

# # Greedy Algorithm → local optimum

\* N meetings in one room :-

→ Sort on the basis of end time

(1,2) (3,4) (0,6) (5,7) (8,9) (5,9)

mEnd = 0

if( mEnd < startVal) → meeting can be conducted.

-1 < 1 ✓

2 < 3 ✓

mEnd = 2

mEnd = 3

3 < 0 X

3 < 5 ✓

7 < 8 ✓

8 < 5 X

→ 4 ans

There is one meeting room in a firm. There are N meetings in the form of (start[i], end[i]) where start[i] is start time of meeting i and end[i] is finish time of meeting i. What is the maximum number of meetings that can be accommodated in the meeting room when only one meeting can be held in the meeting room at a particular time?

Note: Start time of one chosen meeting can't be equal to the end time of the other chosen meeting.

Example 1:

Input:

N = 6

start[] = {1,3,0,5,8,5}

end[] = {2,4,6,7,9,9}

Output:

4

Explanation:

Maximum four meetings can be held with given start and end timings.

The meetings are - (1, 2), (3, 4), (5, 7) and (8, 9)

```
class Solution
{
public:
//Function to find the maximum number of meetings that can
//be performed in a meeting room.
static bool cmp(pair<int, int> a, pair<int, int> b) {
return a.second < b.second;
}
int maxMeetings(int start[], int end[], int n)
{
vector<pair<int, int>> v;
for(int i = 0; i < n; i++) {
v.push_back({start[i], end[i]});
}
sort(v.begin(), v.end(), cmp);

int count = 1;
int ansEnd = v[0].second;

for(int i = 1; i < n; i++) {
if(ansEnd < v[i].first) {
count++;
ansEnd = v[i].second;
}
}
return count;
}
};
```

Time Complexity :  $O(n \log n)$

Space Complexity :  $O(n)$

## \* Shop in Candy store :-

In a candy store, there are **N** different types of candies available and the prices of all the N different types of candies are provided to you.

You are now provided with an attractive offer.

For every candy you buy from the store and get **K** other candies ( all are different types ) for free.

Now you have to answer two questions. Firstly, you have to find what is the minimum amount of money you have to spend to buy all the **N** different candies. Secondly, you have to find what is the maximum amount of money you have to spend to buy all the N different candies.

In both the cases you must utilize the offer i.e. you buy one candy and get **K** other candies for free.

**Example 1:**

**Input:**

N = 4

K = 2

candies[] = {3 2 1 4}

**Output:**

3 7

**Explanation:**

As according to the offer if you buy one candy you can take at most two more for free. So in the first case, you buy the candy which costs 1 and takes candies worth 3 and 4 for free, also you buy candy worth 2 as well. So **min cost** :  $1+2=3$ .

In the second case, you can buy the candy which costs 4 and takes candies worth 1 and 2 for free, also you need to buy candy worth 3 as well.

So **max cost** :  $3+4=7$ .

```
class Solution
{
public:
    vector<int> candyStore(int candies[], int N, int K)
    {
        sort(candies, candies + N);
        int mini = 0;
        int buy = 0;
        int free = N - 1;
        while(buy <= free) {
            mini = mini + candies[buy++];
            free = free - K;
        }

        int maxi = 0;
        buy = N - 1;
        free = 0;

        while(free <= buy) {
            maxi += candies[buy--];
            free += K;
        }
        return {mini, maxi};
    }
};
```

## # Check if possible to survive island:-

Geekina got stuck on an island. There is only one shop on this island and it is open on all days of the week except for Sunday. Consider following constraints:

- N – The maximum unit of food you can buy each day.
- S – Number of days you are required to survive.
- M – Unit of food required each day to survive.

Currently, it's Monday, and she needs to survive for the next S days.

Find the minimum number of days on which you need to buy food from the shop so that she can survive the next S days, or determine that it isn't possible to survive.

**Example 1:**

**Input:** S = 10, N = 16, M = 2

**Output:** 2

**Explanation:** One possible solution is to buy a box on the first day (Monday), it's sufficient to eat from this box up to 8th day (Monday) inclusive. Now, on the 9th day (Tuesday), you buy another box and use the chocolates in it to survive the 9th and 10th day.

```
class Solution{
public:
    int minimumDays(int S, int N, int M){
        int sundays = S / 7;

        int buyingDays = S - sundays;
        int totalFood = S * M;
        int ans = 0;
        if(totalFood % N == 0) ans = totalFood / N;
        else ans = totalFood / N + 1;
        if(ans <= buyingDays) return ans;
        return -1;
    }
};
```

## # Reverse Words in a given string:-

Given a String S, reverse the string without reversing its individual words. Words are separated by dots.

Example 1:

**Input:**

S = i.like.this.program.very.much

**Output:** much.very.program.this.like.i

**Explanation:** After reversing the whole string(not individual words), the input string becomes  
much.very.program.this.like.i

Example 2:

**Input:**

S = pqr.mno

**Output:** mno.pqr

**Explanation:** After reversing the whole string , the input string becomes  
mno.pqr

```
class Solution
{
public:
//Function to reverse words in a given string.
string reverseWords(string S)
{
    string ans = "";
    string temp = "";
    for(int i = S.length() - 1; i ≥ 0; i--) {
        if(S[i] == '.') {
            reverse(temp.begin(), temp.end());
            ans = ans + temp;
            ans.push_back('.');
            temp = "";
        } else {
            temp.push_back(S[i]);
        }
    }
    reverse(temp.begin(), temp.end());
    ans = ans + temp;
    return ans;
}
};
```

## # Chocolate Distribution Problem:-

Given an array A[] of positive integers of size N, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are M students, the task is to distribute chocolate packets among M students such that :

1. Each student gets **exactly** one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

Example 1:

**Input:**

N = 8, M = 5

A = {3, 4, 1, 9, 56, 7, 9, 12}

**Output:** 6

**Explanation:** The minimum difference between maximum chocolates and minimum chocolates is 9 - 3 = 6 by choosing following M packets :{3, 4, 9, 7, 9}.

```
class Solution{
public:
    long long findMinDiff(vector<long long> a, long long n, long long m)
    {
        sort(a.begin(), a.end());
        int i = 0, j = m - 1;
        int mini = INT_MAX;
        while(j < a.size()) {
            int diff = a[j] - a[i];
            mini = min(mini, diff);
            i++;
            j++;
        }
        return mini;
    }
};
```

## # Minimum Cost of ropes :-

There are given **N** ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths.

The task is to connect the ropes with minimum cost. Given **N** size array **arr[]** contains the lengths of the ropes.

Example 1:

```
Input:
n = 4
arr[] = {4, 3, 2, 6}
Output:
29
Explanation:
We can connect the ropes in following ways.
1) First connect ropes of lengths 2 and 3.
Which makes the array {4, 5, 6}. Cost of
this operation 2+3 = 5.
2) Now connect ropes of lengths 4 and 5.
Which makes the array {9, 6}. Cost of
this operation 4+5 = 9.
3) Finally connect the two ropes and all
ropes have connected. Cost of this
operation 9+6 =15
Total cost for connecting all ropes is 5
+ 9 + 15 = 29. This is the optimized cost
for connecting ropes.
Other ways of connecting ropes would always
have same or more cost. For example, if we
connect 4 and 6 first (we get three rope of 3,
2 and 10), then connect 10 and 3 (we get
two rope of 13 and 2). Finally we
connect 13 and 2. Total cost in this way
is 10 + 13 + 15 = 38.
```

```
class Solution{
public:
//Function to return the minimum cost of connecting the ropes.
long long minCost(long long arr[], long long n) {
    priority_queue<long long, vector<long long>, greater<long long>> pq;

    for(int i = 0; i < n; i++) pq.push(arr[i]);
    long long ans = 0;

    while(pq.size() > 1) {
        long long a = pq.top();
        pq.pop();

        long long b = pq.top();
        pq.pop();

        long long sum = a + b;

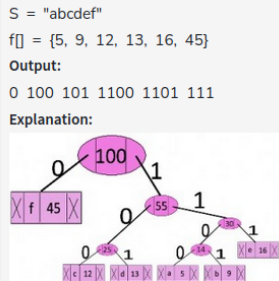
        ans += sum;
        pq.push(sum);
    }
    return ans;
};
```

## # Huffman Coding

Given a string **S** of distinct character of size **N** and their corresponding frequency **f[]** i.e. character **S[i]** has **f[i]** frequency. Your task is to build the Huffman tree print all the Huffman codes in preorder traversal of the tree.

**Note:** While merging if two nodes have the same value, then the node which occurs at first will be taken on the left of Binary Tree and the other one to the right, otherwise Node with less value will be taken on the left of the subtree and other one to the right.

Example 1:



HuffmanCodes will be:

f : 0  
c : 100  
d : 101  
a : 1100  
b : 1101  
e : 111

Hence printing them in the PreOrder of Binary Tree.

**Your Task:**

You don't need to read or print anything. Your task is to complete the function **huffmanCodes()** which takes the given string **S**, frequency array **f[]** and number of characters **N** as input parameters and returns a vector of strings containing all Huffman codes in order of preorder traversal of the tree.

```
class Node{
public:
    int data;
    Node* left;
    Node* right;

    Node(int d){
        data = d;
        left = NULL;
        right = NULL;
    }
};

class cmp{
public:
    bool operator()(Node* a, Node* b) {
        return a->data > b->data;
    }
};

class Solution
{
public:
    void traverse(Node* root, vector<string> &ans, string temp) {
        if(root->left == NULL && root->right == NULL) {
            ans.push_back(temp);
            return;
        }
        traverse(root->left, ans, temp + '0');
        traverse(root->right, ans, temp + '1');
    }

    vector<string> HuffmanCodes(string S, vector<int> f, int N)
    {
        priority_queue<Node*, vector<Node*>, cmp> pq;
        for(int i = 0; i < N; i++) {
            Node* temp = new Node(f[i]);
            pq.push(temp);
        }
        while(pq.size() > 1) {
            Node* left = pq.top();
            pq.pop();
            Node* right = pq.top();
            pq.pop();

            Node* newNode = new Node(left->data + right->data);
            newNode->left = left;
            newNode->right = right;
            pq.push(newNode);
        }

        Node* root = pq.top();
        vector<string> ans;
        string temp = "";
        traverse(root, ans, temp);
        return ans;
    }
};
```

## # Fractional Knapsack

Given weights and values of **N** items, we need to put these items in a knapsack of capacity **W** to get the **maximum** total value in the knapsack.

**Note:** Unlike 0/1 knapsack, you are **allowed** to break the item here.

**Example 1:**

**Input:**

N = 3, W = 50

value[] = {60,100,120}

weight[] = {10,20,30}

**Output:**

240.000000

**Explanation:**

Take the item with value 60 and weight 10, value 100 and weight 20 and split the third item with value 120 and weight 30, to fit it into weight 20. so it becomes  $(120/30)*20=80$ , so the total value becomes  $60+100+80.0=240.0$

Thus, total maximum value of item we can have is 240.00 from the given capacity of sack.

```
class Solution
{
public:
    //Function to get the maximum total value in the knapsack.
    static bool cmp(pair<double, Item>a, pair<double, Item> b) {
        return a.first > b.first;
    }
    double fractionalKnapsack(int W, Item arr[], int n)
    {
        vector<pair<double, Item>> v;
        for(int i = 0; i < n; i++) {
            double perUnitVal = (1.0 * arr[i].value) / arr[i].weight;
            pair<double, Item> p = {perUnitVal, arr[i]};
            v.push_back(p);
        }
        sort(v.begin(), v.end(), cmp);

        double totalVal = 0;

        for(int i = 0; i < n; i++) {
            if(v[i].second.weight > W) {
                totalVal += W * v[i].first;
                W = 0;
            } else {
                totalVal += v[i].second.value;
                W -= v[i].second.weight;
            }
        }
        return totalVal;
    }
};
```

## # Job Sequencing Problem

Given a set of **N** jobs where each **job<sub>i</sub>** has a deadline and profit associated with it.

Each job takes **1** unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with job if and only if the job is completed by its deadline.

Find the number of jobs done and the **maximum profit**.

**Note:** Jobs will be given in the form (Job<sub>i</sub>, Deadline, Profit) associated with that Job. Deadline of the job is the time before which job needs to be completed to earn the profit.

→ Sort on the basis of profit.

**Example 1:**

**Input:**

N = 4

Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

**Output:**

2 60

**Explanation:**

Job<sub>1</sub> and Job<sub>3</sub> can be done with maximum profit of 60 (20+40).

```
class Solution
{
public:
    //Function to find the maximum profit and the number of jobs done.
    static bool cmp(Job a, Job b) {
        return a.profit > b.profit;
    }
    vector<int> JobScheduling(Job arr[], int n)
    {
        sort(arr, arr + n, cmp);

        int maxiDeadline = INT_MIN;

        for(int i = 0; i < n; i++) {
            maxiDeadline = max(maxiDeadline, arr[i].dead);
        }
        int count = 0, maxProfit = 0;
        vector<int> schedule(maxiDeadline + 1, -1);
        for(int i = 0; i < n; i++) {
            int currProfit = arr[i].profit;
            int currJobId = arr[i].id;
            int currDead = arr[i].dead;

            for(int k = currDead; k > 0; k--) {
                if(schedule[k] == -1) {
                    count++;
                    maxProfit += currProfit;
                    schedule[k] = currJobId;
                    break;
                }
            }
        }
        return {count, maxProfit};
    }
};
```

→ Can use disjoint set to reduce the complexity