Algorithms

12. Greedy

Handwritten by pankaj kumar

4. Greedy Algorithm

Date
Page No. 86.

① Introduction:-

- A greedy algorithm is an approach for solving a problem by selecting the best option available at the moment. It doesn't worry whether the current best result will bring the overall optimal result.
- The algorithm never reverses the earlier decision even if the choice is wrong. it works in a top-down approach.
- This algorithm may not produce the best result for all the problems. It's bez it always goes for the local best choice to produce the global best result.

*** To determine greedy problem:**

(i) Greedy choice property:

If an optimal solution to the problem can be found by choosing the best choice at each step without reconsidering the previous steps once chosen, the problem can be solved using a greedy approach. This property is called greedy choice property

(ii) Optimal Substructure:

If the optimal overall solution to the problem corresponds to the optimal solution to its subproblems, then the problem can be solved using a greedy approach. This property is called optimal substructure.

*** The components that can be used in the greedy algorithm are:-**

- Candidate set: A solution that is created from the set is known as a candidate set.
- Selection function: This function is used to choose the candidate or subset which can be added in the solution
- Feasibility function: A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.

- Objective function: A function is used to assign the value to the solution or the partial solution.
- solution function: This function is used to intimate whether the complete function has been reached or not.

* Advantages:

- The algorithm is easier to describe.
- This algorithm can perform better than other algo (not in all cases)

* Disadvantages:

- the greedy algorithm doesn't always produce the optimal. This is the major disadvantages of the algorithm.

* Pseudo code of Greedy Algorithm:

```
Algorithm Greedy(a, n) {
    solution := 0;
    for i=0 to n do
    {
        x := select(a);
        if feasible(solution, x) {
            solution := union(solution, x)
        }
        return solution;
    }
}
```

* Application:

- finding shortest path
- Mst (prims or krustal's)
- Job sequencing with deadline
- fractional knapsack

(2) Activity Selection Problem:

- You are given n activities with their start and finish time 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:**

(1) Consider the following 3 activities sorted by finish time

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 index in start & finish]

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

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

Solution set = {0, 1, 3, 4} [These are indexes in start[] & finish[]]

*** Approach:**

greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity. We can sort the finish time then use greedy approach.

*** Algorithm:**

1. Sort the activities according to their finishing time.
2. Select the first activity from the sorted array and print it
3. Do the following for the remaining activity.
   a) if the start time of activity is greater than or equal to the finish time of the previously selected activity then print it

*** Code:**

// When activity is already sorted to their finishing time.

```
void printMaxActivities (int s[], int f[], int n) {
    int i, j;
    i = 0;
    cout << " " << i;
    for (j=1; j<n; j++) {
        if (s[j] >= f[i]) {
            cout << " " << j;
            i = j
        }
    }
}
int main() {
    int s[] = {1, 3, 0, 5, 8, 5};
    int f[] = {2, 4, 6, 7, 9, 9};
    int n = sizeof(s) / sizeof(s[0]);
    printMaxActivities(s, f, n);
    return 0;
}
```

output:
0 1 3 4

Time complexity:-

(1) It will take O(n) Time when activity is sorted according to finish.
(2) It will take O(n log n) Time when activity is unsorted.

* When activity is unsorted:-

(1) We create a structure/class. We sort all activities by finish time then we apply same algorithm.

```
#include <bits/stdc++.h>
using namespace std;
struct Activity {
    int start, finish;
};
```

```
bool activityCompare (Activity s1, Activity s2) {
        return (s1.finish < s2.finish);
}
void printMaxActivities (Activity arr[], int n) {
        sort (arr, arr+n, activityCompare);
        int i=0;
        cout << "(" << arr[i].start << ", " << arr[i].finish << ");";
        for (int j=1; j<n; j++) {
                if (arr[j].start >= arr[i].finish) {
                        cout << "(" << arr[j].start << ", " <<
                                arr[j].finish << ");";
                        i=j;
                }
        }
}
int main() {
        Activity arr[] = {{5,9},{1,2},{3,4},{0,6},{5,7},{8,9}};
        printMaxActivities (arr, n);
        return 0;
}
```

(3) Job Sequencing Problem :-

- Given an array of Jobs having a specific deadline and associated with a profit, provided the job is completed within the given deadline. The task is to maximize the profit by arranging the jobs in a shedule, such that only one job can be done at a time.

Examples:

Jobs:	J1	J2	J3	J4	J5	J6	J7
Profit:	35	30	25	20	15	12	5
deadline:	3	4	4	2	3	1	2

Output: J4 J3 J1 J2

* Approach 1: Greedy Approach:

Since, the task is to get maxm profit by scheduling the jobs, the idea is to approach this problem greedily.

Algo :-

(1) Sort the jobs based on decreasing order of profit.
(2) Iterate through the jobs and perform the following
- Choose a slot i if:
  - slot i isn't previously selected:
  - i < deadline
  - i is maximum
- If no such slot exist ignore the jobs and continue.

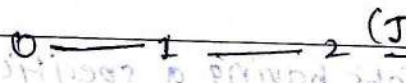
Dry Run :-

Job ID:	1	2	3	4	5
Deadline:	2	3	2	1	3
Profit:	20	38	16	10	30

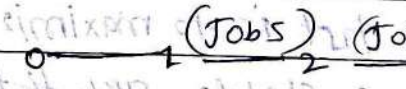
• Sort in decreasing order of profit

Job ID:	2	5	1	3	4
Deadline:	3	3	2	2	1
Profit:	38	30	20	16	10

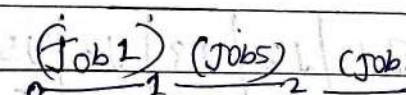
Now, (1) The last empty slot for Job 2, is (2-3) < deadline.



(2) The last empty slot for Job 5, is (1-2) < deadline.



(3) The last empty slot for Job 1, is (0-1) < deadline.



All slots are full so Jobs in sequence is: 1 5 2

Code:

```
Struct Job {
    char id;
    int dead;
    int profit;
};
bool comparison (Job a, Job b) {
    return (a.profit > b.profit);
}
void printJobScheduling (Job arr[], int n) {
    sort (arr, arr+n, comparison);
    int result[n];
    bool slot[n];
    for (int i=0; i<n; i++) {
        slot[i] = false;
    }
    for (int i=0; i<n; i++) {
        for (int j=min(n, arr[i].dead)-1; j>=0; j--) {
            if (slot[j] == false) {
                result[j] = i;
                slot[j] = true;
                break;
            }
        }
    }
    for (int i=0; i<n; i++)
        if (slot[i])
            cout << arr[result[i]].id << " ";
}
int main() {
    Job arr[] = {{'a', 2, 20}, {'b', 3, 30}, {'c', 2, 16}, {'d', 3, 20},
                 {'e', 3, 30}};
    printJobScheduling(arr, 5);
    return 0;
}
```

Time Complexity : O(N^2).
Space Complexity : O(1).

* Efficient Approach: Using Set

- In previous approach, for each job we had to iterate through all the jobs to find a job satisfying the condition
- In this case, the idea is to apply binary search on the set and find the jobs satisfying the given conditions

Algo :-

1. Sort the jobs based on decreasing order of profit
2. Create two variables, total_jobs=0, maxprofit=0.
3. Initialize the set in decreasing order of deadline
4. Iterate through the jobs from 1 and perform following
   - if the set is empty or the deadline of current job is less than the the last set element, ignore the job
   - Else, apply the binary search on set & find the nearest slot i, such that i<deadline and add the profit to maxprofit increment total job by 1 and erase that element from set
5. Return maxm profit.

- **Code:-**

```
bool compare(vector<int> &Job1, vector<int> &Job2){
    return Job1[1] > Job2[1]
}
int JobsScheduling(vector<vector<int>> &Jobs){
    sort(Jobs.begin(), Jobs.end(), compare);
    int maxProfit=0, maxDeadline=0;
    for(int i=0; i<Jobs.size(); i++)
        maxDeadline=max(maxDeadline, Jobs[i][0]);
    set<int, greater<int>> slots;
```

```
for(int i=maxDeadline; i>0; i--)
        slots.insert(i);
for(int i=0; i< Jobs.size(); i++){
    if(slot.size()==0 || Jobs[i][0]<*slots.rbegin())
            continue;
    int availableSlot = *slots.lower_bound(Jobs[i][0]);
    maxProfit = maxProfit + Jobs[i][1];
    slots.erase(availableSlot);
}
return maxProfit;
}
```

Time Complexity : O(N log N)
Space Complexity : O(max(deadline))

(4) Fractional knapsack Problem:

- Given a set of N items each having value V with weight W and the total capacity of a knapsack. The task is to find the maximal value of fractions of items that can be fit into the knapsack.

*** Examples:**

(1) input: A[] = {{60,20}, {100,50}, {120,30}}, Total_capacity = 50
Output: 180.00

- Take first & third item, total value = 60+120 = 180, (weight) 20+30 <= 50

(2) Input A[] = {{500,30}}, Total_capacity = 10
output : 166.67

- since the total capacity of the knapsack is 10, consider one-third of the item.

* Brute Force Approach

- The most basic approach is to try all possible subsets and possible fractions of the given set & find the maxm value among all such fractions.
- The time complexity will be exponential.

* Efficient Approach (Greedy):

Algorithm:-

- sort the given array of items according to ~~weight/value~~ value/weight (v/w) ratio in descending order.
- Start adding the item with the maxm ~~w/v~~ v/w ratio.
- Add the whole item, if the current weight is less than the capacity, else, add a portion of the item to the knapsack
- Stop, when all the items have been considered and the total weight becomes equal to the weight of the given knapsack

value	weight	v/w
60	20	3
100	50	2
120	30	4

sort A/C → to v/w

value	weight	v/w
120	30	4
60	20	3
100	50	2

Dry Run:-

Total capcity = 50

value	weight	x	w*x	curr_cap	max_val
120	30	1	30*1=30	50-30=20	120
60	20	1	20*1=20	20-20=20	60
100	50	0	knapsack filled	0	0
				Total_val = 120+60=180	

* Code:-

```
struct item
{
    int value, weight;
};
```

```
bool cmp (item a, item b){
        double r1 = (double)a.value/a.weight;
        double r2 = (double) b.value/ b.weight;
        return (r1>r2);
}
double fractionalknapsack(item A[], int Total_capacity, int n){
        sort(A, A+n, cmp);
        int cur_weight =0;
        double final_val = 0.0;
        for(int i=0; i<n; i++){
            if (cur_weight + arr[i].weight <= Total_capacity)
            {   cur_weight += A[i].weight;
                Final_val += A[i].value;
            }
            else{
                int remain = Total_capacity - cur_weight;
                Final_val += A[i].value*(double) remain/
                                A[i].weight);
            }
        }return Final_val;
}
// Time Complexity: O(N*logN)
// Space Complexity: O(1)
int main(){
    int W=50
    item arr[] = {{60,20},{100,50},{120,30}};
    int n= sizeof (arr)/sizeof (arr[0]);
    cout << fractionalknapsack(arr, W, n);
    return 0;
}
```

Output :- 180

5. Huffman Coding:

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, length of the assigned code are based on the frequency of corresponding characters. The most frequent character gets the smaller code and the least frequent character gets the largest code.
- The variable-length codes assigned to input character are prefix codes, mean the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of assigned to any other character. this is how Huffman code make sure that there is no ambiguity when decoding.
- let there be four character a, b, c and d their corresponding variable length codes are 00, 01, 0 and 1 but this leads to ambiguity bcz code assigned to c is the prefix of code assigned to a and b. decoding of 00, 01 can cc, cd (instead of a, b).

There are mainly two major part in Huffman coding:
1. Build a Huffman tree from I/p characters.
2. Traverse the Huffman tree and assign codes to characters.

* Steps to build Huffman Tree:-

- I/p is an array of unique characters along with their frequency of occurrence and o/p is Huffman tree
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two node in min Heap. Initially the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min Heap.

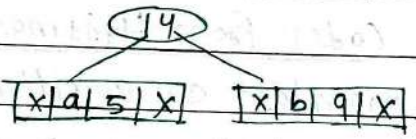
(3) Create a new internal node with a frequency equal to the sum of two nodes frequencies. Make the first extracted node as left child & other as right child. Add this node to min Heap.
(4) Repeat Step #2 and Step #3 until the Heap contains only one node. The remaining node is the root node & the tree is complete.

* Example :- Character - Frequency

[(a, 5), (b, 9), (c, 12), (d, 13), (e, 16), (f, 45)]

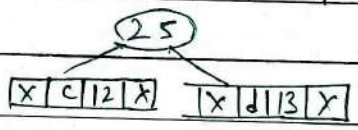
Step 1 :- Build a min Heap that contains 6 nodes where each node represent root of a tree with single node.

Step 2 :- Extract two minimum frequency node from min Heap. Add a new internal node with frequency 5+9 = 14



Now min heap contains 5 node where 4 nodes are root of trees with single element each, and one heap node is root of tree with three element.

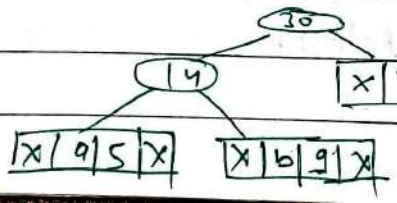
Step 3 :- Extract two min Frequency node from heap 12+13 = 25



Now min Heap contain 4 nodes 2 nodes are root of tree with single element, and two heap nodes are root of tree with more than one.

Character - Frequency
Internal node - 14
e - 16
Internal nod - 25
f - 45

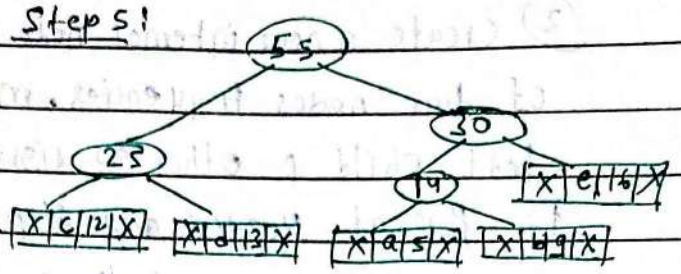
Step 4 :- Extract two nodes from Heap (14+16 = 30)



Date
Page No. 100

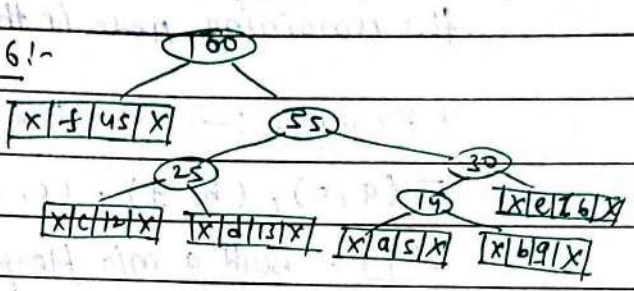
Character – Frequency
Internal Node – 25
Internal Node – 30
f – 45

Step 5:



Character – Frequency
~~Internal Node~~ f – 45
Internal-node – 55

Step 6:-

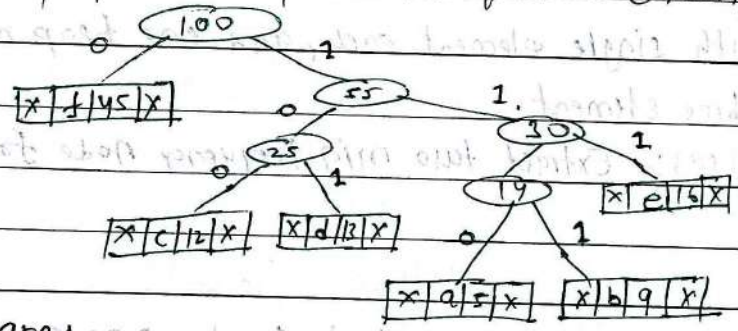


Now min Heap contains only one node.

Character	Frequency.
Internal-Node	100

* Step to Print Codes From Huffman tree

Traverse from the root. maintain an auxiliary array. while moving to left child. write 0 to the array. moving to the right, write 1 to the array. print array when leaf node encountered.



The codes are:

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

* Code Implementation:

Using STL

Date

Page No. 101

```
#include <bits/stdc++.h>
using namespace std;

struct MinHeapNode {        //A min Heap Huffman tree node.
    char data;
    unsigned freq;
    MinHeapNode *left *right;
    MinHeapNode(char data, unsigned freq){
        left = right = NULL;
        this->data = data;
        this-> freq = freq;
    }
};

struct compare{
    bool operator() (MinHeapNode* l, MinHeapNode* r){
        return (l-> freq > r-> freq);
    }
};

void printCodes (struct MinHeapNode * root, string str){
    if (!root)
        return;
    if (root -> data != '$')
        cout << root->data<<":"<< str <<"\n";
    printCodes (root->left, str+"0");
    printCodes (root->right, str+"1");
}

void HuffmanCodes (char data[], int freq[], int size){
    struct MinHeapNode *left, *right, *top;
    priority_queue < MinHeapNode *, vector<MinHeapNode*>,
                    compare>minHeap;
```

```
For(int i=0; i<size; ++i)
    minHeap.push(new MinHeapNode(data[i], freq[i]));

while (minHeap.size() != 1) {
    left = minHeap.top();
    minHeap.pop();
    right = minHeap.top();
    minHeap.pop();
    top = new MinHeapNode('$', left->freq + right->freq);
    top->left = left;
    top->right = right;
    minHeap.push(top);
    printCodes(minHeap.top(), "");
}

int main(){
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr)/sizeof(arr[0]);
    HuffmanCodes(arr, freq, size);
    return 0;
}
```

Time complexity:-

since efficient priority queue data structure require O(log(n)) time per insertion, and a complete binary tree with n leaves has 2n-1 nodes, and Huffman coding tree is a complete binary tree, this algorithm operate in O(n.log n) time where n is the total number of character.

(6) Efficient Huffman Coding for Sorted Input

- If we know that the given array is sorted (by non-decreasing order of frequency), we can generate Huffman codes in O(n) time

*Algorithm:

(1) Create two empty queues

(2) Create a least node for each unique character and Enqueue it to the first queue in non-decreasing order of frequency. Initially second queue is empty

(3) Deque two nodes with the minimum frequency by examines the front of both queues. Repeat following steps two times
   (a.) if second queue is empty, dequeue from first queue.
   (b.) if first queue is empty, dequeue from second queue
   (c.) Else, compare both front of queues and dequeue minm.

(4) Create a new internal node with frequency equal to the sum of two nodes frequencies. Make the first dequeued node as left child and second as a right child. Enqueue this node to second queue.

(5) Repeat Step#3 and #4 while there is more than one node in the queues. The remaining node is root node & tree is complete.

*Time Complexity = O(n)

(7) Find Minimum no. of coin:-

We want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency i.e {1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes to make the change

Examples:

Input: V = 70	Input: V = 121
Output: 2	Output: 3
50 + 20	100 + 20 + 1

```
#include <bits/stdc++.h>
using namespace std;
int deno[] = {1, 2, 5, 10, 20, 50, 100, 500, 1000};
int n = sizeof(deno) / sizeof(deno[0]);
void findMin(int V) {
    sort(deno, deno+n);
    vector<int> ans;
    for(int i = n-1; i>=0; i--) {
        while(V >= deno[i]) {
            V -= deno[i];
            ans.push_back(deno[i]);
        }
    }
    for(int i=0; i<ans.size(); i++)
        cout<<ans[i]<<" ";
}
int main() {
    int n=93;
    findMin(n);
    return 0;
}
```

Output: 50 20 20 2 1

Time: O(V)

Space: O(V)

Note:- This approach may not work for all denominations. For example, it doesn't work for denominations {9, 6, 5, 1} and V=11. The above approach would print 9, 1, 1. But we can use 2 denominations 5 & 6.

(8) K-Centers Problem:

Given n cities and distances b/w every pair of cities, select k cities to place warehouses (or ATMs or cloud servers) such that the maxm distance of a city to warehouse is minimized.

(8) Minimum Number of Platforms Required For a Railway/Bus Station.

Given the arrival & departure times of all trains that reach a railway station, the task is to find the minm no. of platform required for the railway station so that no train waits.

Examples:

Input: arr[] = {9:00, 9:40, 9:50, 11:00, 15:00, 18:00}
dep[] = {9:10, 12:00, 11:20, 11:30, 19:00, 20:00}

O/P: 3

There are at-most three trains at a time (b/w 11:00 to 11:20).

* Naive Approach:

The idea is to check every interval one by one and find the number of intervals that overlap with it. Keep track of maxm number of interval that overlap with an interval. Finally return the maxm no.

```
For(int i=0; i<n; i++){
        plat_needed=1;
        For(int j=i+1; j<n; j++){
            if([arr[i]>=arr[j] && arr[j]<=dep[j]) ||
                (arr[j]>=arr[i] && arr[j]<=dep[i]))
                plat_needed++;
        }
        result= max(result, plat_needed);
}
```

Time = O(n²)

* Efficient solution

The idea is to consider all event (arrival & departure) time into sorted order. Once the events are in sorted order, trace the number of train at any time keeping track of trains that have arrived, but not departed.

Ex:-

Time	Event type	Total Platform needed.
9:00	Arrival	1
9:10	Departure	0
9:40	Arrival	1
9:50	Arrival	2
11:00	Arrival	3
11:20	Departure	2
11:30	Departure	1
12:00	Departure	0
15:00	Arrival	1
18:00	Arrival	2
19:00	Departure	1
20:00	Departure	0

minimum platform needed = max platform needed at any time = 3

Implementation :-

Here we doesn't create a single sorted list of all events, rather it individually sorts arr[] and dep[] arrays, and then uses the merge process of merge sort to process them together as single sorted array.

```
int FindPlatform(int arr[], int dep[], int n){
    sort(arr, arr+n);
    sort(dep, dep+n);
    int plat_needed=1, result=1;
    int i=1, j=0;
```

```
while (i<n && j<n){
    if (arr[i] <= dep[j]){
        plat_needed++;
        i++;
    }
    else if (arr[i] > dep[j]){
        plat_needed--;
        j++;
    }
    if (plat_needed > result)
        result = plat_needed;
}
return result;
}
int main(){
    int arr[] = {900, 940, 950, 1100, 1500, 1800};
    int dep[] = {910, 1200, 1120, 1130, 1900, 2000};
    cout << FindPlatform (arr, dep, 6);
    return 0;
}
```

Time : O(N * logN)

(9.) Find Maxm Meeting in One Room :-

There is one meeting room in a firm. There are N meeting in the form of (s[i], f[i]), i.e start time of meeting & end time. The task is to find the maximum number of meeting that can be accommodated in the meeting room.

Example :-

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

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

Output :- 1, 2, 4, 5

i.e [1, 2], [3, 4], [5, 7] & [8, 9]

* Approach:

- Idea is to solve the problem using the greedy approach which is same as Activity selection problem.
- The idea is we will choose that meeting which finishes early so that we can accomodate more meeting.
- We can do this by sorting the meeting in increasing order of finishing time. and then we select a meeting which start time is greater than the previously selected meeting finish time.

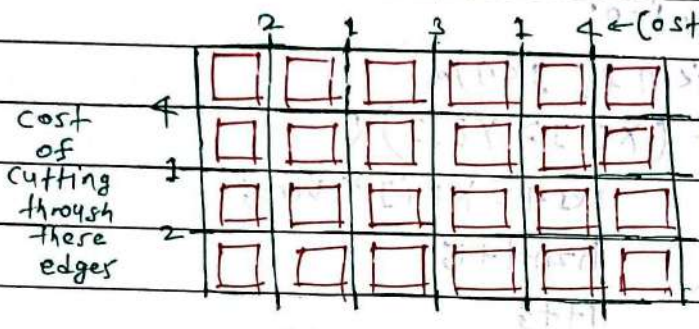
* Code:-

```
void maxMeeting (int s[], int f[], int n){
    pair<int, int> a[n+1];
    For(int i=0; i<n; i++){
        a[i].first = f[i];
        a[i].second = i;
    }
    sort(a, a+n);
    int time_limit = a[0].first;
    vector<int>m;
    m.push_back(a[0].second+1);
    for(int i=1; i<n; i++){
        if (s[a[i].second] > time_limit){
            m.push_back(a[i].second +1);
            time_limit= a[i].first;
        }
    }
    for (int i=0; i<m.size(); i++){
        cout<<m[i]<<" ";
    }
}
```

Time:- O(N log N)

(10) Minimum Cost to cut a board into squares

A board of length m and width n given, we need to break this board into m*n squares such that cost of breaking is minimum. cutting cost for each edge will be given for the board. In short, we need to choose such a sequence of cutting such that cost is minimized.



Total minm cost in above case is 42. It is evaluated using following steps.

initial value: Total_cost = 0 | V (verticle piece) = 1 | H (Horizontal piece) = 1

Total cost = Total_cost + edge_cost * total_pieces

V=1, H=1. Horizontal cut for cost 4 : cost = 0 + 4*1 = 4 (V)

V=1, H=2. vertical cut for cost 4 : cost = 4 + 4*2 = 12 (H)

V=2, H=2. vertical cut for cost 3 : cost = 12 + 3*2 = 18 (H)

V=3, H=2. Horizontal cut for cost 2 : cost = 18 + 2*3 = 24 (V)

V=3, H=3. vertical cut for cost 2 : cost = 24 + 2*3 = 30 (H)

V=4, H=3. Horizontal cut for cost 1 : cost = 30 + 1*4 = 34 (V)

V=4, H=4. vertical cut for cost 1 : cost = 34 + 1*4 = 38 (H)

V=5, H=4. vertical cut for cost 1 : cost = 38 + 1*4 = 42 (H)

* Approach:-

We apply greedy approach here in a sense that when there is minimum no. of piece (i.e initially) then we use maximum cost to cut it and lastly we have more piece so we use minimum cost to cut it. i.e we sort the both Horizontal & vertical cost in decreasing order then we solve it according to mergesort concept.

* code

```
int minimumCostofBreaking (int X[], int Y[], int m, int n){
    int res=0;
    sort ( X, X+m, greater<int>());
    sort (Y, Y+n, greater<int> ());
    int hznt=1, vert=1;
    int i=0, j=0;
    while (i<m && j<n){
        if (X[i] > Y[j]){
            res += X[i] * vert;
            hznt++;
            i++;
        }
        else {
            rest += Y[j] * hznt;
            vert++;
            j++;
        }
    }
    int total=0;
    while (i<m)
        total += X[i++];
    res += total * vert;

    total = 0;
    while (j<n)
        total += Y[j++];
    res += total * hznH;

    return res;
}
```

Time : O(n log n)

(11) Buy Maxm Stocks if 'i' stocks can be bought on 'i-th' day

- You are given an array 'prices' of a stock and a number 'amount'. Each element of the arraylist prices, 'prices[i]' represent the price of stock on the 'i-th' day. your task is to buy the maximum number of stock in 'amount' money. Every stock has an infinite supply.
- Rule to buy a stock. on the 'i-th' day you can buy at max 'i' a number of stock and 'i' is '1' based.

Example:

prices = [10,7,19], amount = 45.

so the total stock will be $(1*10+2*7+1*19=43)$ $1+2+1=4$

i.e Buy one stock on 1st day, 2 stock on 2nd day, 1 stock on last day.

Approach:

- The key idea observation here is that we should always buy stock with minimum price. Suppose the minm stock price is 'x' and you can buy only 'i' number of stock in 'k' amount.
- if k>=i*x then you can buy 'i' stock, remain amount 'k-i*x'. Else you can buy 'k/x' stock and remain amount is 'k-i*(k/x)'

(12) Find the minimum and maximum amount to buy all N candies.

- 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. There is also an attractive offer by the candy store. We can buy a single candy from the store and get at most 'k' other candies (all are different types) for free.

1. Find the minimum amount of money we have to spend to buy all the N different candies.
2. Find the maximum amount of money we have to spend to buy all the N different candies.

*Example

Input: price[] = {3,2,1,4} , K=2

Output:

min=3, max=7

since k is 2, if we buy one candy we can take almost two more free

- so in the first case we buy candy cost '1' and get free 3,4 also we buy 2, ∴ min cost = 1+2=3
- In the second case we buy candy which cost is '4' and get free 1,2 also we buy candy cost 3, ∴ max cost = 3+4=7

* Approach

First sort the price array.

① For finding minimum amount:

Start purchasing candies from starting and reduce k free candies from last with every single purchase.

② For finding maxm amount:

Start purchasing candies from the end and reduce k free candies from starting in every single purchase.

---

(13) Maximum product subset of an array.

Given an array a, we have to find maximum product possible with the subset of elements present in the array. The maximum product can be single element also.

Examples:

Input: a[] = {-1,-1,-2,4,3}

Output: 24 (-2*-1*4*3) = 24

Input: a[] = {-1, 0}

Output: 0

*Approach

(1.) if there are even no. of negative numbers and no zeros, result is simply product of all

(2.) if there are odd no. of negative numbers and no zeros, result is product of all except the negative integer with the least absolute value

(3.) if there are zeros, result is product of all except there zeros with with one exeptional case. The exceptional case is when there is one negative no. and all elements are zero, result is 0.

*Code:

```
int maxProductSubset (int a[], int n){
    if (n==1)
        return a[0];
    int max_neg = INT_MIN;
    int count_neg=0, count_zero=0;
    int prod = 1;
    for(int i=0; i<n; i++) {
        if (a[i]==0) {
            count_zero++;
            continue;
        }
        if (a[i]<0) {
            count_neg++;
            max_neg = max(max_neg, a[i]);
        }
        prod = prod * a[i];
    }
    if (count_zero==n)
        return 0;
    if (count_neg & 1) {
        if (count_neg==1 && count_zero>0 &&
                count_zero+count_neg==n)
            return 0;
        prod = prod / max_neg;
    }
```

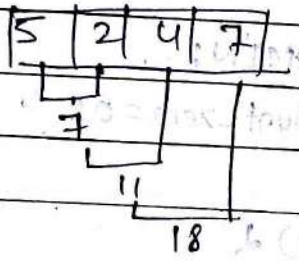
return prod;
}

Time: O(n)

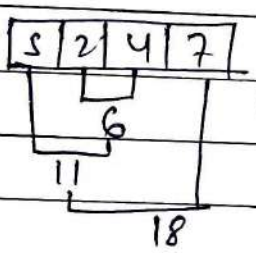
(14) Optimal Merge pattern

- You are given n files with their computation times in an array
- Choose any two files, add their computation times and append it to the list of computation time. {cost = sum of computation time}
- Do this until we are left with only one file in the array. We have to do this operation such that we get minimum cost finally.

*Example:



Total Cost = 7+11+18 = 36
(may not be minimum)



Total cost = 6+11+18 = 35
minimum

*Approach:

1. Push all element to a minheap.
2. Take top 2 element one by one and add the cost to answer. push the merged file to the minheap
3. When single element remains, output the cost.

*Code:

```
int mergel( int a[], int n) {
    priority_queue<int, vector<int>, greater<int>> minheap;
```

(There should be a space)

```
for(int i=0; i<n; i++)
        minheap.push(a[i]);
int ans=0;
while(minheap.size() > 1){
    int e1 = minheap.top();
    minheap.pop();
    int e2 = minheap.top();
    minheap.pop();
    ans += e1+e2;
    minheap.push(e1+e2);
}
return ans;
}
```

Time = main for loop * minheap.push
$O(n)$ $O(\log n)$ $= O(n \log n)$

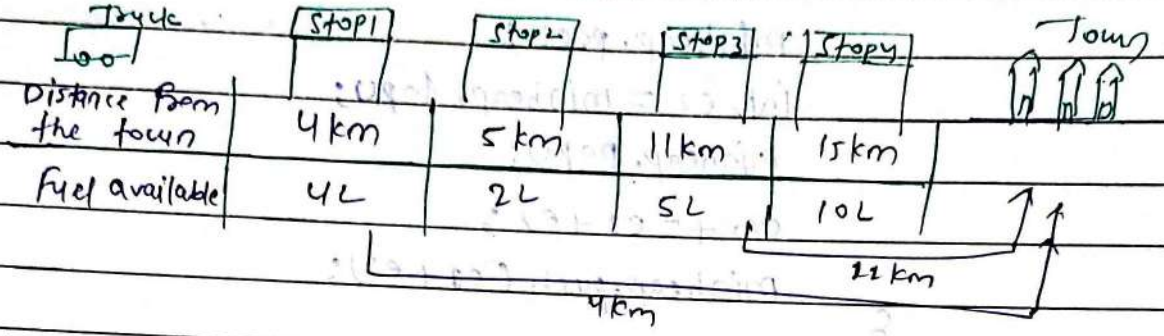
(15) EXPEDI (EXPEDITION)

- A group of cows grabbed a truck and ventured on an expedition deep into the jungle. The truck leak one unit of fuel every unit of distance it travels.
- To repair the truck, the cows need to drive to the nearest town (no more than 10^6 units distance). on this road, between the town and the current location, there are N ($1 \le N \le 10^5$) fuels stops where the cows can stop to acquire additional fuel (1...100 units at each stop).
- The cows want to make the minimum possible no. of stops for fuel on the way to town.
- Capacity of the tank is sufficiently large to hold any amount of fuel

Initial units of fuel: P ($1 \le P \le 10^6$)
Initial distance from town: L
• Determine the minimum no. of stops to reach the town.

* Example

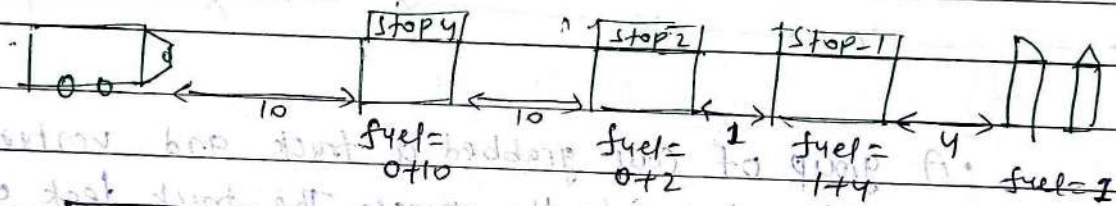
e.g. Petrol Pumps



L = 25km and P = 10L

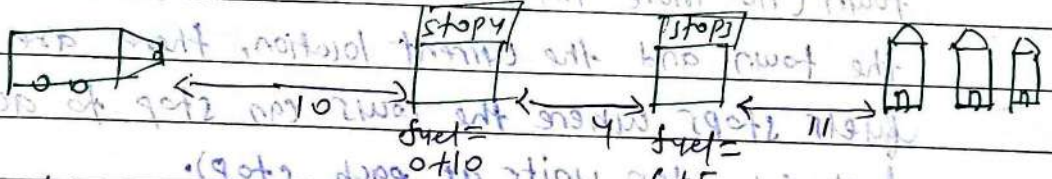
	Stop-1	Stop-2	Stop-3	Stop-4
So distance from initial position of truck	25-4 = 21 km	25-5 = 20 km	25-11 = 14 km	25-15 = 10 km

* One possible way



No. of stop = 3

* other possible way :-



No. of stop = 2

(*) Approach (BRUTE FORCE):-

(1) Generate all subsequences of stops.
(2) Iterate over all subsequences, choose the one that is feasible and has minimum no. of stops ($O(2^n)$)

OPTIMAL SOLUTION

1. create a maxheap, which stores the fuels available at the stops that we have traversed.
2. Sort the stops on the basis of distance of stops from initial position of truck.
3. keep iterating on the stops, and whenever fuel in the tank becomes empty, take the fuel from the maxheap and add it to the truck (greedy step).
4. maintain the count of stops from which we have taken fuel

* Code :-


```
int minstop(int arr1[], int arr2[], int n
            , int l, int p ){
    vector<pair<int,int>> a(n);
    for(int i=0; i<n; i++){
        a[i].first = arr1[i]
        a[i].second = arr2[i]
    }
    for(int i=0; i<n; i++){
        a[i].first = l-a[i].first;
    }
    sort(a.begin(), a.end());
    int ans=0;
    int curr=p;
    priority_queue<int, vector<int>> pq;
    for(int i=0; i<n; i++){
        if(curr>=l)
            break; //return 0;
        while(curr < a[i].first){
            if(pq.empty())
                return -1;
            ans++;
            curr+=pq.top();
```

```
            pq.pop();
        {
            pq.push(q[i].second);
    {
    while (!pq.empty() && curr<l){
            curr+=pq.top();
            pq.pop();
            ans++;
    {
    IF (curr<l)
        return -1;
    return ans;
{
```

(16) Maximum & minimum difference :-

- You are given an array, A, of n elements. You have to remove exactly n/2 elements from array A and add it to another array B (initially empty).
- Find the maximum & minimum values of differences between these two arrays.
- We define difference between these two array as:
  $\sum abs(A[i]-B[i])$

*Example

Input: n = 4
A: 12 -3 10 0

Output:
5 25
↓ ↓
diff maxm minm

For maxm diff.
A: -3 0
B: 12 10

For minm diff
A: -3 12
B: 0 10

*Approach:

To maximise abs (A[i] - A[j])

1. A[i] should bear larger as possible
2. A[j] should be as small as possible

For a sorted array,

maxm diff = (A[n-1] - A[0]) + (A[n-2] - A[1]) .. (A[N/2] - A[N/2-1])

= (A[n-1] + A[n-2] ... A[n/2]) - (A[0] + A[1] + .. A[n/2-1])

To minimize abs (A[i] - A[j])

1. The difference should between A[i] and A[j] should be similar.

for a sorted array,

minm diff = A[1] - A[0] + A[3] - A[2] ...

= (sum of odd idx) - (sum of even idx)