Algorithm Designs

Algorithm

Algorithm is a set of well-defined instructions to solve a particular problem. OR

set of steps used to complete a specific task.

Takes a set of input(s) and produces the desired output.

Algorithms provide a better understanding towards programming and helps in building better problem solving logic.

Algorithms help the programmer to write code faster by following steps in Algorithm.

Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.

Ex.

Preparing tea -

boil a pot of water.

add the ingredients.

Boil for a minute

Strain the tea

For example - An algorithm to add two numbers:

Take two number inputs

Add numbers using the + operator

Display the result

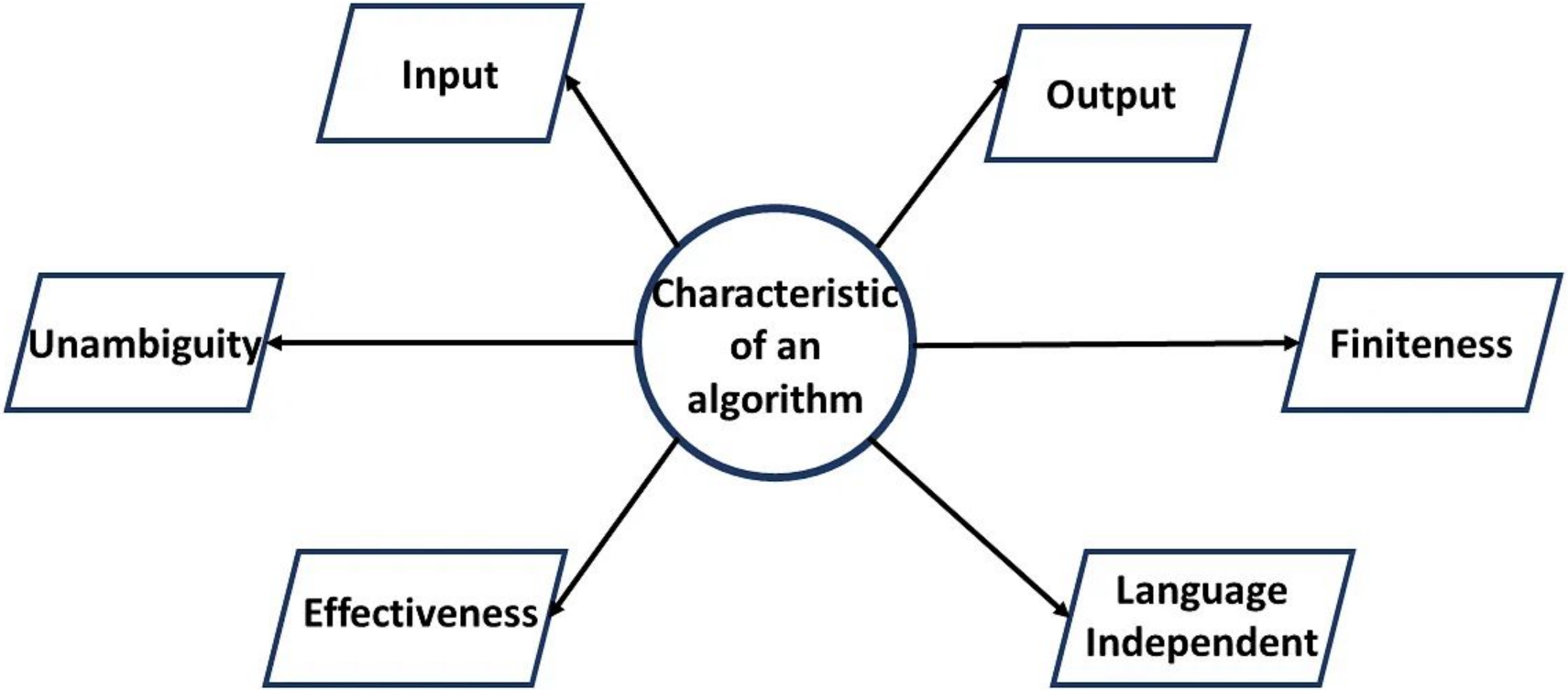
Qualities of a Good Algorithm -

Input and output should be defined precisely.

Each step in the algorithm should be clear and unambiguous.

Algorithms should be most effective among many different ways to solve a problem.

An algorithm shouldn't include computer code. Instead, the algorithm should be written in such a way that it can be used in different programming languages.


Input
Output
Characteristic of an algorithm
Unambiguity
Finiteness
Effectiveness
Language Independent

Analysis of an Algorithm

Measures the performance of an algorithm.

Affecting factors are the space and time complexities.

Process of finding the computational complexity of any algorithm.

Amount of time taken, space, and any other resources needed to execute(run) the algorithm.

The goal of algorithm analysis is to compare different algorithms that are used to solve the same problem.

To evaluate which method solves a certain problem more quickly, efficiently, and memory-wise.

Time complexity is - number of steps taken to execute it based on input size.

space complexity - amount of storage needed by the algorithm to execute and also considers the space for input values with it.

Asymptotic notations -

Three asymptotic notations

Big-O notation Worst case - gives upper bound of time taken to execute algorithm

Omega notation Ω Best case - gives us the lower bound on the running time of the algorithm for any given input.

Theta notation θ Average case - takes the sum of the running time on every possible input, and after that, it takes the average

Space Complexity -

Space Complexity = Auxiliary Space + Space used for input values

For the code instructions like -if/else, return any other statement needs constant space to execute.

Generally space complexity means the space required for execution of program from compilation to execution

1. Instruction Space:to keep or store the code while compilation so as to execute it further. So it’s a space where that code is stored.

2. Environmental Stack: stores the address of the partially executed functions. In Case of recursive fun. Or data of one function is used for other in that case stack reserves memory

3. Data Space: to store the data, variables, and constants of the program

nowadays during the calculation of space complexity of any program, we just consider the data space and ignores the

Examples of Space complexities

Sum Of N Natural Number

```
int sum(int n)
{
 int i,sum=0;
 for(i=n;i>=1;i--)
 sum=sum+i
 return sum;
}
```

input value is 'n'(1 input)so constant will take the space of O(1).

auxiliary space = O(1) because 'i' and 'sum' are also constants.

Total Space Complexity - O(1)

Hence total space complexity is O(1).

2. function sum_of_numbers(arr[],N){

```
    sum=0
    for(i = 0 to N){
      sum=sum+arr[i]
    }
    print(sum)
}
```

Total space complexity= (4*N + 12)bytes

→ 12 bytes considered as constant

→ so it will be totally depend on n ⇒ O(n)

3. factorial(N){

```
    int fact=1;
    for (int i=1; i<=N; i++)
    {
      fact*=i;
    }
    return fact;
}
```

Total Space Complexity = 4*4=16 bytes

4 variable so ⇒ it is considered as constant

algorithm will take constant space that is "O(1)".

4. Factorial of a number(Recursive)

```
factorial(N){
   if(N<=1)
   {
     return 1;
   }
   else
   {
     return (N*factorial(N-1));
   }
}
```

Total Space Complexity = (4 + 4*N) bytes

Function called recursively "N" times so here the complexity of auxiliary space will be "4*N bytes" where N is the number of which factorial have to be found.

By removing constant finally say that this algo have a complexity of "O(N)".

Examples of Time complexities - various cases

1. **O(C)**

```
function(){
        sop("Hello")
        //Some no. of lines of code
}
```

- Requires c no. of seconds like if 4 lines are there requires 4 s if is for a line assumed — so this will be

2. **O(n)**

Any loop run for n times 1-n//0-n-1//n-0

1 time n+1 times n time

```
for(i=0 ;       i<n;      i++){  //for loop runs for n+1 times
      sop(i);      //inner line will be executing for n time
}
```

Total time required = > n+1 +n =>2n +1 ⇒ O(n)

3. For(i=; i<n;i++) —> n+1
For(j=; j<n;j++) —> n X n+1 n+1+n^2+n+n^2 ⇒ 2n^2+2n+1
sop(i*j) —> nXn **O(n^2)**

4. for(i=0;i<n;i++)
for(j=0;j<i;j++) 0+1+2+3+4+....+n-1+n⇒n(n+1)/2⇒n^2
sop() **O(n^2)**

p=0;
5. for(i=1;p<=n;i++) when i=1, 2, 3,4,5,....,k
p=i+p p=0+1,0+1+2,0+1+2+3,...,0+1+2+..+k
p=k(k+1)/2 assume p>n || p==n
k^2=n ⇒ k=sqrt(n) ⇒ **O(sqrt(n))**
6. for(i=1;i<n;i=i*3) → **O(log n base 3)**

6. for(i=1;i<n;i=i*2) i= 1, 2, 4, 8, 16, ..., 2^k assume i=n
sop(i) 2^k=n ⇒ k log2=logn ⇒ k=**O(log n base 2)**

1X2X2X2....X2 ⇒k time

2^k =n ⇒ k log2=logn ⇒ k=log n base 2

7. for(i=n;i>=1;i=i/2) i= n, n/2, n/4, n/8, n/16, ..., n/2^k assume i<1
sop(i) n/2^k<1 ⇒ n = 2^k ⇒ **log n base 2** =k

8. for(i=0;i*i<n;i++) i^2=n ⇒ i=sqrt n → **O(sqrt n)**

p=0;

9. for(i=1;i<n;i=i*2) p= log n
P++

for(j=1;j<p;j=j*2) log p ⇒ **O(log log n)**

10. for(i=0;i<n;i++) —> n+1
for(i=1;i<n;i=i*2) —> n X log n ⇒ n+nlogn+nlogn ⇒ 2n logn +n+1
O(nlogn)

Recurrence Relations

```
print(n){
      if(n>0){
            sop(n);    →1
            print(n-1)  → T(n-1)
}}
      n-k=0    n=k⇒T(n)=T(0)+k
n-k=o  ⇒ n=k

T(k)=T(0)+n ⇒1+O(n)
```

Relation ={ 1 n=0;

{T(n-1)+1 n>0

T(n)=T(n-1)+1 —--- 1

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

T(n)=T(n-2)+2 —--- 2

T(k)=T(n-k)+k —----- k

Recursion

A function that calls itself is known as a recursive function.
technique is known as recursion.

```
void recurse()  //recursion function
{
    … .. …
    recurse();  //recurssive call
    … .. …
}
int main()
{
    … .. …
    recurse();
    … .. …
}
```

A recursive function must have at least one condition where it will stop calling itself.

The condition that stops a recursive function from calling itself is known as **the base case**.

internally stack is maintained for any function call for every function call one active record will be added into stack. Where whole function code will be loaded for execution.

The place recursive call is executed in program no. of entries added into stack till the base condition is not reached

Types of Recursion

Direct & indirect Recursion

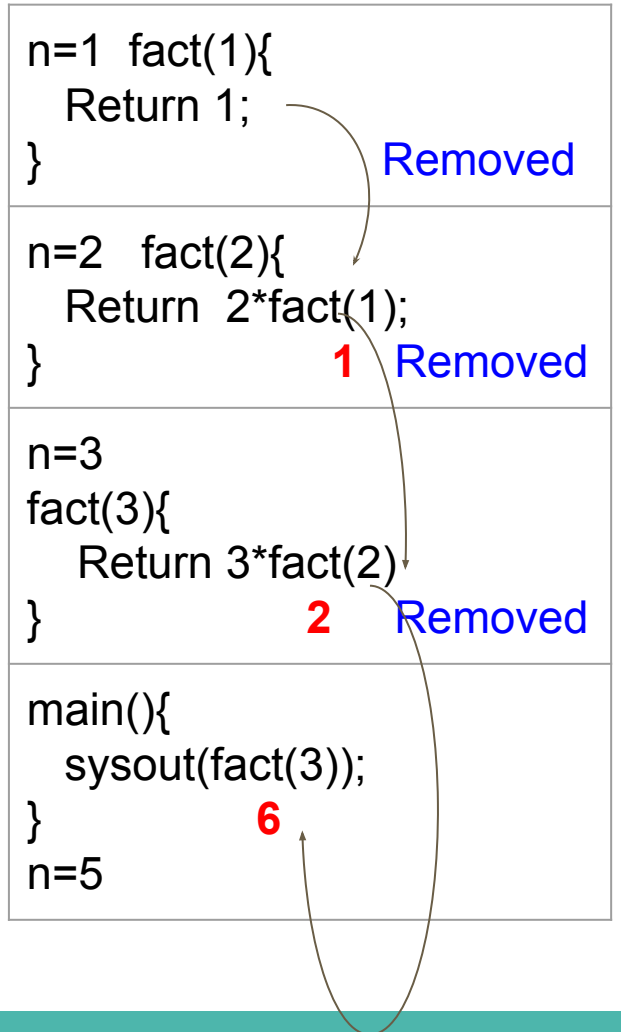
When function call itself within the same function is called direct function

When one function call another function that function calls either one more another or first function so the chain will be created of function call is called indirectly.

In function call has circular structure where either same function or multiple function is direct / indirect recursion

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

```
f1(n){
if(n==1)
            return 1;
      return n*f2(n-1);
}
f2(n){
if(n==1)
            return 1;
      return n*f1(n-1);
}
```



Why recursion

Solving problems that can be broken down into smaller, repetitive problems.

It is especially good for working on things that have many possible branches and are too complex for an iterative approach .

```
n=1  f1(1){
   Return 1;
}                          Removed
```

```
n=2   f2(2){
   Return  2*f1(1);
}                    1     Removed
```

```
n=3
f1(3){
    Return 3*f2(2)
}                    2     Removed
```

```
main(){
   sysout(f1(3));
}                  6
n=5
```

Recursive	Iterative
Definition Recursion refers to a situation where a function calls itself repeatedly until a **base condition** is satisfied, at which point further recursive calls stop.	Iteration refers to a situation where some statements are executed again and again using loops until some condition is satisfied.
Application Recursion is a process because is always called on a function.	Iterative code is applied to variables. It is a set of instructions that are called upon repeatedly.
Program Termination Recursive code terminates when the **base case condition** is satisfied.	

Recursive	Iterative
Code Size Recursive code is smaller in length and neater.	Iterative code is usually extensive and cluttered.
Overhead Time Recursive code has an overhead time for each recursive call that it makes.	Iterative code has no overhead time.
Speed Recursive code is slower than iterative code, since it not only runs the program but must also invoke stack memory.	Iterative code has a relatively faster runtime speed.
Stack Utilization Recursion uses a stack to store the variable changes and parameters for each recursive call.	Iterative code does not use a stack.

Advantages & disadvantages of recursion

Recursion makes program looks shorter and simple

Mostly used in the situation where back tracing like in tree traversal

Simplifies complex problems. – Recursion can simplify complex problems by breaking them down into smaller, more manageable sub-problems. ...

- Saves time and space. ...
- Increases code readability. ...
- Enables efficient data processing. ...
- Facilitates problem solving.

Disadvantages

slower performance than iterative method of looping

- Memory usage. – Recursion can be memory-intensive, as each recursive function call creates a new stack frame. ...
- Stack overflow. ...
- Difficulty in understanding and debugging. ...
- Slower execution time. ...

Algorithm design techniques

Divide and Conquer Algorithm

Greedy algorithm

Dynamic Programming algorithm

Brute force algorithm

Backtracking algorithms

Branch-and-bound algorithms

Divide and Conquer

Is a strategy of solving a large problem by breaking the problem into smaller sub-problems solving the sub-problems, and combining them to get the desired output.

Recursion is used

Steps -

Divide: Divide the given problem into sub-problems using recursion.

Conquer: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.

Combine: Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

Example - Merge Sort , Quick Sort

Greedy Algorithm

Solving a problem by selecting the best option available at the moment.

It doesn't worry whether the current best result will bring the overall optimal result.

The algorithm never reverses the earlier decision even if the choice is wrong.

It works in a top-down approach.

Dynamic programming algorithm

Problem can be divided into subproblems, which in turn are divided into smaller subproblems,

If there are overlapping among these subproblems

Solutions to subproblems can be saved for future reference.

Efficiency of the CPU can be enhanced.

Called dynamic programming.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

Example - print fibonacci series till n term

```
function FibonacciNumber(index n):
    /*Create array of required length*/
    integer array FibArray[n+1]
    FibArray[0]=0
    FibArray[1]=1
    /*Loop over the array and calculate the numbers at all positions*/
    for(i=2 to n+1)
        FibArray[i]=FibArray[i-1]+FibArray[i-2]
    return FibArray[n+1]
```

F(0) = 0 F(1) = 1 F(2) = F(1) + F(0) F(3) = F(2) + F(1)

Brute-Force Search

Simple approach of addressing a problem that relies on huge processing power and testing of all possibilities to improve efficiency.

Suppose you forgot the combination of a 4-digit padlock and to avoid purchasing the new one, you have to open the lock using brute-force search method.

You will have to try all possible 4-digit combinations from 0 to 9 to unlock it.

That combination could be anything between 0000 to 9999, hence there are 10,000 combinations.

So we can say that in the worst case, you have to try 10, 000 times to find your actual combination.

A brute force attack is a trial-and-error method used to decode sensitive data.

Common application cracking passwords and cracking encryption keys

Other common targets for brute force attacks are API keys and SSH logins.

Brute force password attacks are often carried out by scripts or bots(software application) that target a website's login page

a hacking method that uses trial and error to crack passwords, login credentials, and encryption keys.

It is a simple yet reliable tactic for gaining unauthorized access to individual accounts and organizations' systems and networks.

Backtracking algorithms

A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.

The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.

The term backtracking suggests that if the current solution is not suitable, then backtrack and try other solutions.

Thus, recursion is used in this approach.

This approach is used to solve problems that have multiple solutions and we want all those solutions.

If you want an optimal solution, you must go for dynamic programming. Or backtracking is not for optimization problem

Ex- 2 Boys & 1 Girl – Seating arrangement –possible arrangements - 6

want all possible solutions

Backtracking problems has some constraints

Problem: You want to find all the possible ways of arranging 2 boys and 1 girl on 3 benches. Constraint: Girl should not be on the middle bench.

Solution: There are a total of 3! = 6 possibilities. But as per constraint all possibilities will be selected

Backtracking Algorithm Applications

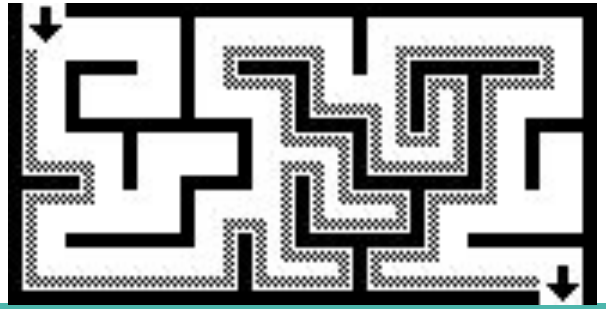
1. To find all Hamiltonian Paths present in a graph.

   Problems of determining whether a Hamiltonian path (a path in an undirected or directed graph that visits each vertex exactly once) or a Hamiltonian cycle exists in a given graph (whether directed or undirected).

2. To solve the N Queen problem.

   n queens problem of placing n non-attacking queens on an n×n chessboard. Solutions exist for all natural numbers n with the exception of n = 2 and n = 3. Although the exact number of solutions is only known for n ≤ 27

3. Maze solving problem



Branch-and-bound algorithms

For combinatory, discrete, and general mathematical optimization problems, branch and bound algorithms are applied to determine the optimal solution.

But only minimization problem.

A branch and bound algorithm searches the set of all possible solutions before recommending the best one.

This algorithm enumerates possible candidate solutions in a stepwise manner by exploring all possible set of solutions.

Job sequencing problem -

j={1,2,3,4}

d={2,1,2,1}

Solutions→

Subset of jobs like {j1,j4} / {1,0,0,1} - 2 ways of representing solution

Branch & bound follows BFS approach to solve problems

And here it create tree of all possible solutions based on constraints