Ahsanullah University of Science and Technology (AUST)

Department of Computer Science and Engineering

LABORATORY MANUAL

Course No. : CSE2208
Course Title: Algorithm Lab

For the students of 2nd Year, 2nd semester of
B.Sc. in Computer Science and Engineering program

Table of Contents

Course Outcomes

Course Outcomes for the course are -

1. Prepare different algorithms from scratch.
2. Apply appropriate algorithm concept to solve some well-known problems.
3. Discover various algorithmic strategies for prospective solutions.

Preferred Tools

1. Codeblocks
2. NetBeans

Reference Books

a. Introduction to Algorithms (3rd Edition)
    i. Thomas H. Cormen
    ii. Charles E. Leiserson
    iii. Ronald L. Rivest
    iv. Clifford Stein
b. Classic Data Structures
    i. Debasis Samanta

Administrative Policy of the Laboratory

1. Class assessment tasks must be performed by students individually, without help of others.
2. Viva for each program will be taken and considered as a performance.
3. Plagiarism is strictly forbidden.

Lab 1 - STL

Objective: Objective of this topic is to learn about Standard Template Library of C++. This knowledge will be required throughout the course to code easily.

Standard Template Library is a C++ library which contains some useful data structures like Stack, Queue, Vector, Priority Queue, Map etc. They come very handy while coding graph theory related codes.

Vector

Vectors are sequence container that can change size. Container is an object that holds data of same type. Sequence containers store elements strictly in linear sequence.

Vector stores elements in contiguous memory locations and enables direct access to any element using operator []. Unlike array, vector can shrink or expand as needed at run time. The storage of the vector is handled automatically.

Following is the way to declare vector type variable:
vector <DATA_TYPE> VARIABLE_NAME;

The vector operations that will be used frequently in this course are:
.push() - It is used to insert an element at the back of a vector.
.size() - It is used to find out the number of elements kept inside the vector.

To use vector we need to add **#include <vector>.**

Example: You are given a list of **N** numbers in a vector. Write a program to find out if summation of all odd numbers in the vector is **greater than** summation of all even numbers.

Sample Input	**Sample Output**
5 1 2 3 4 5 4 1 2 3 4	YES NO

Sample Code for Vector

```
#include <stdio.h>
#include <vector>

using namespace std;

int main() {
   int n;
   scanf("%d", &n);
   vector<int> v;

   for(int i = 0; i < n; ++i) {
      int a;
      scanf("%d", &a);
      v.push_back(a);
   }

   int odd = 0, even = 0;

   for(int i = 0; i < n; ++i) {
      odd += (v[i] % 2 == 0) ? 0 : v[i];
      even += (v[i] % 2 == 0) ? v[i] : 0;
   }

   if(odd > even) printf("YES\n");
   else printf("NO\n");
}
```

Map

Map is dictionary like data structure. It is a sequence of (key, value) pair, where only single value is associated with each unique key. It is often referred as associative array.

Following is the way to declare map type variable:
map <KEY_DATA_TYPE, VALUE_DATA_TYPE> VARIABLE_NAME;

An example of storing data in a **string, double** pair map is given below:
map <string, double> mp;
mp["ABC"] = 10.203;
mp["EFG"] = -5.360;
Now if we print mp["ABC"], then 10.203 will be printed.
And for mp["EFG"] it will be -5.360.
To use map we need to add **#include <map>.**

Example: You are given a list of **N** student names and their achieved numbers in final. You have to query a student name. If the student name and her/his number has already been taken as input then you will check if the number is >= 80. In that case print "A+". Otherwise print "Not A+". If the student's name has not been taken as input, print "NO STUDENT RECORD AVAILABLE". The sample IO is given below:

Sample Input	**Sample Output**
3 Tamim 50 Shakib 80 Mahmudullah 85 Mustafiz	NO STUDENT RECORD AVAILABLE

Sample Code for Map

```
#include <iostream>
#include <map>
using namespace std;

int main() {

  int n;
  scanf("%d", &n);

  map<string, double> mp;

  for(int i = 0; i < n; ++i) {
     string str;
     int a;
     cin >> str >> a;
     mp[str] = a;
  }

  string query;
  cin >> query;

  if(mp[query]) {
     if(mp[query] >= 80.0)
```

```
        cout << "A+" << endl;
      else
        cout << "NOT A+" << endl;
   }
   else {
      cout << "NO STUDENT RECORD AVAILABLE" << endl;
   }

   return 0;
}
```

Priority Queue

Priority queues are a type of container adaptors, specifically designed such that its **first element is always the greatest of the elements** it contains, according to some strict weak ordering criterion.

This context is **similar to a heap**, where elements can be inserted at any moment, and only the max heap element can be retrieved (the one at the top in the priority queue).

Priority queues are implemented as container adaptors, which are classes that use an encapsulated object of a specific container class as its underlying container, providing a specific set of member functions to access its elements. Elements are popped from the "back" of the specific container, which is known as the top of the priority queue.

Following is the way to declare priority queue type variable:
priority_queue <DATA_TYPE> VARIABLE_NAME;

Useful priority queue operations for this course are given below:
.empty() - Test whether container is empty
.top() - Access top element
.push() - Insert element
.pop() - Remove top element

To use priority queue we need to add **#include <queue>.**

Example: You are given a list of N numbers. You need to write a program such that the numbers will be stored in ascending order in a priority queue.

Sample Input	Sample Output
5 1 4 3 1 7	1 1 3 4 7

Sample Code for Priority Queue

```
#include <stdio.h>
#include <queue>

using namespace std;

int main() {
   int n;
   scanf("%d", &n);
   priority_queue<int> q;

   for(int i = 0; i < n; ++i) {
      int a;
      scanf("%d", &a);
      q.push(-a);
   }

   while(!q.empty()) {
      int u = q.top(); q.pop();
      printf("%d ", -u);
   }
}
```

Task:

1. Write a program to demonstrate usage of Vector.
2. Write a program to demonstrate usage of Map.
3. Write a program to demonstrate usage of Priority queue

Lab 2 - Quicksort and Merge sort

Objective: Objective of the topics is to learn about quick sort and merge sort algorithm and which one to use when. We will also learn the advantages and disadvantages of each sorting mechanism.

Quick Sort

Quick sort works in the following steps:

1. Finds a pivot (an arbitrary number)
2. Bring all numbers less than pivot to its left
3. Bring all numbers greater than pivot to its right
4. Keep doing that to left of pivot and right of pivot

The quick sort algorithm is given below:

Algorithm Quicksort(A, lo, hi):

1. **if** lo < hi **then**
2. p = **Partition**(A, lo, hi)
3. **Quicksort**(A, lo, p - 1)
4. **Quicksort**(A, p + 1, hi)

Algorithm Partition(A, lo, hi)

1. pivot = A[hi]
2. i = lo - 1
3. **for** j = lo **to** hi - 1 **do**
4. **if** A[j] < pivot **then**
5. i = i + 1
6. **swap** A[i] with A[j]
7. **swap** A[i + 1] with A[hi]
8. **return** i + 1

Time complexity of Quick sort is :

- **O(nlogn)** (Average Case)
- **O(n^2)** (Worst Case)

Example: A sample input output for sorting a vector of **N** numbers using quick sort.

Sample Input	**Sample Output**
5 4 1 7 -20 41	-20 1 4 7 41

Merge Sort

Merge sort works in the following steps:

1. Keep breaking down main array in half until they are just single numbers
2. Keep merging the broken parts and sort them while merging
3. Get the original array in sorted order

Time complexity of merge sort algorithm is: **O(nlogn)**

The merge sort algorithm is given below:

Algorithm Merge-Sort(A, lo, hi):

```
1. if lo < hi then
2.     mid = (lo + hi) / 2
3.     Merge-Sort(A, lo, mid)
4.     Merge-Sort(A, mid + 1, hi)
5.     Merge(A, lo, mid, hi)
```

Algorithm Merge(A, lo, mid, hi):

```
1.  n1 = mid - lo + 1
2.  n2 = hi - mid
3.  L[1..n1] = A[lo..mid]
4.  R[1..n2] = A[mid+1..hi]
5.  L[n1 + 1] = R[n2 + 1] = INF
6.  i = j = 0
7.  for k = lo to hi do
8.      if L[i] < R[j] then
9.          A[k] = L[i]
10.         i = i + 1
11.     else
12.         A[k] = R[j]
13.         j = j + 1
```

Example: A sample input output for sorting a vector of **N** numbers in **descending** order using merge sort.

Sample Input	**Sample Output**
5 11 747 11 -7 741	747 741 11 11 -7

Tasks:

1. Write a program to sort a list of numbers in ascending order using merge sort
2. Write a program to sort a list of numbers in ascending order using quicksort.

Lab 3 - Heap Sort

Objective: Objective of the topic is to learn about heap sort algorithm and advantage of using it.

Heap sort is a sorting algorithm that runs as fast as Merge sort but doesn't take any extra space. It can be used using Max or Min heapify. Time complexity for heap sort is **O(nlogn).**

Heap sort algorithm is given below:

Algorithm MAX-HEAPIFY (A, i):

```
1.  l = LEFT(i)
2.  r = RIGHT(i)
3.  if l <= A.Heap-Size and A[l] > A[i]
4.      largest = l
5.  else
6.      largest = i;
7.  if r <= A.Heap-Size and A[r] > A[i]
8.      largest = r
9.  if largest != i
10.     exchange A[i] with A[largest]
11.     MAX-HEAPIFY(A, largest)
```

Algorithm BUILD-MAX-HEAP (A):

```
1.  A.heap-size = A.length
2.  for i = floor(A.length / 2) downto 1
3.      MAX-HEAPIFY(A, i)
```

Algorithm HEAPSORT (A):

```
1.  BUILD-MAX-HEAP(A)
2.  for i = A.length downto 2
3.      exchange A[1] with A[i]
4.      A.heap-size = A.heap-size - 1
5.      MAX-HEAPIFY(A, 1)
```

Example: A sample input output for sorting a vector of **N** numbers using Heap sort.

Sample Input	**Sample Output**
5 5 1 -41 2 3	-41 1 2 3 5

Note:

- Time complexity of Heap sort is : O(nlogn)

Tasks:

1. Write a program to sort a list of numbers in ascending order using heap sort

Lab 4 - DFS, BFS

Objective: Objective of this class is to learn about traversal algorithms DFS and BFS. These 2 are one of the basic graph algorithms which will also help getting familiar with graph input and handling adjacency list.

The Depth First Search algorithm is given below:

Color = 1D Array of size **|V|**
Adj = 2D List of **|V|** rows

Algorithm: DFS(u)

```
1. Color[u] = GREY
2. for each v in Adj[u]
3.     if Color[v] = WHITE then
4.         do DFS(v)
5. Color[v] = BLACK
```

The Breadth First Search algorithm is given below:
Color = 1D Array of size **|V|**
Adj = 2D List of **|V|** rows

Algorithm: BFS(s)

```
1. Color[s] = 1
2. ENQUEUE(Q, s)
3. while Q != EMPTY
4.     u  = DEQUEUE(Q)
5.     for each v in Adj[u]
6.         if Color[v] = 0 then
7.             do ENQUEUE(Q, v)
8.                 Color[v] = 1
```

Example: You are given graph containing V vertices and E edges. You need to find the DFS and BFS traversal for the graph. First line will contain V and E. Next E lines will contain the edges.

Sample Input	**Sample Output**
5 4 1 2 2 3 3 4 4 5	DFS: 5 4 3 2 1 BFS: 1 2 3 4 5

Notes:

- Time complexity of DFS is O(V + E)
- Time complexity of BFS is O(V + E)

Tasks:

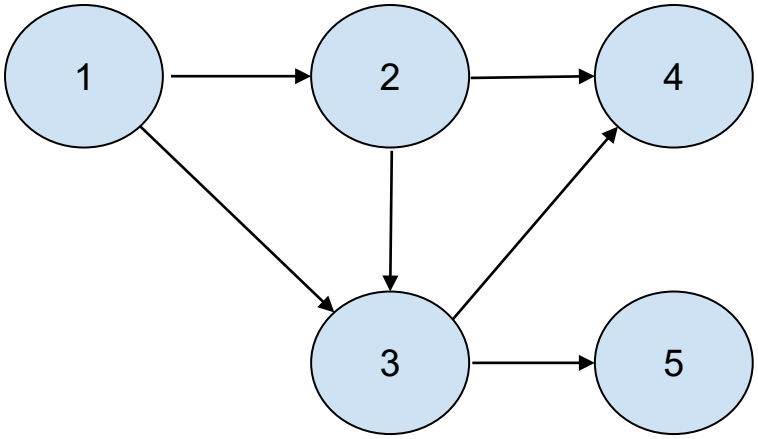
1. Given a graph write a program to print the DFS traversal output.
2. Given a graph write a program to print the BFS traversal output.

Lab 5 - Topological Sort, Strongly Connected Component

Topological Sort

Objective: Suppose we are given a list of tasks where to work on a task **b** we need another task **a** to be completed before. We need to order the tasks in such a way that no **dependent task** will come before the task(s) on which it is dependent. To solve this type of problems, we need to learn **Topological Sorting**.

Topological Sorting of vertices of a **Directed Acyclic Graph** is an ordering of the vertices $v_1, v_2, \ldots, v_n$ in such a way, that if there is an edge directed towards vertex v_j from vertex v_i, then v_i comes before v_j. For example consider the graph given below:



To find out the topological sort order, we can use the following algorithm:

Algorithm: TOPOLOGICAL-SORT(G)

1. Call DFS(G) to compute finishing times f [v] for each vertex v
2. As each vertex is finished, insert it onto the front of a linked list or push in a stack
3. Return the linked list of vertices or return the stack

Notes:

- A topological sort for the above graph is: **1 2 3 4 5**
- Multiple topological sort is possible for a single graph
- We can perform a topological sort in time Θ**(V + E)**, DFS search takes Θ**(V + E)** time and it takes O(1) time to insert each of the |V| vertices onto the front of the linked list or to push in the stack.

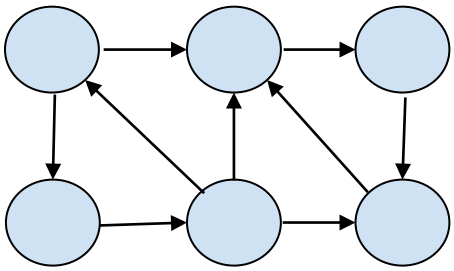
Task:

1. Given a graph where some nodes are dependent on some other nodes, find a topologically sorted order.

Strongly Connected Component (SCC)

Objective: From a directed graph, objective of Strongly Connected Component algorithm is to find such subgraphs where all nodes are connected to one other.

Strongly Connected Component of a directed graph G = (V, E) is a **maximal set** of vertices C ⊆ V such that for every pair of vertices u and v in C, we have both u ->...-> v and v ->...-> u; that is, vertices u and v are reachable from each other. Consider the given graph below (with starting and finishing time for each of the nodes):



To find out the strongly connected components, we can use the following algorithm:

Algorithm: STRONGLY-CONNECTED-COMPONENT(G)

1. Call **DFS(G)** to compute finishing times **f[u]** for each vertex u
2. Compute G^T
3. Call DFS(G^T), but in the main loop of DFS, consider the vertices **in order of decreasing f[u]** (as computed in line 1)

4. Output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

Example - SCC: You are given a directed graph containing **V** vertices and **E** edges. The first line of the input will contain **V** and **E**. The next **E** lines will contain 2 numbers. The endpoints. The sample input output is given below:

Sample Input	**Sample Output**
6 9 1 2 2 3 3 1 1 6 3 6 6 4 4 5 5 6 3 5	2 1 2 3 4 5 6

Notes:

- According to the algorithm - the SCCs for the Graph mentioned above are: **[1 ,2 ,3]** and **[6, 5, 4]**
- $G^T = (V, E^T)$, where $E^T = \{(u, v) : (v, u) \in E\}$. That is, G^T consists of the edges of G with their directions reversed.
- Time to create G^T is O(V + E)

Task:

1. Find out the number of SCCs of a directed graph. Print all the SCCs.

Lab 6 - Minimum Spanning Tree (Kruskal)

Objective: Suppose we are given a set of nodes which **can be connected** by some edges. It takes certain amount of cost to create the edges. Our objective is to reduce the cost by creating only the edges that are necessary. We also need to minimize the total cost of weight. To solve this type of problems, we need Minimum Spanning Tree algorithm. This topic covers **MST Kruskal.**

What is a Spanning Tree?
Given an **undirected and connected** graph **G**=(V,E), a spanning tree of the graph **G** is a tree that spans **G** (that is, it **includes every vertex** of **G**) and **is a subgraph** of **G** (every edge in the tree belongs to **G**)

What is a Minimum Spanning Tree?
The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Kruskal Algorithm uses the following algorithm to find MST of a graph:

Algorithm: Kruskal (G)

1. Sort the edges of G with respect to their weights.
2. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
3. Only add edges which does not form a cycle , edges which connect only disconnected components.

Example: You are given an undirected weighted graph. You have to find the MST using Kruskal algorithm. First line of the input will contain the number of vertices **V** and the number of edges **E**. The next **E** lines will contain the endpoints of each edge and their weight. The output will only contain the edges of the MST.

Sample Input	**Sample Output**
5 7 1 2 1 1 3 7 2 3 5	1 2 1 2 3 5 2 5 3 4 5 2

2 4 4 2 5 3 4 5 2 3 5 6	

Note:

- In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be O(ElogV), which is the overall Time Complexity of the algorithm.

Tasks:

1. Find MST of a given graph using Kruskal Algorithm

Lab 8 - Minimum Spanning Tree (Prim's)

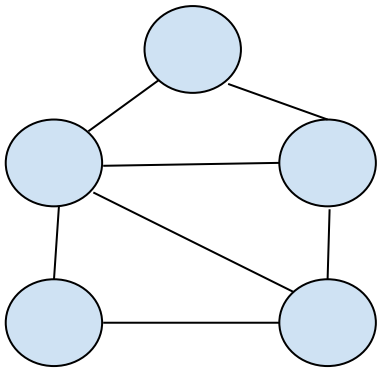
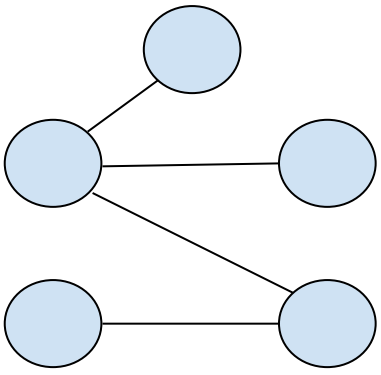
Objective: Objective of this topic is to learn Prim's algorithm to find MST of a graph.

Prim's Algorithm uses the following algorithm to find MST of a graph:

Algorithm: Prim (G, w, r)

```
1.  for each u ϵV[G]
2.        do key[u] ←∞
3.           Π[u]  ←NIL
4.  key[r] ←0
5.  Q ←V[G]
6.  while Q != ϕ
7.     do u ←EXTRACT_MIN(Q)
8.        for each v ϵAdj[u]
9.           do if v ϵQ and w(u, v) < key[v]
10.             then Π[v]  ←u
11.                  key[v] ← w(u, v)
```

Here's a MST algorithms simulation on a sample graph -

Sample Input	**Sample Output**
	

Notes:

- The time complexity of the Prim's Algorithm is O((V+E)logV) because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

Tasks:

1. Find MST of a given graph using Prim's Algorithm

Lab 9 - Dijkstra Algorithm

Objective: The objective of single source shortest path problem is to find paths between source and all vertices in a graph such that the total distance is minimum. This topic covers Dijkstra algorithm for finding shortest path.

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which **all edge weights are nonnegative**. In this section, therefore, we assume that w(u, v) ≥ 0 for each edge (u, v) ∈ E

Algorithm: DIJKSTRA(G, w, s)

1. INITIALIZE-SINGLE-SOURCE(G,s)
2. S ← ϕ
3. Q ←V[G]
4. **while** Q != ϕ
5. **do** u ←EXTRACT-MIN(Q)
6. S ←S U {u}
7. **for** each vertex v ∈ Adj[u]
8. **do** RELAX(u, v, w)

Example: You are given a directed weighted graph with **V** vertices and **E** edges. Each edge will have the end points and the weight. First line of the input will contain **V** and **E**. The next **E** lines will contain the edges. You need to find the distance of all nodes from Node 1 using **Dijkstra** algorithm. The **Dijkstra** algorithm output for a sample input graph is given below:

Sample Input	**Sample Output**
3 3 1 2 10 1 3 60 2 3 40	1 0 2 10 3 50

Note:

- Time Complexity of Dijkstra's Algorithm is O(V^2) but with min-priority queue it drops down to O(V+ElogV).

Task:

1. Find out single source shortest path for a directed graph where nodes are non-negative.

Lab 10 - Bellman-Ford Algorithm

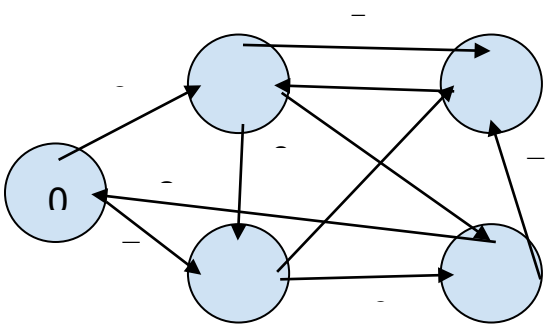
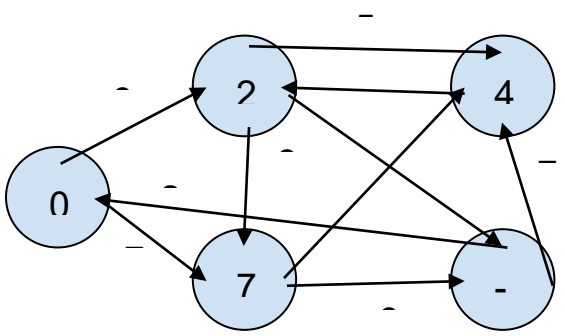
Objective: The objective of single source shortest path problem is to find paths between source and all vertices in a graph such that the total distance is minimum. In this topic, we will be using BELLMAN-FORD algorithm.

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which **edge weights may be negative**.

Algorithm: BELLMAN-FORD(G, w,s)

1. INITIALIZE-SINGLE-SOURCE(G,s)
2. **for** i ← 1 to |V[G]| − 1
3. **do** for each edge (u, v) ∈ E[G]
4. **do** RELAX(u, v, w)
5. **for** each edge (u, v) ∈ E[G]
6. **do** if d[v] > d[u] + w(u, v)
7. **then** return FALSE
8. **return** TRUE

Here's a Bellman-Ford Algorithm Simulation on a sample graph:

Sample Input	**Sample Output**
	

Notes:

- The Bellman-Ford algorithm runs in time O(VE)

Task:

1. Find out single source shortest path for a directed graph where nodes can be negative.

Lab 11 - Floyd-Warshall Algorithm

Objective: The objective of all pair shortest path problem is to find paths between all pairs of vertices in a graph such that the total distance is minimum.

Algorithm: FLOYD-WARSHALL(W)

1. n ← rows[W]
2. D(0) ← W
3. **for** k ← 1 to n
4. **do for** i ← 1 to n
5. **do for** j ← 1 to n
6. **do** $d_{ij}^{k} \leftarrow \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$
7. return D(n)

Example: You are given a directed weighted graph with **V** vertices and **E** edges. Each edge will have the end points and the weight. First line of the input will contain **V** and **E**. The next **E** lines will contain the edges. You need to find the distance of all nodes from every other nodes using **Floyd Warshall** algorithm. The **Floyd Warshall** algorithm output for a sample input graph is given below:

Sample Input	**Sample Output**
3 3 1 2 10 1 3 60 2 3 40	1 1 0 1 2 10 1 3 50 2 1 INF 2 2 0 2 3 40 3 1 INF 3 2 INF 3 3 0

Note:

- Time Complexity of Floyd Warshall Algorithm is O(V^3)

Task:

1. Find out all pair shortest path of a graph.

Lab 11 - Greedy Approach

Objective: Objective of this topic is to learn about Greedy approach and when to use it.

A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

How do you decide which choice is optimal?
Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

Example - Activity Selection Problem: You are given a list of **N** tasks with their starting **Si** and ending time **Ei,** where **1 <= i <= N**. You need to find out the maximum number of tasks you can complete such that no 2 tasks overlap. Given below is a sample input and output which is generated using Greedy algorithm for **Activity Selection problem**.

Sample Input	**Sample Output**
5 1 5 2 3 4 7 4 6 7 10	3

Example - Fractional Knapsack Problem: You are given **N** items and a Capacity **CAP**. Item i (1 <= i <= N) has **Wi** weight and **Ci** cost. You need to find out the maximum cost you can make such that total weight is less than **CAP**. You can take an item partially. A sample input for the problem and output using Greedy approach is given below:

Sample Input	**Sample Output**
3 5 3 4 7 7 7 6	7.5

Task:

- Write a program to demonstrate activity selection problem solution

Lab 12 - Dynamic Programming and Backtracking

Objective: Objective of this topic is to learn about DP and backtracking approach and why are they necessary.

Dynamic Programming Approach

Dynamic programming is basically, recursion plus common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Example - Longest Common Subsequence (LCS): You are given 2 strings. You need to find a longest common subsequence of them. A sample input output using Dynamic Programming is given below:

Sample Input	**Sample Output**
ABCEFGH ABHCEHFH	ABCEFH

Example - 0-1 Knapsack Problem: You are given **N** items and a Capacity **CAP**. Item i (1 <= i <= N) has **Wi** weight and **Ci** cost. You need to find out the maximum cost you can make such that total weight is less than **CAP**. A sample input for the problem and output using Dynamic programming is given below:

Sample Input	**Sample Output**

3 10 6 6 5 5 3 3	9

Backtracking

Backtracking is an algorithm for capturing some or all solutions to given computational issues, especially for constraint satisfaction issues. The algorithm can only be used for problems which can accept the concept of a "partial candidate solution" and allows a quick test to see if the candidate solution can be a complete solution. Backtracking is considered an important technique to solve constraint satisfaction issues and puzzles. It is also considered a great technique for parsing and also forms the basis of many logic programming languages.

Example: You are given an **NxN** chessboard. The task is to place **N** queens on that board such that no 2 queens attack each other. A sample valid output using Backtrack algorithm on a 4x4 chessboard is given below -

	Q		
			Q
Q			
		Q	

Task:

- Write a program to demonstrate 0-1 Knapsack problem solution
- Write a program to demonstrate N-Queens problem using Backtracking

Lab 7 - Midterm Examination

There will be a 20 marks examination containing algorithm problems, simulations and multiple choice questions.

Lab 13 - Lab Final Examination

There will be a 20 marks examination containing algorithm problems, simulations and multiple choice questions.