Write binary search algorithm (only algorithm and not code) and derive it's time complexity.

Algorithm:

Here's an implementation of binary search in C:

```c
int binary_search(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}
```

The function takes an array arr of integers, the indices low and high representing the subarray to search, and a target integer to search for. It returns the index of the target if it is found in the subarray, or -1 if it is not found.

The recurrence relation for the time complexity of binary search is:

T(n) = T(n/2) + O(1)

This is because each time the function is called recursively, the size of the subarray to search is halved. The constant factor O(1) represents the time taken to perform the comparisons and updates in each recursive call.

Using the master theorem, we can solve this recurrence relation to find the time complexity of binary search. The master theorem states that if a recurrence relation is of the form T(n) = aT(n/b) + f(n), where a >= 1, b > 1, and f(n) is an asymptotically positive function, then:

Write a recursive backtracking function for the Sum of subset problem in C language.

```c
#include <stdio.h>
#include <stdbool.h>

void print_subset(int subset[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", subset[i]);
    }
    printf("\n");
}

bool sum_of_subset(int nums[], int n, int target, int subset[], int size, int
index) {
    if (target == 0) {
        print_subset(subset, size);
        return true;
    }
    if (index == n || target < 0) {
        return false;
    }
    subset[size] = nums[index];
    bool included = sum_of_subset(nums, n, target - nums[index], subset, size
+ 1, index + 1);
    if (included) {
        return true;
    }
    return sum_of_subset(nums, n, target, subset, size, index + 1);
}
```

State the Bounding conditions to prune the nodes in the sum of subsets problem.

$$\sum_{i=1}^{k} w_i * x_i + w_{k+1} <= m$$

$$\sum_{i=1}^{k} w_i * x_i + \sum_{i=k+1}^{n} w_i > m$$

What is the Vertex Cover Problem? **Prove that Vertex cover is 'an NP complete problem.**

Problem – Given a graph G (V, E) and a positive integer k, the problem is to find whether there is a subset V' of vertices of size at most k, such that every edge in the graph is connected to some vertex in V'.

Explanation –
First let us understand the notion of an instance of a problem. An instance of a problem is nothing but an input to the given problem. An instance of the Vertex Cover problem is a graph G (V, E) and a positive integer k, and the problem is to check whether a vertex cover of size at most k exists in G.

Since an NP Complete problem, by definition, is a problem which is both in NP and NP hard, the proof for the statement that a problem is NP Complete consists of two parts:

Proof that vertex cover is in NP –
If any problem is in NP, then, given a 'certificate' (a solution) to the problem and an instance of the problem (a graph G and a positive integer k, in this case), we will be able to verify (check whether the solution given is correct or not) the certificate in polynomial time.

The certificate for the vertex cover problem is a subset V' of V, which contains the vertices in the vertex cover. We can check whether the set V' is a vertex cover of size k using the following strategy (for a graph G(V, E)):

```
let count be an integer
set count to 0
for each vertex v in V'
    remove all edges adjacent to v from set E
    increment count by 1
    if count = k and E is empty
    then
        the given solution is correct
    else
        the given solution is wrong
```

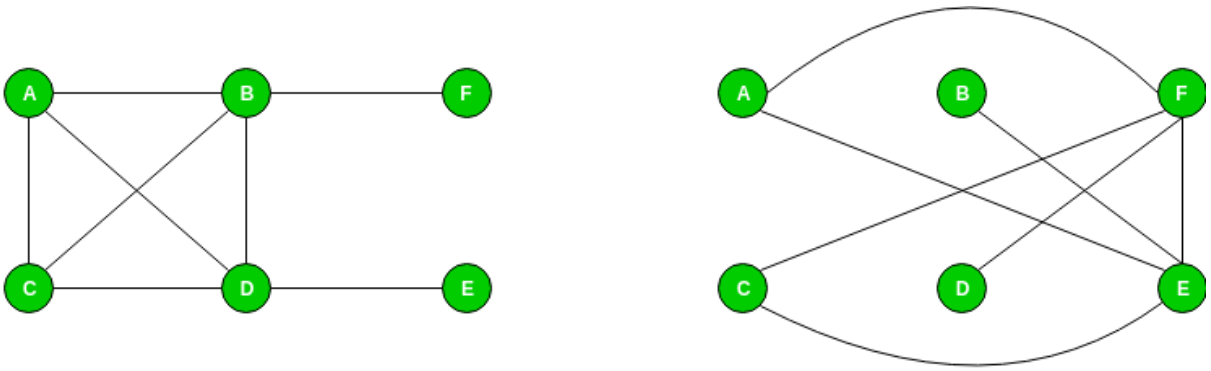
1. It is plain to see that this can be done in polynomial time. Thus the vertex cover problem is in the class NP.

Proof that vertex cover is NP Hard –

To prove that Vertex Cover is NP Hard, we take some problem which has already been proven to be NP Hard and show that this problem can be reduced to the Vertex Cover problem. For this, we consider the Clique problem, which is NP Complete (and hence NP Hard).

"In computer science, the clique problem is the computational problem of finding cliques (subsets of vertices, all adjacent to each other, also called complete subgraphs) in a graph."

Here, we consider the problem of finding out whether there is a clique of size k in the given graph. Therefore, an instance of the clique problem is a graph G (V, E) and a non-negative integer k, and we need to check for the existence of a clique of size k in G.



G
V' ={A, B, C, D}

G'
V'' ={E, F}

Now, we need to show that any instance (G, k) of the Clique problem can be reduced to an instance of the vertex cover problem. Consider the graph G' which consists of all edges not in G, but in the complete graph using all vertices in G. Let us call this the complement of G. Now, the problem of finding whether a clique of size k exists in the graph G is the same as the problem of finding whether there is a vertex cover of size |V| – k in G'. We need to show that this is indeed the case.

Assume that there is a clique of size k in G. Let the set of vertices in the clique be V'. This means |V'| = k. In the complement graph G', let us pick any edge (u, v). Then at least one of u or v must be in the set V – V'. This is because, if both u and v were from the set V', then the edge (u, v) would

belong to V', which, in turn would mean that the edge (u, v) is in G. This is not possible since (u, v) is not in G. Thus, all edges in G' are covered by vertices in the set V – V'.

Now assume that there is a vertex cover V'' of size |V| – k in G'. This means that all edges in G' are connected to some vertex in V''. As a result, if we pick any edge (u, v) from G', both cannot be outside the set V''. This means, all edges (u, v) such that both u and v are outside the set V'' are in G, i.e., these edges constitute a clique of size k.

Thus, we can say that there is a clique of size k in graph G if and only if there is a vertex cover of size |V| – k in G', and hence, any instance of the clique problem can be reduced to an instance of the vertex cover problem. Thus, vertex cover is NP Hard. Since vertex cover is in both NP and NP Hard classes, it is NP Complete.

Give an approximation algorithm in C for Vertex cover problem and justify its approximation ratio with example.

```
greedyVertexCover(numVertices, edges, numEdges):
    visited[numVertices] = { false }  // Array to keep track of visited
vertices

    for i = 0 to numEdges-1 do:
        edge = edges[i]

        if visited[edge.u] = false and visited[edge.v] = false then:
            visited[edge.u] = true
            visited[edge.v] = true

    print "Approximate Vertex Cover: "
    for i = 0 to numVertices-1 do:
        if visited[i] = true then:
            print i
```

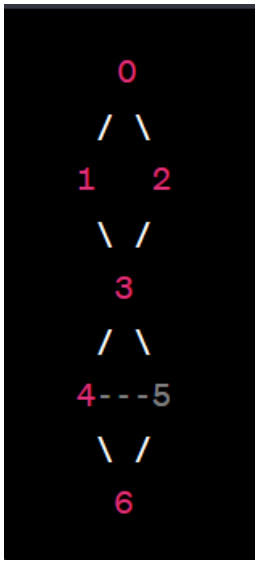
One simple approximation algorithm for the Vertex Cover problem is the following:

1. Initialize an empty set **S** to be the vertex cover

2. For each edge **(u,v)** in the graph: a. If neither **u** nor **v** is in **S**, add both **u** and **v** to **S**
3. Return **S** as the vertex cover

This algorithm is known as the "2-approximation" algorithm for Vertex Cover, as it produces a vertex cover whose size is at most twice the size of the optimal vertex cover. To see why this is the case, consider any edge **(u,v)** in the graph. If **u** and **v** are not in the current vertex cover **S**, then we add both **u** and **v** to **S**. This means that at least one of **u** and **v** must be in the optimal vertex cover, since otherwise the edge **(u,v)** would not be covered. Therefore, we can conclude that the size of **S** is at most twice the size of the optimal vertex cover.

To justify the 2-approximation ratio of this algorithm, consider the following example:

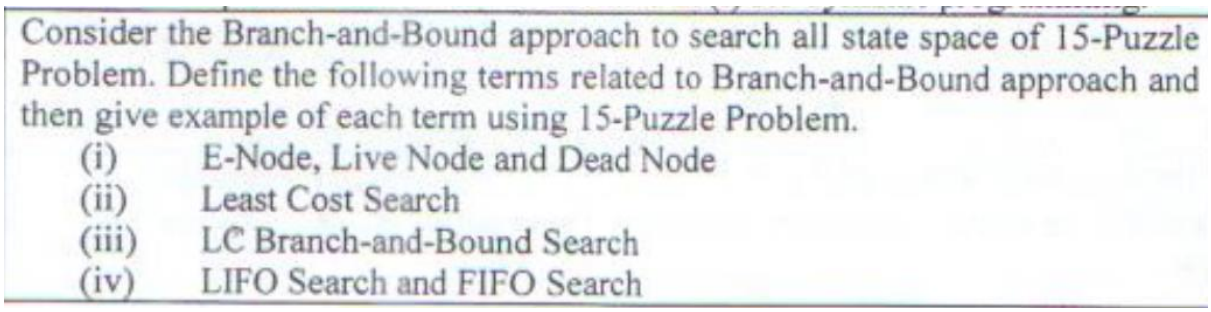


The Greedy Algorithm will work as follows:

1. Initially, no vertices are visited.
2. The algorithm starts with the first edge (0-1). Both vertices are unvisited, so it adds vertices 0 and 1 to the visited set.
3. The algorithm moves to the second edge (1-2). Both vertices are unvisited, so it adds vertices 1 and 2 to the visited set.
4. The algorithm continues in a similar fashion for the remaining edges.

5. The resulting approximate vertex cover is {0, 1, 2, 3, 4, 5}.

In this example, the size of the approximate vertex cover obtained using the Greedy Algorithm is 6. The optimal solution for this graph is {0, 1, 3, 5}, which has a size of 4. Therefore, the approximation ratio of the Greedy Algorithm for this example is 6/4 = 1.5.


Consider the Branch-and-Bound approach to search all state space of 15-Puzzle Problem. Define the following terms related to Branch-and-Bound approach and then give example of each term using 15-Puzzle Problem.

(i) E-Node, Live Node and Dead Node
(ii) Least Cost Search
(iii) LC Branch-and-Bound Search
(iv) LIFO Search and FIFO Search

There are basically three types of nodes involved in Branch and Bound
1. Live node is a node that has been generated but whose children have not yet been generated.
2. E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.
3. Dead node is a generated node that is not to be expanded any further. All children of a dead node have already been expanded.

Cost function:
Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

C(X) = g(X) + h(X) where

g(X) = cost of reaching the current node

from the root

h(X) = cost of reaching an answer node from X.

1. LIFO (Last-In-First-Out) search: LIFO search is a type of search algorithm where the most recently generated node is explored first. This search strategy is commonly used in depth-first search algorithms, where the algorithm explores the deepest unexplored node first before backtracking to explore other nodes.

2. FIFO (First-In-First-Out) search: FIFO search is a type of search algorithm where the oldest generated node is explored first. This search strategy is commonly used in breadth-first search algorithms, where the algorithm explores all the nodes at the current level before moving on to explore the next level.

3. LC (Least-Cost) branch and bound search: LC search is a type of branch and bound search algorithm that selects the node with the lowest cost to expand next. In this search strategy, nodes are sorted by their estimated cost, and the algorithm expands the node with the lowest estimated cost first. This approach is commonly used in optimization problems, where the goal is to find the solution with the lowest cost.

4. Least Cost Search:

   Least Cost Search, also known as Uniform Cost Search (UCS), is an algorithm used to find the path with the lowest cost from a start node to a goal node in a weighted graph. It expands nodes based on cumulative costs, maintains a priority queue, and selects the node with the lowest cost at each step. It updates the costs of neighbouring nodes if a lower-cost path is found. The algorithm explores paths in order of increasing cost and is suitable for finding optimal solutions based on real-world factors. However, it can be computationally expensive and assumes non-negative edge costs.

Analyse the string-matching algorithm with finite automata, compare with brute force string matching.

FINITE AUTOMATA

- Idea of this approach is to build finite automata to scan text T for finding all occurrences of pattern P.
- This approach examines each character of text exactly once to find the pattern. Thus, it takes linear time for matching but pre-processing time may be large.
- **Time Complexity = $O(M^3|\Sigma|)$**

A finite automaton M is a 5-tuple **(Q, q0,A,∑,δ)**, where

Q is a finite set of states,
q_0 ∈ Q is the start state,
A ⊆ Q is a notable set of accepting states,
∑ is a finite input alphabet,
δ is a function from Q x ∑ into Q called the transition function of M.
The finite automaton starts in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves from state q to state δ (q, a). Whenever its current state q is a member of A, the machine M has accepted the string read so far. An input that is not allowed is rejected.

A finite automaton M induces a function ∅ called the called the final-state function, from ∑* to Q such that ∅(w) is the state M ends up in after scanning the string w. Thus, M accepts a string w if and only if ∅(w) ∈ A

Why it is efficient?

These string-matching automatons are very efficient because they examine each text character exactly once, taking constant time per text character. The matching time used is O(n) where n is the length of Text string.

But the pre-processing time i.e., the time taken to build the finite automaton can be large if ∑ is large.

Number of states in Finite Automaton will be **M+1** where M is length of the pattern. The main thing to construct Finite Automaton is to get the next state from the current state for every possible character. Given a character x and a state k, we can get the next state by considering the string "pat[0..k-1]x" which is basically concatenation of pattern characters pat[0], pat[1] …pat[k-1] and the character x. The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of "pat[0..k-1]x". The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character 'C' in the above diagram. We need to consider the string, "pat[0..4]C" which is "ACACAC". The length of the longest prefix of the pattern such that the prefix is suffix of "ACACAC"is 4 ("ACAC"). So the next state (from state 5) is 4 for character 'C'.
In the following code, computeTF() constructs the Finite Automaton. The

time complexity of the computeTF() is O(m^3*NO_OF_CHARS) where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text). The implementation tries all possible prefixes starting from the longest possible that can be a suffix of "pat[0..k-1]x". There are better implementations to construct Finite Automaton in O(m*NO_OF_CHARS)

Time Complexity: O(m²)

Auxiliary Space: O(m)

The comparison between finite automata (FA) and naive string-matching algorithms lies in their approaches, time complexity, and efficiency in different scenarios.

1. Approach:

- Finite Automata: FA algorithm utilizes a pre-processed finite automaton that captures the pattern's internal structure. It constructs a state transition table or diagram based on the pattern. The algorithm then iterates through the text, transitioning between states in the automaton, and efficiently skips comparisons based on the automaton's transitions.
- Naive String Matching: The naive algorithm compares the pattern against every possible position in the text. It involves sliding the pattern across the text and checking character by character for a match. If a mismatch occurs, the pattern is shifted by one position, and the process is repeated until a match is found or the end of the text is reached.

2. Time Complexity:

- Finite Automata: The time complexity of the FA algorithm is O (m + n), where m is the length of the pattern and n is the length of the text. This complexity arises from constructing the finite automaton and performing the string-matching process efficiently using the automaton's transitions.
- Naive String Matching: The time complexity of the naive algorithm is O(mn), where m is the length of the pattern and n is the length of the

text. It compares the pattern against every possible position in the text, resulting in a nested loop structure.

3. Efficiency:

- Finite Automata: The FA algorithm is particularly efficient when the pattern length is large or when the same pattern needs to be matched against multiple texts. Once the finite automaton is constructed, the matching process is performed efficiently using the automaton's transitions, resulting in improved performance compared to the naive approach.
- Naive String Matching: The naive algorithm is simple to implement but can be inefficient for large patterns or texts. It involves a straightforward character-by-character comparison, which can result in redundant comparisons and slower performance for longer patterns or texts.

4. Pattern Pre-processing:

- Finite Automata: The FA algorithm requires an additional pre-processing step to construct the finite automaton. This step involves calculating the failure function or prefix function, which can take some additional time and memory.
- Naive String Matching: The naive algorithm does not require any pre-processing of the pattern. It can be directly applied by comparing the pattern against the text.

In summary, the finite automata algorithm offers a more efficient approach to string matching with a better time complexity of O(m + n) compared to the naive string matching algorithm's O(mn). However, constructing the finite automaton requires additional preprocessing. The choice between the two algorithms depends on the size of the pattern, the text, and the need for efficiency in different scenarios. The FA algorithm is particularly advantageous when pattern matching is a repetitive task or when the pattern length is large. Conversely, the naive algorithm is simple and straightforward to implement but may not be as efficient for large patterns or texts.

Write the algorithm for quick sort and analyse its time complexity.

Algorithm:

```
int partition(int arr[], int low, int high)
{
    // Choosing the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than the pivot
        if (arr[j] < pivot) {

            // Increment index of smaller element
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i + 1], arr[high]);
    return (i + 1);
}

// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index
void quickSort(int arr[], int low, int high)
{
    if (low < high) {

        // pi is partitioning index, arr[p]
        // is now at right place
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Analysis of QuickSort:

Time taken by Quicksort, in general, can be written as follows.

$T(n) = T(k) + T(n-k-1) + \Theta(n)$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.

Worst Case:

The worst case occurs when the partition process always picks the first or last element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is the recurrence for the worst case.

$T(N) = T(0) + T(N-1) + \Theta(n)$ *which is equivalent to*

$T(N) = T(N-1) + \Theta(n)$

The solution to the above recurrence is $O(n^2)$.

Best Case:

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

$T(N) = 2T(N/2) + \Theta(n)$

The solution for the above recurrence is $O(N * logN)$. It can be solved using case 2 of the Master Theorem.

Average Case:

To do an average case analysis, we need to consider all possible permutations of the array and calculate the time taken by every permutation which doesn't look easy.

We can get an idea of an average case by considering the case when partition puts O(N/9) elements in one set and O(9N/10) elements in the other set. Following is the recurrence for this case.

$T(N) = T(N/9) + T(9N/10) + \Theta(n)$

The solution of the above recurrence is also O(N * logN):

Although the worst case time complexity of QuickSort is **$O(N^2)$** which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of $O(N^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

i) You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts a1 < a2 < · · · < an, where each ai is measured from the starting point.

The only places you are allowed to stop are at these hotels, but you can choose which of these hotels to stop at. You must stop at the final hotel (at distance an), which is your destination.

You would ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is (200 – x)^2 .
You want to plan your trip so as to minimize the total penalty – that is, the sum, overall travel days, of the daily penalties.

Give an efficient Dynamic programming approach that determines the optimal sequence of hotels at which to stop.

This problem can be solved using a dynamic programming approach. Let's define an array `penalty[]` where `penalty[i]` represents the minimum total penalty for reaching hotel `i`. We can start by initializing `penalty[0] = 0` since there is no penalty for starting at mile post 0. Then, for each hotel `i`, we can calculate the minimum total penalty for reaching that hotel by considering all previous hotels `j` and choosing the one that results in the minimum penalty for reaching hotel `i`. The penalty for traveling from hotel `j` to hotel `i` is `(200 - (a[i] - a[j]))^2`. Therefore, we can calculate `penalty[i]` as follows:

```
penalty[i] = min(penalty[i], penalty[j] + (200 - (a[i] - a[j]))^2)
```

We can repeat this process for all hotels until we reach the final hotel `n`. The minimum total penalty for reaching the final hotel is given by `penalty[n]`.

Here is an example of pseudocode that implements this dynamic programming approach:

```
function minPenalty(a[], n):
    let penalty[] be an array of size n+1
    penalty[0] = 0
    for i = 1 to n:
        penalty[i] = infinity
        for j = 0 to i-1:
```

```
            penalty[i] = min(penalty[i], penalty[j] + (200 - (a[i] -
a[j]))^2)
    return penalty[n]
```

This algorithm runs in O(n^2) time where n is the number of hotels.

Explain Backtracking approach in general. Write an algorithm for the N queen problem. Apply it to solve 4 queen problems.

Definition:

Backtracking *can be defined as a general algorithmic technique that considers searching every possible combination to solve a computational problem.*

Backtracking is a general algorithmic technique that involves exploring all possible solutions to a problem by incrementally building a solution and then backing out or undoing part of the solution if it is determined to be incorrect or unfeasible. Backtracking is often used to solve problems where the solution can be represented as a sequence of decisions or choices.

The backtracking approach works by incrementally building a solution, one piece at a time. At each step, the algorithm decides and then recursively explores all possible solutions that can be built from that decision. If at any point it is determined that the current decision leads to an incorrect or unfeasible solution, the algorithm backs out or undoes the decision and tries a different one.

Backtracking can be visualized as a depth-first search of a tree where each node represents a partial solution and each edge represents a decision .The algorithm starts at the root of the tree and explores all possible paths until it finds a complete solution or determines that no solution exists.

Backtracking is often used to solve problems in combinatorial optimization, constraint satisfaction, and game playing. Some common examples of problems that can be solved using backtracking include the N-Queens problem, Sudoku, and the Knight's Tour problem.

The efficiency of backtracking algorithms depends on the problem being solved and the pruning techniques used to avoid exploring unfeasible or

suboptimal solutions. In some cases, backtracking can be very efficient while in others it may not be practical due to its exponential time complexity.

There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

N Queen problem:

```
// A utility function to print solution
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if(board[i][j])
                printf("Q ");
            else
                printf(". ");
        }
        printf("\n");
    }
}

// A utility function to check if a queen can
// be placed on board[row][col]. Note that this
// function is called when "col" queens are
// already placed in columns from 0 to col -1.
// So we need to check only left side for
// attacking queens
bool isSafe(int board[N][N], int row, int col)
{
    int i, j;

    // Check this row on left side
    for (i = 0; i < col; i++)
        if (board[row][i])
            return false;

    // Check upper diagonal on left side
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
        if (board[i][j])
            return false;
```

```
    // Check lower diagonal on left side
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
        if (board[i][j])
            return false;

    return true;
}

// A recursive utility function to solve N
// Queen problem
bool solveNQUtil(int board[N][N], int col)
{
    // Base case: If all queens are placed
    // then return true
    if (col >= N)
        return true;

    // Consider this column and try placing
    // this queen in all rows one by one
    for (int i = 0; i < N; i++) {

        // Check if the queen can be placed on
        // board[i][col]
        if (isSafe(board, i, col)) {

            // Place this queen in board[i][col]
            board[i][col] = 1;

            // Recur to place rest of the queens
            if (solveNQUtil(board, col + 1))
                return true;

            // If placing queen in board[i][col]
            // doesn't lead to a solution, then
            // remove queen from board[i][col]
            board[i][col] = 0; // BACKTRACK
        }
    }

    // If the queen cannot be placed in any row in
    // this column col  then return false
    return false;
}
```

This algorithm uses a backtracking approach to incrementally place queens on the board one row at a time. At each step, the algorithm tries to place a queen in a safe position in the current row and then recursively places queens in the next row. If it is

not possible to place a queen in a safe position in the current row, the algorithm backtracks and removes the queen from the previous row and tries a different position.
The `isSafe` function checks if it is safe to place a queen at a given position by checking if there are any queens in the same column or diagonal.
The `printSolution` function prints the current configuration of the board.
This algorithm can be called with `solveNQueens(n)` where `n` is the size of the board and the number of queens to be placed. The algorithm will print all possible solutions to the problem.

Explain Brach and Bound Strategy in general.

Branch and bound is an algorithmic technique for solving optimization problems. It involves systematically exploring the solution space by dividing it into smaller subproblems and calculating upper and lower bounds on the optimal solution. The algorithm uses these bounds to prune subproblems that cannot lead to a better solution than the current best-known solution.

The branch and bound approach works by maintaining a list of active subproblems that need to be explored. At each step, the algorithm selects a subproblem from the list and divides it into smaller subproblems by branching on a decision variable. For each subproblem, the algorithm calculates an upper and lower bound on the optimal solution using a bounding function. If the lower bound of a subproblem is greater than or equal to the upper bound of the current best-known solution, the subproblem can be pruned since it cannot lead to a better solution. Otherwise, the subproblem is added to the list of active subproblems.

The branch and bound algorithm continues exploring subproblems until the list of active subproblems is empty or a termination condition is met. The final solution is given by the best-known solution found during the search.

Branch and bound can be used to solve a wide range of optimization problems including integer programming, traveling salesman problem, and knapsack problem. The efficiency of branch and bound algorithms depends on the problem being solved, the quality of the bounding function, and the strategy used to select subproblems from the list of active subproblems.

Compare Branch and Bound and Backtracking

Parameter	Backtracking	Branch and Bound
Approach	Backtracking is used to find all possible solutions available to a problem. When it realises that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.	Branch-and-Bound is used to solve optimisation problems. When it realises that it already has a better optimal solution that the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution.
Traversal	Backtracking traverses the state space tree by DFS(Depth First Search) manner.	Branch-and-Bound traverse the tree in any manner, DFS or BFS.
Function	Backtracking involves feasibility function.	Branch-and-Bound involves a bounding function.
Problems	Backtracking is used for solving Decision Problem.	Branch-and-Bound is used for solving Optimisation Problem.
Searching	In backtracking, the state space tree is searched until the solution is obtained.	In Branch-and-Bound as the optimum solution may be present anywhere in the state space tree, so the tree need to be searched completely.

Efficiency	Backtracking is more efficient.	Branch-and-Bound is less efficient.
Applications	Useful in solving N-Queen Problem, Sum of subset, Hamilton cycle problem, graph coloring problem	Useful in solving Knapsack Problem, Travelling Salesman Problem.
Solve	Backtracking can solve almost any problem. (Chess, sudoku, etc).	Branch-and-Bound cannot solve almost any problem.
Used for	Typically backtracking is used to solve decision problems.	Branch and bound is used to solve optimization problems.
Nodes	Nodes in state space tree are explored in depth first tree.	Nodes in tree may be explored in depth-first or breadth-first order.
Next move	Next move from current state can lead to bad choice.	Next move is always towards better solution.
Solution	On successful search of solution in state space tree, search stops.	Entire state space tree is search to find optimal solution.

Explain the importance of bounding function in generating the solutions.

The bounding function plays a crucial role in generating solutions in various optimization algorithms. Its primary purpose is to define a region or domain within which the search for optimal solutions is conducted. By imposing constraints on the search space, the bounding function helps to narrow down the feasible solutions and guide the optimization process.

Here are some key points highlighting the importance of bounding functions:

1. Feasibility: Bounding functions ensure that the generated solutions satisfy certain constraints or requirements. These constraints can be based on specific problem characteristics, such as physical limitations, resource constraints, or legal regulations. By bounding the search space, the algorithm focuses only on feasible solutions, eliminating the need to evaluate infeasible or invalid solutions.

2. Efficiency: Bounding functions help in reducing the computational effort required to search for optimal solutions. By restricting the search space, the algorithm can avoid evaluating a large number of unpromising solutions. This leads to more efficient exploration of the feasible region, saving computational time and resources.

3. Optimization Guidance: Bounding functions provide guidance to the optimization algorithm by defining the boundaries of the search space. These boundaries act as a guidepost for the algorithm to direct its search towards regions where promising solutions are likely to be found. By confining the search within a bounded region, the algorithm can concentrate its efforts on areas that are more likely to contain optimal solutions.

4. Solution Quality: The use of bounding functions can significantly improve the quality of the solutions generated by an optimization algorithm. By restricting the search space, the algorithm is encouraged to explore regions that are more likely to contain high-quality solutions. This focus on a bounded region helps in avoiding suboptimal or inefficient solutions that may lie outside the defined boundaries.

5. Problem-specific Considerations: Bounding functions can incorporate problem-specific knowledge or insights into the optimization process. By considering the specific characteristics of the problem, such as symmetry, monotonicity, or known solution bounds, the bounding function can effectively guide the search towards optimal solutions.

There are several types of bounding functions used in optimization algorithms. Here are four common types, along with examples:

Box Bounding:

Box bounding functions define a rectangular bounding box within which the search space is confined. Each variable is bounded within a specified range.

For example, consider a problem where we want to find the maximum value of a function f (x, y) = x^2 + y^2. We can use a box bounding function to restrict the search space by specifying that x and y must lie within certain bounds, such as -5 ≤ x ≤ 5 and -3 ≤ y ≤ 3. The optimization algorithm would then search for the maximum within this bounded region.

Linear Constraint Bounding:

Linear constraint bounding functions impose linear constraints on the search space. These constraints restrict the feasible solutions to lie within specific linear regions.

For instance, consider a linear programming problem where we want to maximize a linear objective function subject to linear constraints. The bounding function can define the feasible region by specifying the linear constraints, such as ax + by ≤ c, where a, b, and c are constants. The optimization algorithm would then search for the optimal solution within this linearly bounded space.

Nonlinear Constraint Bounding:

Nonlinear constraint bounding functions impose nonlinear constraints on the search space. These constraints define regions in which the feasible solutions must lie.

For example, consider an optimization problem where we want to minimize a function f(x) subject to the constraint g(x) ≤ 0, where both f(x) and g(x) are nonlinear functions. The bounding function can incorporate the nonlinear constraint g(x) ≤ 0, restricting the search to the region where the constraint is satisfied. The optimization algorithm would then search for the optimal solution within this nonlinearly bounded space.

Problem-Specific Bounding:

Problem-specific bounding functions are tailored to the characteristics of the specific optimization problem at hand. These functions incorporate domain-specific knowledge to define the search space boundaries.

For instance, consider a scheduling problem where we want to minimize the total completion time of tasks subject to various constraints. A problem-specific bounding function could consider the specific constraints, such as task dependencies or resource limitations, to define the feasible region. The optimization algorithm would then search for the optimal solution within this problem-specific bounded space.

Algorithm for branch and bound strategy:

```
function BranchAndBound(problem):
    UB = Infinity  // Initial upper bound
    LB = -Infinity // Initial lower bound

    subproblems = Priority Queue  // Store subproblems

    initial_subproblem = CreateInitialSubproblem(problem)  // Create initial
subproblem
    subproblems.Enqueue(initial_subproblem)  // Enqueue initial subproblem

    while subproblems is not empty:
        subproblem = subproblems.Dequeue()  // Dequeue subproblem with highest
priority
        if isFeasible(subproblem):  // Check subproblem feasibility
            if isComplete(subproblem):  // Check termination condition
                updateUpperBound(subproblem)  // Update upper bound if solution
quality is better
            else:
                computeBound(subproblem)  // Compute upper and lower bounds for
subproblem
                if bound is better than UB:
                    createSubproblems(subproblem)  // Branch the subproblem and
create new subproblems
                    enqueueSubproblems(subproblems)  // Enqueue new subproblems

    return getOptimalSolution()  // Return optimal solution or best solution found
```

Basic Algorithm for divide and conquer strategy:

```
// Function to perform divide and conquer
function divideAndConquer(problem):
    // Base case: if the problem is small enough, solve it directly
    if problem is small enough:
        return solve problem directly

    // Divide the problem into smaller subproblems
    subproblems = divide(problem)

    // Conquer each subproblem recursively
    solutions = []
    for each subproblem in subproblems:
        solution = divideAndConquer(subproblem)
        solutions.append(solution)

    // Combine the solutions of subproblems into a single solution
    result = combine(solutions)

    // Return the combined solution
    return result
```

Write an algorithm using Divide and conquer approach for finding minimum and maximum number form a given set. Analyze it's time complexity by stating its recurrence relation. Simulate the above algorithm to find Min and Max on the following elements. Show the tree of recursive calls

22 13 -5 -8 15 60 17 31 47

```
1   Algorithm MaxMin(i, j, max, min)
2   // a[1 : n] is a global array. Parameters i and j are integers,
3   // 1 ≤ i ≤ j ≤ n. The effect is to set max and min to the
4   // largest and smallest values in a[i : j], respectively.
5   {
6       if (i = j) then max := min := a[i]; // Small(P)
7       else if (i = j − 1) then // Another case of Small(P)
8            {
9                if (a[i] < a[j]) then
10               {
11                   max := a[j]; min := a[i];
12               }
13               else
14               {
15                   max := a[i]; min := a[j];
16               }
17           }
18           else
19           {   // If P is not small, divide P into subproblems.
20               // Find where to split the set.
21                   mid := ⌊(i + j)/2⌋;
22               // Solve the subproblems.
23                   MaxMin(i, mid, max, min);
24                   MaxMin(mid + 1, j, max1, min1);
25               // Combine the solutions.
26                   if (max < max1) then max := max1;
27                   if (min > min1) then min := min1;
28           }
29  }
```

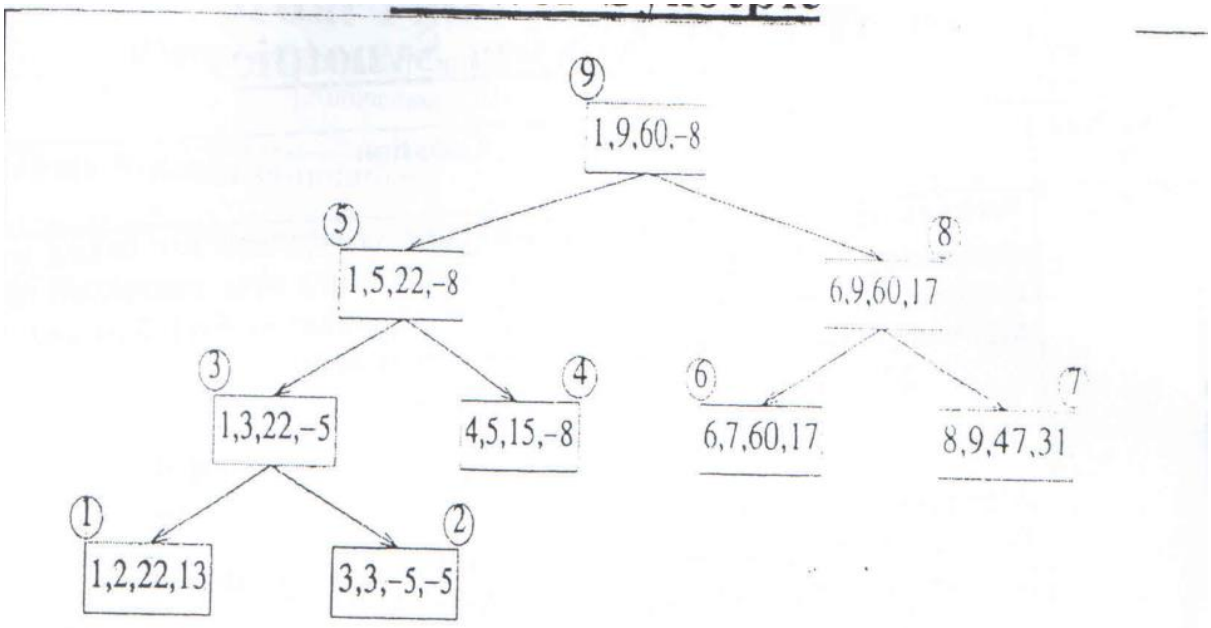


Figure 3.2 Trees of recursive calls of MaxMin

$T(n)$ represents this number, then the resulting recurrence relation is

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

When n is a power of two, $n = 2^k$ for some positive integer k, then

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \\ &= 2(2T(n/2) + 2) + 2 \\ &= 4T(n/2) + 4 + 2 \\ &\vdots \\ &= 2^{k-1}T(2) + \textstyle\sum_{1 \le i \le k-1} 2^i \\ &= 2^{k-1} + 2^k - 2 = 3n/2 - 2 \end{aligned} \qquad (3.3)$$

Note that $3n/2 - 2$ is the best-, average-, and worst-case number of comparisons when n is a power of two.

What are the steps of sequence we should follow to develop a dynamic programming approach. Show how these steps are applicable to solve Longest common subsequence problem efficiently using Dynamic programming approach

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all j $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

A subsequence of a given sequence is just the given sequence with zero or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$ another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all j $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$.

Step 1: Characterizing a longest common subsequence----------02 marks

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step 2: A recursive solution ----------02 marks

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

Step 3: Computing the length of an LCS----------02 marks

Procedure LCS-LENGTH takes two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ as inputs. It stores the c(I, j) values in a table c(0....m; 0 .... n), and it computes the entries in *row-major* order. The procedure also maintains the table b(1...m; 1... n) to help us construct an optimal solution. Intuitively(I, j) points to the table entry corresponding to the optimal sub problem solution chosen when computing c[I, j].

Step 4: Constructing an LCS----------02 marks

We simply begin at b[m, n] and trace through the table by following the arrows. Whenever we encounter a "↖" in entry $b[i, j]$, it implies that $x_i = y_j$ is an element of the LCS that LCS-LENGTH

Dijkstra's algorithm:

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Compare and contrast P and NP Problems:

P and NP are two important classes of computational problems in computer science. Let's compare and contrast them:

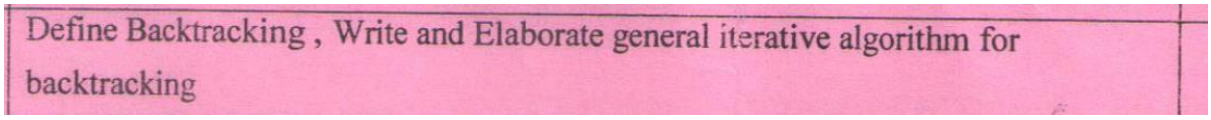
P Problems:

1. Definition: P stands for "Polynomial Time." P problems are the class of decision problems that can be solved by a deterministic Turing machine in polynomial time.
2. Solvability: P problems are efficiently solvable. There exist algorithms that can solve these problems in polynomial time, meaning the time it takes to solve the problem grows polynomially with the input size.
3. Examples: Sorting a list of numbers, finding the shortest path in a graph, checking if a number is prime, and matrix multiplication are examples of problems that belong to the class P.
4. Verification: P problems can be efficiently verified. Given a solution, it can be checked in polynomial time whether the solution is correct.
5. Complexity: P is a subset of NP (P ⊆ NP). If a problem is in P, it is also in NP.

NP Problems:

1. Definition: NP stands for "Nondeterministic Polynomial Time." NP problems are the class of decision problems for which a solution can be verified in polynomial time by a deterministic Turing machine.
2. Solvability: NP problems may not be efficiently solvable. While it may be difficult to find a solution, if one is given, it can be verified efficiently.
3. Examples: The traveling salesman problem, the subset sum problem, the Boolean satisfiability problem (SAT), and the graph colouring problem are examples of problems in NP.
4. Verification: NP problems can be efficiently verified. Given a potential solution, it can be checked in polynomial time whether the solution is correct.
5. Complexity: It is unknown whether P = NP or P ≠ NP. This is one of the most significant unsolved problems in computer science. If P = NP, it means that every problem in NP can be solved in polynomial time, making P and NP equivalent.

In summary, P problems are efficiently solvable, whereas NP problems may be difficult to solve but their solutions can be efficiently verified. While P problems are a subset of NP problems, it is unknown whether the two classes are equivalent. Resolving the P vs. NP question has profound implications for computational complexity theory and practical applications in areas such as cryptography and optimization.



Definition 1:

Backtracking *can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.*

Definition 2:

Backtracking is a general algorithmic technique that involves exploring all possible solutions to a problem by incrementally building candidates to the solutions and abandoning a candidate ("backtracking") as soon as it is determined that the candidate cannot possibly be completed to a valid solution.

```
backtrackingAlgorithm(problem):
    stack = empty stack
    stack.push(initialState)

    while stack is not empty do:
        currentState = stack.pop()

        if isGoalState(currentState) then:
            return currentState  // Found a solution

        if isInvalidState(currentState) then:
            continue  // Skip to the next iteration

        // Generate next valid states and push them onto the stack
        nextStates = generateNextStates(currentState)
        for nextState in nextStates do:
            stack.push(nextState)

    return null  // No solution found
```

Compare divide and conquer, greedy approach and dynamic programming approach

Divide and Conquer, Greedy approach, and Dynamic Programming are three popular algorithmic paradigms used to solve problems. Let's compare and contrast them:

1. Divide and Conquer:
    - Approach: The problem is divided into smaller subproblems, recursively solved, and then combined to obtain the final solution.
    - Process: The problem is broken down into smaller, more manageable subproblems, which are independently solved. The solutions to subproblems are then combined to form the solution to the original problem.
    - Key Concepts: Recursion, subproblem decomposition, conquer and combine.
    - Examples: Merge Sort, Quick Sort, Binary Search, and Strassen's Matrix Multiplication.
2. Greedy Approach:
    - Approach: Makes locally optimal choices at each step with the hope of finding a global optimum.
    - Process: At each step, the greedy algorithm makes the choice that appears to be the best at that moment, without considering the future consequences.
    - Key Concepts: Greedy choice property (optimal choice at each step), and the problem exhibits the optimal substructure property.
    - Examples: Fractional Knapsack Problem, Dijkstra's Shortest Path Algorithm (for non-negative edge weights), and Huffman Coding.
3. Dynamic Programming:

- Approach: Breaks down the problem into overlapping subproblems, solves each subproblem only once, and stores the results for future use.
- Process: Dynamic programming builds the solution to a problem by solving smaller overlapping subproblems and storing their solutions in a table. The stored solutions are then used to solve larger subproblems until the complete problem is solved.
- Key Concepts: Optimal substructure (problem can be solved using solutions to smaller subproblems) and overlapping subproblems (reducing redundant calculations).
- Examples: Fibonacci sequence, Longest Common Subsequence, Knapsack Problem, and Floyd-Warshall Algorithm (all-pairs shortest path).

Comparison:

- Recursion: Divide and Conquer and Dynamic Programming heavily rely on recursion, while Greedy algorithms usually don't require recursion.
- Optimality: Divide and Conquer and Dynamic Programming guarantee an optimal solution, whereas Greedy algorithms may not always produce the globally optimal solution.
- Subproblems: Divide and Conquer solves independent subproblems, whereas Dynamic Programming solves overlapping subproblems. Greedy algorithms typically do not involve subproblems.
- Time Complexity: Divide and Conquer has a typical time complexity of O(n log n), Greedy algorithms have various time complexities depending on the problem, and Dynamic Programming can have a time complexity of O(n^2) or higher, depending on the problem.

Overall, the choice of algorithmic paradigm depends on the problem's characteristics, requirements, and constraints. Divide and Conquer is suitable when the problem can be divided into independent subproblems, Greedy algorithms are efficient for locally optimal solutions, and Dynamic

Programming is beneficial for problems with overlapping subproblems and optimal substructure.

Question

One of the important criteria is to evaluate time which algorithm takes to complete its single task. This is done to get meaningful Comparison between the algorithms , to get the computational time independent from the programming language , compiler, application software , operating system,and computer hardware.

Asymptotic Notation: The notation use to define the asymptotic running time of an algorithms are describe in terms of functions whose domains are set of the natural numbers. Types of Asymptotic Notation used to compare the efficiency and performance of algorithm as follows:

1. O-notation(Big O notation).
2. Θ-notation(Theta notation).
3. Ω -notation(Big Omega notation).

Theta Notation(Θ-notation):

$\Theta(g(n)) = \{ f(n)$: There exist three positive constant C1 ,C2 and n_0 such that $0 \le C1g(n) \le f(n) \le C2g(n)$ for all

$n \ge n_0\}$.

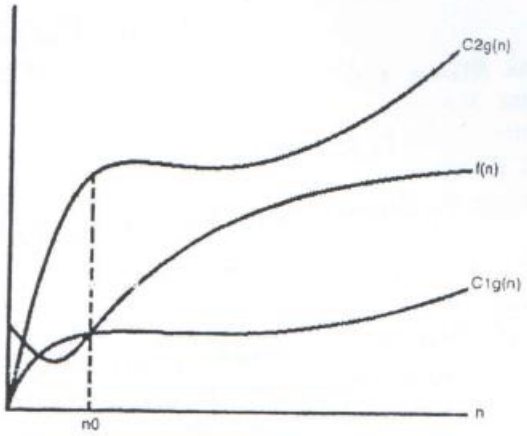
or

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$

or

$\Theta(g(n)) = \{f(n) \mid \exists C1 > 0, \exists C2 > 0, \forall n > n_0 : 0 \le C1g(n) \le f(n) \le C2g(n) \}$

The graph of Θ-notation is as follows:



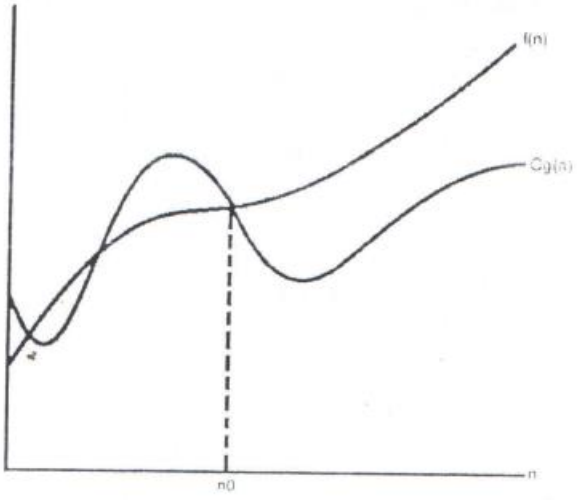
Big O- notation(O-notation):

$O(g(n)) = \{ f(n) : \text{exists positive constant C and } n_0 \text{ such that } 0 \le f(n) \le Cg(n) \text{ for all } n \ge n_0 \}$

or

$O(g(n)) = \{ f(n) : \exists C>0 , \exists n_0>0 , \forall n \ge n_0 : 0 \le f(n) \le Cg(n) \}$

The graph of BigO-notation:



Theta Notation(Θ-notation):

$\Theta(g(n)) = \{ f(n)$: There exist three positive constant C1 ,C2 and n_0 such that $0 \le C1g(n) \le f(n) \le C2g(n)$ for all

$n \ge n_0\}$.

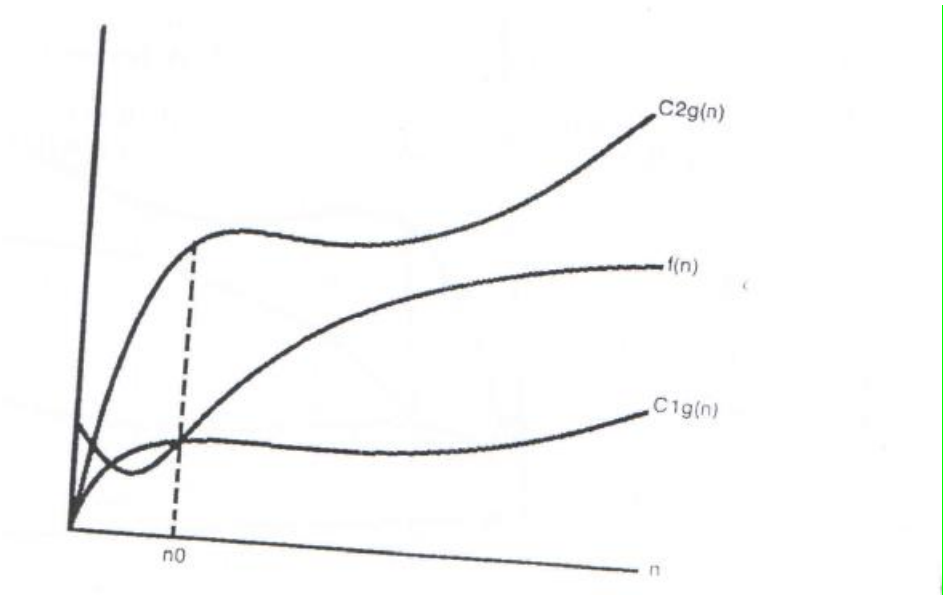
or

$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

or

$\Theta(g(n)) = \{f(n) \mid \exists C1 > 0, \exists C2 > 0, \forall n > n_0 : 0 \le C1g(n) \le f(n) \le C2g(n)\}$

The graph of Θ-notation is as follows:



i)Use definition of θ. Show that $\frac{1}{2}n^2 - 3n = \theta(n^2)$

$c_1 n^2 \le \frac{1}{2}n^2 - 3n \le c_2 n^2$

for all $n \ge n_0$. Dividing by n^2 yields

$c_1 \le \frac{1}{2} - \frac{3}{n} \le c_2$.

The right-hand inequality can be made to hold for any value of $n \ge 1$ by choosing $c_2 \ge 1/2$. Likewise, the left-hand inequality can be made to hold for any value of $n \ge 7$ by choosing $c_1 \le 1/14$. Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$. Certainly, other choices for the constants exist, but the important thing is that *some* choice exists. Note that these constants depend on the function $\frac{1}{2}n^2 - 3n$; a different function belonging to $\Theta(n^2)$ would usually require different constants.