**GREEDY**

# 219. Activity Selection Problem

Given **N** activities with their start and finish day given in array **start[ ]** and **end[ ]**. 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 given day.  
**Note :**Duration of the activity includes both starting and ending day.

**Example 1:**

**Input:**

N = 2

start[] = {2, 1}

end[] = {2, 2}

**Output:**

1

**Explanation:**

A person can perform only one of the

given activities.

**Example 2:**

**Input:**

N = 4

start[] = {1, 3, 2, 5}

end[] = {2, 4, 3, 6}

**Output:**

3

**Explanation:**

A person can perform activities 1, 2

and 4.

**Your Task :**  
You don't need to read input or print anything. Your task is to complete the function ***activityselection()*** which takes array **start[ ],** array **end[ ]** and integer **N** as input parameters and returns the maximum number of activities that can be done.

**Expected Time Complexity** : O(N \* Log(N))  
**Expected Auxilliary Space**: O(N)  
  
**Constraints:**  
1 ≤ N ≤ 2\*105  
1 ≤ start[i] ≤ end[i] ≤ 109

## Solution is same as ques 232 of greedy.

# 220. Job SequencingProblem

Given a set of **N** jobs where each **jobi** 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 (Jobid, Deadline, Profit) associated with that Job.

**Example 1:**

**Input:**

N = 4

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

**Output:**

2 60

**Explanation:**

Job1 and Job3 can be done with

maximum profit of 60 (20+40).

**Example 2:**

**Input:**

N = 5

Jobs = {(1,2,100),(2,1,19),(3,2,27),

  (4,1,25),(5,1,15)}

**Output:**

2 127

**Explanation:**

2 jobs can be done with

maximum profit of 127 (100+27).

**Your Task** :  
You don't need to read input or print anything. Your task is to complete the function **JobScheduling()** which takes an integer **N** and an array of Jobs(Job id, Deadline, Profit) as input and returns the count of jobs and maximum profit.

**Expected Time Complexity**: O(NlogN)  
**Expected Auxilliary Space**: O(N)

**Constraints:**  
1 <= N <= 105  
1 <= Deadline <= 100  
1 <= Profit <= 500

## Solution:

A **Simple Solution** is to generate all subsets of a given set of jobs and check individual subsets for the feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets. The time complexity of this solution is exponential.   
This is a standard [Greedy Algorithm](https://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/)problem.

Following is the algorithm.

*1) Sort all jobs in decreasing order of profit.   
2) Iterate on jobs in decreasing order of profit.For each job , do the following :   
a)Find a time slot i, such that slot is empty and i < deadline and i is greatest.Put the job in   
this slot and mark this slot filled.   
b)If no such i exists, then ignore the job.*

The Following is the implementation of the above algorithm

// Program to find the maximum profit job sequence from a given array

// of jobs with deadlines and profits

#include<iostream>

#include<algorithm>

using namespace std;

// A structure to represent a job

struct Job

{

char id; // Job Id

int dead; // Deadline of job

int profit; // Profit if job is over before or on deadline

};

// This function is used for sorting all jobs according to profit

bool comparison(Job a, Job b)

{

return (a.profit > b.profit);

}

// Returns minimum number of platforms required

void printJobScheduling(Job arr[], int n)

{

// Sort all jobs according to decreasing order of profit

sort(arr, arr+n, comparison);

int result[n]; // To store result (Sequence of jobs)

bool slot[n]; // To keep track of free time slots

// Initialize all slots to be free

for (int i=0; i<n; i++)

slot[i] = false;

// Iterate through all given jobs

for (int i=0; i<n; i++)

{

// Find a free slot for this job (Note that we start

// from the last possible slot)

for (int j=min(n, arr[i].dead)-1; j>=0; j--)

{

// Free slot found

if (slot[j]==false)

{

result[j] = i; // Add this job to result

slot[j] = true; // Make this slot occupied

break;

}

}

}

// Print the result

for (int i=0; i<n; i++)

if (slot[i])

cout << arr[result[i]].id << " ";

}

// Driver code

int main()

{

Job arr[] = { {'a', 2, 100}, {'b', 1, 19}, {'c', 2, 27},

{'d', 1, 25}, {'e', 3, 15}};

int n = sizeof(arr)/sizeof(arr[0]);

cout << "Following is maximum profit sequence of jobs \n";

// Function call

printJobScheduling(arr, n);

return 0;

}

**Output**

Following is maximum profit sequence of jobs

c a e

The **Time Complexity** of the above solution is O(n2).

The costly operation in the Greedy solution is to assign a free slot for a job. We were traversing each and every slot for a job and assigning the greatest possible time slot(<deadline) which was available.

**What does greatest time slot means?**   
Suppose that a job J1 has a deadline of time t = 5. We assign the greatest time slot which is free and less than the deadline i.e 4-5 for this job. Now another job J2 with deadline of 5 comes in, so the time slot allotted will be 3-4 since 4-5 has already been allotted to job J1.

**Why to assign greatest time slot(free) to a job?**   
Now we assign the greatest possible time slot since if we assign a time slot even lesser than the available one than there might be some other job which will miss its deadline. 

**Example:**  
J1 with deadline d1 = 5, profit 40   
J2 with deadline d2 = 1, profit 20   
Suppose that for job J1 we assigned time slot of 0-1. Now job J2 cannot be performed since we will perform Job J1 during that time slot.

**Using Disjoint Set for Job Sequencing**   
All time slots are individual sets initially. We first find the maximum deadline of all jobs. Let the max deadline be m. We create m+1 individual sets. If a job is assigned a time slot of t where t >= 0, then the job is scheduled during [t-1, t]. So a set with value X represents the time slot [X-1, X].   
We need to keep track of the greatest time slot available which can be allotted to a given job having deadline. We use the parent array of Disjoint Set Data structures for this purpose. The root of the tree is always the latest available slot. If for a deadline d, there is no slot available, then root would be 0. Below are detailed steps.  
**Initialize Disjoint Set:** Creates initial disjoint sets.

// m is maximum deadline of a job

parent = new int[m + 1];

// Every node is a parent of itself

for (int i = 0; i ≤ m; i++)

parent[i] = i;

**Find :**Finds the latest time slot available.

// Returns the maximum available time slot

find(s)

{

// Base case

if (s == parent[s])

return s;

// Recursive call with path compression

return parent[s] = find(parent[s]);

}

**Union :**

Merges two sets.

// Makes u as parent of v.

union(u, v)

{

// update the greatest available

// free slot to u

parent[v] = u;

}

**How come find returns the latest available time slot?**   
Initially all time slots are individual slots. So the time slot returned is always maximum. When we assign a time slot ‘t’ to a job, we do union of ‘t’ with ‘t-1’ in a way that ‘t-1’ becomes parent of ‘t’. To do this we call union(t-1, t). This means that all future queries for time slot t would now return the latest time slot available for set represented by t-1.

**Implementation :**  
The following is the implementation of above algorithm.

// C++ Program to find the maximum profit job sequence

// from a given array of jobs with deadlines and profits

#include<bits/stdc++.h>

using namespace std;

// A structure to represent various attributes of a Job

struct Job

{

// Each job has id, deadline and profit

char id;

int deadLine, profit;

};

// A Simple Disjoint Set Data Structure

struct DisjointSet

{

int \*parent;

// Constructor

DisjointSet(int n)

{

parent = new int[n+1];

// Every node is a parent of itself

for (int i = 0; i <= n; i++)

parent[i] = i;

}

// Path Compression

int find(int s)

{

/\* Make the parent of the nodes in the path

from u--> parent[u] point to parent[u] \*/

if (s == parent[s])

return s;

return parent[s] = find(parent[s]);

}

// Makes u as parent of v.

void merge(int u, int v)

{

//update the greatest available

//free slot to u

parent[v] = u;

}

};

// Used to sort in descending order on the basis

// of profit for each job

bool cmp(Job a, Job b)

{

return (a.profit > b.profit);

}

// Functions returns the maximum deadline from the set

// of jobs

int findMaxDeadline(struct Job arr[], int n)

{

int ans = INT\_MIN;

for (int i = 0; i < n; i++)

ans = max(ans, arr[i].deadLine);

return ans;

}

int printJobScheduling(Job arr[], int n)

{

// Sort Jobs in descending order on the basis

// of their profit

sort(arr, arr + n, cmp);

// Find the maximum deadline among all jobs and

// create a disjoint set data structure with

// maxDeadline disjoint sets initially.

int maxDeadline = findMaxDeadline(arr, n);

DisjointSet ds(maxDeadline);

// Traverse through all the jobs

for (int i = 0; i < n; i++)

{

// Find the maximum available free slot for

// this job (corresponding to its deadline)

int availableSlot = ds.find(arr[i].deadLine);

// If maximum available free slot is greater

// than 0, then free slot available

if (availableSlot > 0)

{

// This slot is taken by this job 'i'

// so we need to update the greatest

// free slot. Note that, in merge, we

// make first parameter as parent of

// second parameter. So future queries

// for availableSlot will return maximum

// available slot in set of

// "availableSlot - 1"

ds.merge(ds.find(availableSlot - 1),

availableSlot);

cout << arr[i].id << " ";

}

}

}

// Driver code

int main()

{

Job arr[] = { { 'a', 2, 100 }, { 'b', 1, 19 },

{ 'c', 2, 27 }, { 'd', 1, 25 },

{ 'e', 3, 15 } };

int n = sizeof(arr) / sizeof(arr[0]);

cout << "Following jobs need to be "

<< "executed for maximum profit\n";

printJobScheduling(arr, n);

return 0;

}

**Output**

Following jobs need to be executed for maximum profit

a c e

# 221. 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:**If two elements have same frequency, then the element which occur at first will be taken on the left of Binary Tree and other one to the right.

**Example 1:**

S = "abcdef"

f[] = {5, 9, 12, 13, 16, 45}

**Output:**

0 100 101 1100 1101 111

**Explanation:**

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.

**Expected Time complexity:**O(N \* LogN)   
**Expected Space complexity:**O(N)

**Constraints:**  
1 ≤ N ≤ 26

## Solution:

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.  
The variable-length codes assigned to input characters are [Prefix Codes](http://en.wikipedia.org/wiki/Prefix_code), means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.   
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.  
See [this](http://en.wikipedia.org/wiki/Huffman_coding#Applications)for applications of Huffman Coding.   
There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

***Steps to build Huffman Tree***  
Input is an array of unique characters along with their frequency of occurrences and output 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 nodes 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 the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.  
   Let us understand the algorithm with an example:

character Frequency

a 5

b 9

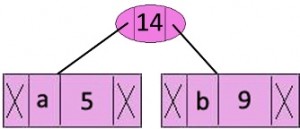
c 12

d 13

e 16

f 45

**Step 1.** Build a min heap that contains 6 nodes where each node represents root of a tree with single node.  
**Step 2** Extract two minimum frequency nodes from min heap. Add a new internal node with frequency 5 + 9 = 14. 



Now min heap contains 5 nodes where 4 nodes are roots of trees with single element each, and one heap node is root of tree with 3 elements

character Frequency

c 12

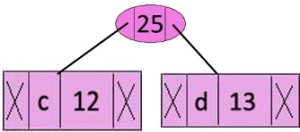
d 13

Internal Node 14

e 16

f 45

**Step 3:** Extract two minimum frequency nodes from heap. Add a new internal node with frequency 12 + 13 = 25



Now min heap contains 4 nodes where 2 nodes are roots of trees with single element each, and two heap nodes are root of tree with more than one nodes

character Frequency

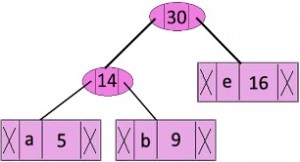
Internal Node 14

e 16

Internal Node 25

f 45

**Step 4:** Extract two minimum frequency nodes. Add a new internal node with frequency 14 + 16 = 30



Now min heap contains 3 nodes.

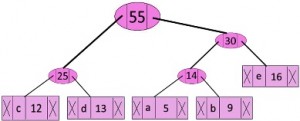
character Frequency

Internal Node 25

Internal Node 30

f 45

**Step 5:** Extract two minimum frequency nodes. Add a new internal node with frequency 25 + 30 = 55



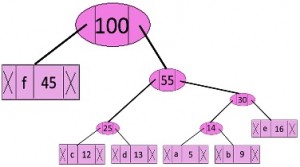
Now min heap contains 2 nodes.

character Frequency

f 45

Internal Node 55

**Step 6:** Extract two minimum frequency nodes. Add a new internal node with frequency 45 + 55 = 100



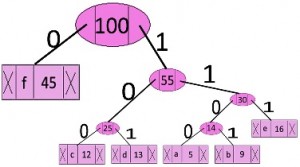
Now min heap contains only one node.

character Frequency

Internal Node 100

Since the heap contains only one node, the algorithm stops here.

***Steps to print codes from Huffman Tree:***  
Traverse the tree formed starting from the root. Maintain an auxiliary array. While moving to the left child, write 0 to the array. While moving to the right child, write 1 to the array. Print the array when a leaf node is encountered.



The codes are as follows:

character code-word

f 0

c 100

d 101

a 1100

b 1101

e 111

Below is the implementation of above approach:

// C program for Huffman Coding

#include <stdio.h>

#include <stdlib.h>

// This constant can be avoided by explicitly

// calculating height of Huffman Tree

#define MAX\_TREE\_HT 100

// A Huffman tree node

struct MinHeapNode {

// One of the input characters

char data;

// Frequency of the character

unsigned freq;

// Left and right child of this node

struct MinHeapNode \*left, \*right;

};

// A Min Heap: Collection of

// min-heap (or Huffman tree) nodes

struct MinHeap {

// Current size of min heap

unsigned size;

// capacity of min heap

unsigned capacity;

// Array of minheap node pointers

struct MinHeapNode\*\* array;

};

// A utility function allocate a new

// min heap node with given character

// and frequency of the character

struct MinHeapNode\* newNode(char data, unsigned freq)

{

struct MinHeapNode\* temp = (struct MinHeapNode\*)malloc(

sizeof(struct MinHeapNode));

temp->left = temp->right = NULL;

temp->data = data;

temp->freq = freq;

return temp;

}

// A utility function to create

// a min heap of given capacity

struct MinHeap\* createMinHeap(unsigned capacity)

{

struct MinHeap\* minHeap

= (struct MinHeap\*)malloc(sizeof(struct MinHeap));

// current size is 0

minHeap->size = 0;

minHeap->capacity = capacity;

minHeap->array = (struct MinHeapNode\*\*)malloc(

minHeap->capacity \* sizeof(struct MinHeapNode\*));

return minHeap;

}

// A utility function to

// swap two min heap nodes

void swapMinHeapNode(struct MinHeapNode\*\* a,

struct MinHeapNode\*\* b)

{

struct MinHeapNode\* t = \*a;

\*a = \*b;

\*b = t;

}

// The standard minHeapify function.

void minHeapify(struct MinHeap\* minHeap, int idx)

{

int smallest = idx;

int left = 2 \* idx + 1;

int right = 2 \* idx + 2;

if (left < minHeap->size

&& minHeap->array[left]->freq

< minHeap->array[smallest]->freq)

smallest = left;

if (right < minHeap->size

&& minHeap->array[right]->freq

< minHeap->array[smallest]->freq)

smallest = right;

if (smallest != idx) {

swapMinHeapNode(&minHeap->array[smallest],

&minHeap->array[idx]);

minHeapify(minHeap, smallest);

}

}

// A utility function to check

// if size of heap is 1 or not

int isSizeOne(struct MinHeap\* minHeap)

{

return (minHeap->size == 1);

}

// A standard function to extract

// minimum value node from heap

struct MinHeapNode\* extractMin(struct MinHeap\* minHeap)

{

struct MinHeapNode\* temp = minHeap->array[0];

minHeap->array[0] = minHeap->array[minHeap->size - 1];

--minHeap->size;

minHeapify(minHeap, 0);

return temp;

}

// A utility function to insert

// a new node to Min Heap

void insertMinHeap(struct MinHeap\* minHeap,

struct MinHeapNode\* minHeapNode)

{

++minHeap->size;

int i = minHeap->size - 1;

while (i

&& minHeapNode->freq

< minHeap->array[(i - 1) / 2]->freq) {

minHeap->array[i] = minHeap->array[(i - 1) / 2];

i = (i - 1) / 2;

}

minHeap->array[i] = minHeapNode;

}

// A standard function to build min heap

void buildMinHeap(struct MinHeap\* minHeap)

{

int n = minHeap->size - 1;

int i;

for (i = (n - 1) / 2; i >= 0; --i)

minHeapify(minHeap, i);

}

// A utility function to print an array of size n

void printArr(int arr[], int n)

{

int i;

for (i = 0; i < n; ++i)

printf("%d", arr[i]);

printf("\n");

}

// Utility function to check if this node is leaf

int isLeaf(struct MinHeapNode\* root)

{

return !(root->left) && !(root->right);

}

// Creates a min heap of capacity

// equal to size and inserts all character of

// data[] in min heap. Initially size of

// min heap is equal to capacity

struct MinHeap\* createAndBuildMinHeap(char data[],

int freq[], int size)

{

struct MinHeap\* minHeap = createMinHeap(size);

for (int i = 0; i < size; ++i)

minHeap->array[i] = newNode(data[i], freq[i]);

minHeap->size = size;

buildMinHeap(minHeap);

return minHeap;

}

// The main function that builds Huffman tree

struct MinHeapNode\* buildHuffmanTree(char data[],

int freq[], int size)

{

struct MinHeapNode \*left, \*right, \*top;

// Step 1: Create a min heap of capacity

// equal to size. Initially, there are

// modes equal to size.

struct MinHeap\* minHeap

= createAndBuildMinHeap(data, freq, size);

// Iterate while size of heap doesn't become 1

while (!isSizeOne(minHeap)) {

// Step 2: Extract the two minimum

// freq items from min heap

left = extractMin(minHeap);

right = extractMin(minHeap);

// Step 3: Create a new internal

// node with frequency equal to the

// sum of the two nodes frequencies.

// Make the two extracted node as

// left and right children of this new node.

// Add this node to the min heap

// '$' is a special value for internal nodes, not

// used

top = newNode('$', left->freq + right->freq);

top->left = left;

top->right = right;

insertMinHeap(minHeap, top);

}

// Step 4: The remaining node is the

// root node and the tree is complete.

return extractMin(minHeap);

}

// Prints huffman codes from the root of Huffman Tree.

// It uses arr[] to store codes

void printCodes(struct MinHeapNode\* root, int arr[],

int top)

{

// Assign 0 to left edge and recur

if (root->left) {

arr[top] = 0;

printCodes(root->left, arr, top + 1);

}

// Assign 1 to right edge and recur

if (root->right) {

arr[top] = 1;

printCodes(root->right, arr, top + 1);

}

// If this is a leaf node, then

// it contains one of the input

// characters, print the character

// and its code from arr[]

if (isLeaf(root)) {

printf("%c: ", root->data);

printArr(arr, top);

}

}

// The main function that builds a

// Huffman Tree and print codes by traversing

// the built Huffman Tree

void HuffmanCodes(char data[], int freq[], int size)

{

// Construct Huffman Tree

struct MinHeapNode\* root

= buildHuffmanTree(data, freq, size);

// Print Huffman codes using

// the Huffman tree built above

int arr[MAX\_TREE\_HT], top = 0;

printCodes(root, arr, top);

}

// Driver code

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;

}

**Output:**

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

***Time complexity:*** O(nlogn) where n is the number of unique characters. If there are n nodes, extractMin() is called 2\*(n – 1) times. extractMin() takes O(logn) time as it calles minHeapify(). So, overall complexity is O(nlogn).  
If the input array is sorted, there exists a linear time algorithm. We will soon be discussing in our next post.

**Applications of Huffman Coding:**

1. They are used for transmitting fax and text.
2. They are used by conventional compression formats like PKZIP, GZIP, etc.

 It is useful in cases where there is a series of frequently occurring characters.

**My Implementation using same approach:**

class Solution{

public:

struct Node{

int freq;

Node \*left, \*right;

};

struct compareFreq{

bool operator()(Node \*a, Node \*b){

return a->freq > b->freq;

}

};

void preorder(Node\* root, string str, vector<string> &res){

if(!root->left && !root->right){

res.push\_back(str);

return;

}

preorder(root->left, str+"0", res);

preorder(root->right, str+"1", res);

}

vector<string> huffmanCodes(string S,vector<int> f,int N)

{

priority\_queue<Node\*, vector<Node\*>, compareFreq> pq;

for(int i=0;i<S.size();i++){

Node\* a = (Node\*)malloc(sizeof(Node));

a->freq = f[i];

a->left = a->right = NULL;

pq.push(a);

}

while(pq.size()>1){

Node \*a = pq.top(); pq.pop();

Node \*b = pq.top(); pq.pop();

Node\* c = (Node\*)malloc(sizeof(Node));

c->freq = a->freq + b->freq;

c->left = a; c->right = b;

pq.push(c);

}

Node\* root = pq.top();

string str = "";

vector<string> res;

preorder(root, str, res);

return res;

}

};

# 222. Water Connection Problem

There are n houses and p water pipes in Geek Colony. Every house has at most one pipe going into it and at most one pipe going out of it. Geek needs to install pairs of tanks and taps in the colony according to the following guidelines.    
1. Every house with one outgoing pipe but no incoming pipe gets a tank on its roof.  
2. Every house with only one incoming and no outgoing pipe gets a tap.  
The Geek council has proposed a network of pipes where connections are denoted by three input values: ai, bi, di denoting the pipe of diameter di from house ai to house bi.  
Find a more efficient way for the construction of this network of pipes. Minimize the diameter of pipes wherever possible.  
**Note**: The generated output will have the following format. The first line will contain t, denoting the total number of pairs of tanks and taps installed. The next t lines contain three integers each: house number of tank, house number of tap, and the minimum diameter of pipe between them.

**Example 1:**

**Input:**

n = 9, p = 6

a[] = {7,5,4,2,9,3}

b[] = {4,9,6,8,7,1}

d[] = {98,72,10,22,17,66}

**Output:**

3

2 8 22

3 1 66

5 6 10

**Explanation:**

Connected components are

***3->1, 5->9->7->4->6 and 2->8***.

Therefore, our answer is **3**

followed by **2 8 22, 3 1 66, 5 6 10.**

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **solve()** which takes an integer n(the number of houses), p(the number of pipes),the array a[] , b[] and d[] (where **d[i]**denoting the diameter of the ith pipe from the house **a[i]** to house **b[i]**) as input parameter and returns the array of pairs of tanks and taps installed i.e ith element of the array contains three integers: house number of tank, house number of tap and the minimum diameter of pipe between them.

**Expected Time Complexity:** O(n+p)  
**Expected Auxiliary Space:** O(n+p)

**Constraints:**  
1<=n<=20  
1<=p<=50  
1<=a[i],b[i]<=20  
1<=d[i]<=100

## Solution:

**Approach:**   
Perform DFS from appropriate houses to find all different connected components. The number of different connected components is our answer t.   
The next t lines of the output are the beginning of the connected component, end of the connected component and the minimum diameter from the start to the end of the connected component in each line.   
Since, tanks can be installed only on the houses having outgoing pipe and no incoming pipe, therefore these are appropriate houses to start DFS from i.e. perform DFS from such unvisited houses.  
Below is the implementation of above approach:

// C++ program to find efficient

// solution for the network

#include <bits/stdc++.h>

using namespace std;

// number of houses and number

// of pipes

int n, p;

// Array rd stores the

// ending vertex of pipe

int rd[1100];

// Array wd stores the value

// of diameters between two pipes

int wt[1100];

// Array cd stores the

// starting end of pipe

int cd[1100];

// Vector a, b, c are used

// to store the final output

vector<int> a;

vector<int> b;

vector<int> c;

int ans;

int dfs(int w)

{

if (cd[w] == 0)

return w;

if (wt[w] < ans)

ans = wt[w];

return dfs(cd[w]);

}

// Function performing calculations.

void solve(int arr[][3])

{

int i = 0;

while (i < p) {

int q = arr[i][0], h = arr[i][1],

t = arr[i][2];

cd[q] = h;

wt[q] = t;

rd[h] = q;

i++;

}

a.clear();

b.clear();

c.clear();

for (int j = 1; j <= n; ++j)

/\*If a pipe has no ending vertex

but has starting vertex i.e is

an outgoing pipe then we need

to start DFS with this vertex.\*/

if (rd[j] == 0 && cd[j]) {

ans = 1000000000;

int w = dfs(j);

// We put the details of component

// in final output array

a.push\_back(j);

b.push\_back(w);

c.push\_back(ans);

}

cout << a.size() << endl;

for (int j = 0; j < a.size(); ++j)

cout << a[j] << " " << b[j]

<< " " << c[j] << endl;

}

// driver function

int main()

{

n = 9, p = 6;

memset(rd, 0, sizeof(rd));

memset(cd, 0, sizeof(cd));

memset(wt, 0, sizeof(wt));

int arr[][3] = { { 7, 4, 98 },

{ 5, 9, 72 },

{ 4, 6, 10 },

{ 2, 8, 22 },

{ 9, 7, 17 },

{ 3, 1, 66 } };

solve(arr);

return 0;

}

**Output:**

3

2 8 22

3 1 66

5 6 10

# 223. Fractional Knapsack Problem

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.

**Example 1:**

**Input:**

N = 3, W = 50

values[] = {60,100,120}

weight[] = {10,20,30}

**Output:**

240.00

**Explanation:**Total maximum value of item

we can have is 240.00 from the given

capacity of sack.

**Example 2:**

**Input:**

N = 2, W = 50

values[] = {60,100}

weight[] = {10,20}

**Output:**

160.00

**Explanation:**

Total maximum value of item

we can have is 160.00 from the given

capacity of sack.

**Your Task** :  
Complete the function ***fractionalKnapsack****()* that receives maximum capacity , array of structure/class and size n and returns a double value representing the maximum value in knapsack.  
**Note:**The details of structure/class is defined in the comments above the given function.

**Expected Time Complexity** : O(NlogN)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 105  
1 <= W <= 105

## Solution:

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.

An **efficient solution** is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can’t add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.  
A simple code with our own comparison function can be written as follows, please see sort function more closely, the third argument to sort function is our comparison function which sorts the item according to value/weight ratio in non-decreasing order.   
After sorting we need to loop over these items and add them in our knapsack satisfying above-mentioned criteria.

Below is the implementation of the above idea:

// C/C++ program to solve fractional Knapsack Problem

#include <bits/stdc++.h>

using namespace std;

// Structure for an item which stores weight and

// corresponding value of Item

struct Item {

int value, weight;

// Constructor

Item(int value, int weight)

{

this->value=value;

this->weight=weight;

}

};

// Comparison function to sort Item according to val/weight

// ratio

bool cmp(struct Item a, struct Item b)

{

double r1 = (double)a.value / (double)a.weight;

double r2 = (double)b.value / (double)b.weight;

return r1 > r2;

}

// Main greedy function to solve problem

double fractionalKnapsack(int W, struct Item arr[], int n)

{

// sorting Item on basis of ratio

sort(arr, arr + n, cmp);

// Uncomment to see new order of Items with their

// ratio

/\*

for (int i = 0; i < n; i++)

{

cout << arr[i].value << " " << arr[i].weight << " :

"

<< ((double)arr[i].value / arr[i].weight) <<

endl;

}

\*/

int curWeight = 0; // Current weight in knapsack

double finalvalue = 0.0; // Result (value in Knapsack)

// Looping through all Items

for (int i = 0; i < n; i++) {

// If adding Item won't overflow, add it completely

if (curWeight + arr[i].weight <= W) {

curWeight += arr[i].weight;

finalvalue += arr[i].value;

}

// If we can't add current Item, add fractional part

// of it

else {

int remain = W - curWeight;

finalvalue += arr[i].value

\* ((double)remain

/ (double)arr[i].weight);

break;

}

}

// Returning final value

return finalvalue;

}

// Driver code

int main()

{

int W = 50; // Weight of knapsack

Item arr[] = { { 60, 10 }, { 100, 20 }, { 120, 30 } };

int n = sizeof(arr) / sizeof(arr[0]);

// Function call

cout << "Maximum value we can obtain = "

<< fractionalKnapsack(W, arr, n);

return 0;

}

**Output**

Maximum value we can obtain = 240

As main time taking step is sorting, the whole problem can be solved in O(n log n) only.

# 224. Choose and swap

You are given a string **s** of lower case english alphabets. You can choose any two characters in the string and replace all the occurences of the first character with the second character and replace all the occurences of the second character with the first character. Your aim is to find the lexicographically smallest string that can be obtained by doing this operation at most once.

**Example 1:**

**Input**:

A = "ccad"

**Output:** "aacd"

**Explanation**:

In ccad, we choose ‘a’ and ‘c’ and after

doing the replacement operation once we get,

aacd and this is the lexicographically

smallest string possible.

**Example 2:**

**Input:**

A = "abba"

**Output:** "abba"

**Explanation:**

In abba, we can get baab after doing the

replacement operation once for ‘a’ and ‘b’

but that is not lexicographically smaller

than abba. So, the answer is abba.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **chooseandswap()** which takes the string A as input parameters and returns the **lexicographically smallest string** that is possible after doing the operation at most once.  
  
**Expected Time Complexity:** O(|A|) length of the string A  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1<= |A| <=105

## Solution:

One was is to generate all possible strings that are possible and then see which among them is the lexicographically smallest. This is a correct solution but something faster can be done.

We can make an array of which stores the left most occurence of each character in the string. Then we traverse from left to right. Whenever we reach a position, we check if there is some character which is smaller than this character and whose left most occurence is to the right of the left most occurence of the present character. If such a character is possible, we take the smallest possible and stop iterating. We now have two characters which need to be interchanged in the string and we can print out the result. If we find nothing after the whole iteration, we just print the original string.

The earlier solution is slower than this by a constant multiple of n.

Implementation:

int left\_most[26];

class Solution{

public:

string chooseandswap(string a){

int n = a.size();

string ans = "";

for(int i = 0; i<=25; i++)

left\_most[i] = -1;

for(int i = 0; i<=n-1; i++)

{

if(left\_most[a[i]-97]==-1)

left\_most[a[i]-97] = i;

}

int j = -1;

int mark = -1;

for(int i = 0; i<=n-2; i++)

{

//if(a[i]>a[i+1])

//{

j = i;

for(int k = 0; k<=25; k++)

{

if(left\_most[k] > left\_most[a[j]-97] && k < (int)(a[j]-97))

{

mark = k;

break;

}

}

//}

if(mark!=-1)

break;

}

// cout<<mark<<endl;

if(mark!=-1)

{

for(int i = 0; i<=n-1; i++)

{

if(a[i]==(char)(mark+97)){

// printf("%c",a[j]);

ans += a[j];

}

else if(a[i]==a[j]){

// printf("%c",mark+97);

ans += ((char)(mark+97));

}

else {

ans += a[i];

}

}

}

else

{

for(int i = 0; i<=n-1; i++)

{

ans += a[i];

}

}

return ans;

}

};

**Time Complexity:** O(|A|) length of the string A  
**Auxiliary Space:** O(1)

**My approach:**

Store all the unique elements of string in sorted order in an ordered set. Then, traverse the string and keep comparing for every element occurred.

string chooseandswap(string a){

set<char> st;

int n = a.size();

for(int i=0;i<n;i++)

if(st.find(a[i])==st.end())

st.insert(a[i]);

int i=0;

auto it = st.begin();

for(;it!=st.end();++it){

if(a[i]!=\*it)

break;

while(i<n && a[i]<=\*it) i++;

}

if(it==st.end())

return a;

int x = a[i], y = \*(it);

for(i=0;i<n;i++){

if(a[i]==x)

a[i] = y;

else if(a[i]==y)

a[i] = x;

}

return a;

}

# 225. Maximum trains for which stoppage can be provided

We are given n-platform and two main running railway track for both direction. Trains which needs to stop at your station must occupy one platform for their stoppage and the trains which need not to stop at your station will run away through either of main track without stopping. Now, each train has three value first arrival time, second departure time and third required platform number. We are given m such trains you have to tell maximum number of train for which you can provide stoppage at your station.

Examples:

Input : n = 3, m = 6

Train no.| Arrival Time |Dept. Time | Platform No.

1 | 10:00 | 10:30 | 1

2 | 10:10 | 10:30 | 1

3 | 10:00 | 10:20 | 2

4 | 10:30 | 12:30 | 2

5 | 12:00 | 12:30 | 3

6 | 09:00 | 10:05 | 1

Output : Maximum Stopped Trains = 5

Explanation : If train no. 1 will left

to go without stoppage then 2 and 6 can

easily be accommodated on platform 1.

And 3 and 4 on platform 2 and 5 on platform 3.

Input : n = 1, m = 3

Train no.|Arrival Time|Dept. Time | Platform No.

1 | 10:00 | 10:30 | 1

2 | 11:10 | 11:30 | 1

3 | 12:00 | 12:20 | 1

Output : Maximum Stopped Trains = 3

Explanation : All three trains can be easily

stopped at platform 1.

## Solution:

If we start with a single platform only then we have 1 platform and some trains with their arrival time and departure time and we have to maximize the number of trains on that platform. This task is similar as [Activity Selection Problem.](https://en.wikipedia.org/wiki/Activity_selection_problem) So, for n platforms we will simply make n-vectors and put the respective trains in those vectors according to platform number. After that by applying greedy approach we easily solve this problem.  
Note : We will take input in form of 4-digit integer for arrival and departure time as 1030 will represent 10:30 so that we may handle the data type easily.  
Also, we will choose a 2-D array for input as arr[m][3] where arr[i][0] denotes arrival time, arr[i][1] denotes departure time and arr[i][2] denotes the platform for ith train.

// CPP to design platform for maximum stoppage

#include <bits/stdc++.h>

using namespace std;

// number of platforms and trains

#define n 2

#define m 5

// function to calculate maximum trains stoppage

int maxStop(int arr[][3])

{

// declaring vector of pairs for platform

vector<pair<int, int> > vect[n + 1];

// Entering values in vector of pairs

// as per platform number

// make departure time first element

// of pair

for (int i = 0; i < m; i++)

vect[arr[i][2]].push\_back(

make\_pair(arr[i][1], arr[i][0]));

// sort trains for each platform as per

// dept. time

for (int i = 0; i <= n; i++)

sort(vect[i].begin(), vect[i].end());

// perform activity selection approach

int count = 0;

for (int i = 0; i <= n; i++) {

if (vect[i].size() == 0)

continue;

// first train for each platform will

// also be selected

int x = 0;

count++;

for (int j = 1; j < vect[i].size(); j++) {

if (vect[i][j].second >=

vect[i][x].first) {

x = j;

count++;

}

}

}

return count;

}

// driver function

int main()

{

int arr[m][3] = { 1000, 1030, 1,

1010, 1020, 1,

1025, 1040, 1,

1130, 1145, 2,

1130, 1140, 2 };

cout << "Maximum Stopped Trains = "

<< maxStop(arr);

return 0;

}

Output:

Maximum Stopped Trains = 3

# 226. Minimum Platforms Problem

Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.  
Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms**.**

**Example 1:**

**Input**: n = 6

arr[] = {0900, 0940, 0950, 1100, 1500, 1800}

dep[] = {0910, 1200, 1120, 1130, 1900, 2000}

**Output**: 3

**Explanation**:

Minimum 3 platforms are required to

safely arrive and depart all trains.

**Example 2:**

**Input**: n = 3

arr[] = {0900, 1100, 1235}

dep[] = {1000, 1200, 1240}

**Output**: 1

**Explanation**: Only 1 platform is required to

safely manage the arrival and departure

of all trains.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findPlatform()** which takes the array arr[] (denoting the arrival times), array dep[] (denoting the departure times) and the size of the array as inputs and returns the minimum number of platforms required at the railway station such that no train waits.

**Note:** Time intervals are in the 24-hour format(**HHMM) ,** where the first two characters represent hour (between 00 to 23 ) and the last two characters represent minutes (this may be > 59).

**Expected Time Complexity:**O(nLogn)  
**Expected Auxiliary Space:**O(n)

**Constraints:**  
1 ≤ n ≤ 50000  
0000 ≤ A[i] ≤ D[i] ≤ 2359

## Solution:

**Naive Solution:**

* **Approach:** The idea is to take every interval one by one and find the number of intervals that overlap with it. Keep track of the maximum number of intervals that overlap with an interval. Finally, return the maximum value.
* **Algorithm:**
  1. Run two nested loops the outer loop from start to end and the inner loop from i+1 to end.
  2. For every iteration of the outer loop find the count of intervals that intersect with the current interval.
  3. Update the answer with the maximum count of overlap in each iteration of the outer loop.
  4. Print the answer.

**Implementation:**

// Program to find minimum number of platforms

// required on a railway station

#include <algorithm>

#include <iostream>

using namespace std;

// Returns minimum number of platforms required

int findPlatform(int arr[], int dep[], int n)

{

// plat\_needed indicates number of platforms

// needed at a time

int plat\_needed = 1, result = 1;

int i = 1, j = 0;

// run a nested loop to find overlap

for (int i = 0; i < n; i++) {

// minimum platform

plat\_needed = 1;

for (int j = i + 1; j < n; j++) {

// check for overlap

if ((arr[i] >= arr[j] && arr[i] <= dep[j]) ||

(arr[j] >= arr[i] && arr[j] <= dep[i]))

plat\_needed++;

}

// update result

result = max(result, plat\_needed);

}

return result;

}

// Driver Code

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 = "

<< findPlatform(arr, dep, n);

return 0;

}

**Output**

Minimum Number of Platforms Required = 3

**Complexity Analysis:**

* **Time Complexity:** O(n^2).   
  Two nested loops traverse the array, so the time complexity is O(n^2).
* **Space Complexity:**O(1).   
  As no extra space is required.

**Efficient Solution:**

* **Approach:** The idea is to consider all events in sorted order. Once the events are in sorted order, trace the number of trains at any time keeping track of trains that have arrived, but not departed.

For example, consider the above example.

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}

**All events are sorted by time.**

Total platforms at any time can be obtained by

subtracting total departures from total arrivals

by that time.

**Time Event Type Total Platforms Needed**

**at this Time**

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 Platforms needed on railway station

= Maximum platforms needed at any time

= 3

**Note:**This approach assumes that trains are arriving and departing on the same date. 

**Algorithm:**

1. Sort the arrival and departure times of trains.
2. Create two pointers i=0, and j=0 and a variable to store *ans* and current count *plat*
3. Run a loop while i<n and j<n and compare the ith element of arrival array and jth element of departure array.
4. If the arrival time is less than or equal to departure then one more platform is needed so increase the count, i.e. plat++ and increment i
5. Else if the arrival time greater than departure then one less platform is needed so decrease the count, i.e. plat– and increment j
6. Update the ans, i.e ans = max(ans, plat).

**Implementation:** This 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](http://geeksquiz.com/merge-sort/) to process them together as a single sorted array.

// Program to find minimum number of platforms

// required on a railway station

#include <algorithm>

#include <iostream>

using namespace std;

// Returns minimum number of platforms required

int findPlatform(int arr[], int dep[], int n)

{

// Sort arrival and departure arrays

sort(arr, arr + n);

sort(dep, dep + n);

// plat\_needed indicates number of platforms

// needed at a time

int plat\_needed = 1, result = 1;

int i = 1, j = 0;

// Similar to merge in merge sort to process

// all events in sorted order

while (i < n && j < n) {

// If next event in sorted order is arrival,

// increment count of platforms needed

if (arr[i] <= dep[j]) {

plat\_needed++;

i++;

}

// Else decrement count of platforms needed

else if (arr[i] > dep[j]) {

plat\_needed--;

j++;

}

// Update result if needed

if (plat\_needed > result)

result = plat\_needed;

}

return result;

}

// Driver code

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 = "

<< findPlatform(arr, dep, n);

return 0;

}

**Output**

Minimum Number of Platforms Required = 3

**Complexity Analysis:**

* **Time Complexity:** O(N \* log N).   
  One traversal O(n) of both the array is needed after sorting O(N \* log N), so the time complexity is O(N \* log N).
* **Space Complexity:**O(1).   
  As no extra space is required.

**O(n) time solution using dynamic programming:**

Use dynamic programming. Create an array of the maximum time possible and initialize all values by 0. Traverse trains and increment value by 1 for each arrival time and decrement by 1 for each departure time. Traverse the array and store sum at each time in a variable and note the maximum possible sum at any point.

// C++ program to find minimum number of platforms

// required on a railway station

#include <bits/stdc++.h>

using namespace std;

int minPlatform(int arrival[], int departure[], int n)

{

// as time range from 0 to 2359 in 24 hour clock,

// we declare an array for values from 0 to 2360

int platform[2361] = {0};

int requiredPlatform = 1;

for (int i = 0; i < n; i++) {

// increment the platforms for arrival

++platform[arrival[i]];

// once train departs we decrease the platform count

--platform[departure[i] + 1];

//we have done +1 in departure time because same arrival time and departure //time are not possible at any instant of time

}

// We are running loop till 2361 because maximum time value

// in a day can be 23:60

for (int i = 1; i < 2361; i++) {

// taking cumulative sum of platform give us required

// number of platform fro every minute

platform[i] = platform[i] + platform[i - 1];

requiredPlatform = max(requiredPlatform, platform[i]);

}

return requiredPlatform;

}

// Driver code

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 = "

<< minPlatform(arr, dep, n);

return 0;

}

**Output:**

Minimum Number of Platforms Required = 3

**Complexity Analysis:**

* **Time Complexity:**O(N).
* **Space Complexity:** O(1).

# 227. Buy Maximum Stocks if i stocks can be bought on i-th day

In a stock market, there is a product with its infinite stocks. The stock prices are given for **N** days, where arr[i] denotes the price of the stock on the ith day. There is a rule that a customer can buy at most i stock on the ith day. If the customer has an amount of **k** amount of money initially, find out the maximum number of stocks a customer can buy.

For example, for 3 days the price of a stock is given as 7, 10, 4. You can buy 1 stock worth 7 rs on day 1, 2 stocks worth 10 rs each on day 2 and 3 stock worth 4 rs each on day 3.

Examples:

Input : price[] = { 10, 7, 19 },

k = 45.

Output : 4

A customer purchases 1 stock on day 1,

2 stocks on day 2 and 1 stock on day 3 for

10, 7 \* 2 = 14 and 19 respectively. Hence,

total amount is 10 + 14 + 19 = 43 and number

of stocks purchased is 4.

Input : price[] = { 7, 10, 4 },

k = 100.

Output : 6

## Solution:

The idea is to use greedy approach, where we should start buying product from the day when the stock price is least and so on.   
So, we will sort the pair of two values i.e { stock price, day } according to the stock price, and if stock prices are same, then we sort according to the day.   
Now, we will traverse along the sorted list of pairs, and start buying as follows:   
Let say, we have R rs remaining till now, and the cost of product on this day be C, and we can buy atmost L products on this day then,   
total purchase on this day (P) = min(L, R/C)   
Now, add this value to the answer.   
total purchase on this day (P) = min(L, R/C)   
Now, add this value to the answer   
total\_purchase = total\_purchase + P, where P =min(L, R/C)   
Now, subtract the cost of buying P items from remaining money, R = R – P\*C.   
Total number of products that we can buy is equal to total\_purchase.

Below is the implementation of this approach:

// C++ program to find maximum number of stocks that

// can be bought with given constraints.

#include <bits/stdc++.h>

using namespace std;

// Return the maximum stocks

int buyMaximumProducts(int n, int k, int price[])

{

vector<pair<int, int> > v;

// Making pair of product cost and number

// of day..

for (int i = 0; i < n; ++i)

v.push\_back(make\_pair(price[i], i + 1));

// Sorting the vector pair.

sort(v.begin(), v.end());

// Calculating the maximum number of stock

// count.

int ans = 0;

for (int i = 0; i < n; ++i) {

ans += min(v[i].second, k / v[i].first);

k -= v[i].first \* min(v[i].second,

(k / v[i].first));

}

return ans;

}

// Driven Program

int main()

{

int price[] = { 10, 7, 19 };

int n = sizeof(price)/sizeof(price[0]);

int k = 45;

cout << buyMaximumProducts(n, k, price) << endl;

return 0;

}

**Output:**

4

**Time Complexity :**O(nlogn).

# 228. 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 to you.  
You are now provided with an attractive offer.  
You can buy a single candy from the store and get at most **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.

**Example 2:**

**Input:**

N = 5

K = 4

candies[] = {3 2 1 4 5}

**Output:**

1 5

**Explanation:**

For minimimum cost buy the candy with

the cost 1 and get all the other candies

for free.

For maximum cost buy the candy with

the cost 5 and get all other candies

for free.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **candyStore()** which takes the array candies[], its size Nand an integer Kas input parameters and returns the minimum amount and maximum amount of money to buy all candies according to the offer.  
  
**Expected Time Complexity:** O(NLogN)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 <= **N**<= 100000  
 0 <= **K** <= N-1  
1 <= **candies[i]** <= 10000

## Solution:

One important thing to note is, we must use the offer and get maximum candies back for every candy purchase. So if we want to minimize the money, we must buy candies at minimum cost and get candies of maximum costs for free. To maximize the money, we must do the reverse. Below is an algorithm based on this.

First Sort the price array.

**For finding minimum amount :**

Start purchasing candies from starting

and reduce k free candies from last with

every single purchase.

**For finding maximum amount :**

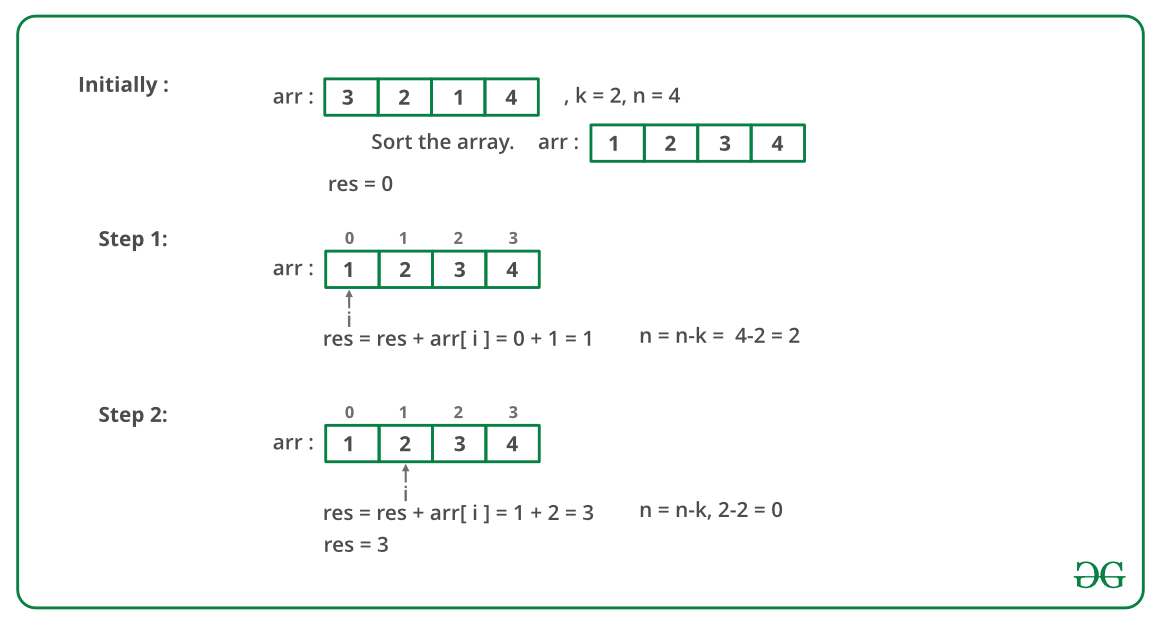
Start purchasing candies from the end

and reduce k free candies from starting

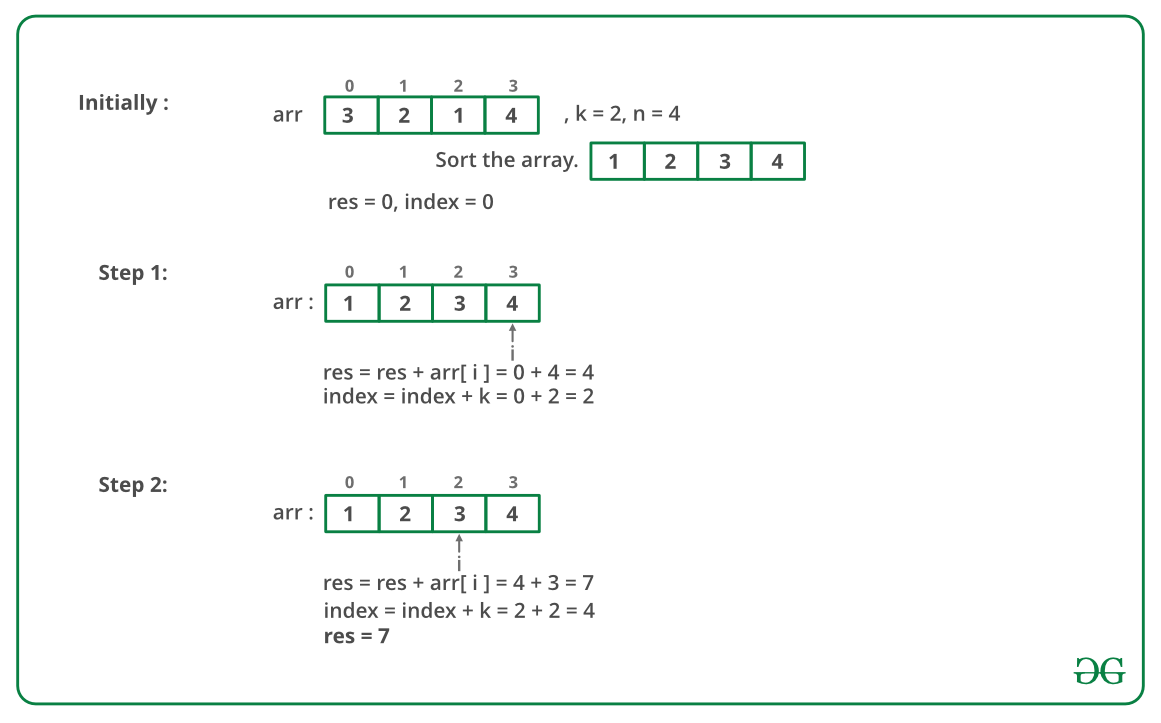
in every single purchase.

Below image is an illustration of the above approach:

Minimum amount :



Maximum amount :



Below is the implementation of the above approach:

// C++ implementation to find the minimum

// and maximum amount

#include <bits/stdc++.h>

using namespace std;

// Function to find the minimum amount

// to buy all candies

int findMinimum(int arr[], int n, int k)

{

int res = 0;

for (int i = 0; i < n; i++) {

// Buy current candy

res += arr[i];

// And take k candies for free

// from the last

n = n - k;

}

return res;

}

// Function to find the maximum amount

// to buy all candies

int findMaximum(int arr[], int n, int k)

{

int res = 0, index = 0;

for (int i = n - 1; i >= index; i--)

{

// Buy candy with maximum amount

res += arr[i];

// And get k candies for free from

// the starting

index += k;

}

return res;

}

// Driver code

int main()

{

int arr[] = { 3, 2, 1, 4 };

int n = sizeof(arr) / sizeof(arr[0]);

int k = 2;

sort(arr, arr + n);

// Function call

cout << findMinimum(arr, n, k) << " "

<< findMaximum(arr, n, k) << endl;

return 0;

}

**Output**

3 7

**Time Complexity:** O(n log n)

**Another Implementation:**

Sort the candies array. Divide N by K+1 and we will get number of candies need to be bought. Then, for minimum cost, choose candies with less cost and for maximum cost, choose candies with high cost.

vector<int> candyStore(int candies[], int N, int K)

{

K++;

int buy = N/K;

if(N%K>0)

buy++;

sort(candies, candies+N);

vector<int> res;

res.push\_back(0);

res.push\_back(0);

for(int i=0;i<buy;i++)

res[0] += candies[i];

for(int i=N-1;i>=N-buy;i--)

res[1] += candies[i];

return res;

}

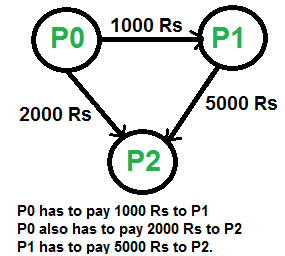
**Time Complexity:** O(n log n)

# 229. Minimize Cash Flow among a given set of friends who have borrowed money from each other

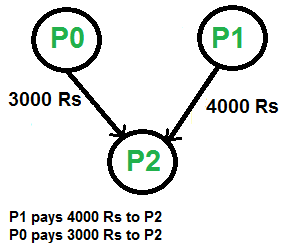
Given a number of friends who have to give or take some amount of money from one another. Design an algorithm by which the total cash flow among all the friends is minimized.

Example:

Following diagram shows input debts to be settled.



Above debts can be settled in following optimized way  



## Solution:

The idea is to use [Greedy algorithm](https://www.geeksforgeeks.org/tag/Greedy-Algorithm/) where at every step, settle all amounts of one person and recur for remaining n-1 persons.   
How to pick the first person? To pick the first person, calculate the net amount for every person where net amount is obtained by subtracting all debts (amounts to pay) from all credits (amounts to be paid). Once net amount for every person is evaluated, find two persons with maximum and minimum net amounts. These two persons are the most creditors and debtors. The person with minimum of two is our first person to be settled and removed from list. Let the minimum of two amounts be x. We pay ‘x’ amount from the maximum debtor to maximum creditor and settle one person. If x is equal to the maximum debit, then maximum debtor is settled, else maximum creditor is settled.  
The following is detailed algorithm.

Do following for every person Pi where i is from 0 to n-1.

1. Compute the net amount for every person. The net amount for person ‘i’ can be computed by subtracting sum of all debts from sum of all credits.
2. Find the two persons that are maximum creditor and maximum debtor. Let the maximum amount to be credited maximum creditor be maxCredit and maximum amount to be debited from maximum debtor be maxDebit. Let the maximum debtor be Pd and maximum creditor be Pc.
3. Find the minimum of maxDebit and maxCredit. Let minimum of two be x. Debit ‘x’ from Pd and credit this amount to Pc
4. If x is equal to maxCredit, then remove Pc from set of persons and recur for remaining (n-1) persons.
5. If x is equal to maxDebit, then remove Pd from set of persons and recur for remaining (n-1) persons.

The following is the implementation of the above algorithm.

// C++ program to fin maximum cash flow among a set of persons

#include<iostream>

using namespace std;

// Number of persons (or vertices in the graph)

#define N 3

// A utility function that returns index of minimum value in arr[]

int getMin(int arr[])

{

int minInd = 0;

for (int i=1; i<N; i++)

if (arr[i] < arr[minInd])

minInd = i;

return minInd;

}

// A utility function that returns index of maximum value in arr[]

int getMax(int arr[])

{

int maxInd = 0;

for (int i=1; i<N; i++)

if (arr[i] > arr[maxInd])

maxInd = i;

return maxInd;

}

// A utility function to return minimum of 2 values

int minOf2(int x, int y)

{

return (x<y)? x: y;

}

// amount[p] indicates the net amount to be credited/debited

// to/from person 'p'

// If amount[p] is positive, then i'th person will amount[i]

// If amount[p] is negative, then i'th person will give -amount[i]

void minCashFlowRec(int amount[])

{

// Find the indexes of minimum and maximum values in amount[]

// amount[mxCredit] indicates the maximum amount to be given

// (or credited) to any person .

// And amount[mxDebit] indicates the maximum amount to be taken

// (or debited) from any person.

// So if there is a positive value in amount[], then there must

// be a negative value

int mxCredit = getMax(amount), mxDebit = getMin(amount);

// If both amounts are 0, then all amounts are settled

if (amount[mxCredit] == 0 && amount[mxDebit] == 0)

return;

// Find the minimum of two amounts

int min = minOf2(-amount[mxDebit], amount[mxCredit]);

amount[mxCredit] -= min;

amount[mxDebit] += min;

// If minimum is the maximum amount to be

cout << "Person " << mxDebit << " pays " << min

<< " to " << "Person " << mxCredit << endl;

// Recur for the amount array. Note that it is guaranteed that

// the recursion would terminate as either amount[mxCredit]

// or amount[mxDebit] becomes 0

minCashFlowRec(amount);

}

// Given a set of persons as graph[] where graph[i][j] indicates

// the amount that person i needs to pay person j, this function

// finds and prints the minimum cash flow to settle all debts.

void minCashFlow(int graph[][N])

{

// Create an array amount[], initialize all value in it as 0.

int amount[N] = {0};

// Calculate the net amount to be paid to person 'p', and

// stores it in amount[p]. The value of amount[p] can be

// calculated by subtracting debts of 'p' from credits of 'p'

for (int p=0; p<N; p++)

for (int i=0; i<N; i++)

amount[p] += (graph[i][p] - graph[p][i]);

minCashFlowRec(amount);

}

// Driver program to test above function

int main()

{

// graph[i][j] indicates the amount that person i needs to

// pay person j

int graph[N][N] = { {0, 1000, 2000},

{0, 0, 5000},

{0, 0, 0},};

// Print the solution

minCashFlow(graph);

return 0;

}

**Output:**

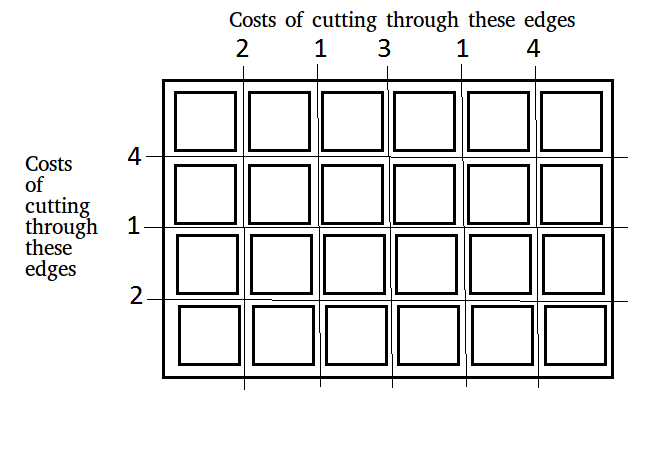
Person 1 pays 4000 to Person 2

Person 0 pays 3000 to Person 2

**Algorithmic Paradigm:** Greedy   
**Time Complexity:** O(N2) where N is the number of persons.

# 230. Minimum Cost to cut a board into squares

A board of length m and width n is 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.   
Examples: 



For above board optimal way to cut into square is:

Total minimum cost in above case is 42. It is

evaluated using following steps.

Initial Value : Total\_cost = 0

Total\_cost = Total\_cost + edge\_cost \* total\_pieces

Cost 4 Horizontal cut Cost = 0 + 4\*1 = 4

Cost 4 Vertical cut Cost = 4 + 4\*2 = 12

Cost 3 Vertical cut Cost = 12 + 3\*2 = 18

Cost 2 Horizontal cut Cost = 18 + 2\*3 = 24

Cost 2 Vertical cut Cost = 24 + 2\*3 = 30

Cost 1 Horizontal cut Cost = 30 + 1\*4 = 34

Cost 1 Vertical cut Cost = 34 + 1\*4 = 38

Cost 1 Vertical cut Cost = 38 + 1\*4 = 42

## Solution:

This problem can be solved using greedy approach, If total cost is denoted by S, then S = a1w1 + a2w2 … + akwk, where wi is a cost of certain edge cutting and ai is corresponding coefficient, The coefficient ai is determined by the total number of cuts we have competed using edge wi at the end of the cutting process. Notice that sum of the coefficients is always constant, hence we want to find a distribution of ai obtainable such that S is minimum. To do so **we perform cuts on highest cost edge as early as possible**, which will reach to optimal S. If we encounter several edges having the same cost, we can cut any one of them first.   
Below is the solution using above approach, first we sorted the edge cutting costs in reverse order, then we loop in them from higher-cost to lower-cost building our solution. Each time we choose an edge, counterpart count is incremented by 1, which is to be multiplied each time with corresponding edge cutting cost.   
Notice below used sort method, sending greater() as 3rd argument to sort method sorts number in non-increasing order, it is predefined function of the library.

// C++ program to divide a board into m\*n squares

#include <bits/stdc++.h>

using namespace std;

// method returns minimum cost to break board into

// m\*n squares

int minimumCostOfBreaking(int X[], int Y[], int m, int n)

{

int res = 0;

// sort the horizontal cost in reverse order

sort(X, X + m, greater<int>());

// sort the vertical cost in reverse order

sort(Y, Y + n, greater<int>());

// initialize current width as 1

int hzntl = 1, vert = 1;

// loop until one or both cost array are processed

int i = 0, j = 0;

while (i < m && j < n)

{

if (X[i] > Y[j])

{

res += X[i] \* vert;

// increase current horizontal part count by 1

hzntl++;

i++;

}

else

{

res += Y[j] \* hzntl;

// increase current vertical part count by 1

vert++;

j++;

}

}

// loop for horizontal array, if remains

int total = 0;

while (i < m)

total += X[i++];

res += total \* vert;

// loop for vertical array, if remains

total = 0;

while (j < n)

total += Y[j++];

res += total \* hzntl;

return res;

}

// Driver code to test above methods

int main()

{

int m = 6, n = 4;

int X[m-1] = {2, 1, 3, 1, 4};

int Y[n-1] = {4, 1, 2};

cout << minimumCostOfBreaking(X, Y, m-1, n-1);

return 0;

}

**Output:**

42

# 231. Check if it is possible to survive on Island

Ishika 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

**Explaination:** 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.

**Example 2:**

**Input:** S = 10, N = 20, M = 30

**Output:** -1

**Explaination:** She can’t survive even if

she buy food because the maximum number

of units she can buy in 1 day is less the

required food for 1 day.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **minimumDays()** which takes S, N, and M as input parameters and returns the minimum number of days Ishika needs to buy food. Otherwise, returns -1 if she cannot survive.

**Expected Time Complexity:** O(1)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N, S ≤ 50  
1 ≤ M ≤ 30

## Solution:

**Approach:**  
In this problem, the greedy approach of buying the food for some consecutive early days is the right direction.   
If we can survive for the first 7 days then we can survive any number of days for that we need to check two things   
-> Check whether we can survive one day or not.   
-> (S >= 7) If we buy food in the first 6 days of the week and we can survive for the week i.e. total food we can buy in a week (6\*N) is greater then or equal to total food we require to survive in a week (7\*M) then we can survive.

**Note : We are buying the food in the first 6 days because we are counting from Monday and the shop will remain close on Sunday.**  
If any of the above conditions are not true then we can’t survive else the minimum number of days required to buy food will be = ceil(total food required/units of food we can buy each day).

// C++ program to find the minimum days on which

// you need to buy food from the shop so that you

// can survive the next S days

#include <bits/stdc++.h>

using namespace std;

// function to find the minimum days

void survival(int S, int N, int M)

{

// If we can not buy at least a week

// supply of food during the first week

// OR We can not buy a day supply of food

// on the first day then we can't survive.

if (((N \* 6) < (M \* 7) && S > 6) || M > N)

cout << "No\n";

else {

// If we can survive then we can

// buy ceil(A/N) times where A is

// total units of food required.

int days = (M \* S) / N;

if (((M \* S) % N) != 0)

days++;

cout << "Yes " << days << endl;

}

}

// Driver code

int main()

{

int S = 10, N = 16, M = 2;

survival(S, N, M);

return 0;

}

**Output:**

Yes 2

**Time Complexity:** O(1)   
**Space Complexity:** O(1)

# 232. Find maximum meetings in one room

**N meetings in one room :**

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)

**Example 2:**

**Input:**

**N** = 3

**start[]** = {10, 12, 20}

**end[]** = {20, 25, 30}

**Output:**

1

**Explanation:**

Only one meetings can be held

with given start and end timings.

**Your Task** :  
You don't need to read inputs or print anything. Complete the function **maxMeetings()**that takes two arrays **start[]**and **end[]**along with their size **N** as input parameters and returns the **maximum** number of meetings that can be held in the meeting room.

**Expected Time Complexity**: O(N\*LogN)  
**Expected Auxilliary Space** : O(N)

**Constraints:**  
1 ≤ N ≤ 105  
0 ≤ **star**t**[i]** < **end[i]** ≤ 105

## Solution:

The 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 activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.  
1) Sort the activities according to their finishing time. While sorting, remember that start time will also change its position according to end time so that pair of start time and end time of a particular activity remain at the same index i.    
2) Select the first activity from the sorted array and print it.   
3) Do the following for the remaining activities in the sorted array.   
…….a) If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it.  
In the following C implementation, it is assumed that the activities are already sorted according to their finish time.

int maxMeetings(int start[], int end[], int n)

{

int count = 0;

if(n>0){

MergeSort(start,end,0,n-1);

count = 1;

int last\_taken = 0;

for(int i=1;i<n;i++){

if(start[i]>end[last\_taken]){

count++;

last\_taken = i;

}

}

}

return count;

}

void MergeSort(int start[], int end[], int i, int j){

if(i<j){

int mid = (i+j)/2;

MergeSort(start,end,i,mid);

MergeSort(start,end,mid+1,j);

Merge(start,end,i,mid,j);

}

}

void Merge(int start[],int end[],int i,int mid,int j){

int n1 = mid-i+1, n2 = j-mid;

int left1[n1], left2[n1], right1[n2], right2[n2];

for(int p=0;p<n1;p++)

left1[p] = start[p+i];

for(int p=0;p<n2;p++)

right1[p] = start[p+mid+1];

for(int p=0;p<n1;p++)

left2[p] = end[p+i];

for(int p=0;p<n2;p++)

right2[p] = end[p+mid+1];

int p=0,q=0,k=0;

while(p<n1 && q<n2){

if(left2[p]<=right2[q]){

end[k+i] = left2[p];

start[k+i] = left1[p];

p++;

k++;

}

else{

end[k+i] = right2[q];

start[k+i] = right1[q];

q++;

k++;

}

}

while(p<n1){

end[k+i] = left2[p];

start[k+i] = left1[p];

p++;

k++;

}

while(q<n2){

end[k+i] = right2[q];

start[k+i] = right1[q];

q++;

k++;

}

}

**Time Complexity:** It takes O(n log n) time if input activities may not be sorted. It takes O(n) time when it is given that input activities are always sorted.

**Space Complexity:** Since, we have chosen Merge sort for sorting so it will take O(n) time complexity.

**Using STL:**

// C++ program to find maximum number of meetings

#include <bits/stdc++.h>

using namespace std;

// Structure for storing starting time,

// finishing time and position of meeting.

struct meeting {

int start;

int end;

int pos;

};

// Comparator function which can compare

// the second element of structure used to

// sort pairs in increasing order of second value.

bool comparator(struct meeting m1, meeting m2)

{

return (m1.end < m2.end);

}

// Function for finding maximum meeting in one room

void maxMeeting(int s[], int f[], int n)

{

struct meeting meet[n];

for (int i = 0; i < n; i++)

{

// Starting time of meeting i.

meet[i].start = s[i];

// Finishing time of meeting i

meet[i].end = f[i];

// Original position/index of meeting

meet[i].pos = i + 1;

}

// Sorting of meeting according to their finish time.

sort(meet, meet + n, comparator);

// Vector for storing selected meeting.

vector<int> m;

// Initially select first meeting.

m.push\_back(meet[0].pos);

// time\_limit to check whether new

// meeting can be conducted or not.

int time\_limit = meet[0].end;

// Check for all meeting whether it

// can be selected or not.

for (int i = 1; i < n; i++) {

if (meet[i].start >= time\_limit)

{

// Push selected meeting to vector

m.push\_back(meet[i].pos);

// Update time limit.

time\_limit = meet[i].end;

}

}

// Print final selected meetings.

for (int i = 0; i < m.size(); i++) {

cout << m[i] << " ";

}

}

// Driver code

int main()

{

// Starting time

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

// Finish time

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

// Number of meetings.

int n = sizeof(s) / sizeof(s[0]);

// Function call

maxMeeting(s, f, n);

return 0;

}

**Output:**

1 2 4 5

**Time Complexity:** O(N log(N))

# 233. 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

**Explanation :** Maximum product will be ( -2 \* -1 \* 4 \* 3 ) = 24

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

**Output:** 0

**Explanation:** 0(single element) is maximum product possible

**Input:** a[] = { 0, 0, 0 }

**Output:** 0

## Solution:

A **simple solution** is to [generate all subsets](https://www.geeksforgeeks.org/power-set/), find product of every subset and return maximum product.  
A **better solution** is to use the below facts.

1. If there are even number of negative numbers and no zeros, result is simply product of all
2. If there are odd number 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 these zeros with one exceptional case. The exceptional case is when there is one negative number and all other elements are 0. In this case, result is 0.

Below is the implementation of the above approach:

// CPP program to find maximum product of

// a subset.

#include <bits/stdc++.h>

using namespace std;

int maxProductSubset(int a[], int n)

{

if (n == 1)

return a[0];

// Find count of negative numbers, count

// of zeros, negative number

// with least absolute value

// and product of non-zero numbers

int max\_neg = INT\_MIN;

int count\_neg = 0, count\_zero = 0;

int prod = 1;

for (int i = 0; i < n; i++) {

// If number is 0, we don't

// multiply it with product.

if (a[i] == 0) {

count\_zero++;

continue;

}

// Count negatives and keep

// track of negative number

// with least absolute value

if (a[i] < 0) {

count\_neg++;

max\_neg = max(max\_neg, a[i]);

}

prod = prod \* a[i];

}

// If there are all zeros

if (count\_zero == n)

return 0;

// If there are odd number of

// negative numbers

if (count\_neg & 1) {

// Exceptional case: There is only

// negative and all other are zeros

if (count\_neg == 1 &&

count\_zero > 0 &&

count\_zero + count\_neg == n)

return 0;

// Otherwise result is product of

// all non-zeros divided by

//negative number with

// least absolute value

prod = prod / max\_neg;

}

return prod;

}

// Driver Code

int main()

{

int a[] = { -1, -1, -2, 4, 3 };

int n = sizeof(a) / sizeof(a[0]);

cout << maxProductSubset(a, n);

return 0;

}

**Output**

24

***Time Complexity:****O(n)*  
***Auxiliary Space:****O(1)*

# 234. Maximize array sum after K negations

Given an array of integers of size **N** and a number **K**., Your must modify array **arr[]** exactly **K** number of times. Here modify array means in each operation you can replace any array element either **arr[i]**by**-arr[i]** or **-arr[i]**by**arr[i]**. You need to perform this operation in such a way that after K operations, the sum of the array must be maximum.

**Example 1:**

**Input:**

N = 5, K = 1

arr[] = {1, 2, -3, 4, 5}

**Output:**

15

**Explanation:**

We have k=1 so we can change -3 to 3 and

sum all the elements to produce 15 as output.

**Example 2:**

**Input:**

N = 10, K = 5

arr[] = {5, -2, 5, -4, 5, -12, 5, 5, 5, 20}

**Output:**

68

**Explanation:**

Here we have k=5 so we turn -2, -4, -12 to

2, 4, and 12 respectively. Since we have

performed 3 operations so k is now 2. To get

maximum sum of array we can turn positive

turned 2 into negative and then positive

again so k is 0. Now sum is

5+5+4+5+12+5+5+5+20+2 = 68

**Your Task:**  
You **don't** have to print anything, print ting is done by the driver code itself. You have to complete the function **maximizeSum()** which takes the array **A[]**, its size **N,**and an integer **K**as inputs and returns the maximum possible sum.

**Expected Time Complexity:**O(N\*logN)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N,K ≤ 105  
-109 ≤ Ai ≤ 109

## Solution:

This problem has very **simple solution**, we just have to replace the minimum element arr[i] in array by -arr[i] for current operation. In this way we can make sum of array maximum after K operations. One interesting case is, once the minimum element becomes 0, we don’t need to make any more changes.

// C++ program to maximize array sum after

// k operations.

#include <bits/stdc++.h>

using namespace std;

// This function does k operations on array

// in a way that maximize the array sum.

// index --> stores the index of current minimum

// element for j'th operation

int maximumSum(int arr[], int n, int k)

{

// Modify array K number of times

for (int i = 1; i <= k; i++)

{

int min = INT\_MAX;

int index = -1;

// Find minimum element in array for

// current operation and modify it

// i.e; arr[j] --> -arr[j]

for (int j = 0; j < n; j++)

{

if (arr[j] < min) {

min = arr[j];

index = j;

}

}

// this the condition if we find 0 as

// minimum element, so it will useless to

// replace 0 by -(0) for remaining operations

if (min == 0)

break;

// Modify element of array

arr[index] = -arr[index];

}

// Calculate sum of array

int sum = 0;

for (int i = 0; i < n; i++)

sum += arr[i];

return sum;

}

// Driver code

int main()

{

int arr[] = { -2, 0, 5, -1, 2 };

int k = 4;

int n = sizeof(arr) / sizeof(arr[0]);

cout << maximumSum(arr, n, k);

return 0;

}

**Output**

10

**Time Complexity:** O(k\*n)   
**Auxiliary Space:**O(1)

**Approach 2(Using Sort):**

1. Sort the array in ascending order. Initialize i = 0.
2. Increment i and multiply all negative elements by -1 till k becomes or a positive element is reached.
3. Check if the end of the array has occurred. If true then go to (n-1)th element.
4. If k ==0 or k is even, return the sum of all elements. Else multiply the absolute of minimum of ith or (i-1) th element by -1.
5. Return sum of the array.

Below is the implementation of the above approach:

// C++ program for the above approach

#include <algorithm>

#include <iostream>

using namespace std;

// Function to calculate sum of the array

long long int sumArray(long long int\* arr, int n)

{

long long int sum = 0;

// Iterate from 0 to n - 1

for (int i = 0; i < n; i++) {

sum += arr[i];

}

return sum;

}

// Function to maximize sum

long long int maximizeSum(long long int arr[], int n, int k)

{

sort(arr, arr + n);

int i = 0;

// Iterate from 0 to n - 1

for (i = 0; i < n; i++) {

if (k && arr[i] < 0) {

arr[i] \*= -1;

k--;

continue;

}

break;

}

if (i == n)

i--;

if (k == 0 || k % 2 == 0) {

return sumArray(arr, n);

}

if (i != 0 && abs(arr[i]) >= abs(arr[i - 1])) {

i--;

}

arr[i] \*= -1;

return sumArray(arr, n);

}

// Driver Code

int main()

{

int n = 5;

int k = 4;

long long int arr[5] = { -3, -2, -1, 5, 6 };

// Function Call

cout << maximizeSum(arr, n, k) << endl;

return 0;

}

**Output**

15

**Time Complexity:**O(n\*logn)

**Auxiliary Space:** O(1)

# 235. Maximize the sum of arr[i]\*i

Given an array **A** of **N** integers. Your task is to write a program to find the maximum value of **∑arr[i]\*i**, where i = 0, 1, 2,…., n – 1.  
You are allowed to rearrange the elements of the array.  
**Note**: Since output could be large, hence module 109+7 and then print answer.

**Example 1:**

**Input :** Arr[] = {5, 3, 2, 4, 1}

**Output :** 40

**Explanation:**

If we arrange the array as 1 2 3 4 5 then

we can see that the minimum index will multiply

with minimum number and maximum index will

multiply with maximum number.

So 1\*0+2\*1+3\*2+4\*3+5\*4=0+2+6+12+20 = 40 mod(109+7) = 40

**Example 2:**

**Input :** Arr[] = {1, 2, 3}

**Output :** 8

**Your Task:**  
This is a function problem. The input is already taken care of by the driver code. You only need to complete the function **Maximize()**that takes an *array****(arr)*,***sizeOfArray****(n)*,** and return the maximum value of an array. The**driver code**takes care of the **printing**.

**Expected Time Complexity:** O(nlog(n)).  
**Expected Auxiliary Space:** O(1).

**Constraints:**  
1 ≤ N ≤ 107  
1 ≤ Ai ≤ N

## Solution:

A **simple solution** is to [generate all permutations of given array](https://www.geeksforgeeks.org/write-a-c-program-to-print-all-permutations-of-a-given-string/). For every permutation, compute the value of Σarr[i]\*i and finally return the maximum value.  
An **efficient solution** is based on the fact that the largest value should be scaled maximum and the smallest value should be scaled minimum. So we multiply the minimum value of i with the minimum value of arr[i]. So, sort the given array in increasing order and compute the sum of arr[i]\*i, where i = 0 to n-1.  
Below is the implementation of this approach:

int Maximize(int a[],int n)

{

int mod = 1e9+7;

sort(a,a+n);

unsigned long long int ans = 0;

for(int i=0;i<n;i++)

ans = (ans + ((unsigned long long)a[i]\*i)%mod )%mod;

return ans;

}

Time Complexity : O(n Log n)

# 236. Maximum sum of absolute difference of an array

Given an array, we need to find the maximum sum of absolute difference of any permutation of the given array.  
**Examples:** 

Input : { 1, 2, 4, 8 }

Output : 18

**Explanation :** For the given array there are

several sequence possible

like : {2, 1, 4, 8}

{4, 2, 1, 8} and some more.

Now, the absolute difference of an array sequence will be

like for this array sequence {1, 2, 4, 8}, the absolute

difference sum is

= |1-2| + |2-4| + |4-8| + |8-1|

= 14

For the given array, we get the maximum value for

the sequence {1, 8, 2, 4}

= |1-8| + |8-2| + |2-4| + |4-1|

= 18

## Solution:

To solve this problem, we have to think greedily that how can we maximize the difference value of the elements so that we can have a maximum sum. This is possible only if we calculate the difference between some very high values and some very low values like (highest – smallest). This is the idea which we have to use to solve this problem. Let us see the above example, we will have maximum difference possible for sequence {1, 8, 2, 4} because in this sequence we will get some high difference values, ( |1-8| = 7, |8-2| = 6 .. ). Here, by placing 8(highest element) in place of 1 and 2 we get two high difference values. Similarly, for the other values, we will place next highest values in between other, as we have only one left i.e 4 which is placed at last.   
**Algorithm:** *To get the maximum sum, we should have a sequence in which small and large elements comes alternate. This is done to get maximum difference.*  
**For the implementation of the above algorithm ->**   
1. We will sort the array.   
2. Calculate the final sequence by taking one smallest element and largest element from the sorted array and make one vector array of this final sequence.   
3. Finally, calculate the sum of absolute difference between the elements of the array.  
**Below is the implementation of above idea :**

// CPP implementation of

// above algorithm

#include <bits/stdc++.h>

using namespace std;

int MaxSumDifference(int a[], int n)

{

// final sequence stored in the vector

vector<int> finalSequence;

// sort the original array

// so that we can retrieve

// the large elements from

// the end of array elements

sort(a, a + n);

// In this loop first we will insert

// one smallest element not entered

// till that time in final sequence

// and then enter a highest element

// (not entered till that time) in

// final sequence so that we

// have large difference value. This

// process is repeated till all array

// has completely entered in sequence.

// Here, we have loop till n/2 because

// we are inserting two elements at a

// time in loop.

for (int i = 0; i < n / 2; ++i) {

finalSequence.push\_back(a[i]);

finalSequence.push\_back(a[n - i - 1]);

}

// If there are odd elements, push the

// middle element at the end.

if (n % 2 != 0)

finalSequence.push\_back(a[n/2]);

// variable to store the

// maximum sum of absolute

// difference

int MaximumSum = 0;

// In this loop absolute difference

// of elements for the final sequence

// is calculated.

for (int i = 0; i < n - 1; ++i) {

MaximumSum = MaximumSum + abs(finalSequence[i] -

finalSequence[i + 1]);

}

// absolute difference of last element

// and 1st element

MaximumSum = MaximumSum + abs(finalSequence[n - 1] -

finalSequence[0]);

// return the value

return MaximumSum;

}

// Driver function

int main()

{

int a[] = { 1, 2, 4, 8 };

int n = sizeof(a) / sizeof(a[0]);

cout << MaxSumDifference(a, n) << endl;

}

**Output :**

18

**Another approach:**

 try to bring elements having greater difference closer.   
Consider the sorted permutation of the given array a1, a1, a2,…., an – 1, an such that a1 < a2 < a3…. < an – 1 < an.   
Now, to obtain the answer having maximum sum of difference between consecutive element, arrange element in following manner:   
a1, an, a2, an-1,…., an/2, a(n/2) + 1   
We can observe that the arrangement produces the optimal answer, as all a1, a2, a3,….., a(n/2)-1, an/2 are subtracted twice while a(n/2)+1 , a(n/2)+2, a(n/2)+3,….., an – 1, an are added twice.   
Note – a(n/2)+1 This term is considered only for even n because for odd n, it is added once and subtracted once and hence cancels out.

// C++ program to maximize the sum of difference

// between consecutive elements in circular array

#include <bits/stdc++.h>

using namespace std;

// Return the maximum Sum of difference between

// consecutive elements.

int maxSum(int arr[], int n)

{

int sum = 0;

// Sorting the array.

sort(arr, arr + n);

// Subtracting a1, a2, a3,....., a(n/2)-1, an/2

// twice and adding a(n/2)+1, a(n/2)+2, a(n/2)+3,.

// ...., an - 1, an twice.

for (int i = 0; i < n/2; i++)

{

sum -= (2 \* arr[i]);

sum += (2 \* arr[n - i - 1]);

}

return sum;

}

// Driver Program

int main()

{

int arr[] = { 4, 2, 1, 8 };

int n = sizeof(arr)/sizeof(arr[0]);

cout << maxSum(arr, n) << endl;

return 0;

}

**Output :**

18

**Time Complexity:**O(nlogn).   
**Auxiliary Space :** O(1)

# 237. Maximize sum of consecutive differences in a circular array

## Same as ques 236 (above ques) of greedy.

# 238. Minimum sum of absolute difference of pairs of two arrays

Given two arrays ***a[]*** and ***b[]*** of equal length ***n***. The task is to pair each element of array ***a*** to an element in array ***b***, such that sum ***S*** of **absolute differences of all the pairs is minimum.**  
Suppose, two elements **a[*i*]** and **a[*j*]** **(*i* != *j*)** of ***a*** are paired with elements **b[*p*]** and **b[*q*]** of ***b*** respectively,   
then ***p*** should not be equal to ***q***.  
**Examples:**

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

b[] = {2, 1, 3}

**Output :** 0

**Explaination :**

1st pairing: |3 - 2| + |2 - 1| + |1 - 3|

= 1 + 1 + 2 = 4

2nd pairing: |3 - 2| + |1 - 1| + |2 - 3|

= 1 + 0 + 1 = 2

3rd pairing: |2 - 2| + |3 - 1| + |1 - 3|

= 0 + 2 + 2 = 4

4th pairing: |1 - 2| + |2 - 1| + |3 - 3|

= 1 + 1 + 0 = 2

5th pairing: |2 - 2| + |1 - 1| + |3 - 3|

= 0 + 0 + 0 = 0

6th pairing: |1 - 2| + |3 - 1| + |2 - 3|

= 1 + 2 + 1 = 4

Therefore, 5th pairing has minimum sum of

absolute difference.

**Input :** n = 4

a[] = {4, 1, 8, 7}

b[] = {2, 3, 6, 5}

**Output :** 6

## Solution:

The solution to the problem is a simple greedy approach. It consists of **two** steps.   
**Step 1** : Sort both the arrays in **O (n log n)** time.   
**Step 2** : Find absolute difference of each pair of **corresponding elements** *(elements at same index)* of both arrays and add the result to the sum ***S***. The time complexity of this step is **O(n)**.  
Hence, the overall time complexity of the program is **O(n log n)**.

// C++ program to find minimum sum of absolute

// differences of two arrays.

#include <bits/stdc++.h>

using namespace std;

// Returns minimum possible pairwise absolute

// difference of two arrays.

long long int findMinSum(long long int a[],long long int b[], int n)

{

// Sort both arrays

sort(a, a+n);

sort(b, b+n);

// Find sum of absolute differences

long long int sum= 0 ;

for (int i=0; i<n; i++)

sum = sum + abs(a[i]-b[i]);

return sum;

}

// Driver code

int main()

{

// Both a[] and b[] must be of same size.

long long int a[] = {4, 1, 8, 7};

long long int b[] = {2, 3, 6, 5};

int n = sizeof(a)/sizeof(a[0]);

printf("%lld\n", findMinSum(a, b, n));

return 0;

}

**Output :**

6

**Time Complexity:**O(n \* logn)

**Auxiliary Space:**O(1)

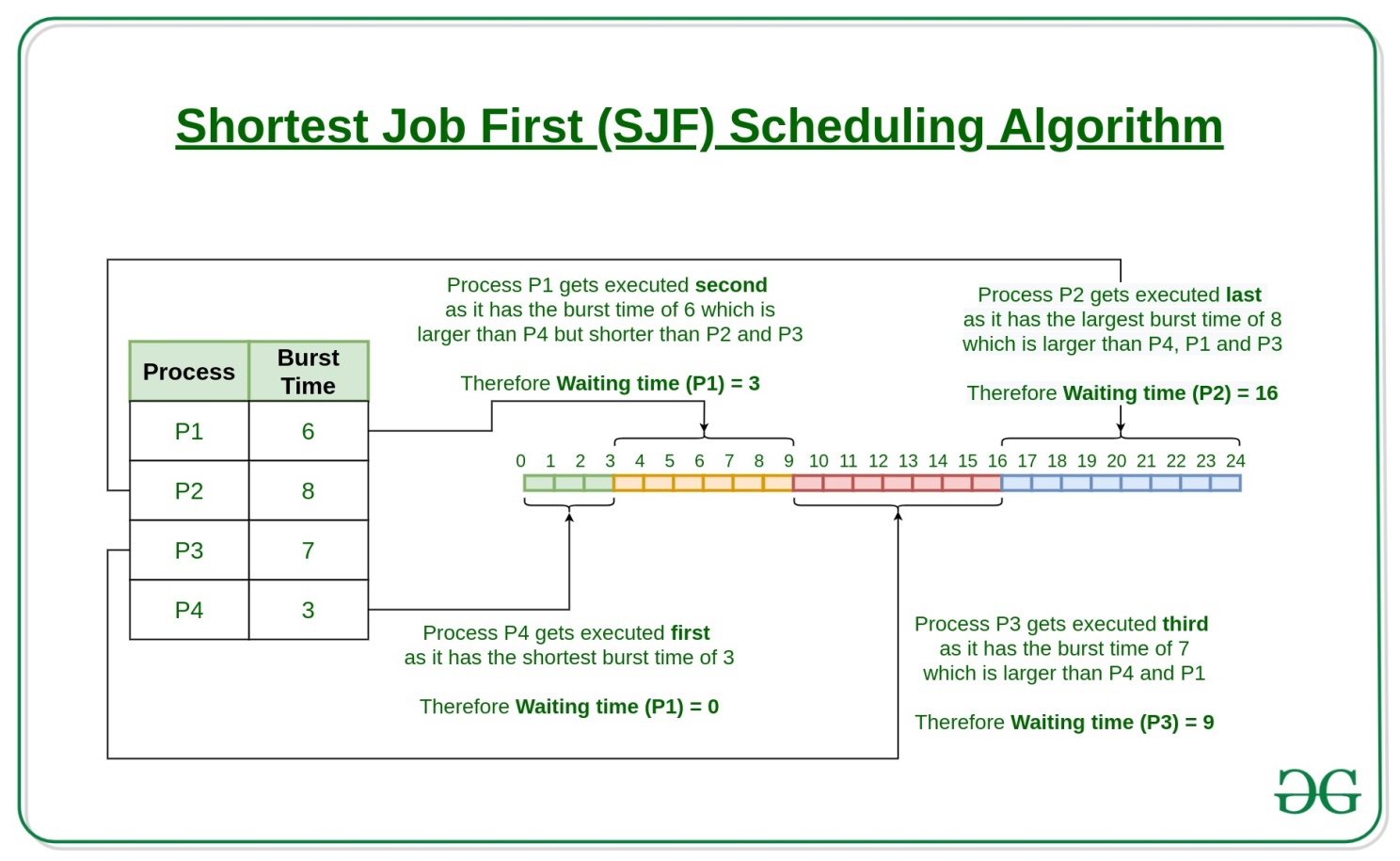
# 239. Program for Shortest Job First (or SJF) CPU Scheduling

Shortest job first (SJF) or shortest job next, is a scheduling policy that selects the waiting process with the smallest execution time to execute next. SJN is a non-preemptive algorithm.

* Shortest Job first has the advantage of having a minimum average waiting time among all scheduling algorithms.
* It is a Greedy Algorithm.
* It may cause starvation if shorter processes keep coming. This problem can be solved using the concept of ageing.
* It is practically infeasible as Operating System may not know burst time and therefore may not sort them. While it is not possible to predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times. SJF can be used in specialized environments where accurate estimates of running time are available.

**Algorithm:**

1. Sort all the process according to the arrival time.
2. Then select that process which has minimum arrival time and minimum Burst time.
3. After completion of process make a pool of process which after till the completion of previous process and select that process among the pool which is having minimum Burst time.



**How to compute below times in SJF using a program?**

1. Completion Time: Time at which process completes its execution.
2. Turn Around Time: Time Difference between completion time and arrival time. Turn Around Time = Completion Time – Arrival Time
3. Waiting Time(W.T): Time Difference between turn around time and burst time.   
   Waiting Time = Turn Around Time – Burst Time

***In this post, we have assumed arrival times as 0, so turn around and completion times are same.***

// C++ program to implement Shortest Job first with Arrival

// Time

#include <iostream>

using namespace std;

int mat[10][6];

void swap(int\* a, int\* b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void arrangeArrival(int num, int mat[][6])

{

for (int i = 0; i < num; i++) {

for (int j = 0; j < num - i - 1; j++) {

if (mat[j][1] > mat[j + 1][1]) {

for (int k = 0; k < 5; k++) {

swap(mat[j][k], mat[j + 1][k]);

}

}

}

}

}

void completionTime(int num, int mat[][6])

{

int temp, val;

mat[0][3] = mat[0][1] + mat[0][2];

mat[0][5] = mat[0][3] - mat[0][1];

mat[0][4] = mat[0][5] - mat[0][2];

for (int i = 1; i < num; i++) {

temp = mat[i - 1][3];

int low = mat[i][2];

for (int j = i; j < num; j++) {

if (temp >= mat[j][1] && low >= mat[j][2]) {

low = mat[j][2];

val = j;

}

}

mat[val][3] = temp + mat[val][2];

mat[val][5] = mat[val][3] - mat[val][1];

mat[val][4] = mat[val][5] - mat[val][2];

for (int k = 0; k < 6; k++) {

swap(mat[val][k], mat[i][k]);

}

}

}

int main()

{

int num, temp;

cout << "Enter number of Process: ";

cin >> num;

cout << "...Enter the process ID...\n";

for (int i = 0; i < num; i++) {

cout << "...Process " << i + 1 << "...\n";

cout << "Enter Process Id: ";

cin >> mat[i][0];

cout << "Enter Arrival Time: ";

cin >> mat[i][1];

cout << "Enter Burst Time: ";

cin >> mat[i][2];

}

cout << "Before Arrange...\n";

cout << "Process ID\tArrival Time\tBurst Time\n";

for (int i = 0; i < num; i++) {

cout << mat[i][0] << "\t\t" << mat[i][1] << "\t\t"

<< mat[i][2] << "\n";

}

arrangeArrival(num, mat);

completionTime(num, mat);

cout << "Final Result...\n";

cout << "Process ID\tArrival Time\tBurst Time\tWaiting "

"Time\tTurnaround Time\n";

for (int i = 0; i < num; i++) {

cout << mat[i][0] << "\t\t" << mat[i][1] << "\t\t"

<< mat[i][2] << "\t\t" << mat[i][4] << "\t\t"

<< mat[i][5] << "\n";

}

}

**Output**

Enter number of Process: ...Enter the process ID...

Before Arrange...

Process ID Arrival Time Burst Time

Final Result...

Process ID Arrival Time Burst Time Waiting Time Turnaround Time

Output: 

Process ID Arrival Time Burst Time

1 2 3

2 0 4

3 4 2

4 5 4

Final Result...

Process ID Arrival Time Burst Time Waiting Time Turnaround Time

2 0 4 0 4

3 4 2 0 2

1 2 3 4 7

4 5 4 4 8

# Program for Shortest Job First (SJF) scheduling | Set 2 (Preemptive)

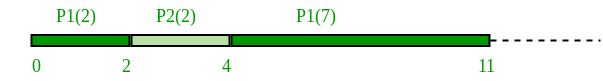
In this post we will discuss the preemptive version of SJF known as Shortest Remaining Time First (SRTF).

### Shortest Remaining Time First (SRTF) scheduling

In the Shortest Remaining Time First (SRTF) scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

### Shortest Remaining Time First (Preemptive SJF): Example

| Process | Duration | Order | Arrival Time |
| --- | --- | --- | --- |
| P1 | 9 | 1 | 0 |
| P2 | 2 | 2 | 2 |



***P1 waiting time: 4-2 = 2******P2 waiting time: 0******The average waiting time(AWT): (0 + 2) / 2 = 1***

**Advantage:**   
1- Short processes are handled very quickly.   
2- The system also requires very little overhead since it only makes a decision when a process completes or a new process is added.   
3- When a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.  
**Disadvantage:**   
1- Like shortest job first, it has the potential for process starvation.   
2- Long processes may be held off indefinitely if short processes are continually added.

**Implementation:** 

1- Traverse until all process gets completely

executed.

a) Find process with minimum remaining time at

every single time lap.

b) Reduce its time by 1.

c) Check if its remaining time becomes 0

d) Increment the counter of process completion.

e) Completion time of current process =

current\_time +1;

e) Calculate waiting time for each completed

process.

wt[i]= Completion time - arrival\_time-burst\_time

f)Increment time lap by one.

2- Find turnaround time (waiting\_time+burst\_time).

// C++ program to implement Shortest Remaining Time First

// Shortest Remaining Time First (SRTF)

#include <bits/stdc++.h>

using namespace std;

struct Process {

int pid; // Process ID

int bt; // Burst Time

int art; // Arrival Time

};

// Function to find the waiting time for all

// processes

void findWaitingTime(Process proc[], int n,

int wt[])

{

int rt[n];

// Copy the burst time into rt[]

for (int i = 0; i < n; i++)

rt[i] = proc[i].bt;

int complete = 0, t = 0, minm = INT\_MAX;

int shortest = 0, finish\_time;

bool check = false;

// Process until all processes gets

// completed

while (complete != n) {

// Find process with minimum

// remaining time among the

// processes that arrives till the

// current time`

for (int j = 0; j < n; j++) {

if ((proc[j].art <= t) &&

(rt[j] < minm) && rt[j] > 0) {

minm = rt[j];

shortest = j;

check = true;

}

}

if (check == false) {

t++;

continue;

}

// Reduce remaining time by one

rt[shortest]--;

// Update minimum

minm = rt[shortest];

if (minm == 0)

minm = INT\_MAX;

// If a process gets completely

// executed

if (rt[shortest] == 0) {

// Increment complete

complete++;

check = false;

// Find finish time of current

// process

finish\_time = t + 1;

// Calculate waiting time

wt[shortest] = finish\_time -

proc[shortest].bt -

proc[shortest].art;

if (wt[shortest] < 0)

wt[shortest] = 0;

}

// Increment time

t++;

}

}

// Function to calculate turn around time

void findTurnAroundTime(Process proc[], int n,

int wt[], int tat[])

{

// calculating turnaround time by adding

// bt[i] + wt[i]

for (int i = 0; i < n; i++)

tat[i] = proc[i].bt + wt[i];

}

// Function to calculate average time

void findavgTime(Process proc[], int n)

{

int wt[n], tat[n], total\_wt = 0,

total\_tat = 0;

// Function to find waiting time of all

// processes

findWaitingTime(proc, n, wt);

// Function to find turn around time for

// all processes

findTurnAroundTime(proc, n, wt, tat);

// Display processes along with all

// details

cout << "Processes "

<< " Burst time "

<< " Waiting time "

<< " Turn around time\n";

// Calculate total waiting time and

// total turnaround time

for (int i = 0; i < n; i++) {

total\_wt = total\_wt + wt[i];

total\_tat = total\_tat + tat[i];

cout << " " << proc[i].pid << "\t\t"

<< proc[i].bt << "\t\t " << wt[i]

<< "\t\t " << tat[i] << endl;

}

cout << "\nAverage waiting time = "

<< (float)total\_wt / (float)n;

cout << "\nAverage turn around time = "

<< (float)total\_tat / (float)n;

}

// Driver code

int main()

{

Process proc[] = { { 1, 6, 1 }, { 2, 8, 1 },

{ 3, 7, 2 }, { 4, 3, 3 } };

int n = sizeof(proc) / sizeof(proc[0]);

findavgTime(proc, n);

return 0;

}

**Output:**

Processes Burst time Waiting time Turn around time

1 6 3 9

2 8 16 24

3 7 8 15

4 3 0 3

Average waiting time = 6.75

Average turn around time = 12.75

# 240. Program for Least Recently Used (LRU) Page Replacement algorithm

In operating systems that use paging for memory management, page replacement algorithm is needed to decide which page needs to be replaced when the new page comes in. Whenever a new page is referred and is not present in memory, the page fault occurs and Operating System replaces one of the existing pages with a newly needed page.

Given a sequence of pages in an array **pages[]**of length **N** and memory capacity **C**, find the number of page faults using Least Recently Used (LRU) Algorithm.

**Example 1:**

**Input:** N = 9, C = 4

pages = {5, 0, 1, 3, 2, 4, 1, 0, 5}

**Output:** 8

**Explaination:** memory allocated with 4 pages 5, 0, 1,

3: page fault = 4

page number 2 is required, replaces LRU 5:

page fault = 4+1 = 5

page number 4 is required, replaces LRU 0:

page fault = 5 + 1 = 6

page number 1 is required which is already present:

page fault = 6 + 0 = 6

page number 0 is required which replaces LRU 1:

page fault = 6 + 1 = 7

page number 5 is required which replaces LRU 3:

page fault = 7 + 1 =8.

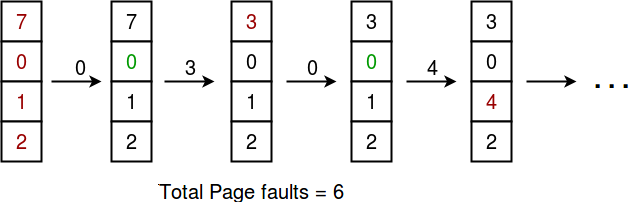
**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **pageFaults()** which takes N, C and pages[] as input parameters and returns the number of page faults.

**Expected Time Complexity:** O(N\*C)  
**â€‹Expected Auxiliary Space:** O(N)

**Constraints:**  
1 ≤ N ≤ 1000  
1 ≤ C ≤ 100  
1 ≤ pages[i] ≤ 1000

## Solution:

In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. Whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.  
In **L**east **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely   
Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.   
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —>**4 Page faults**   
0 is already their so —> **0 Page fault.**   
when 3 came it will take the place of 7 because it is least recently used —>**1 Page fault**   
0 is already in memory so —>**0 Page fault**.   
4 will takes place of 1 —>**1 Page Fault**   
Now for the further page reference string —>**0 Page fault** because they are already available in the memory.



***Given memory capacity (as number of pages it can hold) and a string representing pages to be referred, write a function to find number of page faults.***

Let **capacity** be the number of pages that

memory can hold. Let **set** be the current

set of pages in memory.

1- Start traversing the pages.

i) **If set holds less pages than capacity.**

a) Insert page into the set one by one until

the size of **set** reaches **capacity** or all

page requests are processed.

b) Simultaneously maintain the recent occurred

index of each page in a map called **indexes**.

c) Increment page fault

ii) **Else**

**If** current page is present in **set**, do nothing.

**Else**

a) Find the page in the set that was least

recently used. We find it using index array.

We basically need to replace the page with

minimum index.

b) Replace the found page with current page.

c) Increment page faults.

d) Update index of current page.

2. Return page faults.

Below is implementation of above steps.

//C++ implementation of above algorithm

#include<bits/stdc++.h>

using namespace std;

// Function to find page faults using indexes

int pageFaults(int pages[], int n, int capacity)

{

// To represent set of current pages. We use

// an unordered\_set so that we quickly check

// if a page is present in set or not

unordered\_set<int> s;

// To store least recently used indexes

// of pages.

unordered\_map<int, int> indexes;

// Start from initial page

int page\_faults = 0;

for (int i=0; i<n; i++)

{

// Check if the set can hold more pages

if (s.size() < capacity)

{

// Insert it into set if not present

// already which represents page fault

if (s.find(pages[i])==s.end())

{

s.insert(pages[i]);

// increment page fault

page\_faults++;

}

// Store the recently used index of

// each page

indexes[pages[i]] = i;

}

// If the set is full then need to perform lru

// i.e. remove the least recently used page

// and insert the current page

else

{

// Check if current page is not already

// present in the set

if (s.find(pages[i]) == s.end())

{

// Find the least recently used pages

// that is present in the set

int lru = INT\_MAX, val;

for (auto it=s.begin(); it!=s.end(); it++)

{

if (indexes[\*it] < lru)

{

lru = indexes[\*it];

val = \*it;

}

}

// Remove the indexes page

s.erase(val);

// insert the current page

s.insert(pages[i]);

// Increment page faults

page\_faults++;

}

// Update the current page index

indexes[pages[i]] = i;

}

}

return page\_faults;

}

// Driver code

int main()

{

int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

int n = sizeof(pages)/sizeof(pages[0]);

int capacity = 4;

cout << pageFaults(pages, n, capacity);

return 0;

}

**Output:**

6

**Another approach:** (Without using HashMap)

// Online C++ compiler to run C++ program online

#include <iostream>

#include<bits/stdc++.h>

using namespace std;

int main()

{

int capacity = 4;

int arr[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};

deque<int> q(capacity);

int count=0;

int page\_faults=0;

deque<int>::iterator itr;

q.clear();

for(int i:arr)

{

// Insert it into set if not present

// already which represents page fault

itr = find(q.begin(),q.end(),i);

if(!(itr != q.end()))

{

++page\_faults;

// Check if the set can hold equal pages

if(q.size() == capacity)

{

q.erase(q.begin());

q.push\_back(i);

}

else{

q.push\_back(i);

}

}

else

{

// Remove the indexes page

q.erase(itr);

// insert the current page

q.push\_back(i);

}

}

cout<<page\_faults;

}

**Output:** 

6

Note : We can also find the number of page hits. Just have to maintain a separate count.   
If the current page is already in the memory then that must be count as Page-hit.

# 241. Smallest subset with sum greater than all other elements

Given an array of non-negative integers. Our task is to find minimum number of elements such that their sum should be greater than the sum of rest of the elements of the array.  
**Examples :** 

Input : arr[] = {3, 1, 7, 1}

Output : 1

Smallest subset is {7}. Sum of

this subset is greater than all

other elements {3, 1, 1}

Input : arr[] = {2, 1, 2}

Output : 2

In this example one element is not

enough. We can pick elements with

values 1, 2 or 2, 2. In any case,

the minimum count is 2.

## Solution:

The **Brute force** approach is to find the sum of all the possible subsets and then compare sum with the sum of remaining elements.  
The **Efficient Approach** is to take the largest elements. We sort values in descending order, then take elements from the largest, until we get strictly more than half of total sum of the given array.

// CPP program to find minimum number of

// elements such that their sum is greater

// than sum of remaining elements of the array.

#include <bits/stdc++.h>

#include <string.h>

using namespace std;

// function to find minimum elements needed.

int minElements(int arr[], int n)

{

// calculating HALF of array sum

int halfSum = 0;

for (int i = 0; i < n; i++)

halfSum = halfSum + arr[i];

halfSum = halfSum / 2;

// sort the array in descending order.

sort(arr, arr + n, greater<int>());

int res = 0, curr\_sum = 0;

for (int i = 0; i < n; i++) {

curr\_sum += arr[i];

res++;

// current sum greater than sum

if (curr\_sum > halfSum)

return res;

}

return res;

}

// Driver function

int main()

{

int arr[] = {3, 1, 7, 1};

int n = sizeof(arr) / sizeof(arr[0]);

cout << minElements(arr, n) << endl;

return 0;

}

**Output:** 

1

**Time Complexity :** O(n Log n)

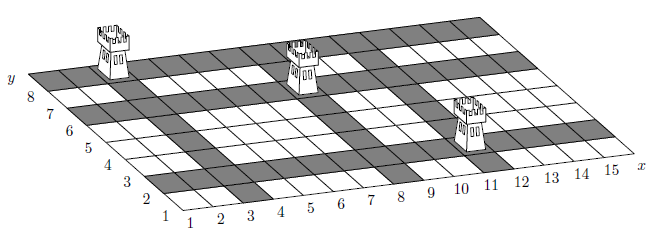
# 242. Chocolate Distribution Problem

## Same as ques 30 of array.

# 243. DEFKIN -Defense of a Kingdom

Theodore implements a new strategy game “Defense of a Kingdom”. On each level a player defends the Kingdom that is represented by a rectangular grid of cells. The player builds crossbow towers in some cells of the grid. The tower defends all the cells in the same row and the same column. No two towers share a row or a column.

The penalty of the position is the number of cells in the largest undefended rectangle. For example, the position shown on the picture has penalty 12.



Help Theodore write a program that calculates the penalty of the given position.

**Input**

The first line of the input file contains the number of test cases.

Each test case consists of a line with three integer numbers: w — width of the grid, h — height of the grid and n — number of crossbow towers (1 ≤ w, h ≤ 40 000; 0 ≤ n ≤ min(w, h)).

Each of the following n lines contains two integer numbers xi and yi — the coordinates of the cell occupied by a tower (1 ≤ xi ≤ w; 1 ≤ yi ≤ h).

**Output**

For each test case, output a single integer number — the number of cells in the largest rectangle that is not defended by the towers.

**Example**

**Input:**

1

15 8 3

3 8

11 2

8 6

**Output:**

12

## Solution:

Sort the array of x and y coordinate of position of defenders. Find the max len undefended on x axis and y axis both separately. Multiply them to get the area of largest rectangle undefended. Handle all the corner cases very carefully.

#include <bits/stdc++.h>

using namespace std;

int main() {

int t;

cin>>t;

while(t--){

int l, b, n;

cin>>l>>b>>n;

if(n==0){

cout<<l\*b<<endl;

continue;

}

int x[n+2], y[n+2];

for(int i=1;i<=n;i++)

cin>>x[i]>>y[i];

x[0] = 0; y[0] = 0;

x[n+1] = l+1; y[n+1] = b+1;

sort(x, x+n+2);

sort(y,y+n+2);

int max\_diff\_x = 0, max\_diff\_y = 0;

for(int i=1;i<n+2;i++){

max\_diff\_x = max(max\_diff\_x, x[i]-x[i-1]-1);

max\_diff\_y = max(max\_diff\_y, y[i]-y[i-1]-1);

//cout<<max\_diff\_x<<" "<<max\_diff\_y<<endl;

}

//cout<<max\_diff\_x<<" "<<max\_diff\_y<<endl;

int res = max\_diff\_x \* max\_diff\_y;

cout<<res<<endl;

}

return 0;

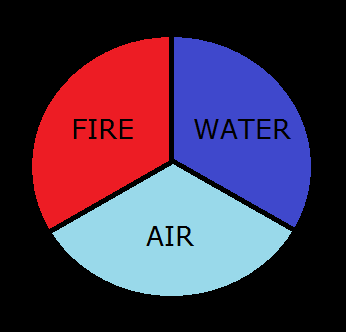
}

**Time Complexity:** O(nlogn)

**Space Complexity:** O(1)

# 244. DIEHARD -DIE HARD

**Problem Statement:**



The game is simple. You initially have ‘H’ amount of health and ‘A’ amount of armor. At any instant you can live in any of the three places - fire, water and air. After every unit time, you have to change your place of living. For example if you are currently living at fire, you can either step into water or air.

If you step into air, your health increases by 3 and your armor increases by 2

If you step into water, your health decreases by 5 and your armor decreases by 10

If you step into fire, your health decreases by 20 and your armor increases by 5

If your health or armor becomes <=0, you will die instantly

Find the maximum time you can survive.

**Input:**

The first line consists of an integer t, the number of test cases. For each test case there will be two positive integers representing the initial health H and initial armor A.

**Output:**  
  
For each test case find the maximum time you can survive.

**Note:** You can choose any of the 3 places during your first move.

**Input Constraints:**  
  
1 <= t <= 10  
1 <= H, A <= 1000  
  
**Example:**

**Sample Input:**

3

2 10

4 4

20 8

**Sample Output:**

1

1

5

## Solution:

When time instant is even move to air. When time instant is odd, move to water if armor>10 else move to fire.

#include <bits/stdc++.h>

using namespace std;

int main()

{

int t;

cin>>t;

while(t--){

int h,a,t=0;

cin>>h>>a;

while(h>0&&a>0){

if(t%2==0){

h+=3; a+=2;

}

else{

if(a>10){

h-=5; a-=10;

}

else{

h-=20; a+=5;

}

}

t++;

}

cout<<t-1<<endl;

}

return 0;

}

**Time Complexity:** O(t)

**Space Complexity:** O(1)

# 245. GERGOVIA -Wine trading in Gergovia

Gergovia consists of one street, and every inhabitant of the city is a wine salesman. Everyone buys wine from other inhabitants of the city. Every day each inhabitant decides how much wine he wants to buy or sell. Interestingly, demand and supply is always the same, so that each inhabitant gets what he wants.

There is one problem, however: Transporting wine from one house to another results in work. Since all wines are equally good, the inhabitants of Gergovia don't care which persons they are doing trade with, they are only interested in selling or buying a specific amount of wine.

In this problem you are asked to reconstruct the trading during one day in Gergovia. For simplicity we will assume that the houses are built along a straight line with equal distance between adjacent houses. Transporting one bottle of wine from one house to an adjacent house results in one unit of work.

**Input**

The input consists of several test cases.

Each test case starts with the number of inhabitants N (2 ≤ N ≤ 100000).

The following line contains n integers ai (-1000 ≤ ai ≤ 1000).

If ai ≥ 0, it means that the inhabitant living in the ith house wants to buy ai bottles of wine. If ai < 0, he wants to sell -ai bottles of wine.

You may assume that the numbers ai sum up to 0.

The last test case is followed by a line containing 0.

**Output**

For each test case print the minimum amount of work units needed so that every inhabitant has his demand fulfilled.

**Example**

**Input:**5  
5 -4 1 -3 1  
6  
-1000 -1000 -1000 1000 1000 1000  
0  
  
**Output:**  
9  
9000

## Solution:

Traverse the array and keep transporting the wine/requirement to its adjacent shop till all requirements is fulfilled and keep calculating the cost.

#include <bits/stdc++.h>

using namespace std;

int main()

{

int n;

cin>>n;

while(n){

int arr[n];

for(int i=0;i<n;i++)

cin>>arr[i];

long long res = 0;

for(int i=1;i<n;i++){

res += abs(arr[i-1]);

arr[i] += arr[i-1];

}

cout<<res<<endl;

cin>>n;

}

return 0;

}

**Time Complexity:** O(n)

**Space Complexity:** O(1)

# 246. Picking Up Chicks

A flock of chickens are running east along a straight, narrow road. Each one is running with its own constant speed. Whenever a chick catches up to the one in front of it, it has to slow down and follow at the speed of the other chick. You are in a mobile crane behind the flock, chasing the chicks towards the barn at the end of the road. The arm of the crane allows you to pick up any chick momentarily, let the chick behind it pass underneath and place the picked up chick back down. This operation takes no time and can only be performed on a pair of chicks that are immediately next to each other, even if 3 or more chicks are in a row, one after the other.

Given the initial locations (**Xi**) at time 0 and natural speeds (**Vi**) of the chicks, as well as the location of the barn (**B**), what is the minimum number of swaps you need to perform with your crane in order to have at least **K** of the **N** chicks arrive at the barn no later than time **T**?

You may think of the chicks as points moving along a line. Even if 3 or more chicks are at the same location, next to each other, picking up one of them will only let one of the other two pass through. Any swap is instantaneous, which means that you may perform multiple swaps at the same time, but each one will count as a separate swap.

**Input**

The first line of the input gives the number of test cases, **C**. **C** test cases follow. Each test case starts with 4 integers on a line -- **N**, **K**, **B** and **T**. The next line contains the **N**different integers **Xi**, in increasing order. The line after that contains the **N** integers **Vi**. All distances are in meters; all speeds are in meters per second; all times are in seconds.

**Output**

For each test case, output one line containing "Case #x: **S**", where x is the case number (starting from 1) and **S** is the smallest number of required swaps, or the word "IMPOSSIBLE".

**Limits**

1 ≤ **C** ≤ 100;  
1 ≤ **B** ≤ 1,000,000,000;  
1 ≤ **T** ≤ 1,000;  
0 ≤ **Xi** < **B**;  
1 ≤ **Vi** ≤ 100;  
1 ≤ **N** ≤ 50;  
0 ≤ **K** ≤ **N**;

All the **Xi**'s will be distinct and in increasing order.

**Example**

**Input:**

3  
5 3 10 5  
0 2 5 6 7  
1 1 1 1 4  
5 3 10 5  
0 2 3 5 7  
2 1 1 1 4  
5 3 10 5  
0 2 3 4 7  
2 1 1 1 4

**Output:**

Case #1: 0  
Case #2: 2  
Case #3: IMPOSSIBLE

## Solution:

Traverse the array of chicks location form end and keep counting the chicks which can reach barn and swaps required. Stop when count=k and print no. of swaps required. or if we reached the starting of array and count!=k then, print "impossible".

#include <bits/stdc++.h>

using namespace std;

int main()

{

int t;

cin>>t;

for(int temp =1;temp<=t;temp++){

long long n, k, b, t;

cin>>n>>k>>b>>t;

long long x[n], v[n];

for(int i=0;i<n;i++)

cin>>x[i];

for(int i=0;i<n;i++)

cin>>v[i];

int count=0, waste=0, swap=0;

for(int i=n-1; i>=0; i--){

long long dist\_possible = v[i]\*t, dist\_needed = b-x[i];

if(dist\_possible >= dist\_needed){

count++;

if(waste>0)

swap += waste;

}

else{

waste++;

}

if(count==k)

break;

}

if (count < k)

{

cout << "Case #" << temp << ": " << "IMPOSSIBLE" << endl;

}

else

{

cout << "Case #" << temp << ": " << swap << endl;

}

}

return 0;

}

**Time Complexity:** O(n)

**Space Complexity:** O(1)

# 247. CHOCOLA –Chocolate

## Same as ques 230 of Greedy soln

# 248. ARRANGE -Arranging Amplifiers

Scientists at the TIFR, Mumbai, are doing some cutting edge research on the Propagation of Signals. A young researcher comes up with a method of progressively amplifying signals, as they progress along a path. The method involves the placing of Amplifiers at regular distances along the line. Each amplifier is loaded with a number a(i), which is called its amplification factor. The method of amplification is simple: an amplifier which recieves a signal of strength X, and has Y loaded in it, results in a signal of strength Y^X [ Y to the power X]. In course of his research, the young scientist tries to find out, that given a set of n amplifiers loaded with a(0), a(1), a(2), ...., a(n-1), which particular permutation of these amplifiers, when placed at successive nodes, with the initial node given a signal of strength 1, produces the strongest output signal.  
  
this is better illustrated by the following example : 5 6 4  
4^(5^(6^1)) is the strength of the strongest signal, which is generated by putting amplifier loaded with 6 in first place, 5 in second place and 4 in third place.  
  
Given a list of integers specifying the set of amplifiers at hand, you must find out the order in which they must be placed, to get the highest signal strength. In case their exist multiple permutations with same output, you should print the one which has bigger amplifiers first.

**Input**

First line of input contains T, the number of test cases. For each test case first line contains a number ni, which is equal to the number of amplifiers available. Next line contains n integers, separated by spaces which denote the values with which the amplifiers are loaded.

**Output**

Output contains T lines, one for each test case. Each line contains ni integers, denoting the order in which the amplifiers should be kept such that the result is strongest.

**Example**

**Input:**

2

3

5 6 4

2

2 3

**Output:**

6 5 4

2 3

**Constraints and Limits**

T ≤ 20, Ni ≤ 10^5.  
Each amplifier will be loaded with a positive integer p, 0 < p ≤ 10^9.  
No two amplifier > 1 shall be loaded with the same integer.

## Solution:

Sort the array in descending order so that exponential power of for each power can be maximised. Handle the exceptional case of 1, 2 & 3 carefully.

#include <bits/stdc++.h>

using namespace std;

int main() {

int t;

cin>>t;

while(t--){

int n, val, one=0;

cin>>n;

vector<int> arr;

for(int i=0;i<n;i++){

cin>>val;

if(val==1)

one++;

else

arr.push\_back(val);

}

sort(arr.begin(), arr.end(), greater<int>());

while(one--)

cout<<"1 ";

if(arr.size()==2 && arr[0]==3 && arr[1]==2)

cout<<"2 3"<<endl;

else{

for(int i=0;i<arr.size();i++)

cout<<arr[i]<<" ";

cout<<endl;

}

}

return 0;

}

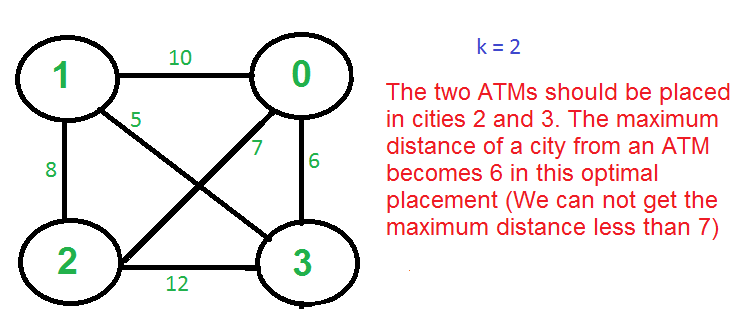
**Time Complexity:** O(nlogn)

**Space Complexity:** O(1)

# 249. K Centers Problem

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2, and 3, and distances between them, how to do place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized. 



## Solution:

There is no polynomial-time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial-time Greedy approximate algorithm, the greedy algorithm provides a solution that is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow [Triangular Inequality](http://en.wikipedia.org/wiki/Triangle_inequality) (The distance between two points is always smaller than the sum of distances through a third point).

**The 2-Approximate Greedy Algorithm:**   
1) Choose the first center arbitrarily.   
2) Choose remaining k-1 centers using the following criteria.   
       Let c1, c2, c3, … ci be the already chosen centers. Choose   
       (i+1)’th center by picking the city which is farthest from already   
       selected centers, i.e, the point p which has following value as maximum   
                 Min[dist(p, c1), dist(p, c2), dist(p, c3), …. dist(p, ci)] 

greedyAlgo

**Example (k = 3 in the above-shown Graph)**   
a) Let the first arbitrarily picked vertex be 0.   
b) The next vertex is 1 because 1 is the farthest vertex from 0.   
c) Remaining cities are 2 and 3. Calculate their distances from already selected centers (0 and 1). The greedy algorithm basically calculates the following values.   
        Minimum of all distanced from 2 to already considered centers   
        Min[dist(2, 0), dist(2, 1)] = Min[7, 8] = 7   
        Minimum of all distanced from 3 to already considered centers   
        Min[dist(3, 0), dist(3, 1)] = Min[6, 5] = 5   
        After computing the above values, city 2 is picked as the value corresponding to 2 is maximum.

Note that the greedy algorithm doesn’t give the best solution for k = 2 as this is just an approximate algorithm with a bound as twice optimal.

**Proof that the above greedy algorithm is 2 approximate.**   
Let OPT be the maximum distance of a city from a center in the Optimal solution. We need to show that the maximum distance obtained from the Greedy algorithm is 2\*OPT.   
The proof can be done using contradiction.   
a) Assume that the distance from the furthest point to all centers is > 2·OPT.   
b) This means that distances between all centers are also > 2·OPT.   
c) We have k + 1 points with distances > 2·OPT between every pair.   
d) Each point has a center of the optimal solution with distance <= OPT to it.   
e) There exists a pair of points with the same center X in the optimal solution (pigeonhole principle: k optimal centers, k+1 points)   
f) The distance between them is at most 2·OPT (triangle inequality) which is a contradiction.

// C++ program for the above approach

#include <bits/stdc++.h>

using namespace std;

int maxindex(int\* dist, int n)

{

int mi = 0;

for (int i = 0; i < n; i++) {

if (dist[i] > dist[mi])

mi = i;

}

return mi;

}

void selectKcities(int n, int weights[4][4], int k)

{

int\* dist = new int[n];

vector<int> centers;

for (int i = 0; i < n; i++) {

dist[i] = INT\_MAX;

}

// index of city having the

// maximum distance to it's

// closest center

int max = 0;

for (int i = 0; i < k; i++) {

centers.push\_back(max);

for (int j = 0; j < n; j++) {

// updating the distance

// of the cities to their

// closest centers

dist[j] = min(dist[j], weights[max][j]);

}

// updating the index of the

// city with the maximum

// distance to it's closest center

max = maxindex(dist, n);

}

// Printing the maximum distance

// of a city to a center

// that is our answer

cout << endl << dist[max] << endl;

// Printing the cities that

// were chosen to be made

// centers

for (int i = 0; i < centers.size(); i++) {

cout << centers[i] << " ";

}

cout << endl;

}

// Driver Code

int main()

{

int n = 4;

int weights[4][4] = { { 0, 4, 8, 5 },

{ 4, 0, 10, 7 },

{ 8, 10, 0, 9 },

{ 5, 7, 9, 0 } };

int k = 2;

// Function Call

selectKcities(n, weights, k);

}

**Output**

5

0 2

# 250. Minimum Cost of ropes

## Same as 324 ques of heap.

# 251. [Find smallest number with given number of digits and sum of digits](https://practice.geeksforgeeks.org/problems/smallest-number5829/1)

The task is to find the smallest number with given sum of digits as **S** and number of digits as **D.**

**Example 1:**

**Input:**

S = 9

D = 2

**Output:**

18

**Explanation:**

18 is the smallest number

possible with sum = 9

and total digits = 2.

**Example 2:**

**Input:**

S = 20

D = 3

**Output:**

299

**Explanation:**

299 is the smallest number

possible with sum = 20

and total digits = 3.

**Your Task:**

You don't need to read input or print anything. Your task is to complete the function smallestNumber() which takes the two integers S and D and returns a string which is the smallest number if possible, else return "-1".

**Expected Time Complexity:** O(D)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ S ≤ 100  
1 ≤ D ≤ 6

## Solution:

A **Simple Solution** is to consider all m digit numbers and keep track of minimum number with digit sum as s. A close upper bound on time complexity of this solution is O(10m).  
There is a **Greedy approach** to solve the problem. The idea is to one by one fill all digits from rightmost to leftmost (or from least significant digit to most significant).   
We initially deduct 1 from sum s so that we have smallest digit at the end. After deducting 1, we apply greedy approach. We compare remaining sum with 9, if remaining sum is more than 9, we put 9 at the current position, else we put the remaining sum. Since we fill digits from right to left, we put the highest digits on the right side. Below is implementation of the idea.

// C++ program to find the smallest number that can be

// formed from given sum of digits and number of digits.

#include <iostream>

using namespace std;

// Prints the smallest possible number with digit sum 's'

// and 'm' number of digits.

void findSmallest(int m, int s)

{

// If sum of digits is 0, then a number is possible

// only if number of digits is 1.

if (s == 0)

{

(m == 1)? cout << "Smallest number is " << 0

: cout << "Not possible";

return ;

}

// Sum greater than the maximum possible sum.

if (s > 9\*m)

{

cout << "Not possible";

return ;

}

// Create an array to store digits of result

int res[m];

// deduct sum by one to account for cases later

// (There must be 1 left for the most significant

// digit)

s -= 1;

// Fill last m-1 digits (from right to left)

for (int i=m-1; i>0; i--)

{

// If sum is still greater than 9,

// digit must be 9.

if (s > 9)

{

res[i] = 9;

s -= 9;

}

else

{

res[i] = s;

s = 0;

}

}

// Whatever is left should be the most significant

// digit.

res[0] = s + 1; // The initially subtracted 1 is

// incorporated here.

cout << "Smallest number is ";

for (int i=0; i<m; i++)

cout << res[i];

}

// Driver code

int main()

{

int s = 9, m = 2;

findSmallest(m, s);

return 0;

}

**Output :**

Smallest number is 18

Time Complexity of this solution is O(m).  
We will soon be discussing approach to find the largest possible number with given sum of digits and number of digits.

# 252. Rearrange characters in a string such that no two adjacent are same

## **Same as ques 80 of string**.

# 253. Find maximum sum possible equal sum of three stacks

Given three stacks of the positive numbers, the task is to find the possible equal maximum sum of the stacks with the removal of top elements allowed. Stacks are represented as an array, and the first index of the array represent the top element of the stack.

Examples:

**Input :** stack1[] = { 3, 10}

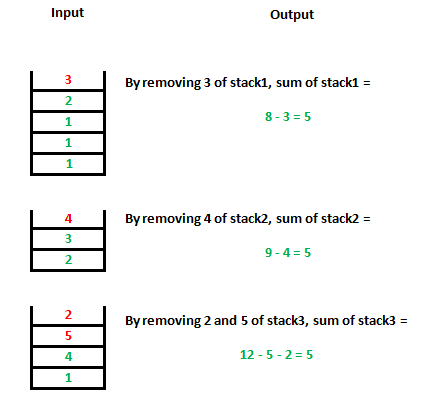
stack2[] = { 4, 5 }

stack3[] = { 2, 1 }

**Output :** 0

Sum can only be equal after removing all elements

from all stacks.



## Solution:

The idea is to compare the sum of each stack, if they are not same, remove the top element of the stack having the maximum sum.  
Algorithm for solving this problem:

1. Find the sum of all elements of in individual stacks.
2. If the sum of all three stacks is the same, then this is the maximum sum.
3. Else remove the top element of the stack having the maximum sum among three of stacks. Repeat step 1 and step 2.

The approach works because elements are positive. To make sum equal, we must remove some element from stack having more sum, and we can only remove from the top.

Below is the implementation of this approach:

// C++ program to calculate maximum sum with equal

// stack sum.

#include <bits/stdc++.h>

using namespace std;

// Returns maximum possible equal sum of three stacks

// with removal of top elements allowed

int maxSum(int stack1[], int stack2[], int stack3[], int n1,

int n2, int n3)

{

int sum1 = 0, sum2 = 0, sum3 = 0;

// Finding the initial sum of stack1.

for (int i = 0; i < n1; i++)

sum1 += stack1[i];

// Finding the initial sum of stack2.

for (int i = 0; i < n2; i++)

sum2 += stack2[i];

// Finding the initial sum of stack3.

for (int i = 0; i < n3; i++)

sum3 += stack3[i];

// As given in question, first element is top

// of stack..

int top1 = 0, top2 = 0, top3 = 0;

while (1) {

// If any stack is empty

if (top1 == n1 || top2 == n2 || top3 == n3)

return 0;

// If sum of all three stack are equal.

if (sum1 == sum2 && sum2 == sum3)

return sum1;

// Finding the stack with maximum sum and

// removing its top element.

if (sum1 >= sum2 && sum1 >= sum3)

sum1 -= stack1[top1++];

else if (sum2 >= sum1 && sum2 >= sum3)

sum2 -= stack2[top2++];

else if (sum3 >= sum2 && sum3 >= sum1)

sum3 -= stack3[top3++];

}

}

// Driven Program

int main()

{

int stack1[] = { 3, 2, 1, 1, 1 };

int stack2[] = { 4, 3, 2 };

int stack3[] = { 1, 1, 4, 1 };

int n1 = sizeof(stack1) / sizeof(stack1[0]);

int n2 = sizeof(stack2) / sizeof(stack2[0]);

int n3 = sizeof(stack3) / sizeof(stack3[0]);

cout << maxSum(stack1, stack2, stack3, n1, n2, n3)

<< endl;

return 0;

}

**Output**

5

**Time Complexity :**O(n1 + n2 + n3) where n1, n2 and n3 are sizes of three stacks.