**DAA ALL ALGORITHMS**

**UNIT-II**

**Binary Search Algorithm** can be implemented in the following two ways

1. Iterative Method
2. Recursive Method

**1. Iteration Method**

binarySearch(arr, x, low, high)

repeat till low = high

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > arr[mid]) // x is on the right side

low = mid + 1

else // x is on the left side

high = mid - 1

**2. Recursive Method**

binarySearch(arr, x, low, high)

if low > high

return False

else

mid = (low + high) / 2

if x == arr[mid]

return mid

else if x > arr[mid] // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the left side

return binarySearch(arr, x, low, mid - 1)

**ALGORITHM FOR MERGE SORT**

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.  
    if left > right  
        return  
    mid= (left+right)/2  
    mergesort(array, left, mid)  
    mergesort(array, mid+1, right)  
    merge(array, left, mid, right)

step 4: Stop

**ALGORITHM FOR QUICK SORT**

QUICKSORT (array A, start, end)

{

**if** (start < end)

{

 p = partition(A, start, end)

 QUICKSORT (A, start, p - 1)

 QUICKSORT (A, p + 1, end)

 }

}

**Partition Algorithm:**

The partition algorithm rearranges the sub-arrays in a place.

PARTITION (array A, start, end)

{

  pivot ? A[end]

 i ? start-1

**for** j ? start to end -1 {

**do** **if** (A[j] < pivot) {

 then i ? i + 1

 swap A[i] with A[j]

  }

}

 swap A[i+1] with A[end]

**return** i+1

}

**GREEDY METHOD: GENERAL**

getOptimal(Item, array[], int num)

Initialize empty result as, result = {}

While (All items are not considered)

 // In order to select an item we make a greedy here.

 j = SelectAnItem()

 // If j is feasible, add j to the result

 if (feasible(j))

   result = result U j

return result

**GREEDY – ALGORITHM FOR KNAPSACK**

for i = 1 to n

do x[i] = 0

weight = 0

for i = 1 to n

if weight + w[i] ≤ W then

x[i] = 1

weight = weight + w[i]

else

x[i] = (W - weight) / w[i]

weight = W

break

return x

**ALGORITHM FOR OPTIMAL MERGE PATTERN**

Algorithm Tree(n)

//list is a global list of n single node

{

For i=1 to i= n-1 do

{

// get a new tree node

Pt: new treenode;

// merge two trees with smallest length

(Pt = lchild) = least(list);

(Pt = rchild) = least(list);

(Pt =weight) = ((Pt = lchild) = weight) = ((Pt = rchild) = weight);

Insert (list , Pt);

}

// tree left in list

Return least(list);

}

**ALGORITHM: MST-PRIM’S (G, w, r)**

for each u є G.V

u.key = ∞

u.∏ = NIL

r.key = 0

Q = G.V

while Q ≠ Ф

u = Extract-Min (Q)

for each v є G.adj[u]

if each v є Q and w(u, v) < v.key

v.∏ = u

v.key = w(u, v)

## ALGORITHM FOR SINGLE SOURCE SHORTEST PATHS(DIJKSTRA ALGORITHM)

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node, lets say C.
3. For each neighbour N of the current node C: add the current distance of C with the weight of the edge connecting C-N. If it is smaller than the current distance of N, set it as the new current distance of N.
4. Mark the current node C as visited.
5. Go to step 2 if there are any nodes are unvisited.

**PSEUDOCODE**

function Dijkstra(Graph, source):

for each vertex v in Graph:

distance[v] = infinity

distance[source] = 0

G = the set of all nodes of the Graph

while G is non-empty:

Q = node in G with the least dist[ ]

mark Q visited

for each neighbor N of Q:

alt\_dist = distance[Q] + dist\_between(Q, N)

if alt-dist < distance[N]

distance[N] := alt\_dist

return distance[ ]

**UNIT-III**

**DYNAMIC PROGRAMMING –KNAPSACK**

for w = 0 to W do

c[0, w] = 0

for i = 1 to n do

c[i, 0] = 0

for w = 1 to W do

if wi ≤ w then

if vi + c[i-1, w-wi] then

c[i, w] = vi + c[i-1, w-wi]

else c[i, w] = c[i-1, w]

else

c[i, w] = c[i-1, w]

**ALGORITHM: LCS-FORMULATION (X, Y)**

m := length(X)

n := length(Y)

for i = 1 to m do

C[i, 0] := 0

for j = 1 to n do

C[0, j] := 0

for i = 1 to m do

for j = 1 to n do

if xi = yj

C[i, j] := C[i - 1, j - 1] + 1

B[i, j] := ‘D’

else

if C[i -1, j] ≥ C[i, j -1]

C[i, j] := C[i - 1, j] + 1

B[i, j] := ‘U’

else

C[i, j] := C[i, j - 1]

B[i, j] := ‘L’

return C and B

**ALGORITHM: PRINT-LCS (B, X, I, J)**

if i = 0 and j = 0

return

if B[i, j] = ‘D’

Print-LCS(B, X, i-1, j-1)

Print(xi)

else if B[i, j] = ‘U’

Print-LCS(B, X, i-1, j)

else

Print-LCS(B, X, i, j-1)

**ALGORITHM: TRAVELING-SALESMAN-PROBLEM**

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

**ALGORITHM: OPTIMAL-BINARY-SEARCH-TREE(P, Q, N)**

e[1…n + 1, 0…n],

w[1…n + 1, 0…n],

root[1…n + 1, 0…n]

for i = 1 to n + 1 do

e[i, i - 1] := qi - 1

w[i, i - 1] := qi - 1

for l = 1 to n do

for i = 1 to n – l + 1 do

j = i + l – 1 e[i, j] := ∞

w[i, i] := w[i, i -1] + pj + qj

for r = i to j do

t := e[i, r - 1] + e[r + 1, j] + w[i, j]

if t < e[i, j]

e[i, j] := t

root[i, j] := r

return e and root

# ALGORITHM: ALL-PAIRS SHORTEST PATHS

Begin

   for k := 0 to n, do

      for i := 0 to n, do

         for j := 0 to n, do

            if cost[i,k] + cost[k,j] < cost[i,j], then

               cost[i,j] := cost[i,k] + cost[k,j]

            done

         done

      done

      display the current cost matrix

End

**UNIT-IV**

**ALGORITHM: Backtracking**

The Algorithm begins to build up a solution, starting with an empty solution set . **S = {}**

1. Add to Backtracking | Set 1Backtracking | Set 1the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree s in the search tree of the algorithm.
2. Check if S+S satisfies each of the constraints in C.
   1. If Yes, then the sub-tree S is “eligible” to add more “children”.
   2. Else, the entire sub-tree S is useless, so recurs back to step 1 using argument S.
3. In the event of “eligibility” of the newly formed sub-tree s, recurs back to step 1, using argument S+s.
4. If the check for S+s returns that it is a solution for the entire data D. Output and terminate the program.   
   If not, then return that no solution is possible with the current **S** and hence discard it.

**PSEUDO CODE FOR BACKTRACKING :**

1. **Recursive backtracking solution.**

void findSolutions(n, other params) :

if (found a solution) :

solutionsFound = solutionsFound + 1;

displaySolution();

if (solutionsFound >= solutionTarget) :

System.exit(0);

return

for (val = first to last) :

if (isValid(val, n)) :

applyValue(val, n);

findSolutions(n+1, other params);

removeValue(val, n);

2. **Finding whether a solution exists or not**

boolean findSolutions(n, other params) :

if (found a solution) :

displaySolution();

return true;

for (val = first to last) :

if (isValid(val, n)) :

applyValue(val, n);

if (findSolutions(n+1, other params))

return true;

removeValue(val, n);

return false;

**ALGORITHM: N-QUEEN PROBLEM**

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing the queen in [row, column] leads to

a solution then return true.

c) If placing queen doesn't lead to a solution then

unmark this [row, column] (Backtrack) and go to

step (a) to try other rows.

4) If all rows have been tried and nothing worked,

return false to trigger backtracking.

## ALGORITHM : SUM OF SUB SET

**Input −** The given set and subset, size of set and subset, a total of the subset, number of elements in the subset and the given sum.

**Output −** All possible subsets whose sum is the same as the given sum.

Begin

   if total = sum, then

      display the subset

      //go for finding next subset

      subsetSum(set, subset, , subSize-1, total-set[node], node+1, sum)

      return

   else

      for all element i in the set, do

         subset[subSize] := set[i]

         subSetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)

      done

End

## NAIVE ALGORITHM- GRAPH COLORING

while there are untried conflagrations

{

generate the next configuration

if ( there are edges between two consecutive vertices of this

configuration and there is an edge from the last vertex to the first ).

{

print this configuration;

break;

}

}

## ALGORITHM : KNAPSACK PROBLEM USING [BACKTRACKING](https://codecrucks.com/backtracking-what-why-and-how/)

Knapsack Problem using [Backtracking](https://codecrucks.com/backtracking-what-why-and-how/) can be solved as follow:

* The knapsack problem is useful in solving resource allocation problems.
* Let X = <x1, x2, x3, . . . . . , xn> be the set of n items,  
  W = <w1, w2, w3, . . . , wn> and V = <v1, v2, v3, . . . , vn> be the set of weight and value associated with each item in X, respectively.
* Let M be the total capacity of the knapsack, i.e. knapsack cannot hold items having a collective weight greater than M.
* Select items from X such that it maximizes the profit and the collective weight of selected items does not exceed the knapsack capacity.
* The knapsack problem has two variants. 0/1 knapsack does not allow breaking up the item, whereas a fractional knapsack does. 0/1 knapsack is also known as a binary knapsack.

**ALGORITHM:**

BK\_KNAPSACK(M, W, V, fw, fp, X)

// Description : Solve knapsack problem using backtracking

// Input :

M: Knapsack capacity

W(1...n): Set of weight of the items

V(1...n): Set of profits associated with items

Fw: Final knapsack weight

Fp: Final earned profit

X(1...n): Solution vector

N: Total number of items

// Output : Solution tuple X, earned profit fp

// Initialization

cw ← 0 // Current weight

cp ← 0 // Current profit

fp ← – 1

k ← 1 // Index of item being processed

## Optimization Problem

Optimization problems are those for which the objective is to maximize or minimize some values. For example,

* Finding the minimum number of colors needed to color a given graph.
* Finding the shortest path between two vertices in a graph.

## Decision Problem

There are many problems for which the answer is a Yes or a No. These types of problems are known as **decision problems**. For example,

* Whether a given graph can be colored by only 4-colors.
* Finding Hamiltonian cycle in a graph is not a decision problem, whereas checking a graph is Hamiltonian or not is a decision problem.

## What is Language?

Every decision problem can have only two answers, yes or no. Hence, a decision problem may belong to a language if it provides an answer ‘yes’ for a specific input. A language is the totality of inputs for which the answer is Yes. Most of the algorithms discussed in the previous chapters are **polynomial time algorithms**.

For input size ***n***, if worst-case time complexity of an algorithm is ***O(nk)***, where ***k*** is a constant, the algorithm is a polynomial time algorithm.

Algorithms such as Matrix Chain Multiplication, Single Source Shortest Path, All Pair Shortest Path, Minimum Spanning Tree, etc. run in polynomial time. However there are many problems, such as traveling salesperson, optimal graph coloring, Hamiltonian cycles, finding the longest path in a graph, and satisfying a Boolean formula, for which no polynomial time algorithms is known. These problems belong to an interesting class of problems, called the **NP-Complete** problems, whose status is unknown.

In this context, we can categorize the problems as follows −

## P-Class

The class P consists of those problems that are solvable in polynomial time, i.e. these problems can be solved in time ***O(nk)*** in worst-case, where **k** is constant.

These problems are called **tractable**, while others are called **intractable or superpolynomial**.

Formally, an algorithm is polynomial time algorithm, if there exists a polynomial ***p(n)*** such that the algorithm can solve any instance of size **n** in a time ***O(p(n))***.

Problem requiring ***Ω(n50)*** time to solve are essentially intractable for large ***n***. Most known polynomial time algorithm run in time ***O(nk)*** for fairly low value of ***k***.

The advantages in considering the class of polynomial-time algorithms is that all reasonable **deterministic single processor model of computation** can be simulated on each other with at most a polynomial slow-d

## NP-Class

The class NP consists of those problems that are verifiable in polynomial time. NP is the class of decision problems for which it is easy to check the correctness of a claimed answer, with the aid of a little extra information. Hence, we aren’t asking for a way to find a solution, but only to verify that an alleged solution really is correct.

Every problem in this class can be solved in exponential time using exhaustive search.

## P versus NP

Every decision problem that is solvable by a deterministic polynomial time algorithm is also solvable by a polynomial time non-deterministic algorithm.

All problems in P can be solved with polynomial time algorithms, whereas all problems in *NP - P* are intractable.

It is not known whether ***P = NP***. However, many problems are known in NP with the property that if they belong to P, then it can be proved that P = NP.

If ***P ≠ NP***, there are problems in NP that are neither in P nor in NP-Complete.

The problem belongs to class **P** if it’s easy to find a solution for the problem. The problem belongs to **NP**, if it’s easy to check a solution that may have been very tedious to find.

## Cook’s Theorem

Stephen Cook presented four theorems in his paper “The Complexity of Theorem Proving Procedures”. These theorems are stated below. We do understand that many unknown terms are being used in this chapter, but we don’t have any scope to discuss everything in detail.

Following are the four theorems by Stephen Cook −

## Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

## Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

## Theorem-3

* For any ***TQ(k)*** of type **Q**, TQ(k)k√(logk)2TQ(k)k(logk)2 is unbounded
* There is a ***TQ(k)*** of type **Q** such that TQ(k)⩽2k(logk)2

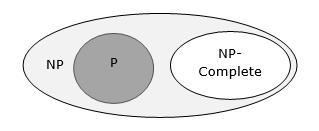
## Theorem-4

If the set S of strings is accepted by a non-deterministic machine within time ***T(n) = 2n***, and if ***TQ(k)*** is an honest (i.e. real-time countable) function of type **Q**, then there is a constant **K**, so **S** can be recognized by a deterministic machine within time ***TQ(K8n)***.

* First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.
* Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.
* Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.
* Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

## NP Hard and NP-Complete Classes

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.



If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is ***NP-complete*** if it satisfies two conditions

* **B** is in NP
* Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

* Determining whether a graph has a Hamiltonian cycle
* Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

* The circuit-satisfiability problem
* Set Cover
* Vertex Cover
* Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove ***TSP is NP-Complete***, first we have to prove that ***TSP belongs to NP***. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus ***TSP belongs to NP***.

Secondly, we have to prove that ***TSP is NP-hard***. To prove this, one way is to show that ***Hamiltonian cycle ≤p TSP*** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume ***G = (V, E)*** to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph ***G' = (V, E')***, where

E′={(i,j):i,j∈Vandi≠jE′={(i,j):i,j∈Vandi≠j

Thus, the cost function is defined as follows −

t(i,j)={01if(i,j)∈Eotherwiset(i,j)={0if(i,j)∈E1otherwise

Now, suppose that a Hamiltonian cycle ***h*** exists in ***G***. It is clear that the cost of each edge in ***h*** is **0** in ***G'*** as each edge belongs to ***E***. Therefore, ***h*** has a cost of **0** in ***G'***. Thus, if graph ***G*** has a Hamiltonian cycle, then graph ***G'*** has a tour of **0** cost.

Conversely, we assume that ***G'*** has a tour ***h'*** of cost at most **0**. The cost of edges in ***E'*** are **0** and **1** by definition. Hence, each edge must have a cost of **0** as the cost of ***h'*** is **0**. We therefore conclude that ***h'*** contains only edges in ***E***.

We have thus proven that ***G*** has a Hamiltonian cycle, if and only if ***G'*** has a tour of cost at most **0**. TSP is NP-complete.