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.

Scanned by CamScanner

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);
```


CODING BLOCKS

Scanned by CamScanner

```
        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.

CODING BLOCKS
Scanned by CamScanner

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.

CODING BLOCKS

Scanned by CamScanner

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)

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

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])
 {
```


Scanned by CamScanner

```
    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:

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

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()
{
```


Scanned by CamScanner

```
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);
```


Scanned by CamScanner

```
    //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()
{
```


CODIN BLOC

Scanned by CamScanner

```
    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.

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

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
```


CODING BLOCKS

Scanned by CamScanner

```
        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

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

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
```


CODING BLOCKS

Scanned by CamScanner

```
        //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())
   {
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
            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()){
```


Scanned by CamScanner

```
            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.

Scanned by CamScanner

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;
```


CODING BLOCKS

Scanned by CamScanner

```
//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];
        }
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
        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
```


Scanned by CamScanner

```
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)
```


CODING BLOCKS
Scanned by CamScanner

```
        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)**

Scanned by CamScanner

Scanned by CamScanner

Scanned by CamScanner

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.

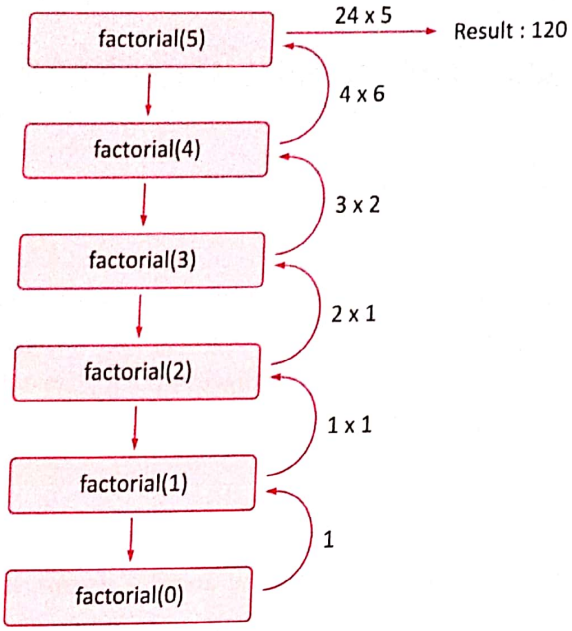
```
function  dream()
{
     print "Dreaming";
     dream();
}
```

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)
}
```


Scanned by CamScanner

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.

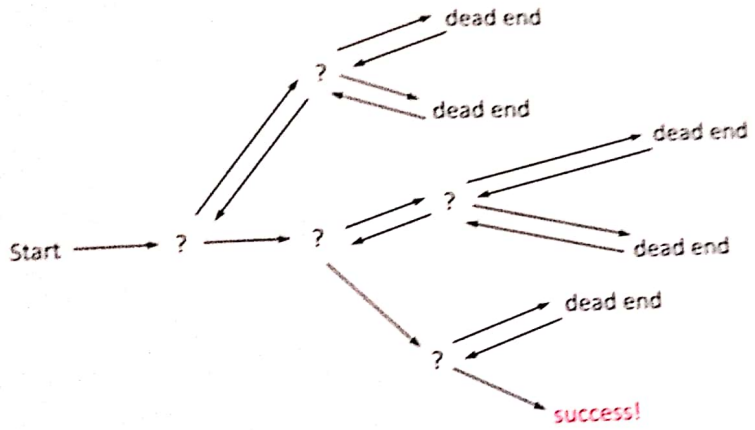
CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

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:

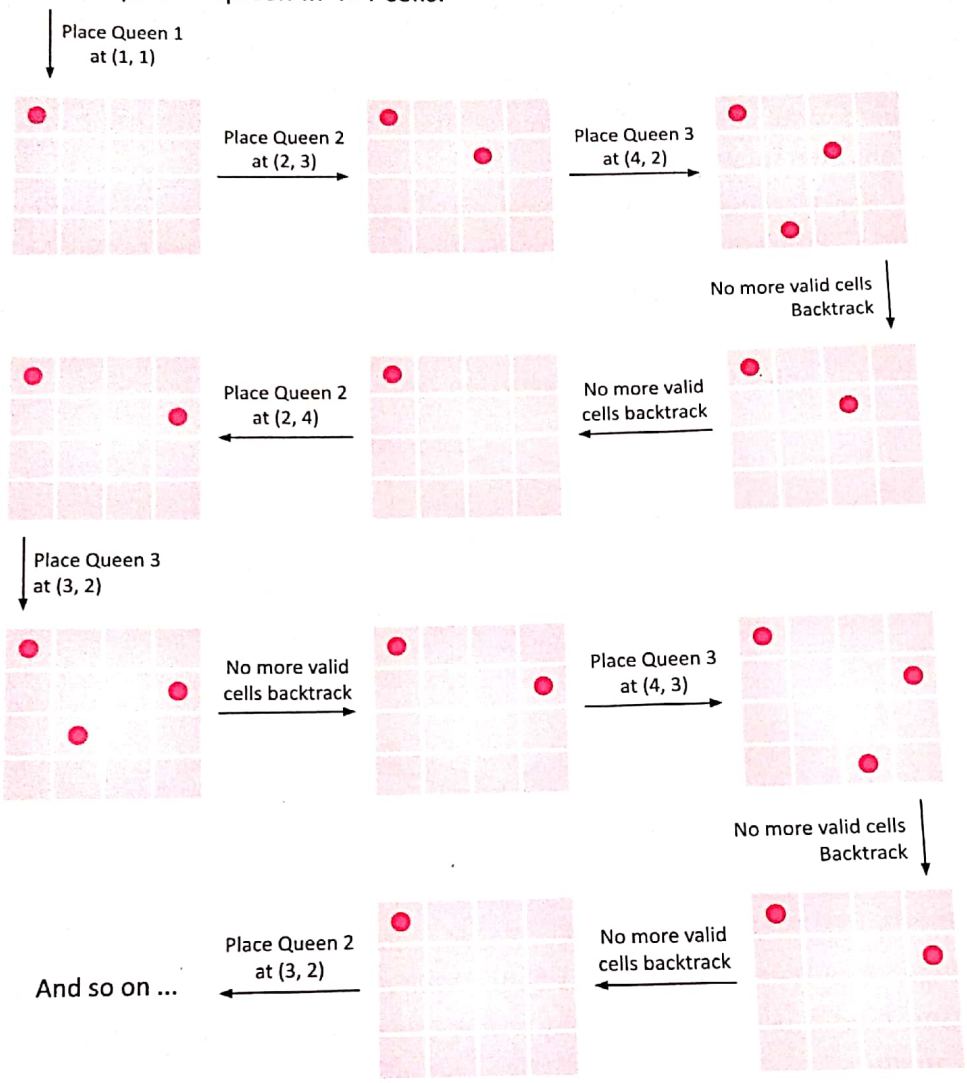
We are having N × N unattacked cells where we need to place N- queens. Let's place the first queen at a cell (i, j), so now the number of unattacked cells is reduced, and number of queens to be placed is N – 1. Place the next queen at some unattacked cell. This again reduces the number of unattacked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

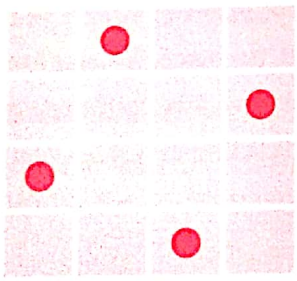
If the number of queens to be placed becomes 0, then it's over, we found a solution. But if the number of unattacked cells become 0, then we need to backtrack, i.e. remove the last placed queen from its current cell, and place it at some other cell.

CODING BLOCKS
Scanned by CamScanner

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.

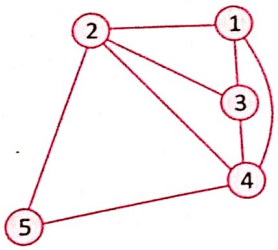
CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

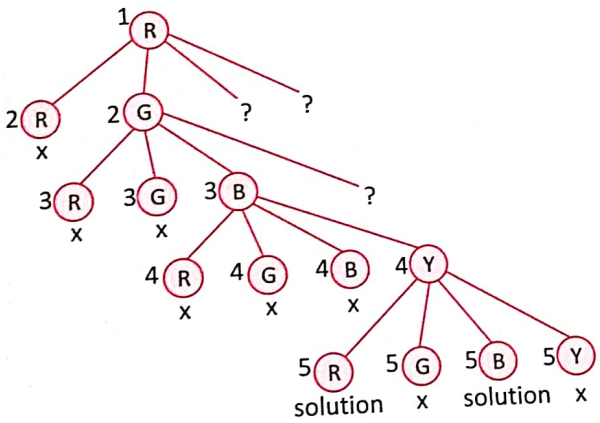
2. M Coloring Problems :

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.

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

Approach towards the solution :

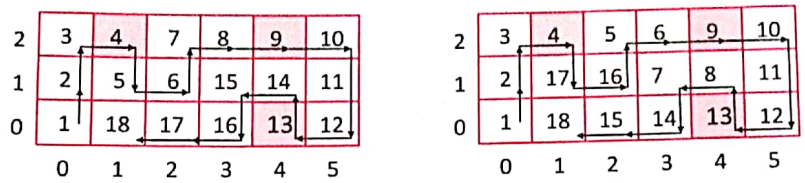
This problem has been break into three main subproblems. In first subproblem we have to check all those path via we can reach to square (2, 1) starting from (0, 0) in 4 steps.

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;
     }
}
```


CODING BLOCKS

Scanned by CamScanner

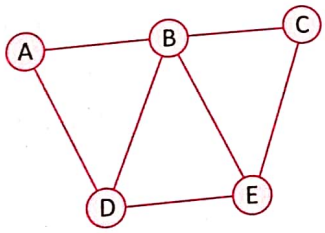
Backtracking Algorithm :

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.
 }

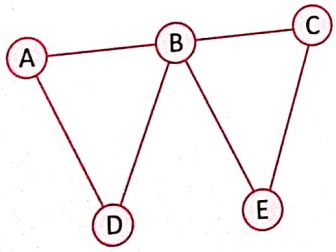
1. Finding Hamiltonian Cycle :

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

{

CODING BLOCKS

Scanned by CamScanner

```
    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)

Scanned by CamScanner

Scanned by CamScanner

Scanned by CamScanner

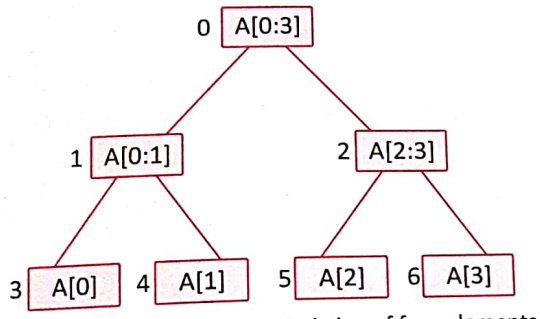
7 Segment Tree

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.**

CODING BLOCKS
Scanned by CamScanner

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;
 }
```


CODING BLOCK

Scanned by CamScanner

```
    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
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
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
```


CODING BLOCKS

Scanned by CamScanner

```
    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;
```


Scanned by CamScanner

```
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
```


CODIN BLOCK

Scanned by CamScanner

```
  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
```


CODING BLOCKS
Scanned by CamScanner

```
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

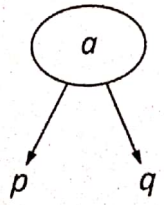
OUTPUT :

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 and q such that node a can give the maximum sum subinterval for its range when queried.

CODIN BLOCK

Scanned by CamScanner

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;
}
```


Scanned by CamScanner

```
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.

CODIN BLOCK

Scanned by CamScanner

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 :

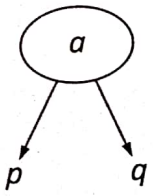
3
1

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)
{
```


Scanned by CamScanner

```
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;
```


CODING BLOCKS

Scanned by CamScanner

```
  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!**

Scanned by CamScanner

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
```


CODIN BLOC

Scanned by CamScanner

```
    {
      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

Scanned by CamScanner

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)
```

Scanned by CamScanner

```
{
 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.

Scanned by CamScanner

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];
```


CODING BLOCKS

Scanned by CamScanner

```
    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;
```


Scanned by CamScanner

```
    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];
    }
```


Scanned by CamScanner

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
```


CODING BLOCKS
Scanned by CamScanner

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)
 {
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
    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;
}
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
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;
}
```


Scanned by CamScanner

SELF STUDY NOTES

CODING BLOCKS

Scanned by CamScanner

Scanned by CamScanner

8 Binary Index Tree

Binary Indexed Tree also called Fenwick Tree provides a way to represent an array of numbers in an array, allowing prefix sums to be calculated efficiently.

Calculating prefix sums efficiently is useful in various scenarios. Let's start with a simple problem.

We are given an array a[], and we want to be able to perform two types of operations on it.

1. Change the value stored at an index i. (This is called a point update operation)
2. Find the sum of a prefix of length k. (This is called a range sum query)

A straightforward implementation of the above would look like this.

```
int a[] = {2, 1, 4, 6, -1, 5, -32, 0, 1};
void update(int i, int v) //assigns value v to a[i]
{
    a[i] = v;
}
int prefixsum(int k) //calculate the sum of all a[i] s
uch that 0 <= i < k
{
   int sum = 0;
   for(int i = 0; i < k; i++)
       sum += a[i];
   return sum;
}
```

This is a perfect solution, but unfortunately the time required to calculate a prefix sum is proportional to the length of the array, so this will usually time out when large number of such operations are performed.

Can We Do Better Than This?

One efficient solution is to use segment tree that can perform both operation in O(logN) time.

Using Binary Indexed Tree also, we can perform both the tasks in O(logN) time. But then why learn another data structure when segment tree can do the work for us. It's because binary indexed trees require less space and are very easy to implement during programming contests (the total code is not more than 8-10 lines).

CODING BLOCKS

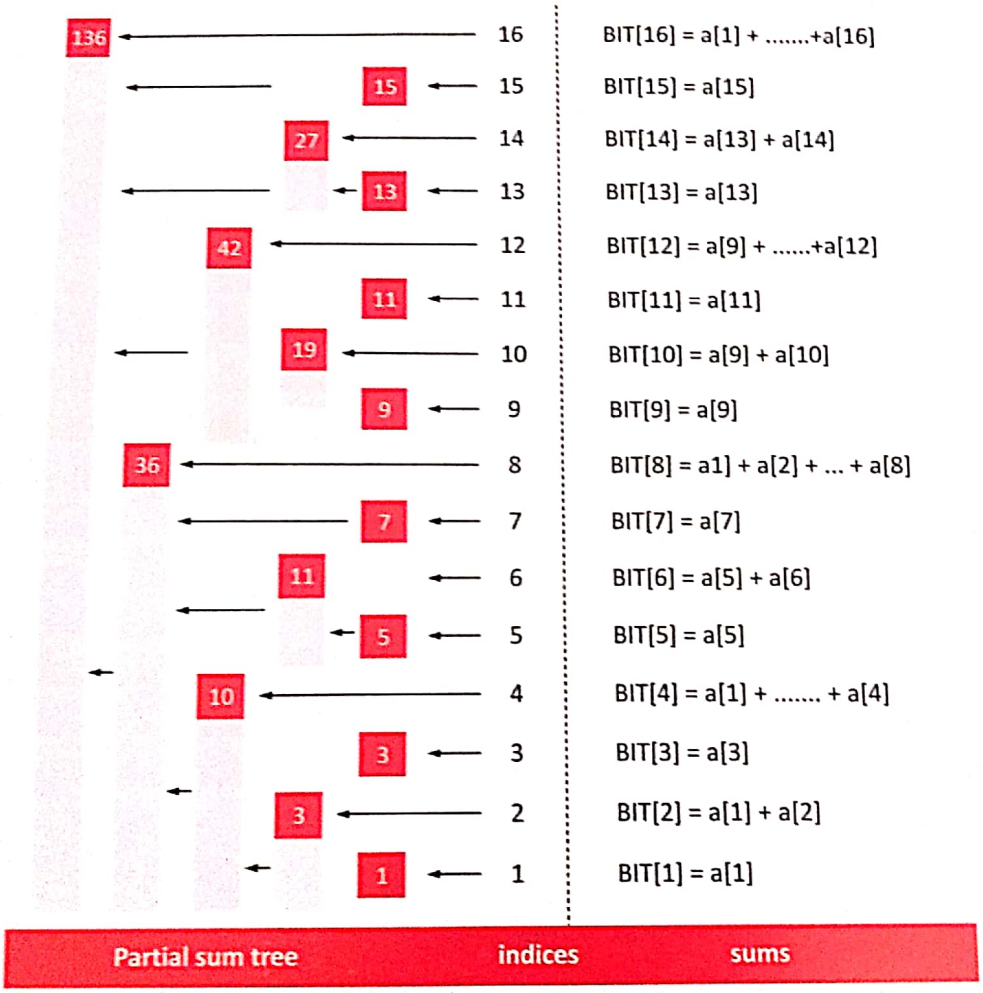
Scanned by CamScanner

Basic Idea of Binary Indexed Tree :

We know the fact that each integer can be represented as sum of powers of two. Similarly, for a given array of size N, we can maintain an array BIT[] such that, at any index we can store sum of some numbers of the given array. This can also be called a partial sum tree.

Let's use an example to understand how BIT[] stores partial sums.

```
//for ease, we make sure our given array is 1-based indexed
int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
```



The value in the enclosed box represents BIT[index]

The above picture shows the binary indexed tree, each enclosed box of which denotes the value BIT[index] and each BIT[index] stores partial sum of some numbers.

Every index i in the BIT[] array stores the cumulative sum from the index i to i - (1<<r) + 1 (both inclusive), where r represents the last set bit in the index i.

CODING BLOCKS

Scanned by CamScanner

Sum of first 12 numbers in array a[] = BIT[12] + BIT[8] = (a[12] + ... + a[9]) + (a[8] + ... + a[1])

Similarly, **sum of first 6 elements** = BIT[6] + BIT[4] = (a[6] + a[5]) + (a[4] + ... + a[1])

Sum of first 8 elements = BIT[8] = a[8] + ... + a[1]

Construct The BIT :

Let's see how to construct this tree and then we will come back to querying the tree for prefix sums.

BIT[] is an array of size = 1 + the size of the given array a[] on which we need to perform operations. Initially all values in BIT[] are equal to 0. Then we call update() operation for each element of given array to construct the Binary Indexed Tree. The update() operation is discussed below.

```
void update(int index,int val) {
    while (index <= maxn) {
        tree[index] += val;
        index += (index & (-index));
    }
}
```

Suppose we call update(13, 2).

Here we see from the above figure that indices 13, 14, 16 cover index 13 and thus we need to add 2 to them also.

Initially x is 13, we update BIT[13]

```
BIT[13] += 2;
```

Now isolate the last set bit of x = 13(1101) and add that to x , i.e. x += x&(-x)

Last bit is of x = 13(1101) is 1 which we add to x, then x = 13+1 = 14, we update BIT[14]

```
BIT[14] += 2;
```

Now 14 is 1110, isolate last bit and add to 14, x becomes 14+2 = 16(10000), we update BIT[16]

```
BIT[16] += 2;
```

In this way, when an update() operation is performed on index x we update all the indices of BIT[] which cover index **x** and maintain the BIT[].

If we look at the for loop in update() operation, we can see that the loop runs at most the number of bits in index x which is restricted to be less or equal to n (the size of the given array), so we can say that the update operation takes at most O(log2(n)) time.

Scanned by CamScanner

Query in BIT :

```
int query(int index) {
    int sum = 0;
    while (index > 0) {
        sum += tree[index];
        index -= (index & (-index));
    }
    return sum;
}
```

The above function query() returns the sum of first x elements in given array. Let's see how it works.

Suppose we call query(14), initially sum = 0

x is 14(1110) we add BIT[14] to our sum variable, thus sum = BIT[14] = (a[14] + a[13])

Now we isolate the last set bit from x = 14(1110) and subtract it from x last set bit in 14(1110) is 2(10), thus x = 14 – 2 = 12

We add BIT[12] to our sum variable, thus **sum =** BIT[14] + BIT[12] = (a[14] + a[13]) + (a[12] + ... + a[9])

again we isolate last set bit from x = 12(1100) and subtract it from x last set bit in 12(1100) is 4(100), thus x = 12 – 4 = 8 we add BIT[8] to our sum variable, thus

sum = BIT[14] + BIT[2] + BIT[8] = (a[14] + a[13]) + (a[12] + ... + a[9]) + (a[8] + ... + a[1])

Once again we isolate last set bit from x = 8(1000) and subtract it from x last set bit in 8(1000) is 8(1000), thus x = 8 – 8 = 0 since x = 0, the for loop breaks and we return the prefix sum.

Talking about complexity, again we can see that the loop iterates at most the number of bits in x which will be at most n(the size of the given array). Thus the query operation takes O(log2(n)) time.

When To use BIT ?

Before going for Binary Indexed tree to perform operations over range, one must confirm that the operation or the function is:

Associative. i.e f(f(a, b), c) = f(a, f(b, c)) this is true even for seg-tree

Has an inverse. eg:

- Addition has inverse subtraction (this example we have discussed)
- Multiplication has inverse division
- gcd() has no inverse, so we can't use BIT to calculate range gcd's

PROBLEM : INVCNT (Inversion Count)

Let A[0...n - 1] be an array of n distinct positive integers. If i < j and A[i] > A[j] then the pair (i, j) is called an inversion of A. Given n and an array A your task is to find the number of inversions of A.

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

For example, the array a={2,3,1,5,4} has three inversions: (1,3), (2,3), (4,5), for the pairs of entries (2,1), (3,1), (5,4).

We'll use BIT to solve the problem of Inversion Count.

Replacing the Values of the Array with the Indexes :

Usually when we are implementing a BIT is necessarily to map the original values of the array to a new range with values between [1,N], where N is the size of the array. This is due to the following reasons:

1. The values in one or more A[i] entry are too high or too low.

   (e.g. 10 – 12 or 10^ – 12).

   For example imagine that we are given an array of 3 integers: {1,10^12,5}

   This means that if we want to construct a frequency table for our BIT data structure, we are going to need at least an array of 10^12 elements.

2. The values in one or more A[i] entry are negative.

   Because we are using arrays it's not possible to handle in our BIT frequency of negative values (e.g. we are not able to do freq[–12]).

A simple way to deal with this issues is to replace the original values of the target array for indexes that maintain its relative order.

For example, given the following array:

A	9	1	0	5	4

The first step is to make a copy of the original array **A** let's call it **B**.

Then we proceed to sort **B** in non-descending order as follow:

sorted copy of the array A

B	0	1	4	5	9

Using binary search over the array B we are going to seek for every element in the array A, and stored the resulting position indexes (1-based) in the new array A.

binary_search(B,9)=4 found at position 4 of the array B

binary_search(B,1)=1 found at position 1 of the array B

binary_search(B,0)=0 found at position 0 of the array B

binary_search(B,5)=3 found at position 3 of the array B

binary_search(B,4)=2 found at position 2 of the array B

The resulting array after increment each position by one is the following:

new array A

A	5	2	1	4	3

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

The following C++ code fragment illustrate the ideas previously explained:

```
for(int i = 0; i < N; ++i)
    B[i] = A[i]; // copy the content of array A to array B

sort(B, B + N); // sort array B

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

    int ind = int(lower_bound(B, B + N, A[i]) - B);
    A[i] = ind + 1;
}
```

Counting inversions with the accumulate frequency :

Initially the cumulative frequency table is empty, we start the process with the element **3**, the last one in our array.

A	5	2	1	4	3

cumulative frequency array

0	0	0	0

how many numbers less than 3 have we seen so far

x=read(3"1) = 0

inv_counter=inv_counter+x

update the count of 3's so far

update(3,+1)

inv_counter=0

A	5	2	1	4	3

cumulative frequency array

0	0	1	1	1

The cumulative frequency of value 3 was increased in the previous step, this is why the read(4"1) count the inversion (4,3).

how many numbers less than 4 have we seen so far

x=read(4"1)=1

inv_counter=inv_counter+x

update the count of 4's so far

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

update(4,+1)

inv_counter=1

The term 1 is the lowest in our array, this is why there is no inversions beginning at 1.

A	5	2	1	4	3

cumulative frequency array

0	0	1	2	2

how many numbers less than 1 have we seen so far

x=read(1"1)=0

inv_counter=inv_counter+x

update the count of 1's so far

update(1,+1)

inv_counter=1

And so on for the other elements...

PROBLEM : BAADSHAH

You are provided an array consisting of 'n' integers and asked to do following operations:

1. Update the position at "x" of array to "y" ie A[x]=y.
2. Find if there is any prefix in the array whose sum equals to "S".If yes, then output "Found" and the last index of prefix, else output "Not Found".

INPUT :

4 4

1 2 3 4

2 6

1 4 10

2 16

2 5

OUTPUT :

Found 3

Found 4

Not Found

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

Construct a BIT(Binary Indexed Tree) from the given array, where internal nodes represents range sum.

Scanned by CamScanner

Use Point updates for query 1, and for query 2, using Binary search on range-query sums, where starting range is 1 to n, check if the prefix sum exists or not.

Code :

```
ll read(ll index) {
    ll sum = 0;
    while (index > 0) {
        sum += BIT[index];
        index -= (index & (-index));
    }
    return sum;
}

void update(ll index, ll val) {
    while (index < maxn) {
        BIT[index] += val;
        index += (index & (-index));
    }
}

int main() {
    cin >> N >> M;
    for (int i = 1; i <= N; ++i) {
        cin >> arr[i];
        update(i, arr[i]);
    }

    while (M—) {
        ll idx;
        cin >> type;
        if (type == 1) {
            cin >> a >> b;
            update(a, b - arr[a]);
            arr[a] = b;
    }
    else {
```


Scanned by CamScanner

```
        bool flag = false;
        cin >> a;
        ll lo = 1, hi = N;
        while (lo <= hi) {
            int mid = (lo + hi) >> 1;
            ll ans = read(mid);
            if (ans < a) {
              lo = mid + 1;
            }
            else if (ans > a) {
                hi = mid - 1;
            }
            else {
              flag = true;
              idx = mid;
              break;
            }
        }
        if (flag)cout << "Found " << idx << "\n";
        else cout << "Not Found\n";
    }
}
return 0;
}
```

PROBLEM : CTRICK (CARD TRICK)

The magician shuffles a small pack of cards, holds it face down and performs the following procedure:

The top card is moved to the bottom of the pack. The new top card is dealt face up onto the table. It is the Ace of Spades.

Two cards are moved one at a time from the top to the bottom. The next card is dealt face up onto the table. It is the Two of Spades.

Three cards are moved one at a time...

This goes on until the nth and last card turns out to be the n of Spades.

Scanned by CamScanner

This impressive trick works if the magician knows how to arrange the cards beforehand (and knows how to give a false shuffle). Your program has to determine the initial order of the cards for a given number of cards, 1 d" n d" 20000.

INPUT :

2

4

5

OUTPUT :

2 1 4 3

3 1 4 5 2

We are asked to place 1 in 2nd free cell of our answer, then to place 2 in 3rd free cell of our answer while starting counting from position where we had placed 1 (and starting from the beginning if we reached end of array), then to place 3 in 4th free cell, and so on.

Now add one more optimization. At every moment we know how many free cells are in our array now. Assume we are at position X now, need to move 20 free cells forward, and you know that at given moment

there are 2 free cells before X and 4 free cells after X. It means that we need to find 22nd free cell starting from beginning of array, which is same as going full circle 3 times and then taking 4th free cell. As we can see, we moved to a problem "find i-th zero in array".

Code :

```
int read(int ind) {
    int sum = 0;
    while (ind > 0) {
        sum += tree[ind];
        ind -= (ind & -ind);
    }
    return sum;
}

void update(int ind, int val) {
    while (ind < 20002) {
        tree[ind] += val;
        ind += (ind & -ind);
    }
}
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
int ans[20005];
int b_search(int val) {

    int l = 1, r = n;
    while (l < r) {

        int mid = (l + r) >> 1;
        int tt = read(mid);

        if (tt > val) {
        r = mid;
        }
        else if (tt < val) {
            l = mid + 1;
        }
        else {
            r = mid;
            if (ans[mid] == 0) return mid;
        }
    }
    return l;

}
int main() {

    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    cin >> t;
    while (t-) {

        cin >> n;
        memset(tree, 0, sizeof(tree));
        for (int i = 1; i <= n;i++) {
            update(i, 1);
        }
```


CODING BLOCKS
Scanned by CamScanner

```
        int fcells = n;
        int cur = 1;
        for (int i = 1; i < n + 1;++i) {
            int free = i + 1;
            int freeb = read(cur - 1);
            int freef = read(n) - freeb;
            int yy = free + freeb;
            int cell = yy % fcells;
            if (cell == 0) cell = fcells;
            int ret_ind = b_search(cell);
            ans[ret_ind] = i;
            fcells–;
            update(ret_ind, -1);
            cur = ret_ind;
        }

        for (int i = 1; i < n + 1;++i) {
            cout << ans[i] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

TIME TO TRY

DQUERY

Given a sequence of n numbers $a_1, a_2, \ldots, a_n$ and a number of d-queries. A d-query is a pair (i, j) (1 = i = j = n). For each d-query (i, j), you have to return the number of distinct elements in the subsequence $a_i, a_{i+1}, \ldots, a_j$.

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

CODING BLOCKS

Scanned by CamScanner

SELF STUDY NOTES

CODING BLOCKS

Scanned by CamScanner

SELF STUDY NOTES

CODING BLOCKS

Scanned by CamScanner

9 Dynamic Programming

"Those who cannot remember the past are condemned to repeat it."

Dynamic Programing is all about remembering answers to the sub-problems you've already solved and not solving it again.

Q. Where do we need Dynamic Programming?

- If you are given a problem, which can be broken down into **smaller sub-problems.**
- These smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some **over- lappping sub-problems.**
- Then you've encountered a DP problem.

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results.

Example

Writes down "1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 =" on a sheet of paper.

"What's that equal to?"

Counting "Eight!"

Writes down another "1+" on the left.

"What about that?"

"Nine!" " How'd you know it was nine so fast?"

"You just added one more!"

"So you didn't need to recount because you remembered the re were eight! Dynamic Programming is just a fancy way to say remembering stuff to save time later!"

Dynamic Programming and Recursion:

- Dynamic programming is basically, recursion plus memoization.
- Recursion allows you to express the value of a function in terms of other values of that function.
- If you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster.
- This is what we call Memoization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

Scanned by CamScanner

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of **Fibonacci numbers.**

Fibonacci (n) = 1; if n = 0

Fibonacci (n) = 1; if n = 1

Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21, 35...

A code for it using pure recursion:

```
int fib (int n) {
if (n < 2)
return 1;
return fib(n-1) + fib(n-2);
}
```

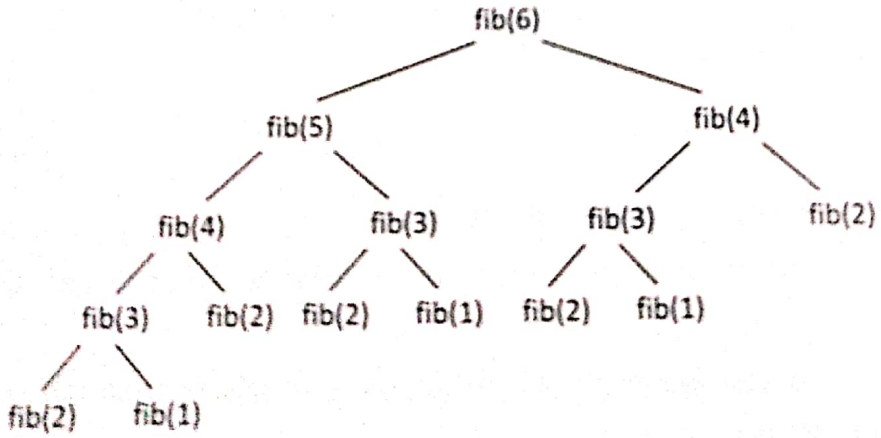
Using Dynamic Programming approach with memoization:

```
void fib () {
fib[0] = 1;
fib[1] = 1;
for (int i = 2; i<n; i++)
fib[i] = fib[i-1] + fib[i-2];
}
```

Q. Are we using a different recurrence relation in the two codes? **No**

Q. Are we doing anything different in the two codes? **Yes**

Let's Visualize :



Here we are running fibonacci function multiple times for the same value of n, which can be preve using memoization.

Scanned by CamScanner

Optimization Problems :

Dynamic Programming is typically applied to optimization problems. In such problems there can be any possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem.

❑ MINIMUM STEPS TO ONE

http://www.spoj.com/problems/MST1/

On a positive integer, you can perform any one of the following 3 steps.

1. Subtract 1 from it.
2. If its divisible by 2, divide by 2.
3. If its divisible by 3, divide by 3. Now the question is, given a positive integer n, find the minimum number of steps that takes n to 1.

Cannot apply **Greedy**! (Why ?)

Recursive Solution :

We will define the recurrence relation as

solve(n):

if n = 1

ans = 0

if n > 1

ans = min{1 + solve(n-1), 1 + solve(n/2), 1 + solve(n/3)}

Code :

```
int minStep(int n){
if(n == 1)
return 0;
int subOne = INF, divTwo = INF, divThree = INF;
//If number is greater than or equal to 2, subtract one
if(n >= 2)
subOne = 1 + minStep(n-1);
//If divsible by 2, divide by 2
if(n % 2 == 0)
divTwo = 1 + minStep(n/2);
//If divisible by 3, divide by 3
if(n%3 == 0)
divThree = 1 + minStep(n/3);
//Return the most optimal answer
return min(subOne, min(divTwo, divThree));}
```

nce we are solving a single sub-problem multiple number of times, T = $O(K^N)$

ODING
LOCKS
Your Way to Success

Scanned by CamScanner

Can we do better?

The recursion tree for the above solution results in subproblem overlapping. So, we can improve the solution using memoization.

Dynamic Programming Solution :

1. **Top Down DP :** In Top Down, you start building the big solution right away by explaining how you build it from smaller solutions.

   Example :

   I will be an amazing coder. How?

   I will work hard like crazy. How?

   I'll practice more and try to improve. How?

   I'll start taking part in contests. How?

   I'll practicing. How?

   I'm going to learn programming.

Code :

```
int minStep(int n){
//if already calculated, return this answer
   if(dp[n] != -1)
         return dp[n];
// Base case
   if(n <= 1)
         return 0;
   int subOne = INF, divTwo = INF, divThree = INF;
//If number is greater than or equal to 2, subtract one
   if(n >= 2)
         subOne = 1 + minStep(n-1);
//If divsible by 2, divide by 2
   if(n % 2 == 0)
         divTwo = 1 + minStep(n/2);
//If divisible by 3, divide by 3
   if(n%3 == 0)
         divThree = 1 + minStep(n/3);
//Return the most optimal answer
   return dp[n] = min(subOne, min(divTwo, divThree));
}
```


CODING BLOCKS

Scanned by CamScanner

2. **Bottom Up DP :** In Bottom Up, you start with the small solutions and then use these small solutions to build up larger solutions.

Example :

I'm going to learn programming.

Then, I will start practicing.

Then, I will start taking part in contests.

Then, I'll practice even more and try to improve.

After working hard like crazy,

I'll be an amazing coder.

Code :

```
void minStep(){
//Base case
dp[1] = 0;
dp[0] = 0;
//Iterate for all possible numbers starting from 2
for(int i = 2;i<=2 * 10000000;i++){
dp[i] = 1 + dp[i-1];
if(i % 2 == 0)
dp[i] = min(dp[i],1 + dp[i/2]);
if(i % 3 == 0)
dp[i] = min(dp[i],1 + dp[i/3]);
}
return;
}
```

❑ MINIMUM COIN CHANGE

Given a value N, if we want to make change for N cents, and we have infinite supply of each of C = { C1, C2, ... , CM} valued coins, what is the minimum number of coins to make the change?

Example :

Input: coins[] = {25, 10, 5}, N = 30

Output: Minimum 2 coins required

We can use one coin of 25 cents and one of 5 cents

Input: coins[] = {9, 6, 5, 1}, N = 13

Output: Minimum 3 coins required

We can use one coin of 6 + 6 + 1 cents coins.

Scanned by CamScanner

Recursive Solution :

Start the solution with initially sum = N cents and at each iteration find the minimum coins required by dividing the problem in subproblems where we take {C1, C2, ..., CM} coin and decrease the sum N by C[i] (depending on the coin we took). Whenever N becomes 0, this means we have a possible solution. To find the optimal answer, we return the minimum of all answer for which N became 0.

If N == 0, then 0 coins required.

If N > 0

minCoins(N , coins[0..m-1]) = min {1 + minCoins(N-coin[i] ,coins[0....m-1])}

where i varies from 0 to m-1 and coin[i] <= N

Code :

```
int minCoins(int coins[], int m, int N)
{
   // base case
    if (N == 0)
      return 0;
   // Initialize result
   int res = INT_MAX;
   // Try every coin that has smaller value than V
   for (int i=0; i<m; i++)
   {
     if (coins[i] <= N)
       {
          int sub_res = 1 + minCoins(coins, m, N-coins[i]);
          // see if result can minimized
          if (sub_res < res)
              res = sub_res;
       }
   }
  return res;
}
```


CODING BLOCK

Scanned by CamScanner

Dynamic Programming Solution :

Since same suproblems are called again and again, this problem has Overlapping Subprolems property. Like other typical Dynamic Programming(DP) problems, re-computations of same subproblems can be avoided by constructing a temporary array dp[] and memoizing the computed values in this array.

1. Top Down DP

Code :

```
int minCoins(int N, int M)
{
    // if we have already solved this subproblem
    // return memoized result
    if(dp[N] != -1)
     return dp[N];
    // base case
    if (N == 0)
      return 0;
    // Initialize result
    int res = INF;
    // Try every coin that has smaller value than N
     for (int i=0; i<M; i++)
      {
       if (coins[i] <= N)
         {
            int sub_res = 1 + minCoins(N-coins[i], M);
            // see if result can minimized
            if (sub_res < res)
               res = sub_res;
         }
      }
   return dp[N] = res;
}
```


CODING BLOCKS

Scanned by CamScanner

2. **Bottom Up DP :** ith state of dp : dp[i] : Minimum number of coins required to sum to i cents.

Code :

```
int minCoins(int N, int M)
{
   //Initializing all values to infinity i.e. minimum coins to make any
   //amount of sum is infinite
      for(int i = 0;i<=N;i++)
         dp[i] = INF;
  //Base case i.e. minimum coins to make sum = 0 cents is 0
      dp[0] = 0;
   //Iterating in the outer loop for possible values of sum between 1 to N
   //Since our final solution for sum = N might depend upon any of these
values
     for(int i = 1;i<=N;i++)
     {
         //Inner loop denotes the index of coin array.
         //For each value of sum, to obtain the optimal solution.
        for(int j = 0;j<M;j++)
         {
          //i -> sum
          //j -> next coin index
          //If we can include this coin in our solution
           if(coins[j] <= i)
            {
              //Solution might include the newly included coin.
              dp[i] = min(dp[i], 1 + dp[i - coins[j]]);
            }
         }
     }
  //for(int i = 1;i<=N;i++)cout<<i<<" "<<dp[i]<<endl;
  return dp[N];
}
T = O(N*M)
```

Solution Link: http://pastebin.com/JFNh3LN3

Scanned by CamScanner

❑ WINE AND MAXIMUM PRICE

Imagine you have a collection of N wines placed next to each other on a shelf. For simplicity, let's number the wines from left to right as they are standing on the shelf with integers from 1 to N, respectively. The price of the ith wine is pi. (prices of different wines can be different).

Because the wines get better every year, supposing today is the year 1, on year ythe price of the ith wine will be y*pi, i.e. y-times the value that current year.

You want to sell all the wines you have, but you want to sell exactly one wine per year, starting on this year. One more constraint - on each year you are allowed to sell only either the leftmost or the rightmost wine on the she lf and you are not allowed to reorder the wines on the shelf (i.e. they must stay in the same order as they are in the beginning).

You want to find out, what is the maximum profit you can get, if you sell the wines in optimal order?

So, for example, if the prices of the wines are (in the order as they are

placed on the shelf, from left to right): p1=1, p2=4, p3=2, p4=3. The optimal solution would be to sell the wines in the order p1, p4, p3, p2 for a total profit 1 * 1 + 3 * 2 + 2 * 3 + 4 * 4 = 29.

Wrong Solution!

Greedy Approach :

Every year, sell the cheaper of the two(leftmost and right most) available wines.

Let the prices of 4 wines are: 2, 3, 5, 1, 4

At t = 1 year: {2,3,5,1,4} sell p1 = 2 to get cost = 2

At t = 2 years: {3,5,1,4} sell p2 = 3 to get cost = 2 + 2*3 = 8

At t = 3 years: {5,1,4} sell p5 = 4 to get cost = 8 + 4*3 = 20

At t = 4 years: {5,1} sell p4 = 1 to get cost = 20 + 1*4 = 24

At t = 5 years: {5} sell p3 = 5 to get cost = 24 + 5*5 = 49

Greedy approach gives an optimal answer of 49, but if we sell in the order of p1, p5, p4, p2, p3 for a total profit 2 * 1 + 4 * 2 + 1 * 3 + 3 * 4 + 5 * 5 = 50, greedy fails.

Recursive Solution : Here we will try out all the possible options and output the optimal Answer.

```
if(start > end)
return 0
if(start <= end)
return maxPrice(price, start, end, year) = max{price[
start] * year + maxPrice(price, start+1, end, year + 1),
price[end] * year + maxPrice(price, start, end + 1,year + 1)},
```


CODING BLOCKS

Scanned by CamScanner

For each endpoint at every function call, we are doing the following:

- Either we take this end point in the solution, increment/decrement the end point index.
- Or we do not take this end point in the solution.

Since we are doing this for each and every n endpoints (n is number of wines on the table). So, T = O(2^n) which is exponential time complexity.

Code :

```
int maxPrice(int price[], int start, int end, int year)
{
//Base case, stop when star of array becomes more than end.
 if(start > end)
   return 0;
 /Including the wine with starting index in our solution
 int incStart = price[start] * year + maxPrice(price,start + 1, end, year + 1);
 //Including the wine with ending index in our solution
 int incEnd = price[end] * year + maxPrice(price, start, end-1, year + 1);
 //return the most optimal answer
 return max(incStart, incEnd);
}
```

Can we do better?

Yes we can! By carefully observing the recursion tree, we can clearly see that we encounter the property of subproblem overlapping which can be prevented using memoization or dynamic programming.

Top Down DP Code :

```
int maxPrice(int start,int end,int N)
{
  //If we have solved this sub-problem, return the memoized answer
  if(dp[start][end] != 0)
     return dp[start][end];
  //base case
  if(start > end)
     return 0;
  //To get the current year
  int year = N - (end - start + 1) + 1;
  //Including the starting index
```


CODING BLOCKS

Scanned by CamScanner

```
    int incStart = year * price[start] + maxPrice(start+1, end, N);
    //Including the ending index
    int incEnd = year * price[end] + maxPrice(start,end-1,N);
    //memoize this solution and return
    return dp[start][end] = max(incStart,incEnd);
}
```

Solution: http://pastebin.com/Yx1FG3ys

We can also solve this problem using the bottom up dynamic programming approach. Here we will need to create a 2-Dimensional array for memoization where the states i and j in **dp[i][j]** denotes the optimal answer between the **starting index i** and **ending index j**.

According to the definition of states, we define:

```
dp[i][j] = max{current_year * price[i] + dp[i+1][j], curr
ent_year * price[j] + dp[i][j+1]}
```

Bottom Up DP Code :

```
int maxPrice(int start,int end,int N){
 //Initialize the dp array
  for(int i = 0;i<N;i++)
  {
    for(int j = 0;j<N;j++)
      dp[i][j] = 0;
  }
//Outer loop denotes the starting index for our solution
 for(int i = N-1;i>=0;i–)
 {
 //Inner loop denotes the ending index
   for(int j = 0;j<N;j++)
   {
    //if (start > end), return 0
      if(i > j)
         dp[i][j] = 0;
     else
     {
    //find the current year
         int year = N - (j - i);
```


ODING
LOCKS

Scanned by CamScanner

```
            //using bottom up dp to solve the problem for smaller sub problems
            //and using it to solve the larger problem i.e. dp[i][j]
             dp[i][j] = max(year * price[i] + dp[i+1][j], year * price[j] +
dp[i][j-1]);
        }
    }
}
//return the final answer where starting index is start = 0 and ending index
is end = n-1
return dp[0][N-1];
}
```

Solution: http://pastebin.com/idsyg4aw

T = O(N^2), where N is the number of wines.

❑ PROBLEMS INVOLVING GRIDS

- Finding a minimum-cost path in a grid
- Finding the number of ways to reach a particular position from a given starting point in a 2-D grid and so on.

Finding Minimum-Cost Path in a 2-D Matrix

Problem : Given a cost matrix Cost[][] where Cost[i][j] denotes the Cost of visiting cell with coordinates (i,j), find a min-cost path to reach a cell (x,y) from cell (0,0) under the condition that you can only travel one step right or one step down. (We assume that all costs are positive integers)

It is very easy to note that if you reach a position (i,j) in the grid, you must have come from one cell higher, i.e. (i-1,j) or from one cell to your left , i.e. (i,j-1). This means that the cost of visiting cell (i,j) will come from the following recurrence relation:

Code :

```
MinCost(i,j) = min{ MinCost(i-1,j), MinCost(i,j-1) } + cost[i][j]
```

We now compute the values of the base cases: the topmost row and the leftmost column. For the topmost row, a cell can be reached only from the cell on the left of it. Assuming zero-based index,

```
MinCost(0,j) = MinCost(0,j-1) + Cost[0][j]
MinCost(i,0) = MinCost(i-1,0) + Cost[i][0]
```

i.e. cost of reaching cell (0,j) = Cost of reaching cell (0,j-1) + Cost of visiting cell (0,j) Similarly,cost of reaching cell (i,0) = Cost of reaching cell (i-1,0) + Cost of visiting cell (i,0).

CODING BLOCKS

Scanned by CamScanner

Code :

```
int MinCost(int Ro, int Col)
{
  //This bottom-up approach ensures that all the sub-problems needed
  // have already been calculated.
   for(int i = 0;i<Ro;i++)
    {
     for(int j = 0;j<Col;j++)
      {
       //Base Cases
       if(i == 0 && j == 0)
         dp[i][j] = cost[0][0];
       else if(i == 0)
         dp[i][j] = dp[0][j-1] + cost[0][j];
       else if(j == 0)
         dp[i][j] = dp[i-1][0] + cost[i][0];
  //Calculate cost of visiting (i,j) using the
  //recurrence relation discussed above
      else
          dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + cost[i][j];
     }
  }
 //Return the optimum cost of reaching the last cell
  return dp[Ro-1][Col-1];
}
```

- **Finding the number of ways to reach from a starting position to an ending position travelling in specified directions only.**

Problem Statement : Given a 2-D matrix with M rows and N columns, find the number of ways to reach cell with coordinates (i,j) from starting cell (0,0) under the condition that you can only travel one step right or one step down.

This problem is very similar to the previous one. To reach a cell (i,j), one must first reach either the cell (i-1,j) or the cell (i,j-1) and then move one step down or to the right respectively to reach cell (i,j). After convincing yourself that this problem indeed satisfies the optimal sub-structure and overlapping subproblems properties, we try to formulate a bottom-up dynamic programming solution.

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

Let **NumWays(i,j)** be the number of ways to reach position (i,j). As stated above, number of ways to reach cell (i,j) will be equal to the sum of number of ways of reaching (i-1,j) and number of ways of reaching (i,j-1). Thus, we have our recurrence relation as :

numWays(i,j) = numWays(i-1,j) + numWays(i,j-1)

Code :

```
int numWays(int Ro, int Col)
{
    //This bottom-up approach ensures that all the sub-problems needed
    // have already been calculated.
   for(int i = 0;i<Ro;i++)
    {
     for(int j = 0;j<Col;j++)
      {
       //Base Cases
       if(i == 0 && j == 0)
         dp[i][j] = 1;
      else if(i == 0)
         dp[i][j] = 1;
      else if(j == 0)
        dp[i][j] = 1;
      //Calculate no. of ways to visit (i,j) using the
      //recurrence relation discussed above
     else
        dp[i][j] = dp[i-1][j] + dp[i][j-1];
    }
  }
    //Return the optimum cost of number of ways
  return dp[Ro-1][Col-1];
}
```

Ques. Finding the number of ways to reach a particular position in a grid from a starting position (given some cells which are blocked)

Problem Statement : Input is three integers M, N and P denoting the number of rows, number of columns and number of blocked cells respectively. In the next P lines, each line has exactly 2 integers *i* and *j* denoting that the cell (i, j) is blocked.

CODING BLOCKS

Scanned by CamScanner

https://www.codechef.com/problems/CD1IT4

The code below explains how to proceed with the solution. The problem is same as the previous one, except for few extra checks (due to blocked cells).

```
int numWays(int Ro, int Col)
{
 //if the initial block is blocked, we cannot move further
   if(dp[0][0] == -1)
   {
     return 0;
   }
 //Number of ways for the first row
   for(int j = 0;j<Col;j++)
   {
     //If any cell is blocked in the first row, we can not visit any
     //cell that are to the right of this blcked cell
      if(dp[0][j] == -1)
        break;
     //There is only one way to reach this cell i.e. from left cell
      dp[0][j] = 1;
   }
 //Number of ways for first column
 for(int i = 0;i<Ro;i++)
 {
  //Similar to first row
   if(dp[i][0] == -1)
     break;
   dp[i][0] = 1;
 }
 //This bottom-up approach ensures that all the sub-problems needed
 //have already been calculated.
 for(int i = 1;i<Ro;i++)
 {
   for(int j = 1;j<Col;j++)
     {
       //If we encounter a blocked cell, do nothing
```


CODING BLOCKS
Scanned by CamScanner

```
                if(dp[i][j] == -1)
                    continue;
                //Calculate no. of ways to visit (i,j)
                dp[i][j] = 0;
                //If the cell on the left is not blocked, we can reach
                //(i,j) from (i,j-1)
                if(dp[i][j-1] != -1)
                    dp[i][j] = dp[i][j-1] % MOD;
                //If the cell above is not blocked, we can reach
                //(i,j) from (i-1,j)
                if(dp[i-1][j] != -1)
                    dp[i][j] = (dp[i][j] + dp[i-1][j]) % MOD;
            }
        }
        //If last cell is blocked, return 0
        if(dp[Ro-1][Col-1] == -1)
            return 0;
        //Return the optimum cost of number of ways
        return dp[Ro-1][Col-1];
    }
```

Solution : http://pastebin.com/LeAhedeQ

Another Variant :

Problem Statement : You are given a 2-D matrix A of n rows and m columns where A[i][j] denotes the calories burnt. Two persons, a boy and a girl, start from two corners of this matrix. The boy starts from cell (1,1) and needs to reach cell (n,m). On the other hand, the girl starts from cell (n,1) and needs to reach (1,m). The boy can move right and down. The girl can move right and up. As they visit a cell, the amount in the cell A[i][j] is added to their total of calories burnt. You have to maximize the sum of total calories burnt by both of them under the condition that they shall meet only in one cell and the cost of this cell shall not be included in either of their total.

http://codeforces.com/contest/429/problem/B

Let us analyse this problem in steps:

The boy can meet the girl in only one cell.

So, let us assume they meet at cell (i,j).

CODING BLOCKS

Scanned by CamScanner

Boy can come in from left or the top, i.e. (i,j-1) or (i-1,j). Now he can move right or down.That is, the sequence for the boy can be:

(i,j-1)—>(i,j)—>(i,j+1)

(i,j-1)—>(i,j)—>(i+1,1)

(i-1,j)—>(i,j)—>(i,j+1)

(i-1,j)—>(i,j)—>(i+1,j)

Similarly, the girl can come in from the left or bottom, i.e. (i,j-1) or (i+1,j) and she can go up or right. The sequence for girl's movement can be:

(i,j-1)—>(i,j)—>(i,j+1)

(i,j-1)—>(i,j)—>(i-1,j)

(i+1,j)—>(i,j)—>(i,j+1)

(i+1,j)—>(i,j)—>(i-1,j)

Comparing the 4 sequences of the boy and the girl, the boy and girl meet only at one position (i,j), iff

Boy: (i,j-1)—>(i,j)—>(i,j+1) and Girl: (i+1,j)—>(i,j)—>(i-1,j)

or

Boy: (i-1,j)—>(i,j)—>(i+1,j) and Girl: (i,j-1)—>(i,j)—>(i,j+1)

Now, we can solve the problem by creating 4 tables:

Boy's journey from start (1,1) to meeting cell (i,j)

Boy's journey from meeting cell (i,j) to end (n,m)

Girl's journey from start (n,1) to meeting cell (i,j)

Girl's journey from meeting cell (i,j) to end (1,n)

The meeting cell can range from 2<= i <= n-1 and 2 <= j <= m-1

See the code below for more details:

```
int maxCalories(int M, int N)
{
  //building boy_start[][] table in bottom up fashion
  //Here boy_start[i][j] -> the max calories that can be burnt if the boy
  //starts from (1,1) and goes up to (i,j)
  for(int i = 1;i<=M;i++)
  {
  for(int j = 1;j<=N;j++)
  {
  boy_start[i][j] = calorie[i][j] + max(boy_start[i-1][j], boy_start[i][j-1]);
  }
 }
//building girl_start[][] table in bottom up fashion
```


CODING BLOCKS

Scanned by CamScanner

```
//Here girl_start[i][j] -> the max calories that can be burnt if the girl
//starts from (M,1) and goes up to (i,j)
  for(int i = M;i>=1;i-)
  {
   for(int j = 1;j<=N;j++)
   {
   girl_start[i][j] = calorie[i][j] + max(girl_start[i+1][j], girl_start[i][j-1]);
   }
  }
//building boy_end[][] table in bottom up fashion which specifies the journey from end
to start
//Here boy_end[i][j] -> the max calories that can be burnt if the boy start
//from end i.e. (M,N) and comes back to (i,j)
  for(int i = M;i>=1;i-)
  {
   for(int j = N;j>=1;j-)
   {
    boy_end[i][j] = calorie[i][j] + max(boy_end[i+1][j], boy_end[i][j+1]);
   }
  }
//building girl_end[][] table in bottom up fashion which specifies the journey from end
to start
//Here girl_end[i][j] -> the max calories that can be burnt if the girl start
//from end i.e. (1,N) and comes back to (i,j)
  for(int i = 1;i<=M;i++)
  {
   for(int j = N;j>=1;j-)
   {
   girl_end[i][j] = calorie[i][j] + max(girl_end[i-1][j], girl_end[i][j+1]);
   }
  }
//Iterate over all the possible meeting points i.e. between (2,2) to (M-1,N-1)
//consider this point as the actual meeting point and calculate the max possible answer
 int ans = 0;
 for(int i = 2;i<M;i++)
 {
  for(int j = 2;j<N;j++)
  {
  int ans1 = boy_start[i][j-1]+boy_end[i][j+1]+girl_start[i+1][j] + girl_end[i-1][j];
```


CODING BLOCKS

Scanned by CamScanner

```
    int ans2 = boy_start[i-1][j] + boy_end[i+1][j]+girl_start[i][j-1]+girl_end[i][j+1];
    ans = max(ans, max(ans1, ans2));
    }
   }
 return ans;
}
```

Solution: http://pastebin.com/cnBqeJEP

❑ BUILDING BRIDGES

Problem Statement: Given two array of numbers which denotes the end points of bridges. What is the maximum number of bridges that can be built if ith point of first array must be connected to ith point of second array and two bridges cannot overlap each other.

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

❑ LONGEST INCREASING SUBSEQUENCE

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the length of LIS for {10, 9, 3, 5, 4, 11, 7, 8} is 4 and LIS is {3, 4, 7, 8}.

Recurrence relation :

```
L(i) = 1 + max{ L(j) } where 0 < j < i and arr[j] < arr[i];
or
L(i) = 1, if no such j exists.
return max(L(i)) where 0 < i < n
```

Code :

```
int LIS(int n)
{
  int i,j,res = 0;
  /* Initialize LIS values for all indexes */
  for (i = 0; i < n; i++ )
   lis[i] = 1;
  /* Compute optimized LIS values in bottom up manner */
  for (i = 1; i < n; i++ )
   for (j = 0; j < i; j++ )
     if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
       lis[i] = lis[j] + 1;
```


CODING BLOCKS
Scanned by CamScanner

```
    /* Pick maximum of all LIS values */
    for (i = 0; i < n; i++ )
     if (res < lis[i])
      res = lis[i];
   return res;
  }
```

T = O(n^2)

We can also print the Longest Increasing Subsequence as:

```
void print_lis(int n)
{
//denotes the current LIS
//initially it is equal to the LIS of the whole sequence
 int cur_lis = LIS(n);
//denotes the previous element printed
//to print the LIS, previous element printed must always be larger than current
//element (if we are printing the LIS backwards)
//Initially set it to infinity
 int prev = INF;
 for(int i = n-1;i>=0;i–)
 {
  //find the element upto which the LIS equal to the cur_LIS
  //and that element is less than the previous one printed
  if(lis[i] == cur_lis && arr[i] <= prev)
  {
   cout<<arr[i]<<" ";
   cur_lis–;
   prev = arr[i];
  }
  if(!cur_lis)
   break;
 }
return;
}
```


CODIN BLOCK

Scanned by CamScanner

In this problem we will use the concept of LIS. Two bridges will not cut each other if both their end points are either in non-increasing or non-decreasing order. To find the solution we will first pair the endpoints i.e. ith point in 1st sequence is paired with ith point in 2nd sequence. Then sort the points w.r.t 1st point in the pair and apply LIS on second point of the pair.

Voila! You have a solution :)

Example: First consider the pairs: (2,6), (5, 4), (8, 1), (10, 2), sort it according to the first element of the pairs (in this case are already sorted) and compute the lis on the second element of the pairs, thus compute the LIS on 6 4 1 2, that is 1 2. Therefore the non overlapping bridges we are looking for are (8, 1) and (10, 2).

Code :

```
int maxBridges(int n)
{
 //for a single point there is always a bridge
 if(n == 1)
  return 1;
 //Sort the points according to the first sequence
  sort(points.begin(), points.end());
 //Apply LIS on the second sequence
 int i,j,res = 0;
 //Initialize LIS values for all indexes
 for (i = 0; i < n; i++ )
  lis[i] = 1;
 //Compute optimized LIS values in bottom up manner
 for (i = 1; i < n; i++ )
  for (j = 0; j < i; j++ )
   if ( points[i].second >= points[j].second && lis[i] < lis[j] + 1)
     lis[i] = lis[j] + 1;
 //Pick maximum of all LIS values
 for (i = 0; i < n; i++ )
  if (res < lis[i])
    res = lis[i];
return res;
}
```

Solution:http://pastebin.com/UdHtgasV

CODING BLOCKS

Scanned by CamScanner

❑ LONGEST COMMON SUBSEQUENCE

LCS Problem Statement: Given two sequences, find the length of longest subsequence present i both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous For example, "abc", "abg", "bdf", "aeg", '"acefg", ... etc are subsequences of "abcdefg". So a string o length n has 2^n different possible subsequences.

Example :

LCS for input Sequences "ABCDGH" and "AEDFHR" is "ADH" of length 3.

LCS for input Sequences "AGGTAB" and "GXTXAYB" is "GTAB" of length 4

Recurrence Relation :

LCS(str1, str2, m, n) = 0, if m = 0 or n = 0 //Base Case

LCS(str1, str2, m, n) = 1 + LCS(str1, str2, m-1, n-1), if str1[m] = str2[n]

LCS(str1, str2, m, n) = max{LCS(str1, str2, m-1, n), LCS(str1, str2, m, n-1)}, otherwise

LCS can take value between 0 and min(m,n).

Code :

```
int lcs( char *str1, char *str2, int m, int n )
{
//Following steps build L[m+1][n+1] in bottom up fashion. Note
//that L[i][j] contains length of LCS of str1[0..i-1] and str2[0..j-1]
for (int i=0; i<=m; i++)
 {
  for (int j=0; j<=n; j++)
   {
    if (i == 0 || j == 0)
      LCS[i][j] = 0;
    else if (str1[i-1] == str2[j-1])
      LCS[i][j] = LCS[i-1][j-1] + 1;
    else
      LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
   }
 }
//Print LCS
```


CODIN BLOCK

Scanned by CamScanner

```
  int i = m, j = n, index = LCS[m][n];
 //array containing the final LCS
  char *lcs_arr = new char[index];
  while (i > 0 && j > 0)
   {
   // If current character in str1[] and str2[] are same, then
   // current character is part of LCS
    if (str1[i-1] == str2[j-1])
     {
      // Put current character in result
       lcs_arr[index-1] = str1[i-1];
      // reduce values of i, j and index
        i–;
        j–;
        index–;
     }
      // If not same, then find the larger of two and
    // go in the direction of larger value
    else if (LCS[i-1][j] > LCS[i][j-1])
      i–;
    else
      j–;
  }
 // Print the lcs
  cout << "LCS of " << str1 << " and " << str2 << " is "<< lcs_arr << endl;
 //L[m][n] contains length of LCS for str1[0..n-1] and
  str2[0..m-1]
  return LCS[m][n];
}
```

T = O(mn)

CODING BLOCKS

Scanned by CamScanner

SHORTEST COMMON SUPERSEQUENCE OF TWO STRINGS

A supersequence is defined as the shortest string that has both str1 and str2 as subsequences.

str1 = "apple", str2 = "les"

"apples"

str1 = "AGGTAB", str2 = "GXTXAYB"

"AGGXTXAYB"

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

Approach:

- First we will find the LCS of the two strings.
- We will insert the non-LCS characters in the LCS found above in their original order.

```
void lcs( string str1, string str2, int m, int n )
{
 //Following steps build L[m+1][n+1] in bottom up fashion. Note
 //that L[i][j] contains length of LCS of str1[0..i-1] and str2[0..j-1]
 for (int i=0; i<=m; i++)
  {
    for (int j=0; j<=n; j++)
     {
      if (i == 0 || j == 0)
       LCS[i][j] = 0;
      else if (str1[i-1] == str2[j-1])
       LCS[i][j] = LCS[i-1][j-1] + 1;
     else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
     }
  }
int i = m, j = n, index = LCS[m][n];
//array containing the final LCS string lcs_arr;
while (i > 0 && j > 0)
  {
  // If current character in str1[] and str2[] are same, then
  // current character is part of LCS
```


CODING BLOCKS

Scanned by CamScanner

```
    if (str1[i-1] == str2[j-1])
     {
      // Put current character in result
       lcs_arr += str1[i-1];
      // reduce values of i, j and index
       i-;
       j-;
       index-;
     }
 // If not same, then find the larger of two and
 // go in the direction of larger value
    else if (LCS[i-1][j] > LCS[i][j-1])
      i-;
   else
      j-;
 }
//Print LCS
//cout<<lcs_arr<<endl;
//Use LCS to get the final answer
 i = m-1, j = n-1;
//Use to make the final array
 int l = 0,k = 0;
//contains the length of longest common subsequence of the two strings
 index = LCS[m][n];
//string containing the final answer
 string ans = "";
while(i>=0 || j>=0)
{
 //If we have not covered the full LCS array
 if(l < index)
 {
   //Find the non-LCS charactes of A and put them in a array
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
        //in reverse order
        while(i >= 0 && str1[i] != lcs_arr[l])
           ans += str1[i-];
       //Find the non-LCS charactes of B and put them in a array
       //in reverse order
        while(j >= 0 && str2[j] != lcs_arr[l])
          ans += str2[j-];
      //Now both the last value of str1 and str2 are equal
      //to the last value of LCS string
         ans += lcs_arr[l++];
        j-;
        i-;
   }
  //If we have exhausted all of our lcs_array and now we
  //just have to merge the non-lcs characters of both the strings
   else
   {
     while(i >= 0)
       ans += str1[i-];
     while(j >= 0)
       ans += str2[j-];
  }
 }
for(int i = ans.length()-1;i>=0;i-)
   cout<<ans[i];
cout<<endl;
}
```

Solution: http://pastebin.com/EB9U8PNu

CODING BLOCKS

Scanned by CamScanner

❑ EDIT DISTANCE

Problem Statement: Given two strings str1 and str2 and below operations can be performed on str1. Find min number of edits(operations) required to convert str1 to str2.

❑ Insert ❑ Remove ❑ Replace

All the above operations are of equal cost.

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

Example :

str1 = "cat"

str2 = "cut"

Replace 'a' with 'u', min number of edits = 1

str1 = "sunday"

str2 = "saturday"

Last 3 characters are same, we only need to replace "un" with "atur".

Replace n—>r and insert 'a' and 't' before 'u', min number of edits = 3

Recurrence Relation :

```
if str1[m] = str2[n]
        editDist(str1, str2, m, n) = editDist(str1, str2, m-1, n-1)
else
        editDist(str1, str2, m, n) = 1 + min{editDist(str1, str2, m-1, n) //Remove
        editDist(str1, str2, m, n-1) //Insert
        editDist(str1, str2, m-1, n-1) //Replace
}
```

Transform "Sunday" to "Saturday" :

Last 3 are same, so ignore. We will transform Sun —> Satur

(Sun, Satur) —> (Su, Satu) //Replace 'n' with 'r', cost= 1

(Su, Satu) —> (S, Sat) //Ignore 'u', cost = 0

(S, Sat) —> (S,Sa) //Insert 't', cost = 1

(S,Sa) —> (S,S) //Insert 'a', cost = 1

(S,S) —> ("","") //Ignore 'S', cost = 0

("", "") —> return 0

CODING BLOCKS

Scanned by CamScanner

Dynamic Programming :

```
int editDist(string str1, string str2){
int m = str1.length();
int n = str2.length();
// dp[i][j] -> the minimum number of edits to transform str1[0...i-1] to
str2[0...j-1]
//Fill up the dp table in bottom up fashion
for(int i = 0;i<=m;i++)
{
for(int j = 0;j<=n;j++)
{
//If both strings are empty
 if(i == 0 && j == 0)
   dp[i][j] = 0;
//If first string is empty, only option is to
//insert all characters of second string
//So number of edits is the length of secondstring
 else if(i == 0)
   dp[i][j] = j;
//If second string is empty, only option is to
//remove all characters of first string
//So number of edits is the length of first string
 else if(j == 0)
   dp[i][j] = i;
//If the last character of the two strings are
//same, ignore this character and recur for the
//remaining string
 else if(str1[i-1] == str2[j-1])
   dp[i][j] = dp[i-1][j-1];
//If last character is different, we need at least one
//edit to make them same. Consider all the possibilities
//and find minimum
 else
   dp[i][j] = 1 + min(min(
                dp[i-1][j], //Remove
```


CODING BLOCKS

Scanned by CamScanner

```
                dp[i][j-1]), //Insert
                dp[i-1][j-1] //Replace
                );
        }
    }
    //Return the most optimal solution
    return dp[m][n];
}
```

Solution: http://pastebin.com/ZVqfmHHJ

❑ MIXTURES

Problem Statement : Given n mixtures on a table, where each mixture has one of 100 different colors (0 - 99). When mixing two mixtures of color 'a' and 'b', resulting mixture have the color (a + b) mod 100 and amount of smoke generated is a*b. Find the minimum amount of smoke that we can get when mixing all mixtures together, given that we can only mix two adjacent mixtures.

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

The first thing to notice here is that, if we mix mixtures i...j into a single mixture, irrespective of the steps taken to achieve this, the final color of the mixture is same and equal to sum(i,j) = sum(color(i).. color(j)) mod 100.

So we define dp(i,j) as the most optimum solution where least amount of smoke is produced while mixing the mixtures from i...j into a single mixture. For achieving this, at the previous steps, we would have had to combine the two mixtures which are resultants of ranges i...k and k+1...j where i <= k <= j.

So it's about splitting the mixture into 2 subsets and each subset into 2 more subsets and so on such that smoke produced is minimized. Hence the recurrence relation will be:

dp(i,j) = min(k: i <= k < j) {dp(i,k) + dp(k+1,j) + sum(i,k)*sum(k+1,j)}

Code :

```
int minSmoke(int n)
{
 //Building the cummulative sum array
  sum[0] = col[0];
  for(int i = 1;i<n;i++)
     sum[i] = (sum[i-1] + col[i]);
 //dp[i][j] -> min smoke produced after mixing {color(i).....color(j)}
 //Note color after mixing {color(i).....color(j)} is sum[i...j] mod M
 //Building the dp in bottom up fashion
```


CODING BLOCKS

Scanned by CamScanner

```
    for(int i = n-1;i>=0;i-)
    {
      for(int j = 0;j<n;j++)
      {
        //Base Case
        //if i and j are equal then we have a single mixture and
        //hence no smoke is produced
        if(i == j)
        {
          dp[i][i] = 0;
           continue;
        }
       dp[i][j] = LONG_MAX;
       for(int k = i;k<j;k++)
       {
        int color_left = (sum[k] - sum[i-1]) % 100;
        int color_right = (sum[j] - sum[k]) % 100;
        dp[i][j] = min(dp[i][j], dp[i][k] + dp[k+1][j] + (color_left *
color_right));
       }
    }
  }
//return the most optimal answer
 return dp[0][n-1];
}
```

❑ 0-1 KNAPSACK PROBLEM

For each item you are given its weight and its value. You want to maximize the total value of all the items you are going to put in the knapsack such that the total weight of items is less than knapsack's capacity. What is this maximum total value?

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

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.

CODING BLOCKS

Scanned by CamScanner

Therefore, the maximum value that can be obtained from n items is max of following two values.

1. Maximum value obtained by n-1 items and W weight (excludingnth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.

Recurrence Relation :

```
//Base case
//If we have explored all the items all we have reached the maximum capacity
of Knapsack
if (n=0 or W=0)
 return 0
//If the weight of nth item is greater than the capacity of knapsack, we cannot
include //this item
if (wieght[n] > W)
 return solve(n-1, W)
otherwise
 return max{ solve(n-1, W), //We have not included the item
solve(n-1, W-weight[n]) //We have included the item in the knapsack
}
```

If we build the recursion tree for the above relation, we can clearly see that the **property of overlapping sub-problems** is satisfied. So, we will try to solve it using dynamic programming.

Let us define the dp solution with states i and j as
dp[i,j] —> max value that can be obtained with objects u
pto index i and knapsack capacity of j.

The most optimal solution to the problem will be **dp[N][W]** i.e. max value that can be obtained upto index N with max capacity of W.

Code :

```
int knapsack(int N, int W)
{
 for(int i = 0;i<=N;i++)
  {
   for(int j = 0;j<=W;j++)
    {
     //Base case
```


CODING BLOCKS

Scanned by CamScanner

```
        //When no object is to be explored or our knapsack's capacity is 0
        if(i == 0 || j == 0)
          dp[i][j] = 0;
        //When the wieght of the item to be considered is more than the
        //knapsack's capacity, we will not include this item
        if(wt[i-1] > j)
          dp[i][j] = dp[i-1][j];
        else
          dp[1][j] = max(
                    //If we include this item, we get a value of val[i-1] but
 the
                    //capacity of the knapsack gets reduced by the weight of
 that
                   //item.
                    val[i-1] + dp[i-1][j - wt[i-1]],
                    //If we do not include this item, max value will be the
                    //solution obtained by taking objects upto index i-1,
capacity
                   //of knapsack will remain unchanged.
                  dp[i-1][j]);
      }
    }
  return dp[N][W];
}
```

Time Complexity = O(NW)

Space Complexity = O(NW)

Can we do better?

If we observe carefully, we can see that the dp solution with states (i,j) will depend on state (i-1, j) or (i-1, j-wt[i-1]). In either case the solution for state (i,j) will lie in the **i-1**th row of the memoization table. So at every iteration of the index, we can copy the values of current row and use only this row for building the solution in next iteration and no other row will be used. Hence, at any iteration we will be using only a **single** row to build the solution for current row. Hence, we can reduce the space complexity to just **O(W).**

CODING BLOCKS

Scanned by CamScanner

Space-Optimized DP Code :

```
int knapsack(int N, int W)
{
 for(int j = 0;j<=W;j++)
      dp[0][j] = 0;
 for(int i = 0;i<=N;i++)
 {
   for(int j = 0;j<=W;j++)
   {
   //Base case
   //When no object is to be explored or our knapsack's capacity is 0
    if(i == 0 || j == 0)
       dp[1][j] = 0;
   //When the wieght of the item to be considered is more than the
   //knapsack's capacity, we will not include this item
    if(wt[i-1] > j)
       dp[1][j] = dp[0][j];
   else
       dp[1][j] = max(
                //If we include this item, we get a value of val[i-1] but the
                //capacity of the knapsack gets reduced by the weight of that
               //item.
                  val[i-1] + dp[0][j - wt[i-1]],
                //If we do not include this item, max value will be the
                //solution obtained by taking objects upto index i-1, capacity
                //of knapsack will remain unchanged.
                dp[0][j]);
   }
       //Here we are copying value of current row into the previous row,
       //which will be used in building the solution for next iteration ofrow.
    for(int j = 0;j<=W;j++)
       dp[0][j] = dp[1][j];
   }
return dp[1][W];
}
```

Time Complexity: O(N*W)

Space Complexity: O(W)

CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

❑ ROD CUTTING PROBLEM

You are given a rod of size n > 1, it can be cut into any number of pieces k. Price for each piece of size i is represented as p(i) and maximum revenue from a rod of size i is r(i) (could be split into multiple pieces). Find r(n) for the rod of size n.

Example :

Let the length of the rod is 6.

Price of size is given below

Length	1	2	3	4	5	6
Price(p)	2	5	8	9	10	11

Sol. Possible number cut on given rod :

Rod(6)	1*6	1*4,2	1*3, 3	1,2,3	1*2,4	1,5	2*3	2,4	3,3
Price(p)	12	13	14	15	13	12	15	13	16

Max Revenue : 16 (3,3)

Approach : Here we have to generate all the configurations of different pieces and find the highest priced configuration. We can get the best price(maximum revenue) by making a cut at different positions and comparing the values obtained after a cut.

Optimal Substructure Property :

Let the maximum revenue be r(n).

then, r(n) = max [p(n), p(1) + r(n - 1), p(2)+r(n-2),....., p(n-1)+r(1)]

$r(n) = \max[p(i) + r(n-i)] \; \forall \; 1 \le i \le (n)$

if RodCut(n) be the function which give the value of r(n) then

RodCut(n) = max[p(i) + RodCut(i)]] $\forall \; 1 \le i \le (n)$

Code :

```
#include <iostream>
#include <bits/stdc++.h>
using namespace std;
int max(int a, int b)
{
    return (a>b)?a:b;
}
```

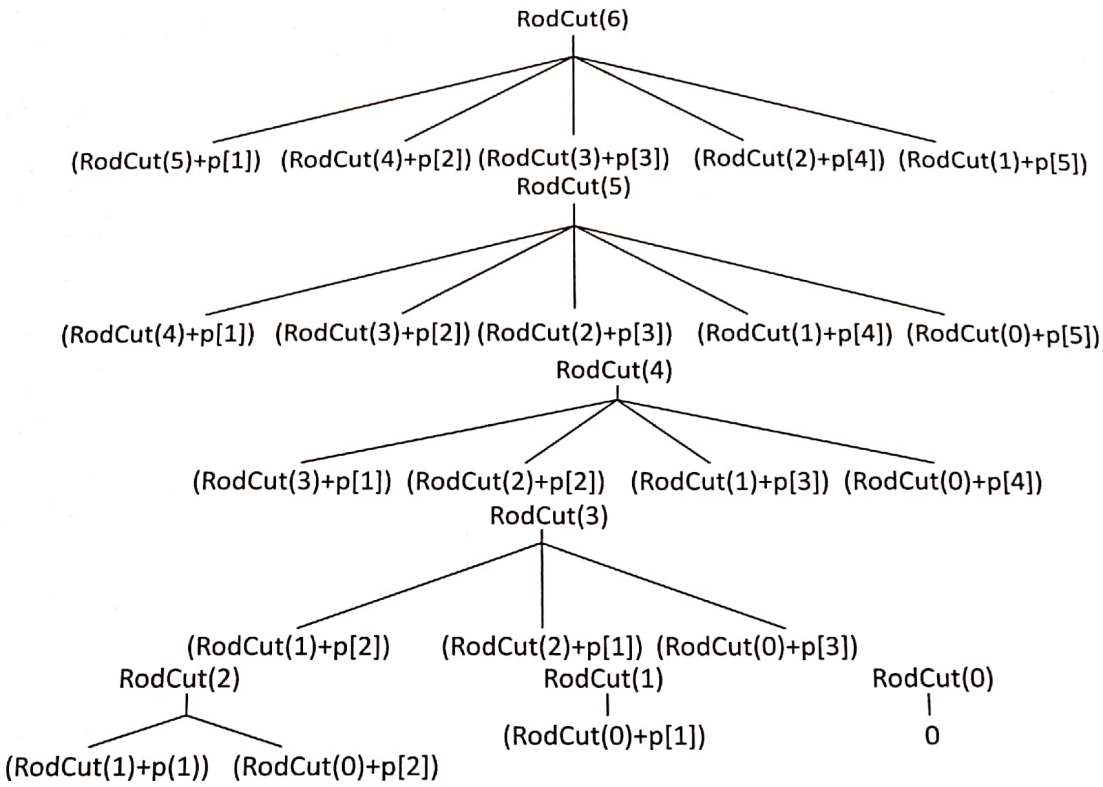

CODING BLOCKS

Scanned by CamScanner

```
int RodCut(int p[], int n)
{
        int max_revenue = INT_MIN;
        if(n<=0)
                return 0;
        else
        {
                for(int i=1;i<=(n);i++)
                {
                                                max_revenue =
max(max_revenue,p[i]+RodCut(p,n-i));
                }
        }
        return max_revenue;
}
int main()
{
        /* code */
        int t;
        cin>>t;
        for(int i=0;i<t;i++)
        {
                int n;
                cin>>n;
                int p[n+1];
                p[0]=0;
                for(int i=1;i<=n;i++)
                {
                        cin>>p[i];
                }
                cout<<"Maximum Revenue of given rod of length "<< n << " =
"<<RodCut(p,n)<<"\n";
        }
        return 0;
}
```


Scanned by CamScanner

Overlapping Subproblems :



Time Complexity : Exponential Time Complexity

A rod of length n can have n – 1 exactly cut positions. We can choose any the k cut (without repetition) anywhere we want so that for each such k the number of different choices is

$$\binom{n-1}{k}$$

when we sum up over all possibilities for 0 £ k £ n – 1 then

$$\sum_{k=0}^{n-1}\binom{n-1}{k}=\sum_{k=0}^{n-1}\frac{(n-1)!}{k!*(n-1-k)!}=2^{n-1}$$

Top Down Method :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
```


CODING BLOCKS
Code Your Way To Success

Scanned by CamScanner

```
int max(int x, int y)
{
      return (x>y)?x:y;
}
int RodCut(int p[], int n)
{
      int Revenue[n+1];
    Revenue[0]=0;
    for(int i=1;i<=n;i++)
   {
      int best_price = INT_MIN;
      for(int j=1;j<=i;j++)
      {
            best_price = max(best_price, p[j]+Revenue[i-j]);
      }
      Revenue[i]=best_price;
   }
   return Revenue[n];
}
int main()
{
      /* code */
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {
            int n;
            cin>>n;
            int p[n+1];
            p[0]=0;
            for(int i=1;i<=n;i++)
            {
                                    cin>>p[i];
            }
            cout<<"Maximum Revenue of given rod of length "<< n << " =
"<<RodCut(p,n)<<"\n";
      }
      return 0;
}
int Revenue[MAX];
int max(int a, int b)
```


CODING BLOCKS

Scanned by CamScanner

```
{
        return (a>b)?a:b;
}
int RodCut(int p[], int n)
{
        int max_revenue = INT_MIN;
        if(n<=0)
                        return 0;
                    else if(Revenue[n]==-1)
        {
            for(int i=1;i<=(n);i++)
            {
                        max_revenue = max(max_revenue,p[i]+RodCut(p,n-i));
            }
            Revenue[n] = max_revenue;
            return Revenue[n];
          }
            else
           return Revenue[n];
}
int main()
{
        /* code */
        int t;
        cin>>t;
        for(int i=0;i<t;i++)
        {
              int n;
              cin>>n;
              int p[n+1];
              p[0]=0;
              for(int i=1;i<=n;i++)
              {
                                        cin>>p[i];
                                        // Revenue[i]=p[i];
```


CODING BLOCKS

Scanned by CamScanner

```
            }
            for(int  i=0;i<=n;i++)
                  Revenue[i]  =  -1;


            cout<<"Maximum Revenue of given rod of length "<< n << " =
"<<RodCut(p,n)<<"\n";
      }
      return 0;
}
```

Bottom Up :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int max(int x, int y)
{
      return (x>y)?x:y;
}
int RodCut(int p[], int n)
{
      int Revenue[n+1];
   Revenue[0]=0;
   for(int i=1;i<=n;i++)
   {
      int best_price = INT_MIN;
      for(int j=1;j<=i;j++)
      {
            best_price = max(best_price, p[j]+Revenue[i-j]);
      }
      Revenue[i]=best_price;
   }
   return Revenue[n];
}
int main()
```


CODING BLOCKS
Scanned by CamScanner

```
{
        /* code */
        int t;
        cin>>t;
        for(int i=0;i<t;i++)
        {
                int n;
                cin>>n;
                int p[n+1];
                p[0]=0;
                for(int i=1;i<=n;i++)
                {
                                        cin>>p[i];
                }
                cout<<"Maximum Revenue of given rod of length "<< n <<" =
"<<RodCut(p,n)<<"\n";
        }
        return 0;
}
```

❑ LONGEST PALINDROME SUBSEQUENCE

Given a sequence of character, find the length of the longest palindromic subsequence in it.

Ex. if the given sequence is "BBABCBCAB", then the output should be 7 as "BABCBAB" is the longest palindromic subsequence in it.

Note : For given sequence "BBABCBCAB", subsquence "BBBBB" and "BBCBB" are also palindromic subsequnce but these are not longest.

Approach : The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence.

Optimal substructure Property :

Let X[0..n-1] be the input sequence of length n and L(0, n-1) be the length of the longest palindromic subsequence of X[0..n-1].

i.e. $X = A_1A_2A_3 \ldots\ldots\ldots\ldots A_{n-1}A_n$

CODING BLOCKS

Scanned by CamScanner

Recursive Relation :

$$L[i, j] = \begin{cases} 2 + L[i+1, j-1], & \text{if } X[i] = X[j] \\ \max(L[i, j-1], m[i+1, j]), & \text{if } X[i] \neq [j] \end{cases}$$

Base Cases :

$$L[i, j] = \begin{cases} 1, & \text{if } i = j \\ 2, & \text{if } i = j-1, X[i] = X[j] \end{cases}$$

Start the code with i = 0 and j = n – 1;

Code :

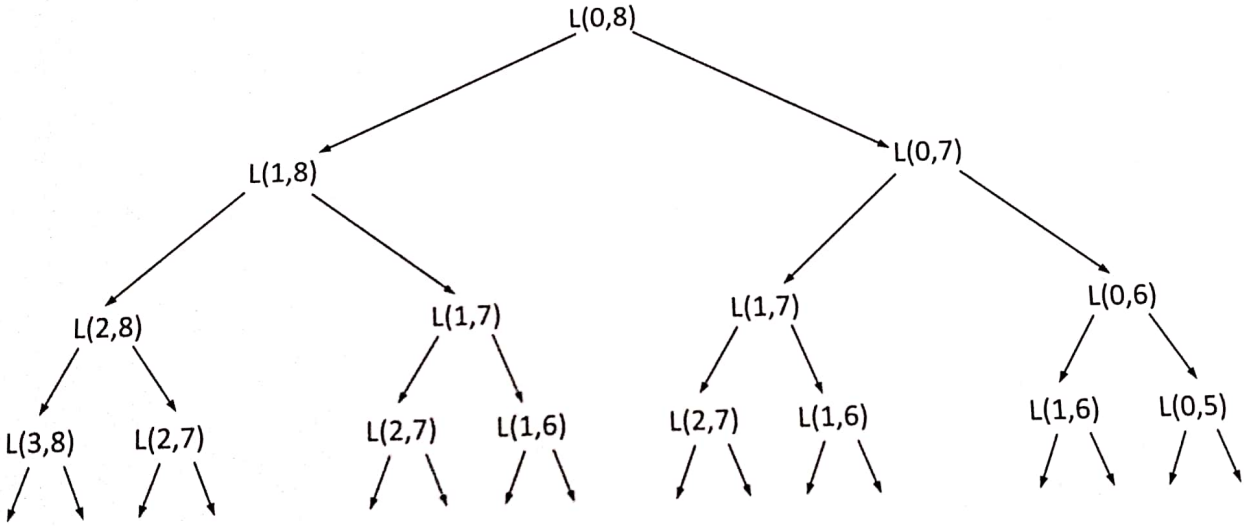
```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int SubSeq(char seq[], int i, int j)
{
      int length_SubSeq =0;
  if(i==j)
      return 1;
  else if(i==(j-1) && seq[i]==seq[j])
  {
            return 2;
  }
  else if(seq[i]==seq[j])
      length_SubSeq = SubSeq(seq,i+1,j-1)+2;
  else
      length_SubSeq = max(SubSeq(seq,i+1,j), SubSeq(seq, i, j-1));
  return length_SubSeq;
}
int main()
{
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {

       int n;
```


CODING BLOCKS

Scanned by CamScanner

```
        cin>>n;
                char seq[n];
        for(int j=0;j<n;j++)
        {
       cin>>seq[j];
        }
          cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,0,n-
1)<<"\n";
       }
}
```

Recursion Tree (Overlapping Subproblems) :



Top Down (Memoization) :

```
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int length[MAX][MAX];
int SubSeq(char seq[], int i, int j)
{
      if(length[i][j]!=-1)
                return length[i][j];
            else
              {
   if(i==j)
```


CODING BLOCKS

Scanned by CamScanner

```
    length[i][j]=1;
   else if(seq[i]==seq[j] && i==(j-1))
   {
      length[i][j]=2;
   }
   else if(seq[i]==seq[j])
    length[i][j] = SubSeq(seq,i+1,j-1)+2;
  else
    length[i][j] = max(SubSeq(seq,i+1,j), SubSeq(seq, i, j-1));
 }

  return length[i][j];
}
int main()
{
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {


       int n;
       cin>>n;
            char seq[n];
       for(int j=0;j<n;j++)
       {
      cin>>seq[j];
       }
       for(int j=0;j<n;j++)
       {
          for(int k=0;k<n;k++)
            length[j][k]=-1;


       }
      cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,0,n-1)<<"\n";
      }
}
```


CODING BLOCKS

Scanned by CamScanner

Bottom Up (Tabulation) :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int SubSeq(char seq[], int n)
{
    int length[n][n];
    for(int i=0;i<n;i++)
        length[i][i]=1;
    for(int k=2;k<=n;k++)
    {
        for(int i=0;i<=(n-k);i++)
        {
            int j = k+i-1;
            if(k==2 && seq[i]==seq[j])
            {
                length[i][j]=2;
            }
            else
            {
                if(seq[i]==seq[j])
                {
                    length[i][j] = 2+length[i+1][j-1];
                }
                else
                {
                    length[i][j]=max(length[i][j-1],length[i+1][j]);
                }
            }
        }
    }
    return length[0][n-1];
}
int main()
{
    int t;
    cin>>t;
    for(int i=0;i<t;i++)
    {
    int n;
```


CODING BLOCKS

Scanned by CamScanner

```
        cin>>n;
              char seq[n];
        for(int j=0;j<n;j++)
        {
        cin>>seq[j];
         }
           cout<<"Length of maximum palindrome Subsequence is "<<SubSeq(seq,n)<<"\n";
        }
}
```

MATRIX CHAIN MULTIPLICATION

We are given a sequence(chain) (A_1 and A_2 , ... , A_n) of n matrices to be multiplied. Our work is to find the most efficient way to multiply these matrices together.

Since matrix multiplication is associative, So we have many option to multiply a chain of matrices.
Ex. Let the chain of matrices is (A_1, A_2, A_3, A_4) and we need to find $A_1 . A_2 . A_3 . A_4$.

There are multiple ways to find this.

1. $((A_1 . A_2) . (A_3 . A_4))$
2. $((A_1 (A_2 . A_3)) . A_4))$
3. $(A_1 . ((A_2 . A_3)) . A_4))$
4. $(((A_1 . A_2) . A_3) . A_4)$
5. $(A_1 . (A_2 . (A_3 . A_4)))$

Multiplication Cost :

Let the size of the matrix is given $A_1 = 40 * 20$, $A_2 = 20 * 30$, $A_3 = 30 * 10$, $A_4 = 10 * 30$

Then multiplication cost for 1st multiplication would be :

(A_1, A_2) : Cost $= (40 * 20 * 30)$, Matrix Size $= 40 * 30$

(A_3, A_4) : Cost $= (30 * 10 * 30)$, Matrix Size $= 30 * 30$

Total Cost $= (40 * 10 * 30) + (30 * 10 * 30) + (40 * 30 * 30) = 70200$ size of Resultant Matrix = $40 * 30$

Total Cost = (40 * 20 * 30) + (30 * 10 * 30) + (40 * 30 * 30) = 70200 Size of Resultant Matrix = 40 * 30

Similarly cost for other multiplication would be given below :

	Multiplication	Cost
1.	$((A_1, A_2), (A_3, A_4))$	$(40 * 20 * 30) + (30 * 10 * 30) + (40 * 30 * 30) = 70200$
2.	$((A_1, (A_2, A_3)), A_4))$	$20 * 30 * 10 + 40 * 20 * 10 + 40 * 10 * 30 = 26000$
3.	$((A_1, (A_2, A_3)), A_4))$	$20 * 30 * 10 + 20 * 10 * 30 + 40 * 20 * 30 = 36000$
4.	$(((A_1, A_2), A_3), A_4)$	$40 * 20 * 30 + 40 * 30 * 10 + 40 * 10 * 30 = 48000$
5.	$(A_1, (A_2, (A_3, A_4)))$	$30 * 10 * 30 + 20 * 30 * 30 + 40 * 20 * 30 = 51000$

CODING BLOCKS

Scanned by CamScanner

Approach :

Approach towards the solution is to place parentheses at all possible places, calculate the cost for each placement and return the minimum value.

Suppose p[n + 1] is an array contain the size of each matrix. i.e.

size of $A_i = p[i-1] * p[i], \forall 1 d \le i d \le n$

Let m[i,j] be the minimum number scalar multiplication needed to compute the multiplication of matrix A_i to A_j (i.e. $A_i \cdot A_{j+1} \cdot A_{i+2} \cdots\cdots\cdots A_j$).

So m[1,n] would be the lowest cost to compute the multiplication of matrix to (i.e. $A_1 \cdot A_2 \cdot A_3 \cdots\cdots A_n$).

A recursive Solution :

We can split the product the product $A_i \cdot A_{i+1} \cdot A_{i+2} \cdots\cdots A_j$ between A_k and A_{k+1} where $i \le k < j$ then m[i,j] equals the minimum cost for computing the subproducts $A_{i\ldots\ldots k}$ and $A_{k+1\ldots\ldots j}$, plus the cost of multiplying these two matrices together.

- Size of $A_{i\ldots\ldots k}$ = p[i-1]*p[k] and size of $A_{k+1\ldots\ldots j}$ = p[k]*p[j]
- then multiplication cost of $A_{i\ldots\ldots k} * A_{k+1\ldots\ldots j} = p[i-1] * p[k] * p[j]$
- $A_{i\ldots\ldots k}$ = minimum cost to multiply the matrices $(A_i \cdot A_{i+1} \cdot A_{i+2} \cdots\cdots\cdots A_k)$ = m[i, k]
- $A_{k+1\ldots\ldots j}$ = minimum cost to multiply the matrices= m[k + 1, j]
- m[i, j] = m[i, k] + m[k + 1, j] + p[i – 1] * p[k] * p[j]

m[i,j] = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]

Possible value of k is j – i i.e. k = i, i + 1, i + 2, ..., j – 1

We need to check them all values of k to find the best(minimum cost i.e optimal solution).

if i==j

m[i, j] = 0;

else

m[i,j] = min(m[i, k] + m[k + 1, j] + p[i – 1] * p[k] * p[j]), $\forall i \le k < j$

Code :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
    return (x>y)?y:x;
}
```


CODING BLOCKS

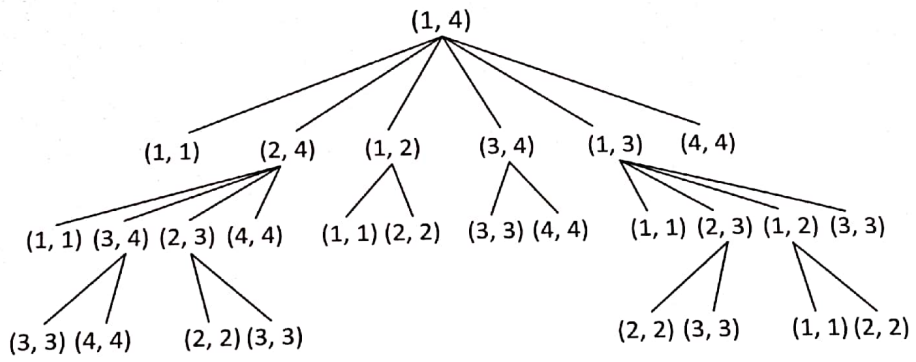
Scanned by CamScanner

```
int Matrix_Chain(int p[], int i,int j)
{
      int mult_cost = INT_MAX;
      if(i==j)
            return 0;
      else
      {
      for(int k=i;k<j;k++)
      {
       mult_cost = min(mult_cost,
                        Matrix_Chain(p,i,k)+Matrix_Chain(p,k+1,j)+p[i-1]*p[k]*p[j]);
      }
       }
      return mult_cost;
}
int main()
{
      int t;
      cin>>t;
      for(int i=0;i<t;i++)
      {
            int n;
            cin>>n;
            int p[n+1];
            for(int j=0;j<=n;j++)
            {
                  cin>>p[j];
            }
      cout<<"Minimum cost for matrix multiplication is :"<<"
"<<Matrix_Chain(p,1,n)<<"\n";
   }
}
```


CODING BLOCKS

Scanned by CamScanner

Recursive Tree Diagram (Overlapping Subproblems) :



Top Down (Memoization):

Code :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int mult[MAX][MAX];
int min(int x, int y)
{
      return (x>y)?y:x;
}
int Matrix_Chain(int p[], int i,int j)
{
      if(mult[i][j]!=-1)
            return mult[i][j];
      else
      {
            int mult_cost = INT_MAX;
          if(i==j)
              mult[i][j]=0;
          else
           {
          for(int k=i;k<j;k++)
           {
```


Scanned by CamScanner

```
                mult_cost = min(mult_cost,
                        Matrix_Chain(p,i,k)+Matrix_Chain(p,k+1,j)+p[i-1]*p[k]*p[j]);
            }
            mult[i][j]=mult_cost;
            }
            return mult[i][j];
        }
        //return mult_cost;
}
int main()
{
        int t;
        cin>>t;
        for(int i=0;i<t;i++)
        {
                int n;
                cin>>n;
                int p[n+1];
                for(int j=0;j<=n;j++)
                {
                        cin>>p[j];
                }
                for(int j=0;j<MAX;j++)
                {
                        for(int k=0;k<MAX;k++)
                        {
                                mult[j][k]=-1;
                        }
                }
        cout<<"Minimum cost for matrix multiplication is :"<<"
"<<Matrix_Chain(p,1,n)<<"\n";
    }
}
```


CODING BLOCKS

Scanned by CamScanner

Bottom Up (Tabulation) :

```
#include <iostream>
#include <bits/stdc++.h>
#define MAX 200
using namespace std;
int min(int x, int y)
{
        return (x>y)?y:x;
}
int Matrix_Chain(int p[], int n)
{
        int mult_cost[n][n];
        for(int i=1;i<n;i++)
        {
                mult_cost[i][i]=0;
        }
        for(int l=2;l<n;l++)
        {
                for(int i=1;i<(n-l+1);i++)
                {
                        int j=i+l-1;
                        mult_cost[i][j] = INT_MAX;
                        for(int k=i;k<j;k++)
                        {
                                mult_cost[i][j] = min(mult_cost[i][j],
                                        mult_cost[i][k]+mult_cost[k+1][j]+p[i-
1]*p[k]*p[j]);
                        }
                }
        }
        return mult_cost[1][n-1];
}
int main()
{
        int t;
```


Scanned by CamScanner

```
    cin>>t;
    for(int i=0;i<t;i++)
    {
        int n;
        cin>>n;
        int p[n+1];
        for(int j=0;j<=n;j++)
        {
            cin>>p[j];
        }

    cout<<"Minimum cost for matrix multiplication is :"<<"
"<<Matrix_Chain(p,n+1)<<"\n";
  }
}
```


CODING BLOCKS
Scanned by CamScanner

DO IT YOURSELVES

- ❑ Follow Coding Blocks tutorials on Dynamic Programming with Bitmasks.
- ❑ Try the Dynamic Programming Section problems on HackerBlocks.

SELF STUDY NOTES

CODING BLOCKS
Code Your Way to Success

Scanned by CamScanner

SELF STUDY NOTES

CODING BLOCKS

Scanned by CamScanner

Scanned by CamScanner