14. ASSIGNMENT TOPICS WITH MATERIAL
Informal Definition:

An Algorithm is any well-defined computational procedure that takes some value or set of values as Input and produces a set of values or some value as output. Thus algorithm is a sequence of computational steps that transforms the i/p into the o/p.

Formal Definition:

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

1. INPUT: Zero or more quantities are externally supplied.
2. OUTPUT: At least one quantity is produced.
3. DEFINITENESS: Each instruction is clear and unambiguous.
4. FINITENESS: If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. EFFECTIVENESS: Every instruction must very basic so that it can be carried out, in principle, by a person using only pencil & paper.

Issues or study of Algorithm:

- How to device or design an algorithm
- How to express an algorithm
- How to analysis an algorithm
- How to validate an algorithm
- Testing the algorithm

Recursive Algorithms:
• A Recursive function is a function that is defined in terms of it.
• Similarly, an algorithm is said to be recursive if the same algorithm is invoked in the body.
• An algorithm that calls itself is Direct Recursive.
• Algorithm ‘A’ is said to be Indirect Recursive if it calls another algorithm which in turns calls ‘A’.

2. Space complexity and time complexity of algorithms?

**Time Complexity:**

The time T(p) taken by a program P is the sum of the compile time and the run time (execution time)
☐ The compile time does not depend on the instance characteristics. Also we may assume that a compiled program will be run several times without recompilation. This run time is denoted by tp(instance characteristics).
☐ The number of steps any problem statement is assigned depends on the kind of statement. For example, comments ☐ 0steps. Assignment statements ☐ 1steps. Interactive statement such as for, while & repeat-until ☐ Control part of the statement.
1. We introduce a variable, count into the program statement to increment count with initial value 0. Statement to increment count by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.

**Algorithm:**

```
Algorithm sum(a,n)
{
  s= 0.0;
  count = count+1; for I=1 to n do
  {
    count =count+1; s=s+a[I]; count=count+1;
  }
  count=count+1; count=count+1; return s;
}
```

- If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.
- The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.
- First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.
- By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.
3. Asymptotic notations

Types of functions, limits, and simplification

Logarithmic Function - \( \log n \)
Linear Function - \( an + b \)
Quadratic Function - \( an^2 + bn + c \)
Polynomial Function - \( an^z + \ldots + a^n + a^n + a^n \), where \( z \) is some constant
Exponential Function - \( a^n \), where \( a \) is some constant

These are some basic function growth classifications used in various notations. The list starts at the slowest growing function (logarithmic, fastest execution time) and goes on to the fastest growing (exponential, slowest execution time). Notice that as \( n \), or the input, increases in each of those functions, the result clearly increases much quicker in quadratic, polynomial, and exponential, compared to logarithmic and linear.

One extremely important note is that for the notations about to be discussed you should do your best to use simplest terms. This means to disregard constants, and lower order terms, because as the input size (or \( n \) in our \( f(n) \) example) increases to infinity (mathematical limits), the lower order terms and constants are of little to no importance. That being said, if you have constants that are \( 2^{9001} \), or some other ridiculous, unimaginable amount, realize that simplifying will skew your notation accuracy.

Since we want simplest form, let's modify our table a bit…
Logarithmic - \( \log n \)

Linear - \( n \)

Quadratic - \( n^2 \)

Polynomial - \( n^z \), where \( z \) is some constant

Exponential - \( a^n \), where \( a \) is some constant

**Big-O**

Big-O, commonly written as \( O \), is an Asymptotic Notation for the worst case, or ceiling of growth for a given function. It provides us with an **asymptotic upper bound** for the growth rate of runtime of an algorithm. Say \( f(n) \) is your algorithm runtime, and \( g(n) \) is an arbitrary time complexity you are trying to relate to your algorithm. \( f(n) \) is \( O(g(n)) \), if for some real constants \( c \) (\( c > 0 \)) and \( n_0 \), \( f(n) \leq c \cdot g(n) \) for every input size \( n \) (\( n > n_0 \)).

*Example 1*

\[ f(n) = 3\log n + 100 \]

\[ g(n) = \log n \]

Is \( f(n) \) \( O(g(n)) \)? Is \( 3 \log n + 100 \) \( O(\log n) \)? Let’s look to the definition of Big-O.

\( 3\log n + 100 \leq c \cdot \log n \)

Is there some pair of constants \( c, n_0 \) that satisfies this for all \( n > 0 \)?

\( 3\log n + 100 \leq 150 \cdot \log n, n > 2 \) (undefined at \( n = 1 \))

**B N JYOTHI** Assistant Professor
Yes! The definition of Big-O has been met therefore $f(n)$ is $O(g(n))$.

**Example 2**

\[
egin{align*}
    f(n) &= 3n^2 \\
    g(n) &= n
\end{align*}
\]

Is $f(n)$ $O(g(n))$? Is $3n^2$ $O(n)$? Let’s look at the definition of Big-O.

\[3n^2 \leq c \cdot n\]

Is there some pair of constants $c$, $n_0$ that satisfies this for all $n > 0$? No, there isn’t. $f(n)$ is NOT $O(g(n))$.

**Big-Omega**

Big-Omega, commonly written as $\Omega$, is an Asymptotic Notation for the best case, or a floor growth rate for a given function. It provides us with an asymptotic lower bound for the growth rate of runtime of an algorithm.

$f(n)$ is $\Omega(g(n))$, if for some real constants $c$ ($c > 0$) and $n_0$ ($n_0 > 0$), $f(n)$ is $\geq c \cdot g(n)$ for every input size $n$ ($n > n_0$).

**Note**
The asymptotic growth rates provided by big-O and big-omega notation may or may not be asymptotically tight. Thus we use small-o and small-omega notation to denote bounds that are not asymptotically tight.

**Small-o**

Small-o, commonly written as $o$, is an Asymptotic Notation to denote the upper bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm. $f(n) = o(g(n))$, if for some real constants $c (c > 0)$ and $n_0 (n_0 > 0)$, $f(n) < cg(n)$ for every input size $n (n > n_0)$. The definitions of $O$-notation and $o$-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $f(n) \leq g(n)$ holds for some constant $c > 0$, but in $f(n) = o(g(n))$, the bound $f(n) < cg(n)$ holds for all constants $c > 0$.

**Small-omega**

Small-omega, commonly written as $\omega$, is an Asymptotic Notation to denote the lower bound (that is not asymptotically tight) on the growth rate of runtime of an algorithm. $f(n) = \omega(g(n))$, if for some real constants $c (c > 0)$ and $n_0 (n_0 > 0)$, $f(n) > cg(n)$ for every input size $n (n > n_0)$. The definitions of $\Omega$-notation and $\omega$-notation are similar. The main difference is that in $f(n) = \Omega(g(n))$, the bound $f(n) \geq g(n)$ holds for some constant $c > 0$, but in $f(n) = \omega(g(n))$, the bound $f(n) > cg(n)$ holds for all constants $c > 0$.

**Theta**

Theta, commonly written as $\Theta$, is an Asymptotic Notation to denote the asymptotically tight bound on the growth rate of runtime of an algorithm. $f(n) = \Theta(g(n))$, if for some real constants
c1, c2 and \( n_0 (c1 > 0, c2 > 0, n_0 > 0) \), \( c1 \ g(n) \) is \( \leq f(n) \) is \( \leq c2 \ g(n) \) for every input size \( n \) (\( n > n_0 \)). \( f(n) \) is \( \Theta(g(n)) \) implies \( f(n) \) is \( O(g(n)) \) as well as \( f(n) \) is \( \Omega(g(n)) \).

4. Divide conquer approach for binary search

1. **Divide** problem into several smaller subproblems
   - Normally, the subproblems are similar to the original
2. **Conquer** the subproblems by solving them recursively
   - Base case: solve small enough problems by brute force
3. **Combine** the solutions to get a solution to the subproblems
   - And finally a solution to the original problem

5. Divide and Conquer algorithms are normally recursive

**Binary Search**

- A Divide and Conquer Algorithm to find a key in an array:

  ```
  -- Precondition: S is a sorted list  index bin search(number n, index low, index high, constkeytype S[], keytype x)
  if low \leq high then
    mid = (low + high) / 2
    if x = S[mid] then
      return mid
    elseif x < S[mid] then
      return binsearch(n, low, mid-1, S, x)
    else
  ```

B N JYOTHI Assistant Professor
return binsearch(n, mid+1, high, S, x)

else
    return 0
end binsearch

• Divide: search lower or upper half
  o Divide: select lower or upper half
  o Conquer: search selected half
  o Combine: None
• Performance:
  o \( T(n) = T(n/2) + \Theta(1) \)
  o \( T(n) = \Omega(\log n) \) \( T(n) = \Theta(\log n) \) (proved earlier)

4. Strassen’s Matrix multiplication?

Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

1) Divide matrices A and B in 4 sub-matrices of size \( \frac{N}{2} \times \frac{N}{2} \) as shown in the below diagram.
2) Calculate following values recursively. \( ae + bg, af + bh, ce + dg \) and \( cf + dh \).
In the above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions. Addition of two matrices takes $O(N^2)$ time. So the time complexity can be written as

$$T(N) = 8T(N/2) + O(N^2)$$

From Master’s Theorem, time complexity of above method is $O(N^3)$ which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(N^3)$, can there be a better way?

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of Strassen’s method is to reduce the number of recursive calls to 7. Strassen’s method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $N/2 \times N/2$ as shown in the above diagram, but in Strassen’s method, the four sub-matrices of result are calculated using following formulae.
Design and Analysis of Algorithms

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C:

\[
\begin{pmatrix}
  a & b \\
  c & d \\
\end{pmatrix}
\begin{pmatrix}
  e & f \\
  g & h \\
\end{pmatrix}
= 
\begin{pmatrix}
  p1 + p2 & p5 + p4 - p2 + p6 \\
  p3 + p4 & p1 + p5 - p3 - p7 \\
\end{pmatrix}
\]

A, B and C are square matrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
\( e, f, g \) and \( h \) are submatrices of B, of size N/2 x N/2
\( p1, p2, p3, p4, p5, p6 \) and \( p7 \) are submatrices of size N/2 x N/2

<table>
<thead>
<tr>
<th>Time Complexity of Strassen’s Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition and Subtraction of two matrices takes ( O(N^2) ) time. So time complexity can be written as ( T(N) = 7T(N/2) + O(N^2) )</td>
</tr>
</tbody>
</table>

From Master’s Theorem, time complexity of above method is \( O(N^{\log_2 7}) \) which is approximately \( O(N^{2.8074}) \)

Generally Strassen’s Method is not preferred for practical applications for following reasons.
1) The constants used in Strassen’s method are high and for a typical application Naive method works better.
2) For sparse matrices, there are better methods especially designed for them.
3) The sub matrices in recursion take extra space.
4) Because of the limited precision of computer arithmetic on non integer values, larger errors accumulate in Strassen’s algorithm than in Naive Method.
UNIT II

1. Union and Find algorithms?

A disjoint-set data structure is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

Find: Determine which subset a particular element is in. This can be used for determining if two elements are in the same subset.

Union: Join two subsets into a single subset.

In this post, we will discuss an application of Disjoint Set Data Structure. The application is to check whether a given graph contains a cycle or not.

Union-Find Algorithm can be used to check whether an undirected graph contains cycle or not. Note that we have discussed an algorithm to detect cycle. This is another method based on Union-Find. This method assumes that graph doesn’t contain any self-loops. We can keeps track of the subsets in a 1D array, lets call it parent[].
Let us consider the following graph:

For each edge, make subsets using both the vertices of the edge. If both the vertices are in the same subset, a cycle is found.

Initially, all slots of parent array are initialized to -1 (means there is only one item in every subset).

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Now process all edges one by one.

Edge 0-1: Find the subsets in which vertices 0 and 1 are. Since they are in different subsets, we take the union of them. For taking the union, either make node 0 as parent of node 1 or vice-versa.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&lt;---1 is made parent of 0 (1 is now representative of subset {0, 1})</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Edge 1-2: 1 is in subset 1 and 2 is in subset 2. So, take union.
A spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph G has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.
We found three spanning trees off one complete graph. A complete undirected graph can have maximum $n^{n-2}$ number of spanning trees, where $n$ is the number of nodes. In the above addressed example, $n$ is 3, hence $3^{3-2} = 3$ spanning trees are possible.

**General Properties of Spanning Tree**

We now understand that one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph $G$ –

- A connected graph $G$ can have more than one spanning tree.
- All possible spanning trees of graph $G$, have the same number of edges and vertices.
- The spanning tree does not have any cycle (loops).
- Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is **minimally connected**.
• Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is **maximally acyclic**.

**Mathematical Properties of Spanning Tree**

• Spanning tree has \( n-1 \) edges, where \( n \) is the number of nodes (vertices).

• From a complete graph, by removing maximum \( e - n + 1 \) edges, we can construct a spanning tree.

• A complete graph can have maximum \( n^{n-2} \) number of spanning trees.

Thus, we can conclude that spanning trees are a subset of connected Graph G and disconnected graphs do not have spanning tree.

**Application of Spanning Tree**

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

• **Civil Network Planning**

• **Computer Network Routing Protocol**

• **Cluster Analysis**

Let us understand this through a small example. Consider, city network as a huge graph and now plans to deploy telephone lines in such a way that in minimum lines we can connect to all city nodes. This is where the spanning tree comes into picture.

**Minimum Spanning Tree (MST)**

In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph. In real-world situations, this weight can be measured as distance, congestion, traffic load or any arbitrary value denoted to the edges.

**Minimum Spanning-Tree Algorithm**

We shall learn about two most important spanning tree algorithms here –
3. Backtracking method and its applications

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.\footnote{\textcite{1}}\footnote{\textcite{2}}

The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other. In the common backtracking approach, the partial candidates are arrangements of \(k\) queens in the first \(k\) rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution. It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test. Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles. It is often the most convenient (if not the most efficient\footnote{\textcite{citation needed}}) technique for parsing, for the knapsack problem and other combinatorial optimization problems. It is also the basis of the so-called logic programming languages such as Icon, Planner and Prolog.
Backtracking depends on user-given "black box procedures" that define the problem to be solved, the nature of the partial candidates, and how they are extended into complete candidates. It is therefore a metaheuristic rather than a specific algorithm—although, unlike many other meta-heuristics, it is guaranteed to find all solutions to a finite problem in a bounded amount of time. The term "backtrack" was coined by American mathematician D. H. Lehmer in the 1950s. The pioneer string-processing language SNOCBOL (1962) may have been the first to provide a built-in general backtracking facility.

4. 8 queen’s problem

Let us discuss N Queen as another example problem that can be solved using Backtracking. The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.
The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for above 4 queen solution.

\[
\begin{align*}
\{ & 0, 1, 0, 0 \\
& 0, 0, 0, 1 \\
& 1, 0, 0, 0 \\
& 0, 0, 1, 0 \\
\end{align*}
\]

**Naive Algorithm**

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

```
while there are untried conflagrations
{
    generate the next configuration
    if queens don't attack in this configuration then
    {
        print this configuration;
    }
}
```

**Backtracking Algorithm**

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.

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 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.
3) If all rows have been tried and nothing worked, return false to trigger backtracking.

5. Graph colouring and Hamiltonian cycles

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 coloured with same colour. Here colouring of a graph means assignment of colours to all vertices.
Input:
1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.
2) An integer m which is maximum number of colours that can be used.

Output:
An array colour[V] that should have numbers from 1 to m. colour[i] should represent the colour assigned to the ith vertex. The code should also return false if the graph cannot be coloured with m colours.

Following is an example of graph that can be coloured with 3 different colours.

Naive Algorithm
Generate all possible configurations of colours and print a configuration that satisfies the given constraints.

while there are untried configurations
Design and Analysis of Algorithms

Design and Analysis of Algorithms

generate the next configuration

if no adjacent vertices are coloured with same colour
{
    print this configuration;
}

There will be $V^m$ configurations of colours.

**Backtracking**

The idea is to assign colours one by one to different vertices, starting from the vertex 0. Before assigning a colour, we check for safety by considering already assigned colours to the adjacent vertices. If we find a colour assignment which is safe, we mark the colour assignment as part of solution. If we do not find a colour due to clashes then we backtrack and return false.

- **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 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 print the path. Following are the input and output of the required function.

- **Input:**
  A 2D array `graph[V][V]` where `V` is the number of vertices in graph and `graph[V][V]` is adjacency matrix representation of the graph. A value `graph[i][j]` is 1 if there is a direct edge from `i` to `j`, otherwise `graph[i][j]` is 0.

- **Output:**
  An array `path[V]` that should contain the Hamiltonian Path. `path[i]` should represent the `i`th vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.
For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}. There are more Hamiltonian Cycles in the graph like {0, 3, 4, 2, 1, 0}

(0)---(1)---(2)
|   /   |
|  /    |
| /     |
(3)-----(4)

And the following graph doesn’t contain any Hamiltonian Cycle

(0)---(1)---(2)
|   /   |
|  /    |
| /     |
(3)  (4)

Recommended: Please solve it on “PRACTICE” first, before moving on to the solution.

Naive Algorithm

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

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;
  | }
  |
• **Backtracking Algorithm**

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.

---

**UNIT III**

1. **Greedy method**

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. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property: At every step, we can make a choice that looks best at the moment, and

B N JYOTHI Assistant Professor
we get the optimal solution of the complete problem. If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming. But Greedy algorithms cannot always be applied. For example, Fractional Knapsack problem (See this) can be solved using Greedy, but 0-1 Knapsack cannot be solved using Greedy.

Following are some standard algorithms that are Greedy algorithms.

1) **Kruskal’s Minimum Spanning Tree (MST)**: In Kruskal’s algorithm, we create a MST by picking edges one by one. The Greedy Choice is to pick the smallest weight edge that doesn’t cause a cycle in the MST constructed so far.

2) **Prim’s Minimum Spanning Tree**: In Prim’s algorithm also, we create a MST by picking edges one by one. We maintain two sets: set of the vertices already included in MST and the set of the vertices not yet included. The Greedy Choice is to pick the smallest weight edge that connects the two sets.

3) **Dijkstra’s Shortest Path**: The Dijkstra’s algorithm is very similar to Prim’s algorithm. The shortest path tree is built up, edge by edge. We maintain two sets: set of the vertices already included in the tree and the set of the vertices not yet included. The Greedy Choice is to pick the edge that connects the two sets and is on the smallest weight path from source to the set that contains not yet included vertices.

4) **Huffman Coding**: Huffman Coding is a loss-less compression technique. It assigns variable length bit codes to different characters. The Greedy Choice is to assign least bit length code to the most frequent character.

The greedy algorithms are sometimes also used to get an approximation for Hard optimization problems. For example, **Traveling Salesman Problem** is a NP Hard problem. A Greedy choice for this problem is to pick the nearest unvisited city from the current city at every step. This

B N JYOTHI Assistant Professor
solutions doesn’t always produce the best optimal solution, but can be used to get an approximate optimal solution.

Let us consider the Activity Selection problem as our first example of Greedy algorithms. Following is the problem statement.

You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Example:

**Example 1:** Consider the following 3 activities sorted by finish time.

```plaintext
start[] = {10, 12, 20};
finish[] = {20, 25, 30};
```

A person can perform at most two activities. The maximum set of activities that can be executed is {0, 2} [ These are indexes in start[] and finish[] ]

**Example 2:** Consider the following 6 activities sorted by finish time.

```plaintext
start[] = {1, 3, 0, 5, 8, 5};
```
Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the 0-1 Knapsack problem, we are not allowed to break items. We either take the whole item or don’t take it.

Input:

Items as (value, weight) pairs

arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, \( W = 50 \);

Output:

Maximum possible value = 220

by taking items of weight 20 and 30 kg

In \textbf{Fractional Knapsack}, we can break items for maximizing the total value of knapsack. This problem in which we can break item also called fractional knapsack problem.

Input :

Same as above

Output :

Maximum possible value = 240

By taking full items of 10 kg, 20 kg and

2/3rd of last item of 30 kg

A \textbf{brute-force solution} would be to try all possible subset with all different fraction but that will be too much time taking.

An \textbf{efficient solution} is to use Greedy approach. The basic idea of greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with highest ratio and add them until we can’t add the next item as whole and at the end add the next item as much as we can. Which will always be optimal solution of this problem.

A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order. After sorting we need to loop over these items and add them in our knapsack satisfying above mentioned criteria.

\textbf{B N JYOTHI Assistant Professor}
3. Job sequencing with deadlines

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 single unit of time, so the minimum possible deadline for any job is 1. How to maximize total profit if only one job can be scheduled at a time.

Examples:

Input: Four Jobs with following deadlines and profits

<table>
<thead>
<tr>
<th>JobID</th>
<th>Deadline</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>40</td>
</tr>
<tr>
<td>d</td>
<td>1</td>
<td>30</td>
</tr>
</tbody>
</table>

Output: Following is maximum profit sequence of jobs

    c, a

Input: Five Jobs with following deadlines and profits

<table>
<thead>
<tr>
<th>JobID</th>
<th>Deadline</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>19</td>
</tr>
</tbody>
</table>
A Simple Solution is to 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. The time complexity of this solution is exponential.

This is a standard Greedy Algorithm problem. Following is algorithm.

1) Sort all jobs in decreasing order of profit.
2) Initialize the result sequence as first job in sorted jobs.
3) Do following for remaining n-1 jobs
   .......a) If the current job can fit in the current result sequence
         without missing the deadline, add current job to the result.
         Else ignore the current job.

4. Single source shortest path problem
Given a graph and a source vertex in graph, find shortest paths from source to all vertices in the given graph.

Dijkstra’s algorithm is very similar to Prim’s algorithm for minimum spanning tree. 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 minimum distance from source.

Below are the detailed steps used in Dijkstra’s algorithm to find the shortest path from a single source vertex to all other vertices in the given graph.

Algorithm

1) Create a set sptSet (shortest path tree set) that keeps track of vertices included in shortest path tree, i.e., whose minimum distance from source is calculated and finalized. Initially, this set is empty.

2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.

3) While sptSet doesn’t include all vertices

… a) Pick a vertex u which is not there in sptSet and has minimum distance value.

… b) Include u to sptSet.

… c) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.
Let us understand with the following example:

The set \( sptSet \) is initially empty and distances assigned to vertices are \{0, INF, INF, INF, INF, INF, INF, INF\} where INF indicates infinite. Now pick the vertex with minimum distance value. The vertex 0 is picked, include it in \( sptSet \). So \( sptSet \) becomes \{0\}. After including 0 to \( sptSet \), update distance values of its adjacent vertices. Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8. Following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in green color.

Pick the vertex with minimum distance value and not already included in SPT (not in \( sptSET \)). The vertex 1 is picked and added to \( sptSet \). So \( sptSet \) now becomes \{0, 1\}. Update the distance values of adjacent vertices of 1. The distance value of vertex 2 becomes 12.
Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes \{0, 1, 7\}. Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (15 and 9 respectively).

We repeat the above steps until sptSet doesn’t include all vertices of given graph. Finally, we get the following Shortest Path Tree (SPT).

How to implement the above algorithm?
We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store shortest distance values of all vertices.

5. minimum cost spanning trees

What is Minimum Spanning Tree?
Given a connected and undirected graph, a spanning tree of that graph is a sub graph that is a tree and connects all the vertices together. A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

How many edges does a minimum spanning tree has?
A minimum spanning tree has (V-1) edges where V is the number of vertices in the given graph.

What are the applications of Minimum Spanning Tree?
See this for applications of MST.

Below are the steps for finding MST using Kruskal’s algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.
The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far. Let us understand it with an example: Consider the below input graph.

![Graph Image]

The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having \((9-1) = 8\) edges.

<table>
<thead>
<tr>
<th>Weight</th>
<th>SrcDest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7 6</td>
</tr>
<tr>
<td>2</td>
<td>8 2</td>
</tr>
<tr>
<td>2</td>
<td>6 5</td>
</tr>
<tr>
<td>4</td>
<td>0 1</td>
</tr>
<tr>
<td>4</td>
<td>2 5</td>
</tr>
<tr>
<td>6</td>
<td>8 6</td>
</tr>
<tr>
<td>7</td>
<td>2 3</td>
</tr>
<tr>
<td>7</td>
<td>7 8</td>
</tr>
<tr>
<td>8</td>
<td>0 7</td>
</tr>
</tbody>
</table>
Now pick all edges one by one from sorted list of edges

1. Pick edge 7-6: No cycle is formed, include it.

2. Pick edge 8-2: No cycle is formed, include it.

3. Pick edge 6-5: No cycle is formed, include it.
4. Pick edge 0-1: No cycle is formed, include it.

5. Pick edge 2-5: No cycle is formed, include it.

6. Pick edge 8-6: Since including this edge results in a cycle, discard it.

7. Pick edge 2-3: No cycle is formed, include it.

8. Pick edge 7-8: Since including this edge results in a cycle, discard it.

B N JYOTHI Assistant Professor
9. Pick edge 0-7: No cycle is formed, include it.

10. Pick edge 1-2: Since including this edge results in cycle, discard it.

11. Pick edge 3-4: No cycle is formed, include it.

Since the number of edges included equals (V - 1), the algorithm stops here.

UNIT IV

1. Applications of dynamic programming

Dynamic Programming is also used in optimization problems. Like divide-and-conquer method, Dynamic Programming solves problems by combining the solutions of subproblems. Moreover,
Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are **overlapping sub-problems and optimal substructure**.

**Overlapping Sub-Problems**

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don’t have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

**Optimal Sub-Structure**

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property –

If a node $x$ lies in the shortest path from a source node $u$ to destination node $v$, then the shortest path from $u$ to $v$ is the combination of the shortest path from $u$ to $x$, and the shortest path from $x$ to $v$.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

**Steps of Dynamic Programming Approach**

Dynamic Programming algorithm is designed using the following four steps –

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.

B N JYOTHI Assistant Professor
Design and Analysis of Algorithms

- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

**Applications of Dynamic Programming Approach**

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

**2. Chained matrix multiplication**

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

\[(ABC)D = (AB)(CD) = A(BCD) = \ldots\]

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a 10 \times 30 matrix, B is a 30 \times 5 matrix, and C is a 5 \times 60 matrix. Then,

\[(AB)C = (10\times30\times5) + (10\times5\times60) = 1500 + 3000 = 4500 \text{ operations}\]

\[A(BC) = (30\times5\times60) + (10\times30\times60) = 9000 + 18000 = 27000 \text{ operations}.\]

Clearly the first parenthesization requires less number of operations.

B N JYOTHI Assistant Professor
Given an array $p[]$ which represents the chain of matrices such that the $i$th matrix $A_i$ is of dimension $p[i-1] \times p[i]$. We need to write a function $\text{MatrixChainOrder()}$ that should return the minimum number of multiplications needed to multiply the chain.

**Input:** $p[] = \{40, 20, 30, 10, 30\}$  
**Output:** 26000  
There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way 
\[(A(BC))D \rightarrow 20*30*10 + 40*20*10 + 40*10*30\]

**Input:** $p[] = \{10, 20, 30, 40, 30\}$  
**Output:** 30000  
There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30. Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way 
\[((AB)C)D \rightarrow 10*20*30 + 10*30*40 + 10*40*30\]

**Input:** $p[] = \{10, 20, 30\}$  
**Output:** 6000  
There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is 10*20*30

1) **Optimal Substructure:**
A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size $n$, we can place the first set of parenthesis in $n-1$ ways. For example, if the given chain is of 4 matrices. let the chain be
ABCD, then there are 3 ways to place first set of parenthesis outer side: (A)(BCD), (AB)(CD) and (ABC)(D). So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size n = Minimum of all n-1 placements (these placements create subproblems of smaller size)

3. All pairs shortest path problem

The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. Example:

**Input:**

```plaintext
graph[][] = { {0, 5, INF, 10},
             {INF, 0, 3, INF},
             {INF, INF, 0, 1},
             {INF, INF, INF, 0} }
```

which represents the following graph

```
  10
(0)------>(3)
  |
  |  \|
  5 |    |
  |    | 1
  |    |
  \|   |
  (1)------>(2)
```
Note that the value of graph[i][j] is 0 if i is equal to j
And graph[i][j] is INF (infinite) if there is no edge from vertex i to j.

**Output:**
Shortest distance matrix

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>5</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>INF</td>
<td>INF</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>INF</td>
<td>INF</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>INF</td>
<td>INF</td>
<td>INF</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

*Recommended: Please solve it on “PRACTICE” first, before moving on to the solution.*

**Floyd Warshall Algorithm**

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices \(\{0, 1, 2, .. k-1\}\) as intermediate vertices. For every pair \((i, j)\) of source and destination vertices respectively, there are two possible cases.

1) k is not an intermediate vertex in shortest path from i to j. We keep the value of dist[i][j] as it is.
2) k is an intermediate vertex in shortest path from i to j. We update the value of dist[i][j] as dist[i][k] + dist[k][j].

4. 0/1 knapsack problem
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).

**0-1 Knapsack Problem**

| value[] = {60, 100, 120}; | Weight = 10; Value = 60; |
| weight[] = {10, 20, 30}; | Weight = 20; Value = 100; |
| W = 50; | Weight = 30; Value = 120; |
| Solution: 220 | Weight = (20+10); Value = (100+60); |
| | Weight = (30+10); Value = (120+60); |
| | Weight = (30+20); Value = (120+100); |
| | Weight = (30+20+10) > 50 |

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.

**1) Optimal Substructure:**

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

Therefore, the maximum value that can be obtained from n items is max of following two values.

1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.
5. Traveling salesperson problem

**Travelling Salesman Problem (TSP):** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

**Naive Solution:**
1) Consider city 1 as the starting and ending point.
2) Generate all \((n-1)!\) Permutations of cities.

3) Calculate cost of every permutation and keep track of minimum cost permutation.

4) Return the permutation with minimum cost.

Time Complexity: \(?(n!)?\)

**Dynamic Programming:**

Let the given set of vertices be \(\{1, 2, 3, 4, \ldots, n\}\). Let us consider 1 as starting and ending point of output. For every other vertex \(i\) (other than 1), we find the minimum cost path with 1 as the starting point, \(i\) as the ending point and all vertices appearing exactly once. Let the cost of this path be \(\text{cost}(i)\), the cost of corresponding Cycle would be \(\text{cost}(i) + \text{dist}(i, 1)\) where \(\text{dist}(i, 1)\) is the distance from \(i\) to 1. Finally, we return the minimum of all \([\text{cost}(i) + \text{dist}(i, 1)]\) values. This looks simple so far. Now the question is how to get \(\text{cost}(i)\)?

To calculate \(\text{cost}(i)\) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems. Let us define a term \(C(S, i)\) be the cost of the minimum cost path visiting each vertex in set \(S\) exactly once, starting at 1 and ending at \(i\).

We start with all subsets of size 2 and calculate \(C(S, i)\) for all subsets where \(S\) is the subset, then we calculate \(C(S, i)\) for all subsets \(S\) of size 3 and so on. Note that 1 must be present in every subset.

If size of \(S\) is 2, then \(S\) must be \(\{1, i\}\),

\[C(S, i) = \text{dist}(1, i)\]

Else if size of \(S\) is greater than 2.

\[C(S, i) = \min \{ C(S - \{i\}, j) + \text{dist}(j, i) \} \text{ where } j \text{ belongs to } S, j \neq i \text{ and } j \neq 1.\]

For a set of size \(n\), we consider \(n-2\) subsets each of size \(n-1\) such that all subsets don’t have \(n\)th in them. Using the above recurrence relation, we can write dynamic programming based solution.
There are at most $O(n \times 2^n)$ subproblems, and each one takes linear time to solve. The total running time is therefore $O(n^2 \times 2^n)$. The time complexity is much less than $O(n!)$, but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

UNIT V

1LC Branch and Bound solution?

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly.

Let us consider below 0/1 Knapsack problem to understand Branch and Bound.

Given two integer arrays $val[0..n-1]$ and $wt[0..n-1]$ that represent values and weights associated with $n$ items respectively. Find out the maximum value subset of $val[]$ such that sum of the weights of this subset is smaller than or equal to Knapsack capacity $W$.

Let us explore all approaches for this problem.

1. A Greedy approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.

2. We can use Dynamic Programming (DP) for 0/1 Knapsack problem. In DP, we use a 2D table of size $n \times W$. The DP Solution doesn’t work if item weights are not integers.

B N JYOTHI Assistant Professor
3. Since DP solution doesn’t always work, a solution is to use Brute Force. With $n$ items, there are $2^n$ solutions to be generated, check each to see if they satisfy the constraint, save maximum solution that satisfies constraint. This solution can be expressed as tree.

4. We can use Backtracking to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.
Branch and Bound

The backtracking based solution works better than brute force by ignoring infeasible solutions. We can do better (than backtracking) if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

Example bounds used in below diagram are, A down can give $315, B$ down can $275, C$ down can $225, D$ down can $125 and E$ down can $30. In the next article, we have discussed the process to get these bounds.
Branch and bound is a very useful technique for searching a solution but in worst case, we need to fully calculate the entire tree. At best, we only need to fully calculate one path through the tree and prune the rest of it.

2. FIFO Branch and Bound solution

Branch & Bound discovers branches within the complete search space by using estimated bounds to limit the number of possible solutions. The different types (FIFO, LIFO, LC) define different 'strategies' to explore the search space and generate branches.

**FIFO** (first in, first out): always the oldest node in the queue is used to extend the branch. This leads to a breadth-first search, where all nodes at depth \(d\) are visited first, before any nodes at depth \(d+1\) are visited.

**LIFO** (last in, first out): always the youngest node in the queue is used to extend the branch. This leads to a depth-first search, where the branch is extended through every 1st child discovered at a certain depth, until a leaf node is reached.

B N JYOTHI Assistant Professor
LC (lowest cost): the branch is extended by the node which adds the lowest additional costs, according to a given cost function. The strategy of traversing the search space is therefore defined by the cost function.

3. Non-deterministic algorithms

In computer science, a nondeterministic algorithm is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition. A probabilistic algorithm's behavior depends on a random number generator. An algorithm that solves a problem in nondeterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during execution. The nondeterministic algorithms are often used to find an approximation to a solution, when the exact solution would be too costly to obtain using a deterministic one.
A deterministic algorithm that performs $f(n)$ steps always finishes in $f(n)$ steps and always returns the same result. A non deterministic algorithm that has $f(n)$ levels might not return the same result on different runs. A non deterministic algorithm may never finish due to the potentially infinite size of the fixed height tree.

4. NP - Hard and NP- Complete classes

We have been writing about efficient algorithms to solve complex problems, like shortest path, Euler graph, minimum spanning tree, etc. Those were all success stories of algorithm designers. In this post, failure stories of computer science are discussed. Can all computational problems be solved by a computer? There are computational problems that can not be solved by algorithms even with unlimited time. For example Turing Halting problem (Given a program and an input, whether the program will eventually halt when run with that input, or will run forever). Alan Turing proved that general algorithm to solve the halting problem for all possible program-input pairs cannot exist. A key part of the proof is, Turing machine was used as a mathematical definition of a computer and program (Source Halting Problem). Status of NP Complete problems
is another failure story, NP complete problems are problems whose status is unknown. No polynomial time algorithm has yet been discovered for any NP complete problem, nor has anybody yet been able to prove that no polynomial-time algorithm exist for any of them. The interesting part is, if any one of the NP complete problems can be solved in polynomial time, then all of them can be solved. What are NP, P, NP-complete and NP-Hard problems? P is set of problems that can be solved by a deterministic Turing machine in Polynomial time. NP is set of decision problems that can be solved by a Non-deterministic Turing Machine in Polynomial time. P is subset of NP (any problem that can be solved by deterministic machine in polynomial time can also be solved by non-deterministic machine in polynomial time). Informally, NP is set of decision problems which can be solved by a polynomial time via a “Lucky Algorithm”, a magical algorithm that always makes a right guess among the given set of choices (Source Ref 1).

NP-complete problems are the hardest problems in NP set. A decision problem L is NP-complete if:
1) L is in NP (Any given solution for NP-complete problems can be verified quickly, but there is no efficient known solution).
2) Every problem in NP is reducible to L in polynomial time (Reduction is defined below).

A problem is NP-Hard if it follows property 2 mentioned above, doesn’t need to follow property 1. Therefore, NP-Complete set is also a subset of NP-Hard set.
Decision vs Optimization Problems

NP-completeness applies to the realm of decision problems. It was set up this way because it’s easier to compare the difficulty of decision problems than that of optimization problems. In reality, though, being able to solve a decision problem in polynomial time will often permit us to solve the corresponding optimization problem in polynomial time (using a polynomial number of calls to the decision problem). So, discussing the difficulty of decision problems is often really equivalent to discussing the difficulty of optimization problems. (Source Ref 2).

For example, consider the vertex cover problem (Given a graph, find out the minimum sized vertex set that covers all edges). It is an optimization problem. Corresponding decision problem is, given undirected graph G and k, is there a vertex cover of size k?

What is Reduction? Let L₁ and L₂ be two decision problems. Suppose algorithm A₂ solves L₂. That is, if y is an input for L₂ then algorithm A₂ will answer Yes or No depending upon whether y...
Design and Analysis of Algorithms

belongs to \( L_2 \) or not. The idea is to find a transformation from \( L_1 \) to \( L_2 \) so that the algorithm \( A_2 \) can be part of an algorithm \( A_1 \) to solve \( L_1 \).

![Algorithm for L1](image)

Learning reduction in general is very important. For example, if we have library functions to solve certain problem and if we can reduce a new problem to one of the solved problems, we save a lot of time. Consider the example of a problem where we have to find minimum product path in a given directed graph where product of path is multiplication of weights of edges along the path. If we have code for Dijkstra’s algorithm to find shortest path, we can take log of all weights and use Dijkstra’s algorithm to find the minimum product path rather than writing a fresh code for this new problem.

How to prove that a given problem is NP complete? From the definition of NP-complete, it appears impossible to prove that a problem \( L \) is NP-Complete. By definition, it requires us to that show every problem in NP is polynomial time reducible to \( L \). Fortunately, there is an alternate way to prove it. The idea is to take a known NP-Complete problem and reduce it to \( L \). If polynomial time reduction is possible, we can prove that \( L \) is NP-Complete by transitivity of reduction (If a NP-Complete problem is reducible to \( L \) in polynomial time, then all problems are reducible to \( L \) in polynomial time).

What was the first problem proved as NP-Complete? There must be some first NP-Complete problem proved by definition of NP-Complete.
problems. SAT (Boolean satisfiability problem) is the first NP-Complete problem proved by Cook (See CLRS book for proof).

It is always useful to know about NP-Completeness even for engineers. Suppose you are asked to write an efficient algorithm to solve an extremely important problem for your company. After a lot of thinking, you can only come up exponential time approach which is impractical. If you don’t know about NP-Completeness, you can only say that I could not come up with an efficient algorithm.

If you know about NP-Completeness and prove that the problem as NP-complete, you can proudly say that the polynomial time solution is unlikely to exist. If there is a polynomial time solution possible, then that solution solves a big problem of computer science many scientists have been trying for years.

5. 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}.

B N JYOTHI Assistant Professor
Theorem-3
- For any $T_Q(k)$ of type $Q$, $T_Q(k)\sqrt{\log k}2T_Q(k)k\log k$ is unbounded
- There is a $T_Q(k)$ of type $Q$ such that $T_Q(k)\leq 2k\log k2T_Q(k)\leq 2k\log k$

Theorem-4
If the set $S$ of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ 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 $T_Q(K8^n)$.

- 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.
15. TUTORIAL TOPICS AND QUESTIONS
16. UNIT WISE QUESTION BANK

Unit I

Two marks questions with answers

1. Define Algorithm.

Answer: algorithm is a sequence of unambiguous instructions for solving a problem in a finite amount of time.

2. Write a short note on Algorithm Design and Analysis of Process.

Answer: Understand the problem
Decide on Computational Device Exact Vs Approximate Algorithms
Algorithm Design Techniques
Design an algorithm
Prove Correctness
Analyze the Algorithm
Code the Algorithm

3. What are the 2 kinds of Algorithm Efficiency?

Answer: Time Efficiency—How fast your algorithm runs Space Efficiency—How much extra memory your algorithm needs

4. How can you specify Algorithms?

Answer: Algorithms can be specified natural language or pseudo code

5. What is Combinatorial Problem?

Answer: This problem that ask to find combinatorial object such as permutations,
combinations or a Subset. Combinatorial problems are most difficult to solve. For
e.g. travelling sales man problem

**Three marks questions with answers**

1. **Discuss the Different approaches to find the time complexity of algorithm.**

**Answer:**
One specification of an algorithm is its correctness. You will probably assume that your algorithm
works after testing it out a few times. However, if you can **mathematically prove** that your
algorithm will work as expected for every input value possible, this is a huge bonus. Another
specification is its efficiency: how does the computing time relate to the amount of input? Is it a
linear relation? Does computing time rise exponentially for the doubling of input.

**Time Complexity**

Time complexity is, as mentioned above, the relation of computing time and the amount of input.
This is usually about the size of an array or an object. Time complexity also isn’t useful for simple
functions like fetching usernames from a database, concatenating strings or encrypting passwords.
It is used more for sorting functions, recursive calculations and things which generally take more
computing time.

This is not because we don’t care about that function’s execution time, but because the difference
is negligible. We don’t care if it takes 10ms instead of 3ms to fetch a username. However, if we
have a recursive sorting algorithm which takes 400ms and we can reduce that to 50ms, that would
be an interesting thing to do. As you might guess, the lower the computing time, the more efficient
the algorithm is. The question that follows is: ‘how can we define time complexity in an universal
way?’. That’s where we’ll use the ‘Big O notation’.

B N JYOTHI Assistant Professor
**Big O notation**

The Big O notation is a notation for the time complexity of an algorithm. It is a mathematical representation of the upper bound of the limit of the scaling factor of the algorithm. For example, if we double the size of an input array, by how much does the computing time increase? This might become clear with two examples:

```php
$numbers = array(14, 82, 4, 0, 24, 28);

foreach($numbers as $number) {
    echo $number;
}
```

Imagine that the `$numbers` array is the argument of the function. We have a foreach loop running through its items. If we calculate the time that the code takes to run, what happens if we double the size of the array? We can easily see in this example that it will double the time to run. We see that there is a linear relationship between the size of the array and the computing time. So if we write the size of the array as `n`, we can write the time complexity as O(n). Another example:

```php
$numbers = array(14, 82, 4, 0, 24, 28);

foreach($numbers as $number1) {
}
```
```
foreach($numbers as $number2)
{
    if($number1 >= $number2)
    {
        echo $number1." is greater than or equal to ". $number2;
    }
    else
    {
        echo $number1." is smaller than ". $number2;
    }
}
```
In this piece of code, there is a for each loop located inside another for each loop. Let’s say ‘n’ is the size of $numbers. Then we loop ‘n’ times through ‘n’. That makes the total amount of loops $n^2$. As you might guess, we write the time complexity as O($n^2$).

The big O notation expresses the scaling of computing time and uses some sort of mixture between the upper bound and the limit of that scaling. For example:

```php
$numbers=array(14,82,4,0,24,28);

foreach($numbers as $number1)
{
    foreach($numbers as $number2)
    {
        if($number1>=$number2)
        {
            echo $number1." is greater than or equal to ".$number2;
        }
    }
else
```
Efficient algorithms

Another special notation: $O(\log(n))$, which shows a logarithmic relationship. An example of an algorithm that uses this is the binary search algorithm. For those too lazy to read the full article: you want to find a name in an alphabetically ordered list and so you go to the centre. If the name you search comes before that, you go to the centre between the centre page and the beginning (so
the 1/4th). You continue that until you find the right name. The time complexity of that algorithm is $O(\log(n))$.

If you were to find the name by looping through the list entry after entry, the time complexity would be $O(n)$. While that isn’t bad, $O(\log(n))$ is many times better. It can be qualified as an efficient algorithm.

**Inefficient algorithms. Just as there are efficient algorithms, we have inefficient algorithms as well. One of them is the bog sort algorithm. While (fortunately) nobody actually uses it, it’s used as a demonstration of how you should.**

2. Explain divide and conquer method?

**Answer:**

Both merge sort and quick sort employ a common algorithmic paradigm based on recursion. This paradigm, divide-and-conquer, breaks a problem into sub problems that are similar to the original problem, recursively solves the sub problems, and finally combines the solutions to the sub problems to solve the original problem. Because divide-and-conquer solves sub problems recursively, each sub problem must be smaller than the original problem, and there must be a base case for sub problems. You should think of a divide-and-conquer algorithm as having three parts:

1. Divide the problem into a number of sub problems that are smaller instances of the same problem.
2. Conquer the sub problems by solving them recursively. If they are small enough, solve the sub problems as base cases.
3. Combine the solutions to the sub problems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as divide, conquer, combine. Here’s how to view one step, assuming that each divide step creates two sub problems (though some divide-and-conquer algorithms create more than two):
If we expand out two more recursive steps, it looks like this:
Because divide-and-conquer creates at least two sub problems, a divide-and-conquer algorithm makes multiple recursive calls.

3. **Explain Strassen’s matrix multiplication?**

**Answer:**

Given two square matrices A and B of size n x n each, find their multiplication matrix.

**Naïve Method** Following is a simple way to multiply two matrices.

```c
void multiply(int A[][N], int B[][N], int C[][N])
{
    for (int i = 0; i < N; i++)
    {
```
for (int j = 0; j < N; j++)
{
    C[i][j] = 0;
    for (int k = 0; k < N; k++)
    {
        C[i][j] += A[i][k]*B[k][j];
    }
}

Time Complexity of above method is $O(N^3)$.

4. Explain binary search time complexity

Answer:

Time complexity of binary search algorithm is $O(\log_2(N))$.

At a glance the complexity table is like this -

Worst case performance : $O(\log_2 n)$

Best case performance : $O(1)$

Average case performance: $O(\log_2 n)$

Worst case space complexity: $O(1)$

But that is not the fact, the fact is why it is $\log_2(N)$?

Here is a mathematical proof which describe why the complexity is $\log_2(N)$. 

B N JYOTHI Assistant Professor
The question is, how many times can you divide $N$ by 2 until you have 1? This is essentially saying, do binary searches (half the elements) until you found it. In a formula this would be this:

$$1 = \frac{N}{2^x}$$

multiply by $2^x$:

$$2^x = N$$

now do the log2:

$$\log_2(2^x) = \log_2 N$$

$$x \cdot \log_2(2) = \log_2 N$$

$$x \cdot 1 = \log_2 N$$

this means you can divide $\log N$ times until you have everything divided. Which means you have to divide $\log N$ ("do the binary search step") until you found your element.

4. **What is the time complexity of linear search?**

**Answer:**

We can have three cases to analyze an algorithm:

1) Worst Case
2) Average Case
3) Best Case

Let us consider the following implementation of Linear Search.

```c
#include <stdio.h>

// Linearly search x in arr[]. If x is present then return the index,
```
/ otherwise return -1
int search(int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
    {
        if (arr[i] == x)
            return i;
    }
    return -1;
}

/* Driver program to test above functions*/
int main()
{
    int arr[] = {1, 10, 30, 15};
    int x = 30;
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("%d is present at index %d", x, search(arr, n, x));

    getchar();
    return 0;
}

Worst Case Analysis (Usually Done)
In the worst case analysis, we calculate upper bound on running time of an algorithm. We must
know the case that causes maximum number of operations to be executed. For Linear Search,
the worst case happens when the element to be searched (x in the above code) is not present in

B N JYOTHI Assistant Professor
the array. When x is not present, the search() functions compares it with all the elements of arr[] one by one. Therefore, the worst case time complexity of linear search would be Θ(n).

Average Case Analysis (Sometimes done)

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in array). So we sum all the cases and divide the sum by (n+1). Following is the value of average case time complexity.

\[
\text{Average Case Time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}
\]

\[
= \frac{\theta((n+1)\times(n+2)/2)}{(n+1)}
\]

\[
= \Theta(n)
\]

<table>
<thead>
<tr>
<th>Best Case Analysis (Bogus)</th>
</tr>
</thead>
</table>

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Θ(1). Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we
guarantee an upper bound on the running time of an algorithm which is good information. The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs. The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn’t provide any information as in the worst case, an algorithm may take years to run.

5. Explain how to identify the repeated elements?

Answer:

Find the two repeating elements in a given array

You are given an array of n+2 elements. All elements of the array are in range 1 to n. And all elements occur once except two numbers which occur twice. Find the two repeating numbers.

For example, array = {4, 2, 4, 5, 2, 3, 1} and n = 5

The above array has n + 2 = 7 elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

Method 1 (Basic)
Use two loops. In the outer loop, pick elements one by one and count the number of occurrences of the picked element in the inner loop.

This method doesn’t use the other useful data provided in questions like range of numbers is between 1 to n and there are only two repeating elements.
Five marks questions with answers

1. Explain algorithm specification

   Answer:

   a. Pseudo code conventions

   1. Comments begin with // and continue until the end offline.
   2. Blocks are indicated and matching braces: {and}.
   3. An identifier begins with a letter
   4. Assignment of values to variables is done using the assignment statement
      \(<variable> := <expression>\)
   5. There are two Boolean values true and false
   6. Elements of multi dimensional array are accessed using [and].
   7. The following looping statements are employed: for, while and repeat – until.
   8. A conditional statement has the following forms: if <condition> then<statement>;
   9. Input and output are done using the instructions read and write.

   b. Recursive algorithms

   An algorithm is said to be recursive if the same algorithm is invoked in the body.
   An algorithm

   That calls itself is Direct recursive. Algorithm A is said to be indeed recursive if it
   calls another algorithm, which in turn calls A.
2. Explain all asymptotic notations Big oh

The function \( f(n) = O(g(n)) \) iff there exist positive constants \( C \) and \( n_0 \) such that
\[
f(n) \leq C \cdot g(n)
\]
for all \( n, n \geq n_0 \).

Omega

The function \( f(n) = \Omega(g(n)) \) iff there exist positive constant \( C \) and \( n_0 \) such that
\[
f(C \cdot g(n)) \text{ for all } n, n \geq n_0.
\]

Theta

The function \( f(n) = \Theta(g(n)) \) iff there exist positive constant \( C_1, C_2 \), and \( n_0 \) such that
\[
C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n) \text{ for all } n, n \geq n_0.
\]

Little oh

The function \( f(n) = o(g(n)) \) iff
\[
g(n)
\]

Little Omega

The function \( f(n) = \omega(g(n)) \) iff
\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0
\]

3. Explain binary search method

If \( q \) is always chosen such that \( a[q] \) is the middle element (that is, \( q = \lfloor (n+1)/2 \rfloor \), then the
resulting search algorithm is known as binary search.

Algorithm BinSearch(a,n,x)
//Given an array a[1:n] of elements in non decreasing order, n>0, determine whether x is present
{ 
low := 1; high := n;
while (low < high) do 
{
mid := (low+high)/2;
if(x \not=a[mid]) then high:= mid-1; else if (x \not=a[mid]) then low:=mid

Lim f(n) = 0 n - \infty + 1; else return mid;
}
return 0;
}

In conclusion we are now able completely describe the computing time of binary search by giving formulas that describe the best, average and worst cases. Successful searches\(\Theta(1) \Theta(\log n) \Theta(\log n)\) best average worst unsuccessful searches \(\Theta(\log n)\) best, average, worst

4. Explain the concepts of quick sort method and analyze its complexity

In quick sort, the division into sub arrays is made so that the sorted sub arrays do not need to be merged later.

Algorithm partition(a,m,p)

v:=a[m]; I:=m;
J:=p;
Repeat
{
repeat I:=I+1
;
Until(a[I]>=v); Repeat J:=j-1;
Until(a[j]<=v);
If(I<j) then interchange(a,I,j);
}until(I>=j);
a[m]:=a[j];
a[j]:=v; return j;
}
In analyzing QUICKSORT, we can only make the number of element comparisons $c(n)$. It is easy to see that the frequency count of other operations is of the same order as $C(n)$.

5. Write the algorithm for finding the maximum and minimum and explain it.

The problem is to find the maximum and minimum items in a set of $n$.

**Algorithm maxmin** ($I,j,max,min$)

```c
{
    if (I = j) then max := min := a[I]; else if (I = j - 1) then
        {
            if (a[I] < a[j]) then
                {
                    max := a[j]; min := a[I];
                }
            else
                {
                    max := a[I]; min := a[j];
                }
        }
    else
        {
            mid := [(I+j)/2]; maxmin(I,mid,maxmin); maxmin(mid+1,j,max1,min1);
            if (max < max1) then max := max1; if (min > min1) then min := min1;
        }
}
```

B N JYOTHI Assistant Professor
10 objective type questions with answers

1. There are ______steps to solve the problem
   A. Seven B. Four C. Six D. Two

2. ______is the first step in solving the problem
   A. Understanding the Problem B. Identify the Problem
   C. Evaluate the Solution D. None of these

3. ______is the last step in solving the problem
   A. Understanding the Problem B. Identify the Problem
   C. Evaluate the Solution D. None of these

4. Following is true for understanding of a problem
   A. Knowing the knowledgebase B. Understanding the subject on which the problem is based
   C. Communication with the client D. All of the above

5. The six-step solution for the problem can be applied to
   I. Problems with Algorithmic Solution
II. Problems with Heuristic Solution

A. Only I B. Only II C. Both I and II D. Neither I nor II

6. ______ solution requires reasoning built on knowledge and experience

A. Algorithmic Solution B. Heuristic Solution
C. Random Solution D. None of these

7. While solving the problem with computer the most difficult step is ______

A. describing the problem B. finding out the cost of the software
C. writing the computer instructions D. testing the solution

8. The correctness and appropriateness of ________solution can be checked very easily.

A. algorithmic solution B. heuristic solution
C. random solution D. none of these

9. The branch of computer that deals with heuristic types of problem is called _____________.

A. system software B. real time software
C. artificial intelligence D. none of these
10. Artificial Intelligence makes use of following prominently

A. Database  B. Data Warehouse  C. Knowledge base

D. None of these

<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>C</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
</tr>
</tbody>
</table>
10 fill up the blanks with answers

1. Assuming that the hash function for a table works well, and the size of the hash table is reasonably large compared to the number of items in the table, the expected (average) time needed to find an item in a hash table containing n items is

2. Heap sort is found to be very efficient

3. In heap sort the input is arranged in the form of a

4. Quick sort is the fastest available method of sorting because of

5. Straight selection sort is basically a method of repeated

6. Number of selections required to sort a file of size N by straight selection requires

7. Which of the following algorithms is NOT a divide & conquer algorithm by nature?

8. Which of the following standard algorithms is not Dynamic Programming based.

9. Which of the following is not true about comparison based sorting algorithms?

10. The recurrence relation capturing the optimal time of the Tower of Hanoi problem with n discs is.
Design and Analysis of Algorithms

<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>2</td>
<td>To free up resources allocated by that class</td>
</tr>
<tr>
<td>3</td>
<td>A a</td>
</tr>
<tr>
<td>4</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>5</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>6</td>
<td>with regard to storage requirement</td>
</tr>
<tr>
<td>7</td>
<td>Heap Sort</td>
</tr>
<tr>
<td>8</td>
<td>Prim’s Minimum Spanning Tree</td>
</tr>
<tr>
<td>9</td>
<td>Heap Sort is not a comparison based sorting algorithm.</td>
</tr>
<tr>
<td>10</td>
<td>$T(n) = 2T(n - 1) + 1$</td>
</tr>
</tbody>
</table>
Unit II

Two marks questions with answers

1. **What is Empirical Analysis?**

   **Answer:** It is performed by running a program implementing the algorithm on a sample of inputs and analyzing the data observed. This involves generating pseudo code and random numbers.

2. **Define Convex Hull Problem.**

   **Answer:** A set of points (finite or infinite) on the plane is called convex if for any two points P and Q in the set, the entire line segment with the end points at P and Q belongs to the set.

3. **What is Divide and Conquer Algorithm**

   **Answer:** It is a general algorithm design techniques that solved problem’s instance by dividing it into several smaller instance, solving of them recursively, and then combining their solutions to the original instance of the Problem.

4. **What are the Features of Algorithm Visualization?**

   **Answer:**
   - Consistent
   - Interactive
   - Very Clear and Concise
   - Emphasize the visual component

5. **Define O-Notation.**

   **Answer:** A function t(n) is said to be \( O(g(n)) \), denoted \( t(n) \in O(g(n)) \), if \( t(n) \) is bounded above by some constant multiple of \( g(n) \) for all large \( n \), i.e., if there exist some positive constant \( c \) and some nonnegative integer \( k \) such that

   \[ t(n) \leq cg(n) \text{ for all } n \geq k \]
Three marks questions with answers

1. What are important problem types? (or) Enumerate some important types of problems.

   Answer:
   1. Sorting 2. Searching

2. Name some basic Efficiency classes


3. What are algorithm design techniques?

   Answer: Algorithm design techniques (or strategies or paradigms) are general approaches to solving problems algorithmically, applicable to a variety of problems from different areas of computing. General design techniques are:
   (i) Brute force (ii) divide and conquer (iii) decrease and conquer (iv) transform and conquer (v) greedy technique (vi) dynamic programming (vii) backtracking (viii) branch and bound.

4. Define direct recursive and indirect recursive algorithms.

   Answer: Recursion occurs where the definition of an entity refers to the entity itself. Recursion can be direct when an entity refers to itself directly or indirect when it refers to other entities which refers to it. A (Directly) recursive routine calls itself. Mutually
recursive routines are an example of indirect recursion. A (Directly) recursive data type contains pointers to instances of the data type.

5. What are the characteristics of an algorithm?

**Answer:**

Every algorithm should have the following five characteristics

(i) Input

(ii) Output

(iii) Definiteness

(iv) Effectiveness

(v) Termination

Therefore, an algorithm can be defined as a sequence of definite and effective instructions, which terminates with the production of correct output from the given input. In other words, viewed little more formally, an algorithm is a step by step formalization of a mapping function to map input set onto an output set.

5. What do you mean by time complexity and space complexity of an algorithm?

**Answer:** Time complexity indicates how fast the algorithm runs. Space complexity deals with extra memory it require. Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size. Basic operation: the operation that contributes most towards the running time of the algorithm. The running time of an algorithm is the function defined by the number of steps required to solve input instances of size
Five marks questions with answers

1. Explain the concept of merge sort.

Answer:

In merge sort, the elements are to be sorted in non-decreasing order. Given a sequence of n elements i.e. a[1], a[2], ..., a[n], the general idea is to imagine them split into 2 sets a[1]...a[(n/2)] and a[(n/2)+1], ..., a[n]. Each set is individually sorted, and the resulting sorted sequence are merge to produce a single sorted sequence of „n” elements. The time complexity is \(O(n \log n)\) for worst case. Insertion sort works exceedingly fast on arrays of less than 16 elements, though for large „n” its computing time is \(O(n^2)\). If the time for the merging operation is proportional to n, then the computing time of merge sort is described by the recurrence relation

\[
T(n) = \begin{cases} 
1, & n=1, \text{ a a constant} \\
2T(n/2)+n, & n \not= 1, c \text{ a constant} 
\end{cases}
\]

Given a sequence of elements a[1], ..., a[n]. The general idea is to imagine them split in to 2 sets a[1], ..., a[n/2] and a[n/2 + 1], ..., a[n] each sets are individually sorted and the resulting sorted sequence are merged to produce a single sorted sequence of n elements.

Algorithm Merge Sort(low, high)

{ 
if (low<high) then 
{ 
mid := (low+high)/2;

mergesort(low, mid); mergesort(mid+1, high); merge(low,mid,high)
}
} 

}Merge Algorithm

Algorithm Merge(low, mid, high)

{ h=high, i=low, j=mid+1; while((h<=mid) and (j<=high)) do

B N JYOTHI Assistant Professor
Design and Analysis of Algorithms

\begin{multicols}{2}

\begin{verbatim}
if (a[h] <= a[j]) then 
  b[i] = a[h], h = h + 1;
else
  i = i + 1;
  b[i] = a[j], j = j + 1;
\}
\end{verbatim}

\begin{verbatim}
if (h > mid) then 
  for k = j to high do 
    b[i] = a[k]; i = i + 1;
else 
  for k = h to mid do 
    b[i] = a[k]; i = i + 1;
\end{verbatim}

\begin{verbatim}
for k = low to high do 
a[k] = b[k]
\end{verbatim}

2. Explain multistage graph

Answer: A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets $V_i$. The multistage graph problem is to find a minimum cost path from $s$ to $t$.

Graph: Using the forward approach we obtain
\begin{align*}
\text{Cost}(i, j) &= \min\{c(j, l) + \text{Cost}(i + 1, 1)\}
\end{align*}

Algorithm $F_{\text{graph}}(G, k, n, p)$
\begin{verbatim}
\{ 
  cost[n] := 0.0;
  for j := n-1 to 1 step -1 do 
\}
\end{verbatim}

\end{multicols}
Let \( r \) be a vertex such that \(<j, r>\) is an edge of \( G \) and \( c[j,r] + cost[r] \) is minimum; \( \text{Cost}[j] := c[j,r] + \text{cost}[r] \); 
\( d[j] := r \);

\[
\text{for } j := 2 \text{ to } k-1 \text{ do } p[j] := d[p[j-1]]; 
\]

From the backward approach we obtain

\[
\text{Bcost}(i,j) = \min\{ \text{bcost}(i-1,l) + c(l,j) \} 
\]

Algorithm Bgraph(G, k, n, p)
{
\[
\text{bcost}[1] := 0.0; \text{ for } j := 2 \text{ to } n \text{ do } 
\]

\[
\text{Let } r \text{ be such that } <r,j> \text{ is an edge of } G \text{ and } \text{bcost}[r]+c[r,j] \text{ is minimum; } 
\text{Bcost}[j] := \text{bcost}[r]+c[r,j]; 
\text{D}[j] := r; 
\]
\[
\text{p}[1] := 1; 
\text{p}[k] := n; 
\]
\[
\text{for } j := k-1 \text{ to } 2 \text{ do } p[j] := d[p[j+1]]; 
\]
}

3. **Explain the concepts of job sequencing with deadlines.**

**Answer:** Let \( j \) be a set of \( k \) jobs and \( \Sigma = i1, i2, \ldots, ik \) a permutation of jobs in \( j \) such that \( d_{i1} \leq d_{i2} \leq \ldots \leq d_k \). Then \( j \) is a feasible solution if and only if the jobs in \( j \) can be processed in the order without violating any deadlines.
* Given a set of \( n \) jobs each job has a deadline \( d_i \) such that \( d_i \geq 0 \) and a profit \( p_i \) such that \( p_i \geq 0 \).

solutions and their values are:

<table>
<thead>
<tr>
<th>Processi ng sequence</th>
<th>v</th>
<th>a</th>
<th>l</th>
<th>u</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B N JYOTHI Assistant Professor
Design and Analysis of Algorithms

* For job i, profit \( p_i \) is earned iff it is completed within deadline.

* Processing the job on the machine is for 1 unit of time.

Only one machine is available.

Example:

Let \( n = 4 \)

\((p_1, p_2, p_3, p_4) = (100, 10, 15, 27)\)

and \((d_1, d_2, d_3, d_4) = (2, 1, 2, 1). \) The feasible algorithm

\[ JS(d, j, n) \]

\[
\begin{align*}
&d[0] := j[0] := 0; \\
&j[1] := 1; \\
&k := 1; \\
&\text{for } k := 2 \text{ to } n \text{ do} \\
&\quad r := k; \\
&\quad \text{while}((d[j[r]] > d[1]) \text{ and } (d[j[r]] <> r)) \text{ do } r := r - 1;
\end{align*}
\]
if((d[j[r]]<=d[I]) and (d[I]>r)) then
{
for q:=k to r+ 1 step –1 do j[q+1] := j[q]; j[r+1]:=I;
k:=k+1;
}
return k;

For job sequence there are two possible parameters in terms of which its complexity can be measured. We can use n the number of jobs, and s, the number of jobs included in the solution j. If we consider the job set p= d_i= n – I + 1, 1 <= I <= n, the algorithm takes \( \varphi(n^2) \) time to determine j. Hence the worst case computing time for job sequence is \( \varphi(n^2) \)

4. Explain quick sort with example and also write algorithm tracing?

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

5. Always pick first element as pivot.

6. Always pick last element as pivot (implemented below)

7. Pick a random element as pivot.

8. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

B N JYOTHI Assistant Professor
Pseudo Code for recursive QuickSort function:

/* low --> Starting index, high --> Ending index */

quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1); // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

/* low --> Starting index, high --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now at right place */
        pi = partition(arr, low, high);
    
}
quickSort(arr, low, pi - 1); // Before pi
quickSort(arr, pi + 1, high); // After pi
}

Pseudo code for partition()

/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1) // Index of smaller element

    for (j = low; j <= high - 1; j++)
    {
    
    
    
}
5. Show that DFS and BFS visit all vertices in a connected graph

G reachable from any one of vertices.

Answer:
In the breadth first search we start at a vertex v and mark it as having been reached. The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices
adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been 
explored. Then newly visited vertices haven’t been explored and are put onto the end of a list of 
unexplored vertices. The first vertex on this list is the next to be explored. Exploration 
continues until no unexplored vertex is left.

Breadth first search
We start at a vertex V and mark it as have been reached. The vertex v is at this time said to be 
unexplored. All visited vertices adjacent from v are visited next.
If G is represented by its adjacent then the time is O(n2). Algorithm BFS(v)
{
u := v; visited[v] := 1; repeat
{
for all vertices w adjacent from u do
{
if (visited[w] = 0)then
{
add w to q; visited[w] :=1;
}
}
if q is empty then return; delete u from q;
} until (false)
}

A depth first search of a graph differs from a breadth first search in that the exploration of a 
vertex v is suspended as soon as a new vertex is reached.
At this time the exploration of the new vertex u begins. When the new vertex has been 
explored, the exploration of v continues.

Depth first search
The exploration of a vertex V is suspended as soon as a new vertex is reached. Algorithm DFS(v)

Algorithm DFS(v)
{
    visited[v]:=1;
    for each vertex q adjacent from v do
    {
        if (visited[w] =0 ) then DFS(w);
    }
}
1. Naming convention for variable is followed in company because

A. it enhances readability B. it allows to work without conflicts C. it enhances the efficiency D. all of the above

2. The true and false values represent

A. logical data B. numeric data C. character data D. alphanumeric data

3. Following operator distinguishes equation from expression

A. +, -, *, / B. < or > C. Logical operators D. Assignment Operator

4. Following are called logical operators

A. +, -, *, / B. <, >, <=, >= C. AND, OR, NOT D. \, MOD

5. The hierarchy of operations is denoted as ____________.

I. +, -
II. Power
III. *, /
IV. \, MOD


6. The hierarchy of operations is denoted as ____________.

I. +, -
II. Power
III. *, /
IV. \, MOD

7. Evaluate $5*(x+y)-4*y/(z+6)$ where $x = 2$, $y = 3$, and $z = 6$
   A. 1 B. 24 C. 5 D. 10

8. Evaluate $a-2>b$ where $a=6$, $b = 8$
   A. False B. True C. 6 D. 7

9. Evaluate for $a = 5$, $b = 4$, $c = 3$, $d = 12$ for the equation $E = a*b+d/c$
   A. 40 B. 24 C. 10 D. 10.66

10. Evaluate for the equation $e = 5*a\backslash d*(b+1)$ where $a = 5$, $b = 4$, $c = 3$, $d = 12$
    A. 10 B. 24 C. 0 D. 10

<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>7</td>
<td>B</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
</tr>
</tbody>
</table>

10 fill up the blanks with answers

B N JYOTHI Assistant Professor
1. A sort which compares adjacent elements in a list and switches where necessary is ____.

2. The way a card game player arranges his cards as he picks them one by one can be compared to

3. Which among the following is the best when the list is already sorted?

4. As part of the maintenance work, you are entrusted with the work of rearranging the library books in a shelf in proper order, at the end of each day. The ideal choice will be

5. In quick sort, the number of partitions into which the file of size $n$ is divided by a selected record is

6. The total number of comparisons made in quick sort for sorting a file of size $n$, is

7. Quick sort efficiency can be improved by adopting

8. For the improvement of efficiency of quick sort the pivot can be

9. Quick sort is the fastest available method of sorting because of

10. Straight selection sort is basically a method of repeated
<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>2</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>3</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>4</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>7</td>
<td>Partitioning</td>
</tr>
<tr>
<td>8</td>
<td>The mean element</td>
</tr>
<tr>
<td>9</td>
<td>Low overhead and also O(n log n) comparisons</td>
</tr>
<tr>
<td>10</td>
<td>Position adjustment</td>
</tr>
</tbody>
</table>
Two marks questions with answers

1. What is articulation point?

Answer:
A vertex of a connected graph G is said to be in articulation point, if its removal with all edges incident to it breaks the graph into disjoint pieces.

2. List the advantages of binary search?

Answer:
1. Less time is consumed
2. The processing speed is fast
3. The number of iterations is less. It take n/2 iterations.
4. Binary search, which is used in Fibonacci Series, involves addition and subtraction rather than division

It is priori analysis, since it can be analysed before execution.

3. Explain the principle used quick sort?

Answer:
It is a partition method using the particular key the given table is partitioned into 2 sub tables so that first, the original key will be its position the sorted sequence and secondly, all keys to the left of this key will be less value and all keys to the right of it will be greater values

4. What is binary search?

Answer:
The binary search algorithm some of the most efficient searching techniques which requires the list to be sort descending order.

Answer:
To search for an amount of the list, the binary search algorithms split the list and locate the middle element of the list.
First compare middle key $K_1$, with given key $K$. If $K_1=K$ then the element is found.

5. Why is bubble sort called by the name?

**Answer:**
The sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up bubbling up the largest element to the last position on the list. The next pass bubbles up the second largest element and so on until, after $n-1$ pass the list is sorted.
1. Define the divide an conquer method.
   
   **Answer:**

   Given a function to compute on 'n' inputs the divide-and-conquer strategy suggests splitting the inputs into 'k' distinct subsets, 1< k < n, yielding 'k' sub problems. The sub problems must be solved, and then a method must be found to combine sub solutions into a solution of the whole. If the sub problems are still relatively large, then the divide-and-conquer strategy can possibly be reapplied.

2. Define control abstraction.
   
   **Answer:**

   A control abstraction we mean a procedure whose flow of control is clear but whose primary operations are by other procedures whose precise meanings are left undefined.

   Write the Control abstraction for Divide-and conquer.

   Algorithm DAndC(P)

   {
   if small(p) then return S(P);
   else
   {
   divide P into smaller instance p1,p2,...,pk, k ≥ 1;
   Apply D And C to each of these sub problems
   Return combine (D And C(p1) D And C(p2),----, D And (pk));
   }
   }

3. Write a algorithm for straightforward maximum and minimum?
Answer:

Algorithm straight MaxMin(a, n, max, min)

// set max to the maximum and min to the minimum of a[1:n]
{
    max := min := a[i];
    for i = 2 to n do
        if(a[i] > max) then max := a[i];
        if(a[i] < min) then min := a[i];
    }

4. Write the algorithm for iterative binary search?

Answer:

Algorithm BinSearch(a, n, x)

// Given an array a[1:n] of elements in nondecreasing
// order, n > 0, determine whether x is present
{
    low := 1;
    high := n;
    while (low < high) do
        mid := [(low + high)/2];
        if(x < a[mid]) then high := mid - 1;
        else if (x > a[mid]) then low := mid + 1;
        else return mid;
5. Write the control abstraction for greedy method.

Answer:
Algorithm Greedy (a, n)
{
    solution=0;
    for i=1 to n do
    {
        x= select(a);
        if feasible(solution ,x) then
            solution=Union(solution ,x);
    }
    return solution;
}
Five marks questions with answers

1. **Insertion and Deletion of a Binary Search Tree.**

**Answer:**
To insert a new element $x$, we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point the search terminated. To insert an element with key 80 into the tree. First we search the element 80 is present in the tree. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new element is inserted as the right child of this node.

Algorithm Insert($x$)
```
{ 
  found := false; p := tree; 
  while ((p ≠ 0) and not found) do 
  { 
    q := p; 
    if (x=(p data)) then found :=true 
    else if (x < (p data)) then p := p lchild; else p := p rchild; 
  } 
  if (not found) then 
  { 
    p := new treenode; 
    p lchild :=; p rchild := 0; p data := x; if (tree ≠ 0) then 
    { 
      if (x < (q data)) then q lchild := p; else q rchild := p; 
    } 
    else tree := p; 
  }
}
b) Deletion from a binary tree.

To delete an element from the tree, the left child of its parent is set to 0 and the node disposed. To delete the 80 from this tree, the right child field of 40 is set to 0. Then the node containing 80 is disposed. The deletion of a nonleaf element that has only one child is also easy. The node containing the element to be deleted is disposed, and the single child takes the place of the disposed node. To delete another element from the tree, simply change the pointer from the parent node to the single child node.

2. Show that DFS and BFS visit all vertices in a connected graph G reachable from any one of vertices.

Answer:
In the breadth first search we start at a vertex v and mark it as having been reached. The vertex v is at this time said to be unexplored. A vertex is said to have been explored by an algorithm when the algorithm has visited all vertices adjacent from it. All unvisited vertices adjacent from v are visited next. These are new unexplored vertices. Vertex v has now been explored. The newly visited vertices haven’t been explored and are put on the end of the list of unexplored vertices. The first vertex on this list is the next to be explored. Exploration continues until no unexplored vertex is left.

Breadth first search

We start at a vertex V and mark it as have been reached. The vertex v is at this time said to be unexplored. All visited vertices adjacent from v are visited next. If G is represented by its adjacent then the time is O(n²). Algorithm BFS(v)

\{ 
\ u := v; \ visited[v] := 1; \ repeat
Design and Analysis of Algorithms

\{
  for all vertices w adjacent from u do
  \{
    if (visited[w] = 0) then
    \{
      add w to q; visited[w] := 1;
    \}
  \}
  if q is empty then return; delete u from q;
  \} until (false)
\}

A depth first search of a graph differs from a breadth first search in that the exploration of a vertex v is suspended as soon as a new vertex is reached. At this time the exploration of the new vertex u begins. When the new vertex has been explored, the exploration of v continues.

Depth first search

The exploration of a vertex v is suspended as soon as a new vertex is reached. Algorithm DFS(v)

Algorithm DFS(v)
\{
  visited[v]:=1;
  for each vertex q adjacent from v do
  \{
    if (visited[w] = 0) then DFS(w);
  \}
\}
1. **Explain the concepts of travelling salesperson problem.**

**Answer:**
Let $G = (V,E)$ be a directed graph with edge costs $c_{ij}$. The variable $c_{ij}$ is defined such that $c_{ij} > 0$ for all $i$ and $j$ and $c_{ij} = \infty$ if $(i,j) \notin E$. Let $|V| = n$ and assume $n > 1$. A tour of $G$ is a directed simple cycle that includes every vertex in $V$. The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem is to find a tour of minimum cost.

Let $G=(V,E)$ be a directed graph defining an instance of the traveling salesperson problem. Let $C_{ij}$ equal the cost of edge $(i,j)$, $c_{ij} = 0$ if $(i,j) \notin E$, and let $|V| = n$. The solution space $s$ is given by $s = \{1, 2, \ldots, n\}$.

Let us now trace the progress of the LCBB algorithm on the problem instance. The portion of the state space tree that gets generated. Matrix $M$ is obtained from that and setting all entries in row 1 and column 4.

### 4. Describe the Backtracking with M-COLORING Graph?

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

1) A 2D array $\text{graph}[V][V]$ where $V$ is the number of vertices in graph and $\text{graph}[V][V]$ is adjacency matrix representation of the graph. A value $\text{graph}[i][j]$ is 1 if there is a direct edge from $i$ to $j$, otherwise $\text{graph}[i][j]$ is 0.
2) An integer $m$ which is maximum number of colors that can be used.

Output: An array $\text{color}[V]$ that should have numbers from 1 to $m$. $\text{color}[i]$ should represent the color assigned to the $i$th vertex. The code should also return false if the graph cannot be colored with $m$ colors.
Algorithm:

If all colors are assigned,

    print vertex assigned colors

Else

    a. Trying all possible colors, assign a color to the vertex
    
    b. If color assignment is possible, recursively assign colors to next vertices
    
    c. If color assignment is not possible, de-assign color, return False

Code:

```python
def is_safe(n, graph, colors, c):
    # Iterate through adjacent vertices
    # and check if the vertex color is different from c
    for i in xrange(n):
        if graph[n][i] and c == colors[i]: return False
    return True
```
# n = vertex nb

def graphColoringUtil(graph, color_nb, colors, n):
    # Check if all vertices are assigned a color
    if color_nb+1 == n :
        return True

    # Trying different color for the vertex n
    for c in xrange(1, color_nb+1):
        # Check if assignment of color c to n is possible
        if is_safe(n, graph, colors, c):
            # Assign color c to n
            colors[n] = c
            # Recursively assign colors to the rest of the vertices
            if graphColoringUtil(graph, color_nb, colors, n+1): return True
            # If there is no solution, remove color (BACKTRACK)
            colors[n] = 0

We test the algorithm for the following graph and test whether it is 3 colorable:

(3)---(2)
|   / |
| /   |
#nb of vertex
vertex_nb = 4

# nb of colors
color_nb = 3

# Initiate vertex colors
colors = [0] * vertex_nb

doctor = [
    [0, 1, 1, 1],
    [1, 0, 1, 0],
    [1, 1, 0, 1],
    [1, 0, 1, 0],
    [1, 0, 1, 0],
]

#beginning with vertex 0
if graphColoringUtil(graph, color_nb, colors, 0):
    print colors
else:

The solution corresponds to the following assignments:

1st node : color 1
2nd node : color 2
3rd node : color 3

5. Explain all the techniques for binary trees

Answer: When the search necessarily involves the examination of every vertex in the object being searched it is called a traversal. There are many operations that we want to perform on binary trees. If we let L, D and R stand for moving left printing the data and moving right when at a node then there are six possible combinations of traversal: LDR, LRD, DLR, DRL, RDL and RLD. If we adopt the convention that we traverse left before right then only three traversals remain: LDR, LRD and DLR. To these we assign the names inorder, postorder and preorder.

Algorithm inorder(t)
{
    if t ≠ 0 then
        { 
            inorder(t->lchild); visit(t);
            inorder(t->rchild);
        }
}

Algorithm preorder(t)
{
    if t ≠ 0 then
        {

visit(t); preorder(t -> lchild); preorder(t -> rchild);
}

Algorithm post order(t)
{
if t ≠ 0 then
{
post order(t->child); post order(t->child); visit(t);
}
}
10 multiple choice questions with answers

1. Evaluate for the following A = TRUE, B = FALSE, C = FALSE
   i. R = NOT ( A OR B ) AND NOT (B OR C)
   ii. R = B AND NOT ( A OR C ) OR NOT (B AND C)
   A. i is true and ii is true
   B. i is true and ii is false
   C. i is false and ii is true
   D. i is false and ii is false

2. An employee came in to work and clocked in at Morning In, clocked out at NoonOut1 for lunch, clocked back in at Noon In, and clocked out to home at NoonOut2. Set up equation to calculate the number of hours worked for the day.
   A. WorkingHrs = (12 - (MorningIn+NoonOut1) + (NoonOut2-NoonIn))
   B. WorkingHrs = (12 – MorningIn + (NoonOut1-12.00) + (NoonOut2-NoonIn))
   C. WorkingHrs = (12 – MorningIn) + (NoonOut1-12.00)-(NoonOut2-NoonIn))
   D. WorkingHrs = (MorningIn+NoonIn) + (12.00-NoonOut2)
   Answer: - B

3. A large department store has its own charge card. The policy for a customer to charge an item is that the customer must have a valid charge card and either a balance of less than Rs.500 or a charge of less than Rs.50.
   A. Charge Card AND (Balance < 500 OR Amount < 50)
   B. Charge Card OR (Balance < 500 AND Amount < 50)
   C. Charge Card OR (Balance < 500 OR Amount < 50)
   D. Charge Card AND (Balance < 500 AND Amount < 50)

4. Consider the use of PAC for obtaining the solution for converting distance in Miles to Kilometres. The use of formula “Kilometres = 1.609* Miles” will be in
   A. given data section
   B. required result section
   C. processing required section
   D. solution alternative section

5. The PAC stands for
   A. Program Analysis Chart
   B. Problem Algorithm Code
   C. Problem Access Code
   D. Problem Analysis Chart

B N JYOTHI Assistant Professor
6. In interactivity chart the darkened circle indicates _______________.
A. duplicate module  B. loop  C. decision  D. no special meaning

7. In interactivity chart the diamond indicates _______________.
A. duplicate module  B. loop  C. decision  D. no special meaning

8. The interactivity chart is also known as _______________.
A. IPO Chart  B. Problem Analysis Chart  C. flow chart  D. structure chart

9. The IPO stands for
A. Input Programming Option  B. Input Programming Output  
C. Input Processing Output  D. Input Operating Operation

10. The difference between /, \ and MOD operator is
A. \ Integer Division, / Division and MOD Modulo Division  
B. / Division, \ escape sequence, MOD remainder
C. / Division, \ not an operator, MOD is module  
D. \ Division /Integer Division, MOD is Modulo Divisio
10 fill up the blanks with answers

1. A list of $n$ strings, each of length $n$, is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is
   Ans $O(n \log n)$

2. Which of the following case does not exist in complexity theory?
   Ans Null case

3. The concept of order Big O is important because
   Ans It can be used to decide the best algorithm that solves a given problem

4. The recurrence relation capturing the optimal execution time of the Towers of Hanoi problem with $n$ discs is
   Ans $T(n) = 2T(n - 1) + 1$

5. Which of the following sorting methods would be most suitable for sorting a list which is almost sorted?
   Ans Insertion Sort

6. Suppose we are sorting an array of eight integers using some quadratic sorting algorithm. After four iterations of the algorithm’s main loop, the array elements are ordered as shown here:
   2 4 5 7 8 1 3 6
   Ans Insertion sort

7. The running time of insertion sort is
   Ans $O(n^2)$

8. A sort which compares adjacent elements in a list and switches where necessary is ____.
   Ans insertion sort

9. The correct order of the efficiency of the following sorting algorithms according to their overall running time comparison is
   Ans bubble > selection > insertion

10. A sort which iteratively passes through a list to exchange the first element with any element less than it and then repeats with a new first element is called
    Ans insertion sort
<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>O (n log n)</td>
</tr>
<tr>
<td>2</td>
<td>Null case</td>
</tr>
<tr>
<td>3</td>
<td>It can be used to decide the best algorithm that solves a given problem</td>
</tr>
<tr>
<td>4</td>
<td>T(n) = 2T(n - 1) + 1</td>
</tr>
<tr>
<td>5</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>6</td>
<td>Insertion sort</td>
</tr>
<tr>
<td>7</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>8</td>
<td>insertion sort</td>
</tr>
<tr>
<td>9</td>
<td>bubble &gt; selection &gt; insertion</td>
</tr>
<tr>
<td>10</td>
<td>insertion sort</td>
</tr>
</tbody>
</table>
Unit IV

Two marks questions with answers

1. Define mode?
A mode is a value that occurs often in a given list of numbers. For example, the list is 5, 1, 5, 7, 6, 5, 7, 5, .. the mode is 5.

2. Define rotation?
A rotation in an AVL tree is a local transformation of its subtree rooted at a node, which is performed, when the balance factor of a node either +2 or -2. If an insertion or deletion of a new node in AVL Tree creates a tree with a violated balance requirement, then the tree is restructured by performing special transformation called rotation, that restore the balance required.

3. What are the different types of rotations? The four types of rotations are.
   • Right rotation
   • Left rotation
   • Double right-left rotation
   • Double left right rotation.

4. What are the drawbacks of AVL Tree?
   1) Frequent rotations are needed to maintain balances from the tree nodes.
   2) Deletion is difficult due to the frequency rotations.
   3) AVL tree is not considered as stranded structure for implementing dictionaries.

5. Define Heap
Heap is partially ordered data structure that is especially suitable for implementing priority queues. A heap is said to be a max heap, then the children of every node have a value less than that node. A heap is said to be a min heap, then the children of every node have a value greater than node.
Three marks questions with answers

1. What do you mean by optimal solution?

Given problem with inputs, we obtain subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. A feasible solution, which either maximizes or minimizes a given objective function is called optimal solution.

2. Define Floyd’s algorithm?

Floyd’s algorithm is an application, which is used to find all the pairs shortest paths problem. Floyd’s algorithm is applicable to both directed and undirected weighted graph, but they do not contain a cycle of a negative length.

3. Define Prim’s algorithm.

Prim’s algorithm is greedy and efficient algorithm, which is used to find the minimum spanning tree of weighted connected graph.

4. How efficient is Prim’s algorithm?

The efficiency of the Prim’s algorithm depends on data structure chosen for the graph.

5. What is path compression?

The better efficiency can be obtained by combining either variation of quick union with path compression. Path compression makes every node encountered during the execution of a find operation point to the tree’s node.
Five marks questions with answers

1. Write the Bellman and ford algorithm, and explain its concepts (10)

The recurrence for dist is given
\[ \text{dist}^k[u] = \min\{ \text{dist}^{k-1}[u], \min_i \{ \text{dist}^{k-1}[i] + \text{cost}[i,u] \} \} \]

Algorithm BellmanFord(v,cost,dist,n)

\{
  \text{for } I := 1 \text{ to } n \text{ do dist}[I] := \text{cost}[v,I]; \text{ for } k := 2 \text{ to } n-1 \text{ do }
  \text{for each } u \text{ such that } u \neq v \text{ and } u \text{ has at least one incoming edge do for each } \langle I,u \rangle \text{ in the graph do }
  \text{if dist}[u] > \text{dist}[I] + \text{cost}[I,u] \text{ then dist}[u] := \text{dist}[I] + \text{cost}[I,u];
\}

The overall complexity is \( O(n^3) \) when adjacency matrices are used and \( O(ne) \) when adjacency lists are used.

2. Discuss the concepts of 8-queens problem

The problem is to place eight queens on a 8 x 8 chessboard so that no two queen “attack” that is, so that no two of them are on the same row, column or on the diagonal.

One solution to the 8 queen problem
Algorithm Nqueens(k,n)
{
    for I:= 1 to n do
    {
        if( place(k,I) then
        {
            x[k]:= I;
            if(k=n) then write(x[1:n])
        }
        else
        Nqueens(k+1,n) }
    }
}

Algorithm place(k,I)
{
    else
}

Nqueens(k+1,n) } }
3. Write and explain the concepts of 0/1 knapsack problem.

A bag or sack is given capacity \( n \) and \( n \) objects are given. Each object has weight \( w_i \) and profit \( p_i \). Fraction of object is considered as \( x_i \) (i.e) \( 0 \leq x_i \leq 1 \). If fraction is 1 then entire object is put into sack. When we place this fraction into the sack we get \( w_i x_i \) and \( p_i x_i \). The solution to the knapsack problem can be viewed as a result of sequence of decisions. We have to decide the value of \( x_i \). \( x_i \) is restricted to have the value 0 or 1 and by using the function \( \text{knap}(l, j, y) \) we can represent the problem as

\[
\max \sum p_i x_i \text{subject to } \sum w_i x_i \leq y
\]

where \( l \) - iteration, \( j \) - number of objects, \( y \) - capacity.

The formula to calculate optimal solution is \( g_0(m) = \max \{g_1, g_1(m-w_1) + p_1\} \).

Algorithm BKnap(k, cp, cw)

{ 
if (cw + w[k] <= m) then 
{ 
y[k] := 1; 
if (k < n) then BKnap(k+1, cp + p[k], cw + w[k]) 
if ((cp + p[k] > fp) and (k = n)) then
{ 
fp := cp + p[k]; fw := cw + w[k]; for j := 1 to k do 
\( x[j] := y[j] \); 
}
}
if (bound (cp, cw, k) >= fp) then 
{ 
y[k] := 0; 
if (k < n) then BKnap(k+1, cp, cw) 
if ((cp > fp) and (k = n)) then
{ 

B N JYOTHI Assistant Professor
4. Write n-queen problem with Algorithm (4-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. For example, following is a solution for 4 Queen problem.

```
fp := cp; fw := cw; for j := 1 to k do
x[j] := y[j]; }
```

The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for above 4 queen solution.

```
{ 0,  1,  0,  0}
{ 0,  0,  0,  1}
{ 1,  0,  0,  0}
{ 0,  0,  1,  0}
```
5. Describe the Backtracking with M-COLORING Graph?

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

1) A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

2) An integer m which is maximum number of colors that can be used.

Output: An array color[V] that should have numbers from 1 to m. color[i] should represent the color assigned to the ith vertex. The code should also return false if the graph cannot be colored with m colors.

Algorithm:

If all colors are assigned,

   print vertex assigned colors

Else

   a. Trying all possible colors, assign a color to the vertex
b. If color assignment is possible, recusively assign colors to next vertices

c. If color assignment is not possible, de-assign color, return False

Code:

def is_safe(n, graph, colors, c):
    # Iterate through adjacent vertices
    # and check if the vertex color is different from c
    for i in xrange(n):
        if graph[n][i] and c == colors[i]: return False
    return True

# n = vertex nb

def graphColoringUtil(graph, color_nb, colors, n):
    # Check if all vertices are assigned a color
    if color_nb+1 == n:
        return True

    # Trying different colors for the vertex n
    for c in xrange(1, color_nb+1):
        # Check if assignment of color c to n is possible
        if is_safe(n, graph, colors, c):
# Assign color c to n

colors[n] = c

# Recursively assign colors to the rest of the vertices

if graphColoringUtil(graph, color_nb, colors, n+1): return True

# If there is no solution, remove color (BACKTRACK)

colors[n] = 0

We test the algorithm for the following graph and test whether it is 3 colorable:

(3)---(2)
|   / |
|  /  |
| /   |
|/    |

(0)---(1)

#nb of vertex
vertex_nb = 4

# nb of colors
color_nb = 3

# Initiate vertex colors

colors = [0] * vertex_nb

graph = ["
[0, 1, 1, 1],
[1, 0, 1, 0],
[1, 1, 0, 1],
[1, 0, 1, 0],

]  

#beginning with vertex 0
if graphColoringUtil(graph, color_nb, colors, 0):
    print colors
else:
    print "No solutions"
[1, 2, 3, 2]

The solution corresponds to the following assignments:
1st node : color 1
2nd node : color 2
3rd node : color 3
10 objective type questions with answers

1. When determining the efficiency of algorithm, the space factor is measured by
   a. Counting the maximum memory needed by the algorithm
   b. Counting the minimum memory needed by the algorithm
   c. Counting the average memory needed by the algorithm
   d. Counting the maximum disk space needed by the algorithm

2. The complexity of Bubble sort algorithm is
   a. O(n) b. O(log n) c. O(n^2) d. O(n log n)

3. Linked lists are best suited
   a. for relatively permanent collections of data
   b. for the size of the structure and the data in the structure are constantly changing
   c. for both of above situation
   d. for none of above situation

4. If the values of a variable in one module are indirectly changed by another module, this situation is called
   a. internal change b. inter-module change
   c. side effect d. side-module update

5. In linear search algorithm the Worst case occurs when
   a. The item is somewhere in the middle of the array
   b. The item is not in the array at all
   c. The item is the last element in the array
   d. The item is the last element in the array or is not there at all

6. For an algorithm the complexity of the average case is
   a. Much more complicated to analyze than that of worst case
   b. Much more simpler to analyze than that of worst case

B N JYOTHI Assistant Professor
c. Sometimes more complicated and some other times simpler than that of worst case
d. None or above

7. The complexity of merge sort algorithm is
a. O(n) b. O(log n)c. O(n^2) d. O(n log n)

8. The complexity of linear search algorithm is
a. O(n) b. O(log n) c. O(n^2) d. O(n log n)

9. When determining the efficiency of algorithm the time factor is measured by a. Counting micro seconds b. Counting the number of key operations c. Counting the number of statements d. Counting the kilobytes of algorithm

10. Which of the following data structure is linear data structure?
   a. Trees b. Graphs c. Arrays d. None of above

<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>D</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>C</td>
</tr>
</tbody>
</table>
10 fill in the blanks with answers

1. The three factors contributing to the sort efficiency considerations are the efficiency in coding, machine run time and the space requirement for running the procedure. 

2. How many passes are required to sort a file of size n by bubble sort method? 

3. How many number of comparisons are required in insertion sort to sort a file if the file is already sorted? 

4. The worst-case time complexity of Quick Sort is .

5. The worst-case time complexity of Bubble Sort is .

6. The algorithm like Quick sort does not require extra memory for carrying out the sorting procedure. This technique is called .

7. Which of the following sorting procedures is the slowest?

8. Two main measures for the efficiency of an algorithm are .

9. The time factor when determining the efficiency of algorithm is measured by .
<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td>2</td>
<td>N-1</td>
</tr>
<tr>
<td>3</td>
<td>N-1</td>
</tr>
<tr>
<td>4</td>
<td>O(n^2)</td>
</tr>
<tr>
<td>5</td>
<td>in-place</td>
</tr>
<tr>
<td>6</td>
<td>Bubble sort</td>
</tr>
<tr>
<td>7</td>
<td>Time and space</td>
</tr>
<tr>
<td>8</td>
<td>Counting the maximum memory needed by the algorithm</td>
</tr>
<tr>
<td>9</td>
<td>Counting the number of key operations</td>
</tr>
<tr>
<td>10</td>
<td>Counting the number of key operations</td>
</tr>
</tbody>
</table>
Unit V

Two marks questions with answers

1. Define backtracking?

Answer: Depth first node generation with bounding function is called backtracking. The backtracking algorithm has its virtue the ability to yield the answer with far fewer than m trials.

2. What is Hamiltonian cycle in an undirected graph?

Answer: A Hamiltonian cycle is round trip along n edges of G that visits every vertex once and returns to its starting position.

3. What is Feasible solution?

Answer: It is obtained from given n inputs Subsets that satisfies some constraints are called feasible solution. It is obtained based on some constraints

4. What is optimal solution?

Answer: It is obtained from feasible solution. Feasible solution that maximizes or minimizes a given objective function. It is obtained based on objective function.

5. Given an application for knapsack problem?

Answer: The knapsack problem is problem combinatorial optimization. It derives its name from the maximum problem of choosing possible essential that can fit too bag to be carried on trip. A similar problem very often appears business, combinatory, complexity theory, cryptography and applied mathematics.

Three marks questions with answers

B N JYOTHI Assistant Professor
1. **What is a minimum cost spanning tree?**

   **Answer:**
   A spanning tree of a connected graph is its connected acyclic subgraph that contains all vertices of a graph. A minimum spanning tree of a weighted connected graph is its spanning tree of the smallest weight where bweight of the tree is the sum of weights on all its edges.
   A minimum spanning subtree of a weighted graph \((G,w)\) is a spanning subtree of \(G\) of minimum weight \(w(T) = \sum w(e)\)
   \(e \in T\)

   Minimum Spanning Subtree Problem: Given a weighted connected undirected graph \((G,w)\), find a minimum spanning subtree

2. **Write the general procedure of dynamic programming.**

   **Answer:**
   The development of dynamic programming algorithm can be broken into a sequence of 4 steps.
   1. Characterize the structure of an optimal solution.
   2. Recursively define the value of the optimal solution.
   3. Compute the value of an optimal solution in the bottom-up fashion.
   4. Construct an optimal solution from the computed information.

   It states that an optimal solution to any of its instances must be made up of optimal solutions to its subinstances.

3. **Write some applications of traveling salesperson problem.**

   **Answer:**
   Routing a postal van to pick up mail from boxes located at \(n\) different sites.
   -> Using a robot arm to tighten the nuts on some piece of machinery on an assembly line.
   -> Production environment in which several commodities are manufactured on the same set of machines. Time complexity is \(O(n^2 2^n)\). Space complexity is \(O(n 2^n)\).
4. Write formula for bounding function in Knapsack problem

Answer:

In knapsack problem upper bound value is computed by the formula:

\[ UB = v + (W-w) \times (vi+1/wi+1) \]

25. Write about traveling salesperson problem.

Let \( g = (V, E) \) be a directed. The tour of \( G \) is a directed simple cycle that includes every vertex in \( V \). The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem is to find a tour of minimum cost. In branch and bound technique of TSP problem Lower bound \( lb = s/2 \)

5. State 8 Queens problem.

Answer:

The problem is to place eight queens on a 8 x 8 chessboard so that no two queen "attack" that is, so that no two of them are on the same row, column or on the diagonal.


Given \( n \) distinct positive numbers usually called as weights, the problem calls for finding all the combinations of these numbers whose sums are \( m \).
Five marks questions with answers

1. Write and explain the techniques in branch and bound method

Answer: The searching techniques that are commonly used in Branch-and-Bound method are:

- FIFO
- LIFO
- LC
- Heuristic search

Control Abstraction for LC – Search

In both LIFO and FIFO branch and bound the selection rule for the next E – node is rather rigid and in a sense blind. The selection rule for the next E – nodedoes not give any preference to a node that has a very good chance of getting the search to an answer not quickly.

The search for an answer node can often be speeded by using an intelligent ranking function for live node. The next E – node is selected on the basis of ranking function.

Let the a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the subtree with root x. c(t) is the cost of a minimum – cost answer in tree t. As remarked earlier, it is usually not possible to find an easily computable function c(). The heuristic search should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then c(x) = C(x).

Listnode = record{
  Listnode * next, *parent; Float coast;
}
Algorithm LCSearch(t)
{
    if *t is an answer node then output *t and return;
    E := t;
    Initialize the list of live nodes to be empty;
    Repeat
    {
        for each child x of E do
        {
        }
    }

2. Non deterministic Algorithm

    Answer :

    Algorithm has the property that the result of every operation whose outcome are not
    uniquely defined. The machine executing such operation is allowed to choose any one of
    these outcomes. To specify such algorithms, introduce three new functions.

    Choice (s) – Choose any one of the element of the set s. Failure
    () – Signals an unsuccessful completion
    Success () – Signals a successful completion.

    The assignment statement x: = choice(1, n) could result in x being assigned any one of the
    integer in the range[1, n]. The failure() and Success() signals are used to define a
    computation of the algorithm. The set of choices that leads to a successful completion,
    then one such set of choice is always made and the algorithm terminates successfully.

    Consider the problem of searching for an element x in a set of elements A[1, n] where n>=
Design and Analysis of Algorithms

1.

Step 1: \( J: = \text{choice} \ (1, \ n); \text{ Step} \)

2: If \( A[j] = x \) then

Write (j); Success()

; Step 3: \( A[j] \neq x \)

Write(0); Failure();

3. NP – hard Graph Problems

Answer :

1Pick a problem \( L_1 \) already known to be \( \text{NP – hard} \).

2Show how to obtain an instance of \( L_2 \) from any instance of \( L_1 \), such that from the solution of \( L_1 \) we can determine the solution to instance of \( L_1 \).

3Conclude from step 2 that \( L_1 \supset L_2 \).

4From step 1 and step 3, the transitivity of \( \supset \) that \( L_2 \) is \( \text{NP – hard} \)

4. Clique Decision Problem(CDP)

Answer : A maximal complete sub graph of a graph \( G = (V, E) \) is a clique. The size of the clique is the number of vertices in it. The Max clique problem is an optimization problem that has to determine the size of a largest clique in \( G \). The corresponding decision problem is to determine wheather \( G \) has a clique of size \( k \) for some \( k \).

The Input to the max clique decision problem can be provided as a sequence of edged and an integer \( k \). Each edge in \( E(G) \) is a pair of numbers \( (I, j) \). The size of the input for each edge \( (I, j) \) is \( \log I + \log j + 2 \). if a binary representation is assumed. The input size of any instance is
a) **Node Cover Decision problem**

A set \( V \) is a node cover for a graph \( G = (V, E) \) iff all edges in \( E \) are incident to at least one vertex in \( s \). The size \( |s| \) of the cover is the number of vertices in \( s \).

b) **Chromatic Number Decision Problem (CNDP)**

A coloring of a graph \( G = (V, E) \) is a function \( f: \{1, 2, \ldots, k\} \) defined for all \( x \in V \). If \( (V, E) \in E \), then \( f(u) \neq f(v) \).

c) **NP-hard scheduling problem**

This problem requires us to decide whether a given multiset \( A = \{a_1, a_2, a_3, \ldots, a_n\} \) of \( n \) positive integer has a partition \( P \) such that

d) **Job shopscheduling.**

A job shop like a flow shop has \( m \) different processors. The \( n \) job to be scheduled require the completion of several tasks. The time of the \( j \)th task for job \( j \). The task for any job \( j \) are to be carried out in the order 1, 2, 3, \ldots and so on.

3. **Approximation & \( \varepsilon \)-Approximation Algorithms**

Answer:

- \( A \) is an absolute approximation algorithm for problem \( p \) iff for every instance \( I \) of \( p \),

\[
\begin{align*}
\frac{f^*(I)}{f(I)} & \leq \varepsilon \quad \text{for some constant } k. \\
\end{align*}
\]

\( A \) is an \( \varepsilon \)-approximate algorithm iff for every instance \( I \) of size \( n \),

\[
\begin{align*}
\frac{f^*(I)}{f(I)} & \leq \varepsilon \quad \text{for every instance } I \\
anf(n) & \leq \varepsilon \\
\end{align*}
\]

\( B. \ N. \ JYOTHI \ Assistant \ Professor \)
An $\varepsilon$-Approximation algorithm is an $f(n)$ approximate algorithm for which $f(n) \leq \varepsilon$ for some constant $\varepsilon$.

A(e) is an approximation scheme iff for every given $\varepsilon > 0$ and problem instance I, A(e)

$\text{f}^*(I) - f(I)$

generate

$\alpha$

feasible

such

that

$f^*(I) \leq f(I) - f(I)$

$\leq \varepsilon$

$e$

$w$

$h$

$e$

$e$

$f^*$

$(I)$

$> 0$

$\text{f}^*(I)$

An approximation scheme is a polynomial time approximation scheme iff
for every fixed \( e > 0 \) it has a computing time ie, a polynomial in the 
problem size.

- An approximation scheme whose computing time is a polynomial both in 
the problem size \( s \) in \( 1/e \) is a fully polynomial time approximationscheme.

Maximum program stored problem

Assumewehaven programs and two storage device. Let \( l_i \) be the 
amount of storage needed to store the \( i^{th} \) program. Let \( L \) be the storage 
capacity of each disk. Determine the maximum number of these \( n \) programs 
that can be stored on two disks is NP-hard

i) Partition of maximum program

ii) Let \( I \) be any instance of the maximum programs stored problem. Let \( f^*(I) \) be 
the maximum number of programs that can be stored on two disks each of 
length \( L \).
1. The elements of an array are stored successively in memory cells because
   a. by this way computer can keep track only the address of the first element
      and the addresses of other elements can be calculated
   b. the architecture of computer memory does not allow arrays to store other than serially
   c. both of above
   d. none of above

2. Which of the following data structure is not linear data structure?
   a. Arrays
   b. Linked lists
   c. Both of above
   d. None of above

3. The Average case occur in linear search algorithm
   a. When Item is somewhere in the middle of the array
   b. When Item is not in the array at all
   c. When Item is the last element in the array
   d. When Item is the last element in the array or is not there at all

4. Two main measures for the efficiency of an algorithm are
   a. Processor and memory
   b. Complexity and capacity
   c. Time and space
   d. Data and space

5. Finding the location of the element with a given value is:
a. Traversal  
b. Search  
c. Sort  
d. None of above

6. Which of the following case does not exist in complexity theory  
a. Best case  
b. Worst case  
c. Average case  
d. Null case

7. The operation of processing each element in the list is known as  
a. Sorting  
b. Merging  
c. Inserting  
d. Traversal

8. Arrays are best data structures  
a. for relatively permanent collections of data  
b. for the size of the structure and the data in the structure are constantly changing  
c. for both of above situation  
d. for none of above situation

9. Each array declaration need not give, implicitly or explicitly, the information about  
a. the name of array  
b. the data type of array  
c. the first data from the set to be stored  
d. the index set of the array
10. The complexity of Binary search algorithm is
a. O(n)
b. O(log n)
c. O(n^2)
d. O(n log n)

<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
</tr>
<tr>
<td>9</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>B</td>
</tr>
</tbody>
</table>
1. The help menus or user manuals are the part of ______________.

2. The main measure for efficiency algorithm are ____________.

3. What does the algorithmic analysis count ________________?

4. Examples of O(1) algorithms are ________________.

5. There are four algorithms A1, A2, A3, A4 to solve the given problem with the order log(n), nlog(n), log(log(n))n/log(n), Which is the best algorithm. ____________

6. Express the formula (n-1)*(n-5) in terms of big Oh notation ________________.

7. The time complexity of binary search is ____________.

8. The time complexity of linear search is ____________.

9. In quick sort, the number of partitions into which the file of size n is divided by a selected record is ________________.

10. A sort technique is said to be stable when the original relative order of records with equal keys are retained after sorting. ________________.

B. N. JYOTHI Assistant Professor
<table>
<thead>
<tr>
<th>Question no</th>
<th>Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>External Documentation</td>
</tr>
<tr>
<td>2</td>
<td>Time and space</td>
</tr>
<tr>
<td>3</td>
<td>The number of arithmetic and the operations that are required to run the program</td>
</tr>
<tr>
<td>4</td>
<td>Multiplying two numbers assigning some value to a variable displaying some integer on console</td>
</tr>
<tr>
<td>5</td>
<td>A3</td>
</tr>
<tr>
<td>6</td>
<td>O(n2)</td>
</tr>
<tr>
<td>7</td>
<td>O(log n)</td>
</tr>
<tr>
<td>8</td>
<td>O(n)</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>True</td>
</tr>
</tbody>
</table>
17. BEYOND THE SYLLUBUS TOPICS MATERIAL

The study of ML algorithms has gained immense traction post the Harvard Business Review article terming a ‘Data Scientist’ as the ‘Sexiest job of the 21st century’. So, for those starting out in the field of ML, we decided to do a reboot of our immensely popular Gold blog The 10 Algorithms Machine Learning Engineers need to know - albeit this post is targeted towards beginners.

ML algorithms are those that can learn from data and improve from experience, without human intervention. Learning tasks may include learning the function that maps the input to the output, learning the hidden structure in unlabeled data; or ‘instance-based learning’, where a class label is produced for a new instance by comparing the new instance (row) to instances from the training data, which were stored in memory. ‘Instance-based learning’ does not create an abstraction from specific instances.

II. Types of ML algorithms

There are 3 types of ML algorithms:
1. Supervised learning:
Supervised learning can be explained as follows: use labeled training data to learn the mapping function from the input variables (X) to the output variable (Y).

\[ Y = f (X) \]

Supervised learning problems can be of two types:

a. Classification: To predict the outcome of a given sample where the output variable is in the form of categories. Examples include labels such as male and female, sick and healthy.
b. Regression: To predict the outcome of a given sample where the output variable is in the form of real values. Examples include real-valued labels denoting the amount of rainfall, the height of a person.

The 1st 5 algorithms that we cover in this blog— Linear Regression, Logistic Regression, CART, Naïve Bayes, KNN are examples of supervised learning.
Ensembling is a type of supervised learning. It means combining the predictions of multiple different weak ML models to predict on a new sample. Algorithms 9-10 that we cover– Bagging with Random Forests, Boosting with XGBoost are examples of ensemble techniques.

2. Unsupervised learning:
Unsupervised learning problems possess only the input variables (X) but no corresponding output variables. It uses unlabeled training data to model the underlying structure of the data.

Unsupervised learning problems can be of two types:

a. Association: To discover the probability of the co-occurrence of items in a collection. It is extensively used in market-basket analysis. Example: If a customer purchases bread, he is 80% likely to also purchase eggs.
b. Clustering: To group samples such that objects within the same cluster are more similar to each other than to the objects from another cluster.
c. Dimensionality Reduction: True to its name, Dimensionality Reduction means reducing the number of variables of a dataset while ensuring that important information is still conveyed. Dimensionality Reduction can be done using Feature Extraction methods and Feature Selection methods. Feature Selection selects a subset of the original variables. Feature Extraction performs data transformation from a high-dimensional space to a low-dimensional space. Example: PCA algorithm is a Feature Extraction approach.

Algorithms 6-8 that we cover here - Apriori, K-means, PCA are examples of unsupervised learning.

3. Reinforcement learning:
Reinforcement learning is a type of machine learning algorithm that allows the agent to decide the best next action based on its current state, by learning behaviours that will maximize the reward.

Reinforcement algorithms usually learn optimal actions through trial and error. They are typically used in robotics – where a robot can learn to avoid collisions by receiving negative feedback after bumping into obstacles, and in video games – where trial and error
these rewards to understand the optimal state of game play and choose the next action.

III. Quantifying the popularity of ML algorithms

Survey papers such as these have quantified the 10 most popular data mining algorithms. However, such lists are subjective and as in the case of the quoted paper, the sample size of the polled participants is very narrow and consists of advanced practitioners of data mining. The persons polled were the winners of the ACM KDD Innovation Award, the IEEE ICDM Research Contributions Award; the Program Committee members of the KDD-06, ICDM’06 and SDM’06; and the 145 attendees of the ICDM’06.

The Top 10 algorithms in this blog are meant for beginners and are primarily those that I learnt from the ‘Data Warehousing and Mining’ (DWM) course during my Bachelor’s degree in Computer Engineering at the University of Mumbai. The DWM course is a great introduction to the field of ML algorithms. I have especially included the last 2 algorithms (ensemble methods) based on their prevalence to win Kaggle competitions. Hope you enjoy the article!

IV. Supervised learning algorithms

1. Linear Regression
In ML, we have a set of input variables \((x)\) that are used to determine the output variable \((y)\). A relationship exists between the input variables and the output variable. The goal of ML is to quantify this relationship.
Figure 1: Linear Regression is represented as a line in the form of \( y = a + bx \). Source

In Linear Regression, the relationship between the input variables \((x)\) and output variable \((y)\) is expressed as an equation of the form \( y = a + bx \). Thus, the goal of linear regression is to find out the values of coefficients \(a\) and \(b\). Here, \(a\) is the intercept and \(b\) is the slope of the line.

Figure 1 shows the plotted \(x\) and \(y\) values for a dataset. The goal is to fit a line that is nearest to most of the points. This would reduce the distance (‘error’) between the \(y\) value of a data point and the line.

2. Logistic Regression

Linear regression predictions are continuous values (rainfall in cm), logistic regression predictions are discrete values (whether a student passed/failed) after applying a transformation function.

Logistic regression is best suited for binary classification (datasets where \( y = 0 \) or \(1\), where \(1\) denotes the default class. Example: In predicting whether an event will occur or not, the event that it occurs is classified as 1. In predicting whether a person will be sick or not, the sick instances are denoted as 1). It is named after the transformation function used in it, called the logistic function \( h(x) = \frac{1}{1 + e^{-x}} \), which is an S-shaped curve.
In logistic regression, the output is in the form of probabilities of the default class (unlike linear regression, where the output is directly produced). As it is a probability, the output lies in the range of 0-1. The output (y-value) is generated by log transforming the x-value, using the logistic function \( h(x) = \frac{1}{1 + e^{-x}} \). A threshold is then applied to force this probability into a binary classification.

![The Logistic Function, \( h(x) = \frac{1}{1 + e^{-x}} \)](image)

Figure 2: Logistic Regression to determine if a tumour is malignant or benign. Classified as malignant if the probability \( h(x) \geq 0.5 \). Source

In Figure 2, to determine whether a tumour is malignant or not, the default variable is \( y=1 \) (tumour= malignant); the x variable could be a measurement of the tumour, such as the size of the tumour. As shown in the figure, the logistic function transforms the x-value of the various instances of the dataset, into the range of 0 to 1. If the probability crosses the threshold of 0.5 (shown by the horizontal line), the tumour is classified as malignant.

The logistic regression equation \( P(x) = \frac{e^{b0 + b1*x}}{1 + e^{b0 + b1*x}} \) can be transformed into \( \ln(p(x) / 1-p(x)) = b0 + b1*x \).

The goal of logistic regression is to use the training data to find the values of coefficients \( b0 \) and \( b1 \) such that it will minimize the error between the predicted outcome and the actual outcome. These coefficients are estimated using the technique of Maximum Likelihood Estimation.

B. N. JYOTHI Assistant Professor
3. CART

Classification and Regression Trees (CART) is an implementation of Decision Trees, among others such as ID3, C4.5.

The non-terminal nodes are the root node and the internal node. The terminal nodes are the leaf nodes. Each non-terminal node represents a single input variable (x) and a splitting point on that variable; the leaf nodes represent the output variable (y). The model is used as follows to make predictions: walk the splits of the tree to arrive at a leaf node and output the value present at the leaf node.

The decision tree in Figure 3 classifies whether a person will buy a sports car or a minivan depending on their age and marital status. If the person is over 30 years and is not married, we walk the tree as follows: ‘over 30 years?’ -> yes -> ‘married?’ -> no. Hence, the model outputs a sportscar.

Figure 3: Parts of a decision tree. Source

4. Naïve Bayes
To calculate the probability that an event will occur, given that another event has already occurred, we use Bayes’ Theorem. To calculate the probability of an outcome given the value of some variable, that is, to calculate the probability of a hypothesis(h) being true, given our prior knowledge(d), we use Bayes’ Theorem as follows:

\[ P(h|d) = \frac{P(d|h) \times P(h)}{P(d)} \]

where

- \( P(h|d) = \) Posterior probability. The probability of hypothesis h being true, given the data d, where \( P(h|d) = P(d1| h)* P(d2| h)*...*P(dn| h)* P(d) \)
- \( P(d|h) = \) Likelihood. The probability of data d given that the hypothesis h was true.
- \( P(h) = \) Class prior probability. The probability of hypothesis h being true (irrespective of the data)
- \( P(d) = \) Predictor prior probability. Probability of the data (irrespective of the hypothesis)

This algorithm is called ‘naive’ because it assumes that all the variables are independent of each other, which is a naive assumption to make in real-world examples.

---

![Weather Table](image)

Figure 4: Using Naive Bayes to predict the status of ‘play’ using the variable ‘weather’.

Using Figure 4 as an example, what is the outcome if 'weather='sunny'?

To determine the outcome play= ‘yes’ or ‘no’ given the value of variable weather=’sunny’, calculate \( P(\text{yes}|\text{sunny})\) and \( P(\text{no}|\text{sunny})\) and choose the outcome with higher probability.

\[-P(\text{yes}|\text{sunny})= \frac{(P(\text{sunny}|\text{yes}) \times P(\text{yes}))}{P(\text{sunny})}\]
\[
=(3/9 \times 9/14) / (5/14)
\]
\[
= 0.60
\]

\[
\Rightarrow P(\text{no} | \text{sunny}) = \frac{P(\text{sunny} | \text{no}) \times P(\text{no})}{P(\text{sunny})}
\]
\[
=(2/5 \times 5/14) / (5/14)
\]
\[
= 0.40
\]
Thus, if the weather = 'sunny', the outcome is play = 'yes'.

5. KNN
The k-nearest neighbor algorithm uses the entire dataset as the training set, rather than splitting the dataset into a training set and test set.

When an outcome is required for a new data instance, the KNN algorithm goes through the entire dataset to find the k-nearest instances to the new instance, or the k number of instances most similar to the new record, and then outputs the mean of the outcomes (for a regression problem) or the mode (most frequent class) for a classification problem. The value of k is user-specified.

The similarity between instances is calculated using measures such as Euclidean distance and Hamming distance.