Design and Analysis of Algorithms

Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- **O Notation**
  The notation O(n) is the formal way to express the upper bound of an algorithm's running time.
- **Ω Notation**
  The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time
- **θ Notation**

  The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

Divide and Conquer

A **divide-and-conquer** algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

(1) Binary Search

This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

Time Complexity : O (LogN)

--- ---

(2) Quick Sort

The algorithm picks a pivot element, rearranges the array elements in such a way that all elements smaller than the picked pivot element move to left side of pivot, and all greater elements move to right side. Finally, the algorithm recursively sorts the subarrays on left and right of pivot element. The partitioning of the array according to one element i.e pivot is done using Lomuto or Hoare's Partition method as a sub-routine.

Time Complexity :

Best : O (n Log n)
Average : O (n Log n)
Worst : O (n^2)

--- ---

(3) Merge Sort

It divides the input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one in a sorted manner.

Time Complexity : ϴ(n l og n)

Auxilary Space : O(n)

--- ---

(4) Strassen's Algorithm

it is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices need 3 nested loops and is O(n^3). Strassen's algorithm multiplies two matrices in O(n^2.8974) time.
It divides matrices to sub-matrices of size N/2 x N/2 and calculates the resultant matrix with the help of certain formulas.

Time Complexity : 0(n^2)

-- ---

(5) <u>Median of two sorted arrays</u>

- Create a recursive function that takes two arrays and the sizes of both the arrays.
- Find the middle elements of both the arrays. i.e element at (n – 1)/2 and (m – 1)/2 of first and second array respectively. Compare both the elements.
- If the middle element of the smaller array is less than the middle element of the larger array then the first half of smaller array is bound to lie strictly in the first half of the merged array. It can also be stated that there is an element in the first half of the larger array and second half of the smaller array which is the median. So, reduce the search space to the first half of the larger array and second half of the smaller array.
- Similarly, If the middle element of the smaller array is greater than the middle element of the larger array then reduce the search space to the first half of the smaller array and second half of the larger array.

Time Complexity : O(min(log m, log n))

-- ---

(5) <u>Maximum and Minimum element of an array</u>

- the array is divided into two halves
- Then using recursive approach maximum and minimum numbers in each halves are found
- Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

Time Complexity : O (n)

-- ---

(5) <u>Bitonic point in a Bitonic Sequence</u>

Given :

- *a bitonic sequence (*A Bitonic Sequence is a sequence of numbers which is first strictly increasing then after a point strictly decreasing.)

Find a bitonic point (A Bitonic Point is a point in bitonic sequence before which elements are strictly increasing and after which elements are strictly decreasing)

Note : A Bitonic point doesn’t exist if array is only decreasing or only increasing.

Naive :

- use linear search and find out an element which is smaller than its previous and next.

  Time Complexity : O (n)

Binary Search :

- If **arr[mid-1] < arr[mid]** and **arr[mid] > arr[mid+1]** then we are done with bitonic point.
- If **arr[mid] < arr[mid+1]** then search in right sub-array, else search in left sub-array.

Time Complexity : O (n log n)

----------------------------Divide and Conquer Done !! -------------------------------------

Other Sorting Algos

(1) Insertion Sort

- Here we consider first element as sorted initially.
- Now we start inserting other elements in this assumed sorted array in sorted manner.
- We keep on moving the current element to its left till we find out a smaller element.

Time Complexity : O(n^2)

(2) Selection Sort

- find out the maximum element swap it with the last element.
- now reduce the size of array as the last element is in its correct positition.
- Repeat the process until we reach one element/

Time Complexity : O (n^2)
pros : less swaps.

(3) Heap Sort

- use build heap method to bulid the heap of the array in O (n) time.
- Take out the top most element swap with the last element reduce the size of the heap.
- Repeat until we reach root.

Time Complexity : O (n log n)

(4) Count Sort

Naive :

- find out the maximum element
- create a empty array count of size of the maximum element
- now traverse the array use elements as an index for the empty array and increment values in the count array accordingly. (Simple hashing)
- now traverse the empty array and print non zero indexes the times of values inside them.

Time Complexity : O(n+k) were elements range from n to k.

Cons: when elements are in range from 1 to n^2 it takes O (n^2)
Its also not a stable sorting algo.

Improved approach to make it stable :

- repeat the first two steps.
- now create one more array to store cummulative counts of the values.
- Traverse the count array and build up cummulative frequencies of each element in the new array.
- So that we may know how many elements are smaller than the current element.
- Now start traversing the main array from the back and find out the position of the current element by comparing with the cummulative array for example if a element has 2 elements smaller than it then its postion is 3.
- Now as you print the elements decrease the value of that particular index in the cummulative array.
- Repeat till all the values become zero.

Time Complexity : O (n+k) but now it becomes stable.

Radix Sort

- uses count sort as a subroutine
- loop runs the times of no of digits in the highest element.
- Now keep on dividing this element and other elements by 10 and apply count sort.
- When the highest element becomes zero we are done sorting.

Time Complexity : O (d*(n+b)) d=no. Of digits in the max element and n=no of elements in the array and b=base (10 for decimal)
Auxilary space : O (n + b)

---------------------------*Sorting Done !!* --

Greedy

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy.

(1) <u>Activity Selection Problem</u>

Given :

- *a vector of vectors in which each vector is of size two that denotes start and the end time to perform that particular activity.*

- *A machine that can perform only one activity at a time . So if the time of any two or more activities overlap each other then we can perform only one.*

  Find out the maximum number of activities we may perform using our machine by selecting the time intervals that do not overlap each other.

Naive Approach:

- consider each activity as the first activity of the solution and now recursively find out maximum activities that could be done.
- When done traversing for all the activites return the maximum answer.

  Time Complexity : exponential

Greedy Approach :

- *Sort the vector according to the finish time.*
- *Initialize solution as first activity.*
- *Do following with the remaining activities :*

  - *if current activity overlaps the last chosen activity then ignore the current activity.*
  - *Else add the current activity to the solution.*

  Time Complexity: O (n Log n)

-- --

(2) <u>Fractional Knapsack Problem</u>

Given :

- *set of items with their weights and the amount of profit they carry.*
- *A bag known as Knapsack of capacity M.*

Find the maximum profit you can achieve by filling up the knapsack with certain items where choice of fraction of certain item is allowed.

Naive approach:

try all possible subset with all different fractions

Greedy approach :

- *calculate the ratio value/weight for each item and sort the item on basis of this ratio*
- Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

Time Complexity : O(n log n)

-- --

(3) <u>Job Sequencing Problem</u>

Given :

- *an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline*
- *It is also given that every job takes the single unit of time, so the minimum possible deadline for any job is 1*

maximize total profit if only one job can be scheduled at a time.

Naive approach:

generate all subsets of given set of jobs and check individual subset for feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets

Time Complexity : exponential

Greedy Approach :

- *Sort all jobs in decreasing order of profit.*
- *Iterate on jobs in decreasing order of profit.For each job , do the following :*

  a)Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in this slot and mark this slot filled.

  b)If no such i exists, then ignore the job.

Time Complexity : O (n^2)

-- --

<u>Greedy algos in Graph</u>

Prerequisites :

Spanning Tree : ***i****ts is a sub-graph of a graph and has no cycle.*

if we have a graph containing V vertices and E edges, then the graph can be represented as:

G(V, E)

If we create the spanning tree from the graph, then the spanning tree would have the same number of vertices as the graph

The edges in the spanning tree would be equal to the number of edges in the graph minus 1.

Suppose the spanning tree is represented as:

G\`(V\`, E\`)

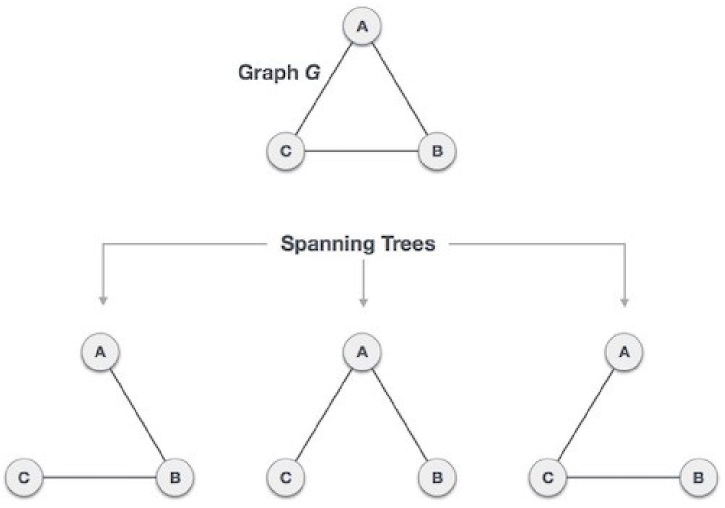
where,

V=V\`

E\`ϵ E -1

E\`=|V| - 1

Demonstration :



Minimum Spanning Trees (MST):

In a weighted graph (a graph were edges have some value or weight associated with them), a minimum spanning tree is a spanning tree that has minimum weight (sum of all weights) than all other spanning trees of the same graph.

--- --

A graph can have lots of spanning trees now in order to find MST checking all spanning trees is not an efficient way so we follow these two algorithms :

(4) <u>Prims's Algorithm</u>

- select the minimum weightage edge initially.
- Now keep on selecting the minimum weightage connected edges until you have | V | -1 edges.

  Time Complexity : O (V^2)

Now we may improve it using binary heap to ***O(E log(V))***.

--- --

(4) <u>Kruskal's Algorithm</u>

- select the minimum weightage edges avoiding the formation of a cycle until you have | V | - 1 edges.

  Time Complexity : O(|E| . |V|)

- Now to improve the time complexity use Min-Heap as you can get minimum weight edge everytime without searching in only log(n) time.

So the improved time complexity is ***O (n log(n))*** *were n is the no. Of vertices.*

--- --

(5) <u>Dijkstra's Algorithm</u>

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) . In easy terms it is used to find the minimum distance between source and every other node.

- Like Prim's MST, we generate a *SPT (shortest path tree)* with given source as root.
- We maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree.
- At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has a minimum distance from the source.

  Time Complexity : O (V^2)

Now we may improve it using binary heap to ***O(E log(V))***.

Fundamental Graph Algorithms :

BFS :
Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

- Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- If no adjacent vertex is found, remove the first vertex from the queue.
- Repeat Rule 1 and Rule 2 until the queue is empty.

Time Complexity : O (V + E)

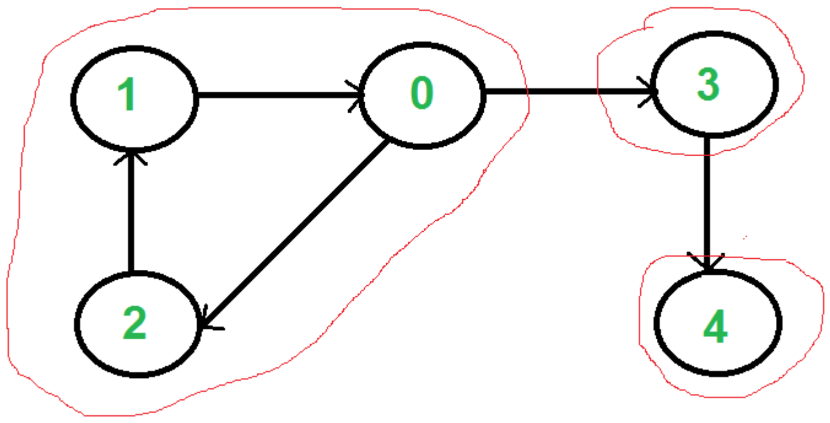
DFS :
Depth First Traversal (or Search) for a graph is similar to Depth First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, a node may be visited twice. To avoid processing a node more than once, use a boolean visited array.

- Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- Repeat Rule 1 and Rule 2 until the stack is empty.

Time Complexity : O (V + E)
Space Complexity : O (V)

Kosaraju's Algorithm

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. For example, there are 3 SCCs in the following graph.



We can find all strongly connected components using Kosaraju's algorithm. Following is detailed Kosaraju's algorithm.

- Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the above graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.

- Reverse directions of all arcs to obtain the transpose graph.

- One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS (call DFSUtil(v)). The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).
  Time Complexity : O (V + E)

Backtracking

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

(1) N-Queens Problem

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other.

Naive Approach:
Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

Backtracking Approach :

- The idea is to place queens one by one in different columns, starting from the leftmost column.
- When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution.
- If we do not find such a row due to clashes then we backtrack and return false.

(2) Graph Coloring Problem

Given an undirected graph and a number m, determine if the graph can be coloured with at most m colours such that no two adjacent vertices of the graph are colored with the same color. Here coloring of a graph means the assignment of colors to all vertices.

Naive Approach: Generate all possible configurations of colours. Since each node can be coloured using any of the m available colours, the total number of colour configurations possible are m^V. After generating a configuration of colour, check if the adjacent vertices have the same colour or not. If the conditions are met, print the combination and break the loop.

Time Complexity**:** $O(m^v)$.
Space Complexity : O (v)

Backtracking Approach:

- The idea is to assign colors one by one to different vertices, starting from the vertex 0.
- Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not.
- If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution.
- If no assignment of color is possible then backtrack and return false.

Time Complexity: $O(m^v)$.
Space Complexity : O (v)

Hamiltonian Problem:

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Naive Approach : Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! (n factorial) configurations.

Backtracking Approach:

- Create an empty path array and add vertex 0 to it.
- Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added.
- If we find such a vertex, we add the vertex as part of the solution.
- If we do not find a vertex then we return false.

Time Complexity : $O(n^n)$

Dynamic Programming

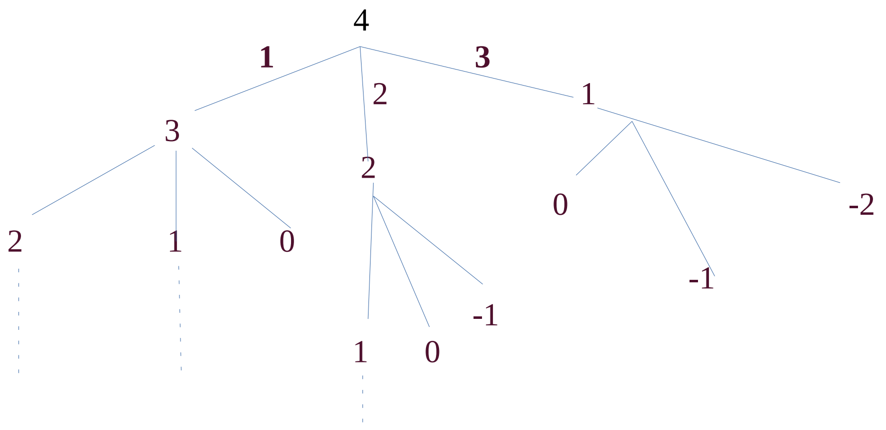
Nothing but an intelligent recursion . We store the results of the recursive call in ana array or vector and before making any call we check weather the same call was made before or not if yes then return the ans from the array or vector.

Coin-Exchange Problem

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.
For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

DP approach :
Like the ribbon cutting problem we decrement all posible combinations form the N in each recursive call until a call produces value less than or equal to zero also we keep storing the results as we keep on incrementing a counter for each valid call so that we may return them without doing further calls.

Like this :



If we follow these approach without memoization it bears time complexity O (m^n)
were m is the number of coins given but using memoization we can actually reduce it to O(m.n)

Single Source Shortest path for a directed graph (Bellman-Ford Algorithm)

Given a graph and a source vertex src in graph, find shortest paths from src to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstra’s algorithm for this problem. Dijkstra’s algorithm is a Greedy algorithm and time complexity is O(VLogV) (with the use of Fibonacci heap). Dijkstra doesn’t work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.

- Start with the weighted graph
- choose a starting vertex and assign infinity path values to all other vertices.
- Visit each edge and relax the path distances if they are inaccurate.
- Do this V times
- Notice how the vertex at the top right had its path length adjusted.
- After all vertices have their path length we check if a negative cycle is present

Time Complexity : O (V.E)

<u>All pair Shortest path problem for a directed graph (Floyd-Warshall Algorithm)</u>

Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph. As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.

Time Complexity : O (V^3)

<u>0-1 Knapsack Problem</u>

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item or don’t pick it (0-1 property).

Naive Approach :
A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.
Optimal Sub-structure: To consider all subsets of items, there can be two cases for every item.

1. **Case 1:** The item is included in the optimal subset.
2. **Case 2:** The item is not included in the optimal set.

Time Complexity: O(2^n).

Dp Approach :

- In the Dynamic programming we will work considering the same cases as mentioned in the recursive approach
- In a DP[][] table let's consider all the possible weights from '1' to 'W' as the columns and weights that can be kept as the rows.

The state DP[i][j] will denote maximum value of 'j-weight' considering all values from '1 to ith'. So if we consider 'wi' (weight in 'ith' row) we can fill it in all columns which have 'weight values > wi'. Now two possibilities can take place:

- Fill 'wi' in the given column.
- Do not fill 'wi' in the given column.

Time Complexity: O(N*W).

Auxiliary Space**:** O(N*W).

String

Knuth-Morris-Pratt Algorithm

Knuth Morris Pratt (KMP) is an algorithm, which checks the characters from left to right. When a pattern has a sub-pattern appears more than one in the sub-pattern, it uses that property to improve the time complexity, also for in the worst case.

For more :

https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

Time Complexity : O (m+n) m for preparing table and n for finding pattern in the string.