4 Binary Search

Divide and Conquer

Searching an element in a Sorted Sequence

```
binary_search(A, target):
lo = 1, hi = size(A)
while lo <= hi:
    mid = lo + (hi-lo)/2
    if A[mid] == target:
        return mid
else if A[mid] < target:
      lo = mid+1
else:
     hi = mid-1
```

First Occurrence of an Element

```
FirstOccur(A, target):
   lo = 1, hi = size(A), result=-1;
 while lo <= hi:
         mid = lo + (hi-lo)/2
         if A[mid] == target:
              result=mid; high=mid-1;
       else if A[mid] < target:
            lo = mid+1
       else:
      hi = mid-1
 return res;
```

Last Occcurrence of an Element

```
LastOccur(A, target):
  lo = 1, hi = size(A), result=-1;
 while lo <= hi:
   mid = lo + (hi-lo)/2
```

```
  if A[mid] == target:
    result=mid;  low=mid+1;
  else if A[mid] < target:
    lo = mid+1
  else:
    hi = mid-1
 return  res;
```

Beyond Sorted arrays

Binary search obviously works on searching for elements in a sorted array. But if you think about the reason why it works is because the array itself is monotonic (either increasing or decreasing). So, if you are at a particular position, you can make a definite call whether the answer lies in the left part of the position or the right part of it. But Most importantly you need to know the range [start,end].

Similar thing can be done with monotonic functions (monotonically increasing or decreasing) as well.

Lets say we have f(x) which increases when x increases.

So, given a problem of finding x so that f(x) = p, I can do a binary search for x.

/Take Example/

Therefore

1. if f(current_position) > p, then I will search for values lower than current position.
2. if f(current_position) < p, then I will search for values higher than current position
3. if f(current_position) = p, then I have found my answer.

Optimisation Problems

Consider an optimisation problem where you need to minimise a quantity X satisfying some constraint Y such that:

- If X satisfies the constraint, anything more than X will satisfy the constraint too.
- For a given value of X, you know how to check whether the constraint is satisfied.

Painter's Partition Problem

You have to paint N boards of length {A_0, A_1, A_2, A_3 … A_{N-1}}. There are K painters available and you are also given how much time a painter takes to paint 1 unit of board. You have to get this job done **as soon as** possible under the constraints that any painter will only paint contiguous sections of board.

/Take example/

Observations :

- The lowest possible value for costmax must be the maximum element in A (name this as lo).
- The highest possible value for costmax must be the entire sum of A,
  (name this as hi).

- As costmax increases, x decreases. The opposite also holds true.

```
int painterNum(vector<int> &C, long long X){
    long long sum = 0;
    long long num = 1;
    for(auto c: C){
        if(sum + c > X){
            sum = c;num++;
        }else sum += c;
    }
    return num;
}
int paint(int A, int B, vector<int> &C) {

    int high=0,low=0;
    for(int x: C){low = max(x, low); high += x;}
    while(low < high){
        long long mid = low + (high-low)>>1;
        if(painterNum(C, mid) <= A)high = mid;
        else low = mid+1;
    }

    return low;
}
```

Allocate Books

N number of books are given.

The ith book has Pi number of pages.

You have to allocate books to M number of students so that maximum number of pages alloted to a student is minimum. A book will be allocated to exactly one student. Each student has to be allocated at least one book. Allotment should be in contiguous order, for example: A student cannot be allocated book 1 and book 3, skipping book 2.

Return -1 if a valid assignment is not possible

Problem :

Aggressive Cows :

http://www.spoj.com/problems/AGGRCOW/

Approach:

Define the following function:

Func(x) = 1 if it is possible to arrange the cows in stalls such that the distance between any two cows is at least x

Func(x) = 0 otherwise

The problem satisfies the monotonicity condition necessary for binary search.Why??

Check that if Func(x)=0, Func(y)=0 for all y>x and if Func(x)=1,

Funx(y)=1 for all y < x

Find low and high values

Func(0)=1 trivially since the distance between any two cows is at least 0. Also, since we have at least two cows, the best we can do is push them towards the stalls at the end - so there is no way we can achieve better. Hence Func(maxbarnpos-minbarnpos+1)=0.

```
#include <bits/stdc++.h>
using namespace std;
int n,c;
int func(int num,int array[])
{
    int cows=1,pos=array[0];
    for (int i=1; i<n; i++)
    {
        if (array[i]-pos>=num)
        {
            cows++;
            if (cows==c)
                return 1;
            pos=array[i];
        }
     }
     return 0;
 }
 int bs(int array[])
```

```
{
    int ini=0,last=array[n-1],max=-1;
    while (last>ini)
    {
        int mid=(ini+last)/2;
        if (func(mid,array)==1)
        {
           if (mid>max)
               max=mid;
           ini=mid+1;
        }
        else
           last=mid;
    }
    return max;
}
int main()
{
    int t;
    scanf("%d",&t);
    while (t--)
    {
         scanf("%d %d",&n,&c);
         int array[n];
         for (int i=0; i<n; i++)
             scanf("%d",&array[i]);
         sort(array,array+n);
         int k=bs(array);
         printf("%dn",k);
     }
     return 0;
}
```

Time to Try

- EKO (Spoj)
- PRATA (Spoj)
- BEAUTIFUL TRIPLETS (Hackerearth)

SELF STUDY NOTES

SELF STUDY NOTES

5 Greedy Algorithm

Greedy Algorithms are one of the most intuitive algorithms. Whenever we see a problem we first try to apply some greedy strategy to get the answer(we humans are greedy, aren't we :P ?).

Greedy approaches are quite simple and easy to understand/formulate.But many times the proving part might be difficult.

- Greedy Algorithm always makes the choice that looks best at **the moment.**
- You hope that by choosing a local optimum at each step, you will end up at a global optimum.

Counting Money

Problem Statement: Suppose you want to count out a certain amount of money,using the fewest possible notes or coins.

Greedy Approach

At each step, take the largest possible note or coin that does not overshoot the sum of money and include it in the solution.

Example **:**

To make Rs 39 with fewest possible notes or coins.

Rs 10 note, to make 29

Rs 10 note, to make 19

Rs 10 note, to make 9

Rs 5 note, to make 4

Rs 2 coin, to make 2

Rs 2 coin, to make 0

Total is 4 notes and 2 coins, which is the optimum solution.

Note : For Indian currency, the greedy algorithm, even after Demonetization always gives the optimum solution.

Is this true for any currency?

Now that Donald Trump is the new President of US, he decides to do something extraordinary like PM Modi. So he decides to change all the currency notes to 1 dollars, 7 dollars and 10 dollars.

Suppose you went to US and used the same greedy approach to count minimum numbers of notes and coins in exchange for a Burger which costs $15.

To make $15:

1 $10 note

5 $1 notes

Total = 6 notes

Can we do better than 6?

$7 + $7 +$1 – Only 3 notes required!

PROBLEMS

- **BUSYMAN**

  Activity Scheduling Problem

  http://www.spoj.com/problems/BUSYMAN/

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
struct cmpr
{
      bool operator()(pair<int,int> l, pair<int,int> r)
    {
      return l.second < r.second;
    }
}cmp;
int main()
{
    int t,n,s,e,res;
   scanf("%d",&t);
    while(t—)
  {
    res = 1;
    vector<pair<int,int> > activity;
    scanf("%d",&n);
   for(int i = 0;i<n;i++)
   {
         scanf("%d %d",&s,&e);
```

```
        activity.push_back(make_pair(s,e));
    }
    //Sort the activities according to their finish time
           sort(activity.begin(),activity.end(),cmp);
    //fin denotes the finish time of the choosen activity
    //i.e. activity with least finish time
      int fin = activity[0].second;
       for(int i = 1;i<activity.size();i++)
       {
          //To find the next compatible activity with
          //least finish time
           //Find the first activity whose start time
               //is more than finish time of previous activity
             if(activity[i].first >= fin)
             {
                  //update the finish time as the finish time
                 //of this activity
                   fin = activity[i].second;
                  //Since we have choosen a new activity,
                  //increment the count by 1
                     res++;
              }
        }
         printf("%d\n",res);
     }
    return 0;
}
```

- **CONNECTING WIRES**
  - There are n white dots and n black dots, equally spaced, in a line.
  - You want to connect each white dot with some one black dot, with a minimum total length of "wire".

Greedy Approach : Suppose you have a sorted list of the white dot positions and the black dot positions. Then you should match the i-th white dot with the i-th black dot.

Why should greedy work?

Let b0 and w0 be the first black and white balls respectively and let bi and wj be the next white and black balls.

Case 1: w0 and wj < b0

b0 bi

w0 wj

(i) C(w0, bi) + C(wj, b0) = (bi - w0) + (b0 - wj)

(ii) C(w0, b0) + C(wi, bj) = (b0 - w0) + (bi - wj)

Both are same.

Case 2: w0 < b0 and b0 < wj < bi

b0 bi

w0 wj

(i) C(w0, bi) + C(wj, b0) = (bi - w0) + (wj - b0)

(ii) C(w0, b0) + C(wi, bj) = (b0 - w0) + (bi - wj)

(i) - (ii) = 2(wj - b0) is extra

Case 3: w0 < b0 and bi < wj

b0 bi

w0 wj

(i) C(w0, bi) + C(wj, b0) = (bi - w0) + (wj - b0)

(ii) C(w0, b0) + C(wi, bj) = (b0 - w0) + (wj - bi)

(i) - (ii) = 2(bi - b0) is extra

Hence, in all the cases connecting 1st b to 1st w and 2nd b to 2nd w yeilds more optimal solution.

❑ BIASED STANDINGS

In a competition, each team is be able to enter their preferred place in the ranklist. Suppose that we already have a ranklist. For each team, compute the distance between their preferred place and their place in the ranklist. The sum of these distances will be called the badness of this ranklist. Find one ranklist with the minimal possible badness.

http://www.spoj.com/problems/BAISED/

This is question is similar to the connecting wires problem, where the dseired ranks denote the black balls and the actual ranks denote the white balls. Hence, we can apply the greedy approach by sorting the desired ranks and calculating the distance between i-th desired rank and i-th actula rank.

Code :

```
#include <bits/stdc++.h>
using namespace std;
#define abs(a,b) a > b ? a - b : b - a
#define ll long long
int main()
{
    int t;
   cin>>t;
   while(t—)
  {
     vector<int> v;
    string name;
    ll sum=0;
    cin>>n;
     for(int i=0;i<n;i++)
   {
       cin>>name;
       cin>>temp;
     //Insert the desired rank in vector v
        v.push_back(temp);
   }
   //Sort the vector containing the ranks
     sort(v.begin(),v.end());
   //This is the greedy approach
   for(int i=1;i<=n;i++)
   {
      //Take the difference between the i-th actual rank
      //and the i-th desired rank
       sum += abs(v[i-1],i);
     }
  printf(“%lld\n”,sum);
 }
return 0;
}
```

T = O(nlogn)

Can we do better?

```
#include <bits/stdc++.h>
using namespace std;
#define abs(a,b) a > b ? a - b : b - a
#define ll long long
int arr[100000+10];
int main()
{
  int t,n,temp;
  cin>>t;
   while(t—)
  {
     memset(arr,0,sizeof arr);
      string name;
      ll sum=0;
      cin>>n;
   for(int i=0;i<n;i++)
  {
     cin>>name;
    cin>>temp;
    //Increment the desired rank index in array arr
    arr[temp]++;
  }
   int pos = 1;
  //This is the greedy approach
  for(int i=1;i<=n;i++)
  {
    //If the i-th rank was desired by atleast one team
    //Assign and increment the actual rank index 'pos'
  //till this rank is desired
   while(arr[i])
  {
```

```
    sum += abs(pos,i);
    arr[i]—;
   pos++;
  }
 }
  printf("%lld\n",sum);
 }
return 0;
}
```

T = O(n)

❑ LOAD BALANCER

Rebalancing proceeds in rounds. In each round, every processor can transfer at most one job to each of its neighbors on the bus. Neighbors of the processor i are the processors i-1 and i+1 (processors 1 and N have only one neighbor each, 2 and N-1 respectively). The goal of rebalancing is to achieve that all processors have the same number of jobs.Determine the minimal number of rounds needed to achieve the state when every processor has the same number of jobs, or to determine that such rebalancing is not possible.

http://www.spoj.com/problems/BALIFE/

Greedy Approach: First find the final load that each processor will have. We can find it by sum(arr[1], arr[2],...,arr[n])/n, let us call it **load.**

So in the final state, each of the processor will have final load = load.

For each index i from 1 to n-1, create a partition of (1...i) and (i+1...n) and find the amount of load that is to be shared between these two partitions. The answer will be maximum of all the loads shared between all the partitions with i varying from 1 to n-1.

Example **:**

Initial state: 4, 8, 12, 16

Final state: 10 10 10 10

i = 1:

partition: (4) (8,12,16)

Load —> (4) needs 6 and (8,12,16) needs to give 6.

max_load = 6

i = 2:

partition: (4,8) (12,16)

Load—> (4,8) needs 8 and (12,16) needs to give 8.

max_laod = 8

i = 3:

partition: (4,8,12) (16)

Load—> (4,8,12) needs 6 and (16) needs to give 6.

max_laod = 6

Final answer = 8

Why does it works?

In all the partitions where less than max_load is transferred, we can internally transfer the load between these partitions when max_laod is being transferred between the max_load partition.

t = 2s:

when load = 2 is transferred between (4,8) and (12,16)

we can transfer load = 2 between (8) —> (4) and (16)—> (12)

State: 6 8 12 14

t = 4s:

when load = 2 is transferred between (6,8) and (12,14)

we can transfer load = 2 between (8) —> (6) and (14) —> (12)

State: 8 8 12 12

t = 6s:

when load = 2 is transferred between (8,8) and (12,12)

State: 8 10 10 12

t = 8s:

when load = 2 is transferred between (8,10) and (10,12)

we can transfer load = 2 between (10) —> (8) and (12) —> (10)

State: 10 10 10 10

Code :

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
```

```
int arr[9000],n, i, val, diff;
while(1)
{
  int max_load = 0, load = 0;
  cin>>n;
  if(n == -1)
    break;
  for(int i = 0;i<n;i++)
  {
     cin>>arr[i];
      load += arr[i];
  }
  //If we cannot divide the load equally
if(load % n)
{
    cout<<-1<<endl;
    continue;
 }
 //Find the load that is to be divided equally
load /= n;
 //Greedy step
for(int i = 0;i<n;i++)
{
   //At each iteration, find the value
    //of difference between final load to be assigned
     //and current load
       //Keep adding this difference in 'diff'
       diff += (arr[i] - load);
      //If the net difference is negative i.e.
      //we need diff amount till i-th index
 if(diff < 0)
    max_load = max(max_load, -1*diff);
```

```
    //If diff is positive i.e. we have to
   //give diff amout to (n-i) processors
    else
      max_load = max(max_load, diff);
    //calculate the max of load that can be given or
    //taken at each iteration .
  }
  cout<<max_load<<endl;
 }
 return 0;
}
```

T = O(n)

- **DEFENSE OF A KINGDOM, SPOJ**

Problem Statement

Given H,W of a field, and location of towers which guard the horizontal and the vertical lines corresponding to their positions. Find out the largest unbounded area(white rect). Refer the diagram on spoj.

http://www.spoj.com/problems/DEFKIN/

Greedy Approach: Given w, h as width and height of the playing field, and the coordinates of the towers as (x1,y1) ... (xN,yN), split the coordinates into two lists x1...xN, y1...yN, sort both of those coordinate lists.

Then calculate the empty spaces, e.g. dx[] = { x1, x2-x1, ..., xN - xN-1, (w + 1) - xN }. Do the same for the y coordinates: dy[] = { y1, y2-y1,..., yN - yN-1, (h + 1) - yN }. Multiply max(dx)-1 by max(dy)-1 and you should have the largest uncovered rectangle. You have to decrement the delta values by one because the line covered by the higher coordinate tower is included in it, but it is not uncovered.

Code :

```
#include<iostream>
#include<algorithm>
using namespace std;
int point_x[40000+10], point_y[40000+10];
int main()
{
```

```
    int t,w,h,n,x,y;
    scanf("%d",&t);
   while(t—)
   {
         scanf("%d %d %d",&w,&h,&n);
           for(int i = 0;i<n;i++)
            {
                scanf("%d %d",&point_x[i],&point_y[i]);
            }
          //sort the x-coordinates of the list
           sort(point_x, point_x + n);
           //sort the y-coordinates of the list
           sort(point_y, point_y + n);
            //dx —> maximum uncovered tiles in x coordinate
            //dy —> maximum uncocered tiles in y coordinate
            //Initially dx and dy are the first guars's position
             int dx = point_x[0],dy = point_y[0];
            //calculate the maximum uncovered gap
             //in x and y coordinate
            for(int i = 1;i<n;i++)
            {
                  dx = max(dx,point_x[i] - point_x[i-1]);
                   dy = max(dy,point_y[i] - point_y[i-1]);
            }
           dx = max(dx, w + 1 - point_x[n-1]);
           dy = max(dy, h + 1 - point_y[n-1]);
       printf("%d\n",((dx-1) * (dy-1)));
    }
   return 0;
}
```

T = O(nlogn)

- **QUES. CHOPSTICKS**

Given N sticks of length L[1], L[2], ..., L[N] and a positive integer D. Two sticks can be paired if the difference of their length is at most D.

Find the max number of pairs.

https://www.codechef.com/problems/TACHSTCK

Greedy Appraoch :

- Sort the list of sticks according to their lengths.
- If L[1] and L[2] cannot be paired, L[1] is useless.
- Else pair L[1] and L[2] and remove them from the list.
- Repeat steps 2-3 till the list become empty.

Why does this works?

- By pairing starting stick to its immediate next, we have max number of options left for next pairing.
- If L[1] and L[2] can be paired, but instead we pair (L[1], L[M]) and (L[2], L[N]).

Code :

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
typedef long long ll;
int main()
{
    ll n,d,a;
   vector<ll> v;
   cin>>n>>d;
  for(int i = 0;i<n;i++)
  {
    cin>>a;
    v.push_back(a);
  }
   //Sort the list of sticks
   sort(v.begin(),v.end());
  int res = 0;
   for(int i = 0;i<n-1;)
   {
    //If the adjacent difference is less than d
      //we can make a pair
     //Now remove these two sticks from the list
```

```
      if(v[i+1] - v[i] <= d)
       {
         res++;
          i+=2;
        }
       //If we cannot make a pair with adjacent stick
        //This stick is useless, remove this stick from the list
        else
            i++;
      }
   cout<<res<<endl;
  return 0;
}
```

T = O(nlogn)

- **EXPEDITION, SPOJ**

A damaged truck needs to travel a distance of L. The truck leaks one unit of fuel for every unit distance it travels.There are N (<=10^4)fuel stops on the path, what is the minimum number of refueling stops that the truck needs to make given that it has P amout of fuel initially.

http://www.spoj.com/problems/EXPEDI/

The first observation is, if you want to reach the city, say, point L, you **have to ensure that every single point between the current position and city must also be reachable.**

Now, the task is to minimize the number of stoppages for fuel, which is at most 10000. So, we sort the fuel stations, and start from current **position. For every fuel station, if we want to reach it, we must have fuel f more than or equal to the distance d. Also, using the** larger capacities will always reduce the number of stations we must stop.

How does greedy work?

If we make sure that we only stop at the largest capicity fuel stations

upto the point where the fuel capacity of truck becomes 0, number of

stops will be minimized.

Example:

Initial capacity = 11

Fuel Stations: 3 8 10 12

Capacity: 4 10 2 3

Initially, we don't have enough fuel to reach 12.

So we can reach upto maximum 11th city.

In order to minimize our stops, we will stop at 8th city

where fuel capacity is maximum.

Code:

```
#include<iostream>
#include<vector>
#include<queue>
#include<algorithm>
using namespace std;
bool cmpr(pair<int,int> l,pair<int,int> r)
{
   return l.first > r.first;
}
int main()
{
  int n,t,x,d,f,D,F,prev = 0;
  scanf("%d",&t);
 while(t—)
 {
   int flag = 0,ans = 0;
  vector<pair<int,int> > v;
  priority_queue<int> pq;
  scanf("%d",&n);
  for(int i = 0;i<n;i++)
  {
       scanf("%d %d",&d,&f);
      //Insert the city index and fuel capacity
       v.push_back(make_pair(d,f));
  }
  //Sort the cities according to their location
        sort(v.begin(),v.end(),cmpr);
        scanf("%d %d",&D,&F);
       //Calculate the difference between the current city and the
          //destination i.e. v[i] = j means that we need to travel j
```

```
        //units to rach our destination
      for(int i = 0;i<n;i++)
     {
       v[i].first = D - v[i].first;
      }
      //prev denotes the previous city visited
       prev = 0;
      //x will denote the current city that we are in
         x  = 0;
  while(x < n)
{
//cout<<x<<" "<<F<<" "<<v[x].first<<" "<<prev<<" "<<endl;
//If we have enough fuel to travel from prev to current city
//Push this fuel station to priority_queue
 //Reduce the amount of fuel used
 //update the previous city
   if(F >= (v[x].first - prev))
     {
        F -= (v[x].first - prev);
       pq.push(v[x].second);
      prev = v[x].first;
     }
    //If we dont have enough fuel to visit
   //the current city
    //Find the max capacity fuel station between
    //prev and current city and use it to refuel
  else
   {
//If no fuel station is left
  //i.e. we have used all of
//the fuel stations and still not able
   //to reach the city
  //return FAIL!
  if(pq.empty())
    {
```

```
            flag = 1;
            break;
          }
      //Increment the fuel capacity of truck
      //by the maximum fuel station capacity
       F += pq.top();
      //Remove that fuel station from heap
       pq.pop();
       //Increment the number of used fuel
       //station by 1
        ans++;
      continue;
     }
     //If we have visited the current city
      //Visit next city
        x++;
    }
 if(flag)
{
  printf("-1\n");
 continue;
}
  //Find the distance between the destination
  //and last city
   //Check if it is possible to visit the destination
     //from the last city.
D = D - v[n-1].first;
if(F >= D)
{
   printf("%d\n",ans);
  continue;
}
while(F < D)
 {
    if(pq.empty()){
```

```
            flag = 1;
            break;
            }
        F += pq.top();
        pq.pop();
        ans++;
    }
    if(flag){
        printf("-1\n");
        continue;
      }
    printf("%d\n",ans);
  }
 return 0;
}
```

T = O(NlogN)

- **GREEDY KNAPSACK PROBLEM, CODECHEF**

You are given N items, each item has two parameters: the weight and the cost. Let's denote M as the sum of the weights of all the items. Your task is to determine the most expensive cost of the knapsack, for every capacity 1, 2, ..., M. The capacity C of a knapsack means that the sum of weights of the chosen items can't exceed C.

https://www.codechef.com/problems/KNPSK

The key observation here is that weight of each item is either 1 or 2.

Greedy Approach: Greedily pickup the most costliest item with

weight <= 2, which can be taken in the knapsack.

Case 1: W is even

Select the most expensive item with sum of weight = 2. This can be done in two ways:

- Take the most expensive item of weight 2
- Or take at most two most expensive item of weight 1

Note that after picking up the most expensive item with sum of weight <= 2, we will remove the item taken and will recursively select the items to fill the most expensive elements.

Case 2: W is odd

We can simply select the most expensive item of weight 1. Now, we don't consider this item again. Now we have to select the most expensive weights from the remaining items. Since W-1 is even, we can solve this problem similar to case 1.

Example:

1 7 1 100 2 70

2 44 1 56 2 44

1 56 1 33 2 1

2 1 1 18

1 18

1 33

2 70

Case 1: Even

W = 2: (1,100) + (1,56) = 156

W = 4: 156 + (2,70) = 226

W = 6: 226 + (1,33) + (1,18) = 277

W = 8: 277 + (2,44) = 321

W = 10: 321 + (2,1) = 322

Case 2: Odd

W = 1: (1,100) = 100

W = 3: 100 + (1,56) + (1,33) = 189

W = 5: 189 + (2,70) = 259

W = 7: 259 + (2,44) = 303

W = 9: 321 + (1,18) = 321

Code :

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<stdio.h>
using namespace std;
long long cost[200000+10];
int main()
{
   int n,w,c,m,W = 0;
```

```
 //Separate one and two for even and odd casses
 vector<int> one,two,One,Two;
scanf("%d",&n);
for(int i = 0;i<n;i++)
  {
     scanf("%d %d",&w,&c);
     //Insert weight 1 items in one vector
     if(w == 1)
        one.push_back(c);
      //Insert weight 2 items in two vector
      else
         two.push_back(c);
      W += w;
    }
   //sort the two vectors
     sort(one.begin(),one.end());
      sort(two.begin(),two.end());
    One = one;
    Two = two;
    long long sum = 0,cur = 0,res = 0;
     //even case
     for(int i = 2;i<=W;i+=2)
        {
             //res1 —> when we take atmost two 1 wight items
             //res2 —> when we take single 2 weight item
              long long res1 = 0, res2 = 0;
              //calc res2
          if(two.size() > 0)
            {
                  res2 = two[two.size()-1];
             }
          //calculate res1
          if(one.size() > 1)
            {
               res1 = one[one.size()-1] + one[one.size()-2];
            }
```

```
        else if(one.size() > 0)
          {
                res1 = one[one.size() - 1];
          }
          //if res2 > res1 remove the 2 weight item
         if(res2 > res1)
            {
               two.pop_back();
                res += res2;
             }
           //remove at most two 1 weight items
          else
            {
              if(one.size() > 1)
                {
                   one.pop_back();
                   one.pop_back();
                }
               else
                  one.pop_back();
              res += res1;
            }
         //update the max cost for weight i
          cost[i] = res;
        }
      //odd case
         //subtract 1 to make the weight sum even
        res = 0;
       //Find the most expensive 1 weight item
      //and remove it from the list
      if(One.size() > 0){
           res = One[One.size()-1];
            One.pop_back();
         }
        cost[1] = res;
       //Similar to even case
```

```
        for(int i = 3;i<=W;i+=2)
        {
            long long res1 = 0, res2 = 0;
            if(Two.size() > 0)
                {
                    res2 = Two[Two.size()-1];
                }
            if(One.size() > 1)
                {
                    res1 = One[One.size()-1] + One[One.size()-2];
                }
              else if(One.size() > 0)
              {
                  res1 = One[One.size()-1];
              }
              if(res2 > res1)
              {
                 Two.pop_back();
                 res += res2;
              }
              else
              {
                    if(One.size() > 1)
                      {
       One.pop_back();
       One.pop_back();
      }
       else
          One.pop_back();
    res += res1;
   }
  cost[i] = res;
 }
for (int i = 1; i <= W; i++)
{
  if (i > 1)
```

```
        printf(" ");
      printf("%lld", cost[i]);
     }
   printf("\n");
  return 0;
}
```

T = O(nlogn)

Problems to Try :

1. **Fractional Knapsack :** Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.We can break items for maximizing the total value of knapsack.
2. **Huffman Coding :** 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. Read about this problem and implement it yourself.
3. **Maximum Circles (HackerBlocks) :** There are *n* circles arranged on x-y plane. All of them have their centers on x-axis. You have to remove some of them, such that no two circles are overlapping after that. Find the minimum number of circles that need to be removed.
4. **Maximum Unique Segment(Codechef) :** You are given 2 arrays W = (W_1, W_2, .., W_N) and C = (C_1, C_2, .., C_N) with N elements each. A range [l, r] is *unique* if all the elements C_l, C_{l+1}, .., C_r are unique (ie. no duplicates). The *sum* of the range is $W_l + W_{l+1} + ... + W_r$

   You want to find an *unique* range with the maximum *sum* possible, and output this sum.
5. **Station Balance - UVa 410**

Read and solve the problem statement Online

https://uva.onlinejudge.org/external/4/410.pdf

6. **DIE HARD (Spoj)**
7. **GERGOVIA (Spoj)**
8. **SOLDIER (Spoj)**
9. **CHOCOLA (Spoj)**
10. **CMIYC (Spoj)**

SELF STUDY NOTES

SELF STUDY NOTES

6 Recursion & Backtracking

Introduction :

Backtracking is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem. Backtracking is also known as depth-first search or branch and bound. Recursion technique is always used to solve backtracking problems. In this article, we will see the concept of recursion and backtracking.

Recursion :

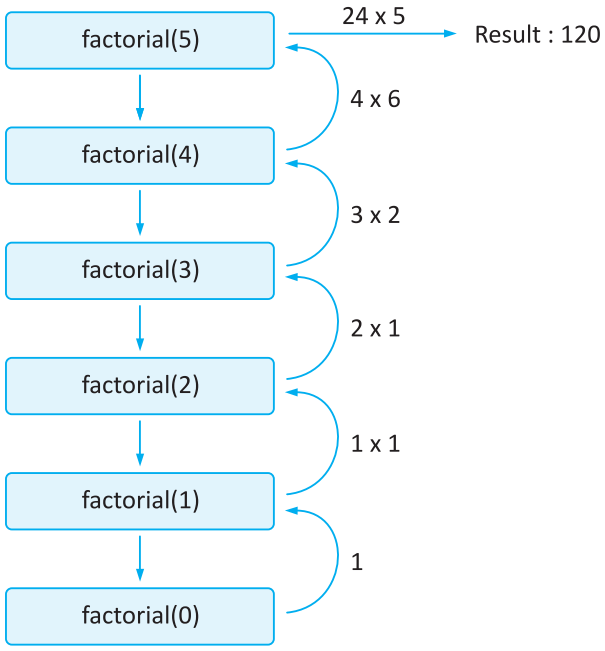
When a function calls itself, its called Recursion. It will be easier for those who have seen the movie Inception. Leonardo had a dream, in that dream he had another dream, in that dream he had yet another dream, and that goes on. So it's like there is a function called dream(), and we are just calling it in itself.

```
function  dream()
{
     print "Dreaming";
     dream();
}
```

Recursion is useful in solving problems which can be broken down into smaller problems of the same kind. But when it comes to solving problems using Recursion there are several things to be taken care of. Let's take a simple example and try to understand those. Following is the pseudo code of finding factorial of a given number X using recursion.

```
function factorial(x)
{
   if (x==  0)              // base case
      return 1;
    return x*factorial(x-1);       // break into smaller problem(s)
}
```

The following procedure shows how it works for factorial(5)



Base cases for Recusive Program :

Any recursive method must have a terminating condition. Terminating condition is one for which the answer is already known and we just need to return that. Ex. For the factorial problem, we know that factorial(0) = 1 , so when x is 0 we simply return 1, otherwise we break into smaller problem i.e. find factorial of (xd1). If we don't include a Base Case, the function will keep calling itself, and ultimately will result in stack overflow. For example, the dream() function given above has no base case. If you write a code for it in any language, it will give a runtime error.

Limitation of Recursive Call :

There is an upper limit to the number of recursive calls that can be made. To prevent this make sure that your base case is reached before stack size limit exceeds.

So, if we want to solve a problem using recursion, then we need to make sure that:

- The problem can broken down into smaller problems of same type.
- Problem has some base case(s).
- Base case is reached before the stack size limit exceeds.

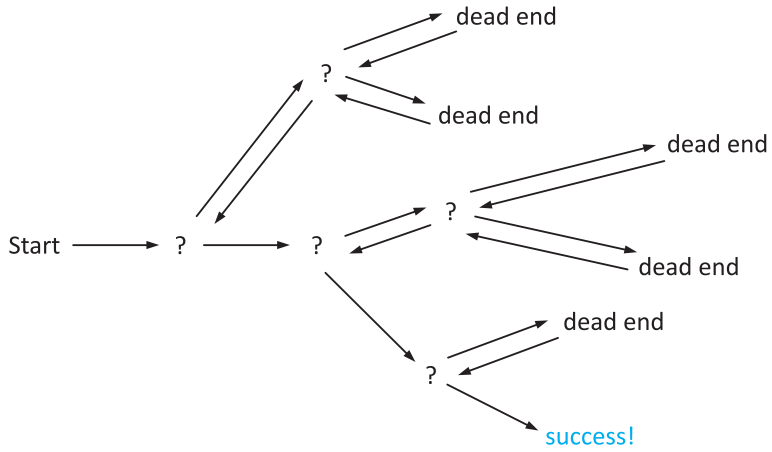
Backtracking :

When we solve a problem using recursion, we break the given problem into smaller ones. Let's say we have a problem **PROB** and we divided it into three smaller problems **P1, P2** and **P3**. Now it may be the case that the solution to PROB does not depend on all the three subproblems, in fact we don't even know on which one it depends.

Let's take a situation. Suppose you are standing in front of three tunnels, one of which is having a bag of gold at its end, but you don't know which one. So you'll try all three. First go in tunnel 1, if that is not the one, then come out of it, and go into tunnel 2, and again if that is not the one, come out of it and go into tunnel 3. So basically in backtracking we attempt solving a subproblem, and if we don't reach the desired solution, then undo whatever we did for solving that subproblem, and again try solving another subproblem.

Basically **Backtracking** is an algorithmic paradigm that tries different solutions until finds a solution that "works". Problems which are typically solved using backtracking technique have following property in common. These problems can only be solved by trying every possible configuration and each configuration is tried only once.

Note : We can solve this by using **recursion method**.



Problems based on Backtracking :

1. **N-Queen Problem :** Given a chess board having N×N cells, we need to place N queens in such a way that no queen is attacked by any other queen. A queen can attack horizontally, vertically and diagonally.

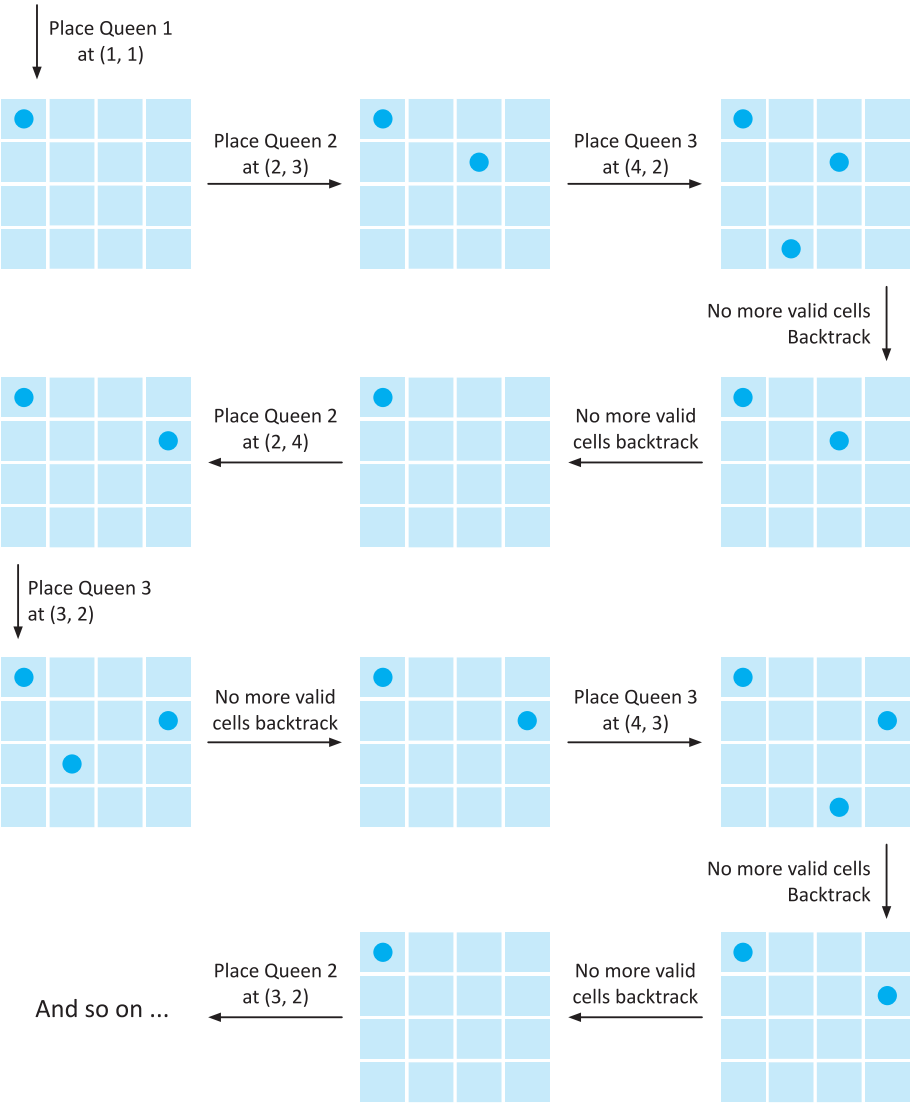
   ### Approach towards the Solution:

   We are having N × N unattacked cells where we need to place N- queens. Let's place the first queen at a cell (i, j), so now the number of unattacked cells is reduced, and number of queens to be placed is N – 1. Place the next queen at some unattacked cell. This again reduces the number of unattacked cells and number of queens to be placed becomes N – 2. Continue doing this, as long as following conditions hold.

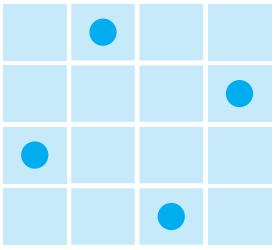
   - The number of unattacked cells is not equal to 0.
   - The number of queens to be placed is not equal to 0.

   If the number of queens to be placed becomes 0, then it's over, we found a solution. But if the number of unattacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.

Let's N = 4, We have to place 4 queen in 4×4 cells.



So, final solution of this problem is

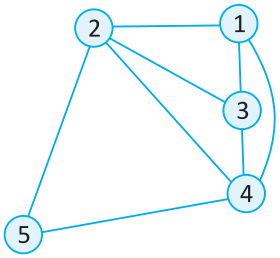


So, clearly, we tries solving a subproblem, if that does not result in the solution, it undo whatever changes were made and solve the next subproblem. If the solution does not exists (like N=2), then it returns false.

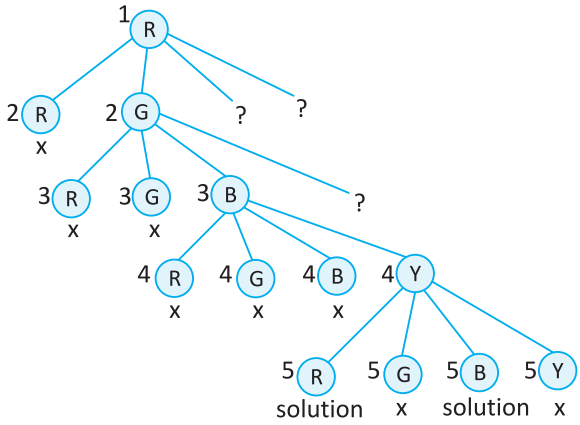
2. M Coloring Problems :

Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with same color. Here coloring of a graph means assignment of colors to all vertices.

Suppose we have a graph G in the given figure and we need to color the vertex of this graph in such away that no two adjacent vertices have same color.



This graph coloring problem can be solved using the mechanism of Backtracking concept.
The state space tree for the above map is shown below :



Hence the above graph can be colored using four colors and four colors are Red, Green, Blue and Yellow.

3. Robots on Ice :

In this problem , we have given a grid size and a sequence of three check-in points. We need to find how many different tours are possible for to check-in all those points with equal time interval. Let's try to understand this problem with example.

Suppose the given grid size is 3 × 6 and that the three check-in points , in order to visitation are (2,1), (2, 4) and (0, 4). Robot must start at (0, 0) and end at (0, 1) . It must visit location (2, 1) on step 4 (= #18⁄4#), location (2, 4) on step 9 (= #18⁄2#), and location (0, 4) on step 13 (= #3 × 18⁄4#) and at the end it must visit all 18 squares.

Approach towards the solution :

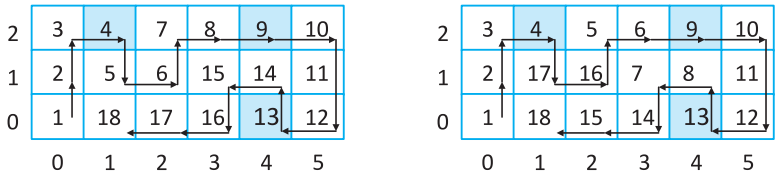
This problem has been break into three main subproblems. In first subproblem we have to check all those path via we can reach to square (2, 1) starting from (0, 0) in 4 steps.

After this we have to check all those path via we can reach to square (2,4) starting from (2,1) in 5 (= 9-4) steps. Then we have to check all those paths via we can reach to square (0,4) starting from (2,4) in 4 (=13-9) steps. Now finally we need to rached at squares (0, 1) in 5 (=18 – 13) steps.

In above process, we have to remind that Robot must visit every square only once.

This is a backtracking problem where we have to check all possible paths to check-in all three given points.

In this given example , there are only two ways to do this.



4. Rat in a Maze :

Given a maze, NxN matrix. A rat has to find a path from source to des-ti-na-tion. maze[0][0] (left top corner)is the source and maze[N-1][N-1](right bot-tom cor-ner) is des-ti-na-tion. There are few cells which are blocked, means rat can-not enter into those cells. The rat can move only in two directions: forward and down.

Approach Towards the Solution :

Here we have to generate all paths from source to destination and one by one check if the generated path satisfies the constraints or not.

Let's the given maze is in the form of matrix whose entries are either 0 or 1. Entry 0 represent the block path from where the rat can't move. We have to print another matrix whose entries are either 0 or 1. In output matrix, entry 1 represents the the path of Rat from starting point to destination point.

Algorithm to generate all the Paths :

While there are untried paths

```
{
     Generate the next paths
     If this path has all blocks as 1
       {
              Print this path;
       }
}
```

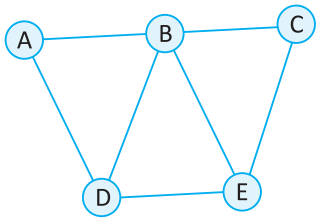
Backtracking Algorithm :

If destination is reached
  Print the solution
Else
 {
a) Mark current cell in solution matrix as 1.
b) Move forward in horizontal direction and recursively check if this move leads to a solution.
c) If the move chosen in the above step doesn't lead to a solution then move down and check of this move leads to a solution.
d) If none of the above solutions work then unmark this cell as 0 (Backtrack) and return false.
 }

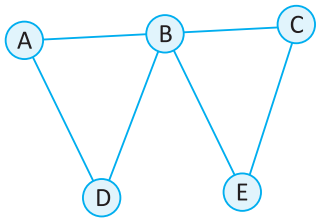
1. Finding Hamiltonian Cycle :

Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A **Hamiltonian cycle (or Hamiltonian circuit)** is a Hamiltonian Path such that there is an edge (in graph) from the last vertex to the first vertex of the Hamiltonian Path.Determine whether a given graph contains Hamiltonian Cycle or not.

Example :



A Hamiltonian Cycle in the above graph is {A, B, C, E, D, A}. There are more Hamiltonian Cycles in the graph like {A, D, E, C, B, A}.



There is no any Hamiltonian Cycle in the above graph.

Approach towards the solution :

A naive algorithm is to generate all possible configurations of vertices and print a configuration that satisfies the given constraints. It will be n! (n factorial) configurations.

While there are untried configurations
{

Generate the next configuration

If (there are edges between two consecutive vertices of this configuration and there is an edge from the last vertex to the first)

```
{
    Print this configuration;
    break;
}

}
```

Backtracking Algorithm :

Create an empty path array and add vertex 0 to it. Add other vertices, starting from the vertex 1. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return false.

TIME TO TRY

1.	Knight's Tour	2.	N-Queen	3.	Sudoku Solver
4.	Permutations	5.	Rat in a Maze	6.	T-shirts (Cochef)

SELF STUDY NOTES

SELF STUDY NOTES

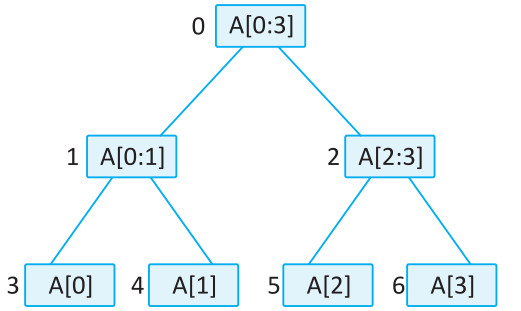
7 Segment Tree

Divide & Conquer

What is a Segment Tree?

A Segment Tree is a full binary tree where each node represents an interval.

Generally a node would store one or more properties of an interval which can be queried later.



Segment Tree of an Array consisting of four elements

Why do we need Segment Tree?

Many problem require that we give results based on query over a range or segment of available data. This can be a tedious and slow process, especially if the number of queries is large.

A segment tree let's us process such queries efficiently in logarithmic order of time.

How do we make a Segment Tree?

Let the data be in array arr[]

1. The root of our tree will represent the entire interval of data we are interested in, i.e, arr[0...n-1].
2. Each leaf of the tree will represent a range consisting of just a single element. Thus the leaves represent arr[0], arr[1], … , arr[n-1].
3. The internal nodes will represent the merged result of their child nodes.
4. Each of the children nodes could represent approximately half of the range represented by their parent.

Segment Tree generally contain three types of methods: **BUILD, QUERY, UPDATE.**

Let's Start with an Example

Problem : Range Minimum Query

You are given a list of N numbers and Q queries. There are two types of query:

1. 0 l r - find the minimum element in range [l,r].
2. 1 i a - update the element at ith position of array to val.

The problem states that we have to find minimum element in a given range [i,j], or update the element of the given array.

Basic Approach

For any range [i, j], we can answer the minimum element in O(n). But as we'll be having Q queries then we have to find minimum of a range, Q times.

So the overall complexity will be O(QxN).

So, to solve range based problems and to bring the complexity to logarithmic time we use Segment Tree.

We'll use tree[] to store the nodes of our Segment Tree(initialized to all zeros).

- The root of the tree is at index 1, i.e, tree[1] is the root of our tree.
- The children of tree[i] are stored at tree[i * 2] and tree[i * 2 +1].

```
#include <iostream>
using namespace std;
int tree[400005]; //will store our segment tree structure
int arr[100005]; //input array
```

BUILD Function :

Here, Build Function takes three parameter, **node, a, b.**

```
void build_tree(int node, int a, int b)
{ //BUILD function
// node represents the current node number
// a, b represents the current node range
// for leaf, node range will be single element
// so 'a' will be equal to 'b' for leaf node
if (a == b)
  { //checking if node is leaf
    //for single element, minimum will be the element itself
    tree[node] = arr[a]; //storing the answer in our tree structure
    return;
  }
```

```
   int mid = (a + b) >> 1; //middle element of our range
   build_tree(node * 2, a, mid); //recursively call for left half
   build_tree(node * 2 + 1, mid + 1, b); //call for right half
    //left child and right child are build
    // now build the parent node using its child nodes
  tree[node] = __min(tree[node*2], tree[node * 2 + 1]);
}
```

QUERY Function

```
int query_tree(int node, int a, int b, int i, int j)
{
// i, j represents the range to be queried
 if (a > b || a > j || b < i) return INT_MAX; //out of range
 if (a >= i && b <= j)
 { //current segment is totally within range[i,j]
   return tree[node];
 }
 int mid = (a + b) >> 1;
 //Query left child
 int q1 = query_tree(node * 2, a, mid, i, j);
 //Query right child
 int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
 return __min(q1, q2); // return final result
}
```

UPDATE Function

```
// updates the ith element to val
void update_tree(int node, int a, int b, int i, int val)
{
 if (a > b || a > i || b < i) return; //out of range
 if (a == b) { //leaf node
 tree[node] = val;
 return;
}
 int mid = (a + b) >> 1;
 update_tree(node * 2, a, mid, i, val); // updating left child
```

```
update_tree(node * 2 + 1, mid + 1, b, i, val);// updating right child
// updating node with min value
tree[node] = __min(tree[node * 2], tree[node * 2 + 1]);
}
```

Complexity Analysis

BUILD

We visit each leaf of the Segment Tree. That makes n leaves. Also there will be n-1 internal nodes. So we process about 2 * n nodes. This makes the Build process run in O(n) linear complexity.

UPDATE

The update process discards half of the range for every level of recursion to reach the appropriate leaf in the tree. This is similar to binary search and takes logarithmic time. After the leaf is updated, its direct ancestors at each level of the tree are updated. This takes time linear to height of the tree. So the complexity will be O(lg(n)).

QUERY

The query process traverses depth-first through the tree looking for node(s) that match exactly with the queried range. At best, we query for the entire range and get our result from the root of the segment tree itself. At worst, we query for a interval/range of size 1 (which corresponds to a single element), and we end up traversing through the height of the tree. This takes time linear to height of the tree. So the complexity will be O(lg(n)).

Problem : Range Sum

We have an array arr{0,.....n,– 1} and we have to perform two types of queries:

1. Find sum of elements from index l to r where 0 <= l <= r <= n
2. Increase the value of all elements from l to r by a given number.

As we can build the parent node using its two child nodes. So, we'll use the Segment Tree to answer the sum of elements from l to r.

Code :

```
int tree[400005]; //will store our segment tree structure
int arr[100005]; //input array
void build_tree(int node, int a, int b)
{ //BUILD function
if (a == b)
{ //checking if node is leaf
  //for single element, sum will be the element itself
```

```
   tree[node] = arr[a]; //storing the answer in our tree structure
   return;
}
int mid = (a + b) >> 1; //middle element of our range
build_tree(node * 2, a, mid); //recursively call for left half
build_tree(node * 2 + 1, mid + 1, b); //call for right half
//left child and right child are build
// now build the parent node using its child nodes
tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
int query_tree(int node, int a, int b, int i, int j)
{
// i, j represents the range to be queried
if (a > b || a > j || b < i)
   return 0; //out of range
if (a >= i && b <= j)
{ //current segment is totally within range[i,j]
 return tree[node];
}
int mid = (a + b) >> 1;
int q1 = query_tree(node * 2, a, mid, i, j); //Query left child
int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
//Query right child
return (q1 + q2); // return final result
}
void update_tree(int node, int a, int b, int i,int j, int val)
{
if (a > b || a > j || b < i)
   return; //out of range
if (a == b)
{ //leaf node
 tree[node] += val;
 return;
}
int mid = (a + b) >> 1;
```

```
update_tree(node * 2, a, mid, i, j, val);;// updating left child
update_tree(node * 2 + 1, mid + 1, b, i, j, val); // updating right child
tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

The above approach will query the tree in O(lg(n)) time. But the problem will arise in the update function. As updating a single element takes O(lg(n)) time, now we are doing a range update. So, the overall complexity for update will be O(nlg(n)), which will get TLE as we have to answer for multiple queries.

Lazy Propagation

In the current version when we update a range, we branch its childs even if the segment is covered within range. In the lazy version we only mark its child that it needs to be updated and update it when needed.

In short, we try to postpone updating descendants of a node, until the descendants themselves need to be accessed.

We use another array lazy[] which is the same size as our segment tree array tree[] to represent a lazy node. lazy[i] holds the amount by which the node tree[i] needs to be incremented, when that node is finally accessed or queried. When lazy[i] is zero, it means that node tree[i] is not lazy and has no pending updates.

UPDATE function

```
void update_tree(int node, int a, int b, int i,int j, int val)
{
if (lazy[node] != 0)
{ // lazy node
 tree[node] += (b - a + 1)*lazy[node]; //normalize current node by removing
laziness
 if (a != b)
 { // update lazy[] for children nodes
 lazy[node * 2] += lazy[node];
 lazy[node * 2 + 1] += lazy[node];
 }
 lazy[node] = 0; // remove laziness from current node
}
if (a > b || a > j || b < i)
  return; //out of range
if (a >= i && b <= j)
{ // segment is fully within update range
```

```
 tree[node] += (b - a + 1)*val;
 if (a != b)
 { // update lazy[] for children nodes
  lazy[node * 2] += lazy[node];
  lazy[node * 2 + 1] += lazy[node];
 }
return;
}
int mid = (a + b) >> 1;
update_tree(node * 2, a, mid, i, j, val);;// updatingleft child
update_tree(node * 2 + 1, mid + 1, b, i, j, val); //updating right child
tree[node] = tree[node * 2] + tree[node * 2 + 1];
}
```

QUERY function

```
int query_tree(int node, int a, int b, int i, int j)
{
// i, j represents the range to be queried
if (a > b || a > j || b < i)
 return INT_MAX; //out of range
if (lazy[node] != 0)
{ // lazy node
  tree[node] += (b - a + 1)*lazy[node]; //normalize current node by removing
laziness
 if (a != b)
 { // update lazy[] for children nodes
   lazy[node * 2] += lazy[node];
   lazy[node * 2 + 1] += lazy[node];
 }
 lazy[node] = 0; // remove laziness from current node
}
if (a >= i && b <= j)
{ //current segment is totally within range[i,j]
 return tree[node];
}
int mid = (a + b) >> 1;
int q1 = query_tree(node * 2, a, mid, i, j); //Query left child
```

```
int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
//Query right child
return (q1 + q2); // return final result
}
```

Using Lazy Propagation the range update will be done in O(lg(n))time complexity.

Note :

The following lines:

tree[node] += (b - a + 1)*lazy[node]; //normalize currentnode by removing laziness and

tree[node] += (b - a + 1)*val; // update segment and

tree[node] = tree[node * 2] + tree[node * 2 + 1]; //merge updates are specific to Range Sum Query problem. Different problems may have different updating and merging schemes.

Problem : GSS1 - Can you answer these queries

You are given a sequence A[1], A[2], ..., A[N] . (|A[i]| ≤ 15007, 1 ≤ N ≤ 50000). A query is defined as follows:

Query(x,y) = Max { a[i]+a[i+1]+...+a[j] ; x ≤ i ≤ j ≤ y }.

Given M queries, your program must output the results of these queries.

INPUT :

3

-1 -2 -3

1

1 2

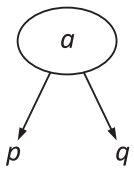
OUTPUT :

2

http://www.spoj.com/problems/GSS1/

Now we need to use a Segment Tree. But what data to store in each node, such that it is easy to compute the data associated with a given node if we already know the data associated with its two child nodes.

We need to find a maximum sum subarray in a given range.



Say we have a as a parent node and p and q as its child nodes. Now we need to build data for a from p and q such that node a can give the maximum sum subinterval for its range when queried.

So for this what do we need??

Maximum sum subarray in a can be equal to either:

1. Maximum sum subarray in p
2. Maximum sum subarray in q
3. Elements including from both p and q

So for each node we need to store:

1. Maximum Prefix Sum
2. Maximum Suffix Sum
3. Total Sum
4. Best Sum (Maximum Sum Subarray for a)

Maximum Prefix Sum can be calculated for a node as

a.prefix = max(p.prefix,p.total + q.prefix)

Maximum Suffix Sum can be calculated for a node as

a.suffix = max(p.suffix,q.total + p.suffix)

Total Sum

a.total = p.total + q.total

Best Sum

a.best = max(p.suffix + q.prefix , max(p.best,q.best))

Code :

```
long long int arr[50005]; //input array
struct Tree
{
 long long int prefix, suffix, total, best;
};
Tree tree[200005];
void build_tree(long long int node, long long int a, long long int b)
{
if (a == b)
{ //leaf node
 tree[node].prefix = arr[a]; // prefix sum
 tree[node].suffix = arr[a]; // suffix sum
 tree[node].total = arr[a]; // total sum
 tree[node].best = arr[a]; // best sum
 return;
}
```

```
int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
tree[node].prefix = max(tree[node * 2].prefix, tree[node*2].total + tree[node *
2 + 1].prefix); //prefix sum
tree[node].suffix = max(tree[node * 2 + 1].suffix, tree[node * 2].suffix +
tree[node * 2 + 1].total); //suffix sum
tree[node].total = tree[node * 2].total + tree[node *2 + 1].total; //total sum
tree[node].best = max(tree[node * 2].suffix + tree[node * 2 + 1].prefix,
max(tree[node * 2].best, tree[node *2 + 1].best)); //best sum
}
Tree query_tree(long long int node, long long int a, long long int b, long long int
i, long long int j)
{
Tree t;
if (a > b || a > j || b < i)
{
 t.prefix = t.suffix = t.best = INT_MIN ;
 t.total = 0;
 return t;
}
if (a >= i && b <= j)
{
 return tree[node];
}
long long int mid = (a + b) >> 1;
Tree q1 = query_tree(node * 2, a, mid, i, j);
Tree q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
t.prefix = max(q1.prefix, q1.total + q2.prefix);
t.suffix = max(q2.suffix, q1.suffix + q2.total);
t.total = q1.total + q2.total;
t.best = max(q1.suffix + q2.prefix, max(q1.best, q2.best));
return t;
}
```

Similar Problem: http://www.spoj.com/problems/GSS3/

The problem also includes an update operation, rest of the methods are same.

Problem : QSET (Queries on the String)

You have a string of N decimal digits, denoted by numbers A1,A2, ..., AN.

Now you are given M queries, each of whom is of following two types.

1. 1 X Y: Replace AX by Y.
2. 2 C D: Print the number of sub-strings divisible by 3 in range [C,D]

Formally, you have to print the number of pairs (i,j) such that the string Ai,Ai+1...Aj, ($C \le i \le j \le D$), when considered as a decimal number, is divisible by 3.

INPUT :

5 3
01245
2 1 3
1 4 5
2 3 5

OUTPUT :

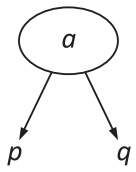
3
1

https://www.codechef.com/problems/QSET

A number is divisible by 3 if sum of its digits are divisible by 3.

Use segment trees. Store in node for each interval,

- the answer for that interval
- prefix[], where prefix[i] denotes number of prefixes of interval which modulo 3 give i.
- suffix[], where suffix[i] denotes number of suffixes of interval which modulo 3 give i.
- total sum of interval modulo 3.



Answer for interval denoted by **a** will be:

a.ans = p.ans + q.ans.

(Number of Suffixes in p which when taken with modulo 3 gives 1) + (Number of Prefixes in q which when taken with modulo 3 gives 2) —> also gives the substring divisible by 3 in **a**.

Similarly, for all possible combinations we can calculate the answer for an interval.

```
a.ans = p.ans + q.ans;
for (int i = 0; i < 3; ++i)
{
```

```
for (int j = 0; j < 3; ++j)
  {
   if ((i + j) % 3 == 0)
    {
      a.ans += p.suffix[i] * q.prefix[j];
    }
  }
}
```

How to build prefix and suffix array for a node?

There are p.prefix[i] prefixes of p which when taken modulo with 3 gives i, and there are some prefixes which are made by combining whole of p and some prefixes of q.

So, total prefixes in a which when taken modulo with 3 gives i will be:
(Same for suffix)

```
for (int i = 0; i < 3; ++i)
{
a.prefix[i] = p.prefix[i] + q.prefix[(3 - q1.total + i) % 3];
a.suffix[i] = q.suffix[i] + p.suffix[(3 - q2.total + i) % 3];
}
```

Code :

```
int arr[100005]; //input array
struct Tree
{
int ans;
int prefix[3], suffix[3];
int total;
};
Tree tree[400005];
void build_tree(int node, int a, int b)
{
if (a == b)
{
 tree[node].prefix[arr[a] % 3] = 1;
 tree[node].suffix[arr[a] % 3] = 1;
 tree[node].total = arr[a] % 3;
```

```
 tree[node].ans = (arr[a] % 3 == 0 ? 1 : 0);
 return;
}
int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
for (int i = 0; i < 3; ++i)
{
 tree[node].prefix[i] = tree[node * 2].prefix[i] +
 tree[node * 2 + 1].prefix[(3 - tree[node * 2].total + i)% 3];
tree[node].suffix[i] = tree[node * 2 + 1].suffix[i] + tree[node * 2].suffix[(3 -
tree[node * 2 + 1].total+ i) % 3];
}
tree[node].total = (tree[node * 2].total + tree[node* 2 + 1].total) % 3;
tree[node].ans = tree[node * 2].ans + tree[node * 2 +1].ans;
for (int i = 0; i < 3; ++i)
```

PROBLEM : JTREE (JosephLand)

Nick lives in a country named JosephLand. JosephLand consists of **N** cities. **City 1** is the capital city. There are **N - 1** directed roads. It's guaranteed that it is possible to reach capital city from any city, and in fact there is a unique path from any city to the capital city.

Besides, you can't cross roads for free. To pass a road, you must have a ticket. There are total **M** tickets. You can not have more than one ticket at a time! Each ticket is represented by three integers:

v k w : you can buy a ticket with cost w in the city v. This ticket can be used at max k times. That means, after using this ticket for k roads ticket can't be used!

By the way, you can tear your ticket any time and buy a new one. But you are not allowed to buy a ticket if you are still having a ticket with you!

Nick's home is located in the capital city. He has Q friends, and he wants to invite all of them for dinner! So he is interested in knowing about how much each of his friends is going to spend in the journey!

His friends are quite smart and always choose a route to capital city that minimizes his/her spending! Nick has to prepare dinner, so he doesn't have time to figure out himself, Can you please help him?

Please note that **it's guaranteed that, one can reach the capital from any city using the tickets!**

INPUT :

7 7
3 1
2 1
7 6
6 3
5 3
4 3
7 2 3
7 1 1
2 3 5
3 6 2
4 2 4
5 3 10
6 1 20
3
5
6
7

OUTPUT :

10
22
5

https://www.codechef.com/problems/JTREE

The problem in simple words is:

Given a tree with edges directed towards root and nodes having some ticket information which allows you to travel k units towards the root with cost w. Answer Q queries i.e output the minimum cost for travelling from a node x to root.

Let's try to solve problem for a node X at depth H from root 1. Think of this path as a 1D vector V where we have all the H-1 nodes between root and X. The nodes are stored in vector in increasing order of depth.

Let dp[x] be the minimum cost to travel from X to the root.

So, for a given node, iterate over all the tickets present at node X and DP state will be like this:

```
for(int  i=0;i<total_tickets;i++)
{
   K=current_ticket_jump_info;
   W=weight_ticket_info;
   for(int  j=V.size()-1;j>=  max(0,V.size()-1-k);j++)  //loop  2
```

```
   {
     DP[X]=min( DP[X] , DP[V[j]] + W );
   }
 }
```

We will use DFS to find vector V for every node X.

Now the code will be :

```
void dfs(int u,int p)
{
V.push_back(u);
for(int l=0 ; l< adj[u].size() ; l++)
{
   if(adj[u][l]==p)
     continue;
   int x =adj[u][l];
   for(int i=0;i<total_tickets;i++) // loop 1
  {
     K=current_ticket_jump_info;
    W=weight_ticket_info;
     for(int j=V.size()-1;j>= max(0,V.size()-1-k);j++) //loop 2
     {
       DP[x]=min( DP[x] , DP[V[j]] + W );
     }
  }
   dfs(adj[u][l],u);
 }
V.pop_back();
}
```

The complexity will be **O(n + m * n).**

But If we look at the inner loop of the code which goes up to K times and we find the minimum of DP, this can be done using any data structure that supports **RMQ(Range Minimum QUery) + Point updates** in **O(logn)** time . Which will make the total complexity to be **O(N + MlogN).**

So we can build a segment tree that supports two operations.

First : find the minimum element in range L,R and

Second: update an element's value to VAL.

And we need to query for the minimum element between [H-K , H] so we will build the tree over **height H**.

Now just print **DP[x]** for every query.

Code :

```
void dfs(long long int cur, long long int h)
{
for (int j = 0; j < n_info[cur].size(); ++j)
{ //no. of tickets
   long long int k1 = n_info[cur][j].first;
   long long int w1 = n_info[cur][j].second;
  dp[cur] = min(dp[cur], query_tree(1, 0, N - 1, max(0LL, h - k1), h) + w1); //
find the min cost from cur to root
}
update_tree(1, 0, N - 1, h, dp[cur]); //update the
tree at posn h with its DP value
for(int i = 0; i < g_tree[cur].size(); ++i)
{
 long long int x = g_tree[cur][i];
 dfs(x, h + 1);
}
update_tree(1, 0, N - 1, h, INF_n); //update the tree at posn h with INF as we
are done with its subtree
}
int main()
{
for(int i = 0; i < 400003; ++i)
{
tree[i] = INF_n;
 dp[i / 4] = INF_n;
}
 cin >> N >> M;
for(int i = 0; i < N - 1; ++i)
{
 cin >> a >> b;
  g_tree[b].push_back(a);
}
for(int i = 0; i < M; ++i)
```

```
{
 cin >> v >> k >> w;
  n_info[v].push_back(make_pair(k,  w));
}
dp[1] = 0;
update_tree(1, 0, N - 1, 0, dp[1]);
dfs(1, 0);
cin >> Q;
while (Q—)
{
 cin >> a;
 cout << dp[a] << “\n”;
}
return 0;
}
```

PROBLEM : COUNZ (Counting Zeroes)

You are given an array of N integers(Indexed at 1)

For the given array you have to answer some queries given later. The queries are of 2 types:

1. **TYPE 1 -> 1 L R (where L and R are integers)**

   For this query you have to calculate the product of elements of the array in the range L to R (both inclusive) and print the number of zeros at the end of the result.

2. **TYPE 2 -> 0 L R V (where L,R,V are integers)**

   For this query you have to set the value of all the elements in the array ranging from L to R (both inclusive) to V.

INPUT :

5
1 3 5 8 9
3
1 2 5
0 1 4 10
1 1 5

OUTPUT :

1
4

https://www.codechef.com/problems/COUNZ

We’ll use Segment Tree to answer the queries.

To determine the number of zeroes at the end of a number, we should know number of 2's and 5's in its prime factorization.

Number Of Zeroes = min(No. of 2's, No. of 5's)

Number of Zeroes in product = min(sum of 2's in elements in product, sum of 5's in elements in product).

Be careful of element having value 0 in product as number of zeroes will be 1.*

Since any array element can have value 0 at any point of time we store 3 values at a node in segment tree sum of number of twos in prime factorization of all the elements in range, sum of number of twos in prime factorization of all the elements in range and number of elements in range with value 0.

Now, using lazy propagation we can answer queries in **O(log2(MaxA[i])+logN) time.**

Total Complexity : O(Nlog2(Max A[i])+Q(log2(Max A[i])+logN))

Code :

```
void process_node(int node,int num)
{
tree[node][0] = tree[node][1] = tree[node][2] = 0;
if (num == 0)
   tree[node][0]++;
while (num % 2 == 0 && num != 0)
{
 num /= 2;
 tree[node][1]++; //no of 2 in PF of arr[a];
}
while (num % 5 == 0 && num != 0)
{
 num /= 5;
tree[node][2]++; //no of 5 in PF of arr[a];
}
return;
}
void build_tree(int node, int a, int b)
{
if (a == b)
{
int num = arr[a];
```

```
 process_node(node, num);
 return;
}
int mid = (a + b) >> 1;
build_tree(node * 2, a, mid);
build_tree(node * 2 + 1, mid + 1, b);
for (int i = 0; i < 3;++i) //build the parent node
 tree[node][i] = tree[node * 2][i] + tree[node * 2+ 1][i];
}
void update_tree(int node, int a, int b, int i, int j,int val)
{
if (lazy[node] != -1)
{ //if lazy
 process_node(node, lazy[node]);
 for (int i = 0; i < 3; ++i)
  tree[node][i] *= (b - a + 1);
 if (a != b)
 { //not a leaf node
  lazy[node * 2] = lazy[node * 2 + 1] = lazy[node]; //mark lazy
 }
lazy[node] = -1; //unmark lazy
}
if (a > b || a > j || b < i)return;
if (a >= i && b <= j)
{
 process_node(node, val);
 for (int i = 0; i < 3;++i)
  tree[node][i] *= (b - a + 1);
 if (a != b)
{
  lazy[node * 2] = lazy[node * 2 + 1] = val;
 }
 return;
}
int mid = (a + b) >> 1;
```

```
update_tree(node * 2, a, mid, i, j, val);
update_tree(node * 2 + 1, mid + 1, b, i, j, val);
for (int i = 0; i < 3; ++i)
  tree[node][i] = tree[node * 2][i] + tree[node * 2 + 1][i];
}

void query_tree(int node, int a, int b, int i, int j)
{
if (lazy[node] != -1)
{ //if lazy
 process_node(node, lazy[node]);
  for (int i = 0; i < 3; ++i)
   tree[node][i] *= (b - a + 1); //multiply value by range times
 if (a != b)
 { //if not leaf
 lazy[node * 2] = lazy[node * 2 + 1] = lazy[node];//mark as lazy
 }
 lazy[node] = -1; //unmark lazy
}
if (a > b || a > j || b < i)
{
 q_tree[node][0] = q_tree[node][1] = q_tree[node][2] = 0;
 return;
}
if (a >= i && b <= j)
{
 for (int i = 0; i < 3;++i)
    q_tree[node][i] = tree[node][i];
 return;
}
int mid = (a + b) >> 1;
query_tree(node * 2, a, mid, i, j);
query_tree(node * 2 + 1, mid + 1, b, i, j);
for (int i = 0; i < 3; ++i)
q_tree[node][i] = q_tree[node * 2][i] + q_tree[node * 2 + 1][i];
}
```

PROBLEM : DIVMAC (Dividing Machine)

Chef has created a special dividing machine that supports the below given operations on an array of positive integers.

There are two operations that Chef implemented on the machine.

```
Type 0 Operation
Update(L,R):
for i = L to R:
  a[i] = a[i] / LeastPrimeDivisor(a[i])
Type 1 Operation
Get(L,R):
result = 1
for i = L to R:
   result = max(result, LeastPrimeDivisor(a[i]))
return result;
```

The function **LeastPrimeDivisor(x)** finds the smallest prime divisor of a number. If the number does not have any prime divisors, then it returns 1.

Chef has provided you an array of size N, on which you have to apply M operations using the special machine. Each operation will be one of the above given two types. Your task is to implement the special dividing machine operations designed by Chef. Chef finds this task quite easy using his machine, do you too?

INPUT :

```
2
6 7
2 5 8 10 3 44
1 2 6
0 2 3
1 2 6
0 4 6
1 1 6
0 1 6
1 4 6
2 2
1 3
0 2 2
1 1 2
```

OUTPUT :

5 3 5 11

1

The problem is :

Given an array of numbers A, support the following two queries:

- for a given range [L, R], reduce each number in A[L...R] by its smallest prime factor
- for a given range [L, R], find the number in A[L...R] with largest smallest prime factor.

When we get the range modification entry, we modify each element in the range one by one, and hence perform (R - L + 1) single element update queries. This approach would be to slow, as it makes the complexity of range update query to **O ((R - L + 1) lg N)**

Note that, if the value of an element A[x] is 1, then update query on A[x] does not have any effect on the segment tree. We call such queries as degenerate queries, and can discard them. Also note that a number M can be written as a product of at most O (lg M) prime numbers, Hence, we can perform at most O (lg M) non-degenerate modification queries on A[x] = M all the latter queries will be degenerate. This means that if all numbers in the array are smaller than M, then a most O (N lg M) non-degenerate single element update queries would be performed. Each single element update query takes O (lg N) time, and hence all update queries can be performed in **O (N lg M lg N) time.**

Code :

```
void  modified_sieve()
{ //linear  time  sieve
lp[1] = 1;
for (int i = 2; i < maxn; ++i)
{
 if (lp[i] == 0)
 {
  lp[i] = i;
   prime.push_back(i);
 }
 for (int j = 0; j < prime.size() && i*prime[j] <maxn; ++j)
 {
  lp[i*prime[j]] = prime[j];
 }
}
}
void build_tree(int node, int a, int b)
{
 if (a == b)
 {
```

```
    tree[node] = lp[arr[a]];
    ones[node] = (arr[a] == 1 ? 1 : 0);
    return;
 }
 int mid = (a + b) >> 1;
 build_tree(node * 2, a, mid);
 build_tree(node * 2 + 1, mid + 1, b);
 ones[node] = ones[node * 2] + ones[node * 2 + 1];
 tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
}
void update_tree(int node, int a, int b, int i, int j)
{
if (a > b || a > j || b < i)
  return;
if (a >= i && b <= j)
{
 if (ones[node] == (b - a + 1))
 {
  return;
 }
 if (a == b)
 { //leaf node
  arr[a] /= lp[arr[a]];
  tree[node] = lp[arr[a]];
  ones[node] = (arr[a] == 1 ? 1 : 0);
  return;
 }
}
int mid = (a + b) >> 1;
update_tree(node * 2, a, mid, i, j);
update_tree(node * 2 + 1, mid + 1, b, i, j);
ones[node] = ones[node * 2] + ones[node * 2 + 1];
tree[node] = max(tree[node * 2], tree[node * 2 + 1]);
}
int query_tree(int node, int a, int b, int i, int j)
{
if (a > b || a > j || b < i)
{
 return INT_MIN;
}
```

```
if (a >= i && b <= j)
{
 return tree[node];
}
int mid = (a + b) >> 1;
int q1 = query_tree(node * 2, a, mid, i, j);
int q2 = query_tree(node * 2 + 1, mid + 1, b, i, j);
return q1 > q2 ? q1 : q2;
}
```

SELF STUDY NOTES

SELF STUDY NOTES