightarrow (b)$$

When $t_j = j$ $c_8(n-1)$

$$\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \left[\frac{n(n+1)}{2} - 1\right] = \frac{n^2+n-2}{2} \quad (3.)$$

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

$$= n^2 + n - 2 - 2n + 2$$

$$= \frac{n^2 - n}{2} = \frac{n(n-1)}{2} \; // \quad (4.)$$

Sub (3.) & (4.) in b.

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n^2+n-2}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$$= c_1 n + c_2 n - c_2 + c_4 n - c_4 + \frac{c_5}{2} n^2 + \frac{c_5 n}{2} - c_5 + \frac{c_6 n^2}{2} - \frac{c_6 n}{2} + \frac{c_7 n^2}{2} - \frac{c_7 n}{2} + c_8 n - c_8$$

$$= n^2\left(\frac{C_5 + C_6 + C_7}{2}\right) + n\left(\frac{C_5}{2} - \frac{C_7}{2} - \frac{C_6}{2} + C_1 + C_2 + C_4 + C_8\right) - (C_4 + C_5 + C_8)$$

$$= n^2\left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right) + n\left(\frac{C_8}{2} + \frac{C_4}{2} + \frac{C_5}{2} + C_7 + C_8\right)$$

$$= C \quad (\text{or})$$

$$= C_1 n + C_2 n - C_2 + C_4 n - C_4 + \frac{n^2 C_5}{2} + \frac{nC_5}{2} - C_5 + \frac{n^2 C_6}{2} - \frac{nC_6}{2} + \frac{n^2 C_7}{2} - \frac{nC_7}{2} + nC_8 - C_8$$

$$= \frac{n^2 C_5}{2} + \frac{n^2 C_6}{2} + \frac{n^2 C_7}{2} + \frac{nC_5}{2} + \frac{nC_7}{2} - \frac{nC_6}{2} + nC_1 + nC_2 + nC_4 + nC_8 - C_4 - C_5 - C_8.$$

$$= n^2\left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right) + n\left(\frac{C_5}{2} - \frac{C_7}{2} - \frac{C_6}{2} + C_1 + C_2 + C_4 + C_8\right) - [C_4 + C_5 + C_8]$$

$$= an^2 + bn + c.$$

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