ASSIGNMENT 2

ACTIVITY SELECTION

1)

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool activityComparator(pair<int, int> a, pair<int, int> b) {
    return a.second < b.second;
}

void activitySelection(int start[], int finish[], int n) {
    vector<pair<int, int>> activities;
    for (int i = 0; i < n; i++) {
        activities.push_back({start[i], finish[i]});
    }

    // Sort based on finish time
    sort(activities.begin(), activities.end(), activityComparator);

    cout << "Selected activities are:\n";
    int lastEnd = activities[0].second;
    cout << "(" << activities[0].first << ", " << activities[0].second << ")\n";

    for (int i = 1; i < n; i++) {
        if (activities[i].first >= lastEnd) {
            cout << "(" << activities[i].first << ", " << activities[i].second << ")\n";
            lastEnd = activities[i].second;
        }
    }
}

int main() {
    int start[] = {1, 3, 0, 5, 8, 5};
    int finish[] = {2, 4, 6, 7, 9, 9};
    int n = sizeof(start) / sizeof(start[0]);

    activitySelection(start, finish, n);
}
```

```
Selected activities are:
(1, 2)
(3, 4)
(5, 7)
(8, 9)
```

MIN PLATFORM

2)

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

int findMinimumPlatforms(int arr[], int dep[], int n) {
   sort(arr, arr + n);
   sort(dep, dep + n);

   int platforms_needed = 1, max_platforms = 1;
   int i = 1, j = 0;

   while (i < n && j < n) {
      if (arr[i] <= dep[j]) {
         platforms_needed++;
         i++;
      } else {
         platforms_needed--;
         j++;
      }
      max_platforms = max(max_platforms, platforms_needed);
   }
   return max_platforms;
}

int main() {
   int arr[] = {900, 940, 950, 1100, 1500, 1800};
   int dep[] = {910, 1200, 1120, 1130, 1900, 2000};
   int n = sizeof(arr) / sizeof(arr[0]);

   cout << "Minimum Number of Platforms Required = " << findMinimumPlatforms(arr, dep, n)
<< endl;
}
```

```
Minimum Number of Platforms Required = 3
```

3) JOB SCHEDULE WITH PROFIT GIVEN

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

struct Job {
   char id;
   int deadline;
   int profit;
};

bool compare(Job a, Job b) {
   return a.profit > b.profit;
}

void jobSequencing(Job jobs[], int n) {
   sort(jobs, jobs + n, compare);

   int maxDeadline = 0;
   for (int i = 0; i < n; i++)
      maxDeadline = max(maxDeadline, jobs[i].deadline);

   vector<char> result(maxDeadline + 1, '-'); // Job slots

   int totalProfit = 0;
   for (int i = 0; i < n; i++) {
      for (int j = jobs[i].deadline; j > 0; j--) {
         if (result[j] == '-') {
            result[j] = jobs[i].id;
            totalProfit += jobs[i].profit;
            break;
         }
      }
   }

   cout << "Scheduled Jobs: ";
   for (int i = 1; i <= maxDeadline; i++) {
      if (result[i] != '-')
         cout << result[i] << " ";
   }
```

```
    cout << "\nTotal Profit: " << totalProfit << endl;
}

int main() {
    Job jobs[] = { {'a', 4, 20}, {'b', 1, 10}, {'c', 1, 40}, {'d', 1, 30} };
    int n = sizeof(jobs) / sizeof(jobs[0]);

    jobSequencing(jobs, n);
    return 0;
}
```

```
Scheduled Jobs: c a
Total Profit: 60
```

FFRACTIONAL KNAPSACK

4)

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

struct Item {
    int profit, weight;
};

// Compare items by profit/weight ratio (descending)
bool compare(Item a, Item b) {
    double r1 = (double)a.profit / a.weight;
    double r2 = (double)b.profit / b.weight;
    return r1 > r2;
}

double fractionalKnapsack(Item items[], int n, int capacity) {
    sort(items, items + n, compare);

    double totalProfit = 0.0;

    cout << "Items included in the knapsack:\n";

    for (int i = 0; i < n; i++) {
        if (capacity >= items[i].weight) {
            // Take the whole item
```

```
            capacity -= items[i].weight;
            totalProfit += items[i].profit;
            cout << "- Taken 100% of item with profit = " << items[i].profit
                 << " and weight = " << items[i].weight << endl;
        } else {
            // Take fraction of the item
            double fraction = (double)capacity / items[i].weight;
            totalProfit += items[i].profit * fraction;
            cout << "- Taken " << fraction * 100 << "% of item with profit = " << items[i].profit
                 << " and weight = " << items[i].weight << endl;
            break; // Knapsack full
        }
    }
    return totalProfit;
}

int main() {
    Item items[] = { {60, 10}, {100, 20}, {120, 30} };
    int capacity = 50;
    int n = sizeof(items) / sizeof(items[0]);

    double maxProfit = fractionalKnapsack(items, n, capacity);
    cout << "Maximum profit in knapsack = " << maxProfit << endl;
}
```

```
Items included in the knapsack:
- Taken 100% of item with profit = 60 and weight = 10
- Taken 100% of item with profit = 100 and weight = 20
- Taken 66.6667% of item with profit = 120 and weight = 30
Maximum profit in knapsack = 240
```

5)

HUFFMAN

```
#include <iostream>
#include <queue>
#include <vector>
#include <string>
using namespace std;

// Max number of ASCII characters (0-255)
#define CHAR_RANGE 256

// Huffman tree node
struct Node {
```

```
    char ch;
    int freq;
    Node *left, *right;

    Node(char c, int f) {
        ch = c;
        freq = f;
        left = right = nullptr;
    }
};

// Comparator for min-heap
struct Compare {
    bool operator()(Node* a, Node* b) {
        return a->freq > b->freq;
    }
};

// Generate Huffman codes recursively
void generateCodes(Node* root, string code, string codes[]) {
    if (!root) return;

    // If leaf node
    if (!root->left && !root->right) {
        codes[(unsigned char)root->ch] = code;
    }

    generateCodes(root->left, code + "0", codes);
    generateCodes(root->right, code + "1", codes);
}

// Encode the input using Huffman codes
string encodeString(const string& input, string codes[]) {
    string encoded = "";
    for (char ch : input) {
        encoded += codes[(unsigned char)ch];
    }
    return encoded;
}

// Decode encoded string using Huffman Tree
string decodeString(Node* root, const string& encoded) {
    string decoded = "";
    Node* current = root;
```

```
    for (char bit : encoded) {
        if (bit == '0') current = current->left;
        else current = current->right;

        if (!current->left && !current->right) {
            decoded += current->ch;
            current = root;
        }
    }

    return decoded;
}

int main() {
    string input;
    cout << "Enter a string: ";
    getline(cin, input);

    // Step 1: Count character frequencies
    int freq[CHAR_RANGE] = {0};
    for (char ch : input) {
        freq[(unsigned char)ch]++;
    }

    // Step 2: Create min-heap and insert all non-zero frequency characters
    priority_queue<Node*, vector<Node*>, Compare> pq;

    for (int i = 0; i < CHAR_RANGE; i++) {
        if (freq[i] > 0) {
            pq.push(new Node((char)i, freq[i]));
        }
    }

    // Step 3: Build Huffman Tree
    while (pq.size() > 1) {
        Node* left = pq.top(); pq.pop();
        Node* right = pq.top(); pq.pop();

        Node* merged = new Node('\0', left->freq + right->freq);
        merged->left = left;
        merged->right = right;

        pq.push(merged);
```

```
    }

    Node* root = pq.top();

    // Step 4: Generate Huffman codes
    string codes[CHAR_RANGE];
    generateCodes(root, "", codes);

    cout << "\nHuffman Codes:\n";
    for (int i = 0; i < CHAR_RANGE; i++) {
        if (!codes[i].empty()) {
            cout << (char)i << ": " << codes[i] << endl;
        }
    }

    // Step 5: Encode the input
    string encoded = encodeString(input, codes);
    cout << "\nEncoded string: " << encoded << endl;

    // Step 6: Decode back
    string decoded = decodeString(root, encoded);
    cout << "Decoded string: " << decoded << endl;

    return 0;
}
```

```
Enter a string: abacabad

Huffman Codes:
a: 0
b: 10
c: 110
d: 111

Encoded string: 01001100100111
Decoded string: abacabad
```