DAA - Lab Programs C++

Min Max

```cpp
#include <iostream>
#include <vector>
#include <climits>
using namespace std;

int mini, maxi;

void minMax(vector<int> arr, int i, int j) {
	if(i == j) {
		mini = arr[i];
		maxi = arr[i];
	}

	else if(i == j - 1) {
		maxi = max(arr[i], arr[j]);
		mini = min(arr[i], arr[j]);
	} else {
		int mid = (i + j) / 2;

		int max1 = maxi;
		int min1 = mini;

		minMax(arr, i, mid);
		minMax(arr, mid + 1, j);

		maxi = max(max1, maxi);
		mini = min(min1, mini);
	}
}

int main() {

	int n;
	cout << "Enter no of elements in the array: ";
	cin >> n;

	vector<int> arr(n);
	for(int i = 0; i < n; i++) {
		cin >> arr[i];
	}
```

```
      mini = INT_MAX;
      maxi = INT_MIN;

      int i = 0;
      int j = arr.size() - 1;

      minMax(arr, i , j);

      cout << "Maximum: " << maxi << endl;
      cout << "Minimum: " << mini << endl;
}
```

Knapsack

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

class Item {
public:
    int id;
    int weight;
    int value;

    Item(int i, int wt, int val) {
        this -> id = i;
        this -> weight = wt;
        this -> value = val;
    }
};

bool compare(Item* a, Item* b) {
    int pw_ratio_a = (double)a -> value / a -> weight;
    int pw_ratio_b = (double)b -> value / b -> weight;

    return pw_ratio_a > pw_ratio_b;
}

int main()
{
    int capacity;
```

```
    cin >> capacity;

    int n; // no of items in the knapsack
    cin >> n;

    vector<Item*> items;
    for(int i = 0; i < n; i++) {
        int wt, val;
        cout << "Enter weight and profit of " << i + 1 << "th elemet: ";
        cin >> wt >> val;

        Item* item = new Item(i, wt, val);
        items.push_back(item);
    }

    sort(items.begin(), items.end(), compare);

    vector<double> solution(n, 0);
    double total_profit = 0;
    for(int i = 0; i < n; i++) {
        Item* item = items[i];
        if(capacity - item -> weight >= 0) {
            capacity -= item -> weight;
            solution[item->id] = 1;
            total_profit += item -> value;
        } else {
            solution[item->id] = (double)capacity / item -> weight;
            total_profit += item -> value * solution[item->id];
            capacity = 0;
        }

        if(capacity <= 0) {
            break;        }
    }

    for(int i = 0; i < n; i++) {
        cout << solution[i] << " ";
    } cout << endl;
    cout << "Total profit: " << total_profit << endl;

    return 0;
}
```

Optimal merge pattern

```
#include <iostream>
#include <vector>

using namespace std;

class MinHeap {
	vector<int> nums;

	public:

	int parent(int i) { return (i-1)/2; }
	int lchild(int i) { return 2 * i + 1; }
	int rchild(int i) { return 2 * i + 2; }
	int size() { return nums.size(); }

	void push(int num) {
		nums.push_back(num);
		int i = size() - 1;
		percolateUp(i);
	}

	void percolateUp(int i) {
		if(i <= 0) {
			return;
		}

		int p = parent(i);
		if(nums[p] > nums[i]) {
			swap(nums[p], nums[i]);
			percolateUp(p);
		}
	}

	int top() {
		return size() == 0 ? -1 : nums[0];
	}

	void pop() {
		if(size() == 1) {
			nums.pop_back();
		} else {
			swap(nums[0], nums[size() - 1]);
```

```
                nums.pop_back();
                percolateDown(0);
            }
        }

        void percolateDown(int i) {
            int lc = lchild(i);
            int rc = rchild(i);

            int imin = i;

            if(lc < size() && nums[lc] < nums[imin]) {
                imin = lc;
            }

            if(rc < size() && nums[rc] < nums[imin]) {
                imin = rc;
            }

            if(i != imin) {
                swap(nums[i], nums[imin]);
                percolateDown(imin);
            }
        }
};

int optimal_merge(int n, vector<int> nums) {
        MinHeap minHeap;
        for(int num : nums) {
            minHeap.push(num);
        }

        int totalMergeCost = 0;
        while(minHeap.size() > 1) {
            int record1 = minHeap.top();
            minHeap.pop();

            int record2 = minHeap.top();
            minHeap.pop();

            int mergeCost = record1 + record2;
            totalMergeCost += mergeCost;

            cout << "Cost of merging " << record1 << " and " << record2 << "
is " << mergeCost << endl;

            minHeap.push(mergeCost);
        }
```

```
	return totalMergeCost;
}

int main() {
	int n;
	cout << "Enter no of records: ";
	cin >> n;

	vector<int> nums(n);
	cout << "Enter sizes of each record: ";
	for(int i = 0; i < n; i++) {
		cin >> nums[i];
	}

	int optimal_merge_value = optimal_merge(n, nums);

	cout << optimal_merge_value << endl;

	return 0;
}
```

Bellman-Ford

```
#include <iostream>
#include <climits>
#include <vector>
using namespace std;

class Edge {
public:
  int u, v, w;
};

class Graph {
public:
  int vertices;
  int edges;
  vector<Edge*> edge;
};

void display(vector<int> arr) {
  for(int i = 0; i <arr.size(); i++) {
      cout << arr[i] << " ";
  }
```

```
  cout << endl;
}

void bellmanford(Graph *g, int source) {
  int i, j, u, v, w;

  int noOfVertices = g->vertices;
  int noOfEdges = g->edges;

  vector<int> distance(noOfVertices, INT_MAX);
  vector<int> parent(noOfVertices, 0);


  distance[source] = 0;
  for(i = 1; i <= noOfVertices; i++) {
      for(j = 0; j < noOfEdges; j++) {
          u = g->edge[j]->u;
          v = g->edge[j]->v;
          w = g->edge[j]->w;

          // relaxation
          if(distance[u] != INT_MAX && distance[v] > distance[u] + w) {
              distance[v] = distance[u] + w;
              parent[v] = u;
          }

      }
  }

  for(j = 0; j < noOfEdges; j++) {
      u = g->edge[j]->u;
      v = g->edge[j]->v;
      w = g->edge[j]->w;

      // relaxation
      if(distance[u] != INT_MAX && distance[v] > distance[u] + w) {
          cout << "\nNegativie cycle exists!" << endl;
          return;
      }
  }

  cout << "Distance array: ";
  display(distance);

  cout << "Predecessor array: ";
  display(parent);
}
```

```
int main()
{
    // create Graph
    Graph *g = new Graph();

    cout << "Enter no of vertices/nodes: ";
    cin >> g->vertices;

    cout << "Enter no of edges: ";
    cin >> g->edges;

    cout << "Enter edges (from to weight): " << endl;
    g->edge.resize(g->edges);

    for(int i = 0; i < g->edges; i++) {
        g->edge[i] = new Edge();
        cin >> g->edge[i]->u;
        cin >> g->edge[i]->v;
        cin >> g->edge[i]->w;
    }

    bellmanford(g, 0);

    return 0;
}

/*
 * Sample Output:

Enter no of vertices/nodes: 7
Enter no of edges: 10
Enter edges (from to weight):
0 1 6
0 2 5
0 3 5
1 4 -1
2 1 -2
3 2 -2
2 4 1
3 5 -1
4 6 3
5 6 3
Distance array: 0 1 3 5 0 4 3
Predecessor array: 0 2 3 0 1 3 4

 *
 * */
```

Floyd-Warshall

```
#include <iostream>
#include <vector>
#include <climits>

using namespace std;

void floyd_warshall(vector<vector<int>>& dist, int n) {
      for(int k = 1; k <= n; k++) {
            for(int i = 1; i <= n; i++) {
                  for(int j = 1; j <= n; j++) {
                        if(i == j) {
                              dist[i][j] = 0;
                        } else {
                              dist[i][j] = min(dist[i][j], dist[i][k] +
dist[k][j]);
                        }
                  }
            }
      }
}

int main() {
      int vertices;
      cout << "Enter no of vertices: ";
      cin >> vertices;

      int edges;
      cout << "Enter no of edges: ";
      cin >> edges;

      vector<vector<int>> cost(vertices+1, vector<int>(vertices+1, 1000));

      int u, v, wt;
      cout << "Enter edges: " << endl;
      for(int edge = 0; edge < edges; edge++) {
            cout << "Enter edge (format: u v weight): ";
            cin >> u >> v >> wt;
            cost[u][v] = wt;
      }

      cout << "\n ----- COST MATRIX -----\n";
      for(int i = 1; i <= vertices; i++) {
```

```
            for(int j = 1; j <= vertices; j++) {
                  if(cost[i][j] == 1000)
                        cout << "inf" << " ";
                  else
                        cout << cost[i][j] << " ";
            }
            cout << endl;
      }

      floyd_warshall(cost, vertices);

      cout << "\n ----- SHORTEST PATH MATRIX -----\n";
      for(int i = 1; i <= vertices; i++) {
            for(int j = 1; j <= vertices; j++) {
                  cout << cost[i][j] << " ";
            }
            cout << endl;
      }

      for(int i = 1; i <= vertices; i++) {
            for(int j = 1; j <= vertices; j++) {
                  if(i != j) {
                        cout << "Shortest path from " << i << " to " << j <<
" is " << cost[i][j] << endl;
                  }
            }
      }
}

/*
Sample output:
Enter no of vertices: 4
Enter no of edges: 6
Enter edges:
Enter edge (format: u v weight): 1 2 3
Enter edge (format: u v weight): 2 1 2
Enter edge (format: u v weight): 1 4 5
Enter edge (format: u v weight): 2 4 4
Enter edge (format: u v weight): 3 2 1
Enter edge (format: u v weight): 4 3 2

 ----- COST MATRIX -----
inf 3 inf 5
2 inf inf 4
inf 1 inf inf
inf inf 2 inf

 ----- SHORTEST PATH MATRIX -----
```

```
0 3 7 5
2 0 6 4
3 1 0 5
5 3 2 0
Shortest path from 1 to 2 is 3
Shortest path from 1 to 3 is 7
Shortest path from 1 to 4 is 5
Shortest path from 2 to 1 is 2
Shortest path from 2 to 3 is 6
Shortest path from 2 to 4 is 4
Shortest path from 3 to 1 is 3
Shortest path from 3 to 2 is 1
Shortest path from 3 to 4 is 5
Shortest path from 4 to 1 is 5
Shortest path from 4 to 2 is 3
Shortest path from 4 to 3 is 2

*/
```

N Queens

```
// pastebin.com/deenFDkZ

#include <iostream>
#include <vector>
using namespace std;

int count;
vector<int> solution;

bool canPlace(int row) {
    int col = solution[row];
    for(int i = 1; i < row; i++) {
        // if they are in same column
        // or in diagonals
//        (1, 2) (1, 3) (1, 4)
//        (2, 2) (2, 3)
//        (3, 2)
        if((solution[i] == col) || abs(solution[i] - col) == abs(row - i)) {
            return false;
        }
    }

    return true;
```

```
}

void printSolution(int n) {
    count++; // increment no of solutions
    cout << "Solution " << count << ": " << endl;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= n; j++) {
            if(solution[i] == j) {
                cout << "Q ";
            } else {
                cout << "* ";
            }
        }
        cout << endl;
    }
}

void nqueens(int n) {
    int row = 1;

    while(row != 0) {
        solution[row]++;  // by default we place the queen at first column

        // we are not at end && we cant place queen at col
        while((solution[row] <= n) && !canPlace(row)) {
            solution[row]++;
        }

        if(solution[row] <= n) {
            // if we are at last row we found
            if(row == n) {
                printSolution(n);
            } else {
                row++;
            }
        } else {
            solution[row] = 0;
            row--; // backtrack
        }
    }
}

int main() {

    int n;
    cout << "Number of N queens: ";
    cin >> n;
```

```
    count = 0;
    solution.resize(n+1, 0);

    nqueens(n);

    cout << "No of possible solutions: " << count << endl;

    return 0;
}
```

Subset Sum

```
#include <iostream>
#include <vector>

using namespace std;

int count;

void printSolution(vector<int> arr, vector<int> solution, int n) {
    count++;
    cout << "Solution " << count << ": ";
    for(int i = 0; i < n; i++) {
        cout << solution[i] << " ";
    }  cout << endl;

    cout << "Elements of the set are: ";
    for(int i = 0; i < n; i++) {
        if (solution[i])
            cout << arr[i] << " ";
    }  cout << endl;
}

void findSubset(vector<int> arr, vector<int> solution, int k, int curSum, int
i, int n) {
    if(i >= n || curSum >= k) {
        if(curSum == k)
            printSolution(arr, solution, n);
        return;
    }

    findSubset(arr, solution, k, curSum, i + 1, n);
    solution[i] = 1;
    findSubset(arr, solution, k, curSum + arr[i], i + 1, n);
```

```
}

int main() {
    int n;
    cout << "Enter no of elements in the subset: ";
    cin >> n;

    vector<int> arr(n);
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }

    int k;
    cout << "Enter sum: ";
    cin >> k;

    vector<int> solution(n, 0);
    count = 0;

    findSubset(arr, solution, k, 0, 0, n);

    return 0;
}
```

Graph coloring

```
#include <iostream>
#include <vector>


using namespace std;

int m; // no of colors
int n; // no of vertices or nodes in the graph

int c = 0;
int sol = 0; // no of solutions

vector<int> x; // Solution vector
vector<vector<int>> g; // graph

void next_value(int k) {
      int j;
      while(1) {
            x[k] = (x[k] + 1) % (m + 1);
```

```
            if(x[k] == 0) {
                  return;
            }

            for(j = 1; j <= n; j++) {
                  if(g[k][j] == 1 && x[k] == x[j]) {
                        break;
                  }
            }

            if(j == (n + 1)) {
                  return;
            }
      }
}

void graph_coloring(int k) {
      while(1) {
            next_value(k);
            if(x[k] == 0) {
                  return;
            }

            if(k == n) {
                  c = 1;
                  for(int i = 1; i <= n; i++) {
                        cout << x[i] << " ";
                  }
                  sol++;
                  cout << endl;
            } else {
                  graph_coloring(k+1);
            }
      }
}

int main() {

      cout << "Enter no of vertices in the graph: ";
      cin >> n;

      cout << "Enter graph edges in form of adj list" << endl;
      g.resize(n+1, vector<int>(n+1, 0)); // graph
      x.resize(n + 1);

      for(int i = 1; i <= n; i++) {
            for(int j = 1; j <= n; j++) {
                  cin >> g[i][j];
```

```
            }
      }

      cout << "All possible solutions are:\n";
      for(m = 1; m <= n; m++) {
            if(c == 1) {
                  break;
            }

            graph_coloring(1);
      }

      cout << "\nChromatic number is: " << (m-1) << endl;
      cout << "Total no of solutions: " << sol << endl;

}
```

Hamiltonian Cycles

```
#include <iostream>
#include <vector>

using namespace std;

int n, c = 0; // n => no of vertices, c = no of solutions
vector<vector<int>> g; // graph (adj matrix)
vector<int> x; // visited array

void display_cycle() {
      for(int i = 1; i <= n; i++) { cout << x[i] << " "; }
      cout << x[1] << endl;
}

void next_value(int k) {
      int j;
      while(1) {
            x[k] = (x[k] + 1) % (n + 1); // trying the next val

            // if soln is not possible, traceback
            if(x[k] == 0) {
                  return;
            }

            // if there is an edge between prev node & this node
```

```
            if(g[x[k - 1]][x[k]]) {
                  // check if this node is already visited or not
                  for(j = 1; j <= k - 1; j++) {
                        // if already visited, break
                        if(x[j] == x[k]) {
                              break;
                        }
                  }

                  // not visited, then
                  if(j == k) {

                        // if there are still vertexes to visit
                        // or this is last vertex and has edge to source
                        // then solution is found, so return

                        if((k < n) || (k == n && g[x[n]][x[1]] != 0)) {
                              return;
                        }
                  }
            }
      }
}

void hamiltonian(int k) {
      while(1) {
            next_value(k);

            // if solution is not possible, trace back
            if(x[k] == 0) {
                  return;
            }

            // if solution is found, display it
            if(k == n) {
                  c = c + 1;
                  display_cycle();
            } else {
                  hamiltonian(k + 1);
            }
      }
}

int main() {

      int i, j;
      cout << "Enter no of vertices: ";
      cin >> n;
```

```
	g.resize(n + 1, vector<int>(n + 1, 0));
	x.resize(n + 1, 0);

	cout << "Enter the graph (adj matrix): " << endl;
	for(int i = 1; i <= n; i++) {
		for(int j = 1; j <= n; j++) {
			cin >> g[i][j];
		}
	}

	x[1] = 1; // mark start node 1 as visited
	cout << "\nHamiltonian cycles possible are: " << endl;

	hamiltonian(2);

	if(c == 0) {
		cout << "Solution not possible" << endl;
	} else {
		cout << "Total " << c << " solutions are found" << endl;
	}

	return 0;
}
```