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.

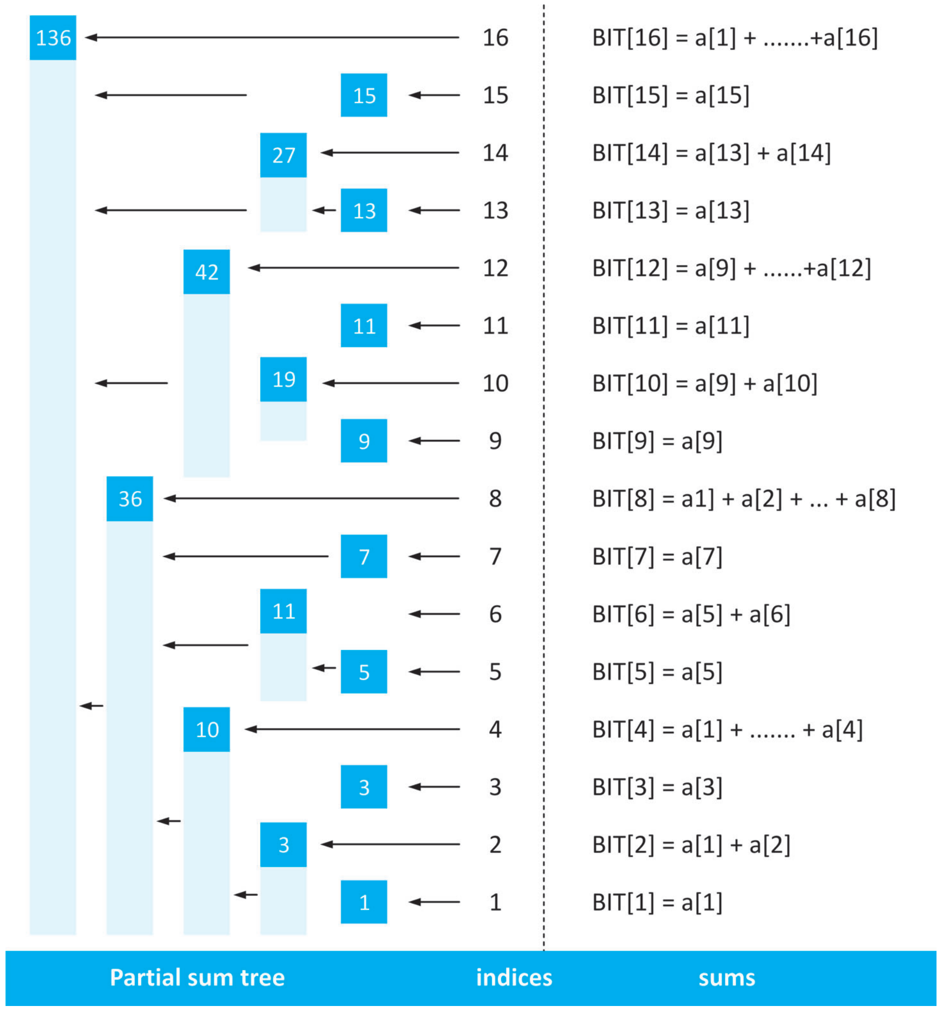
Using Binary Indexed Tree also, we can perform both the tasks in O(logN) time. But then why learn another data structure when segment tree can do the work for us. It's because binary indexed trees require less space and are very easy to implement during programming contests (the total code is not more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

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.

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.

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.

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

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

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.

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 {
```

```
        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.

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);
     }
}
```

```
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);
        }
```

```
        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, ..., 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}, ..., a_j.

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

SELF STUDY NOTES

SELF STUDY NOTES

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.

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);
}
```

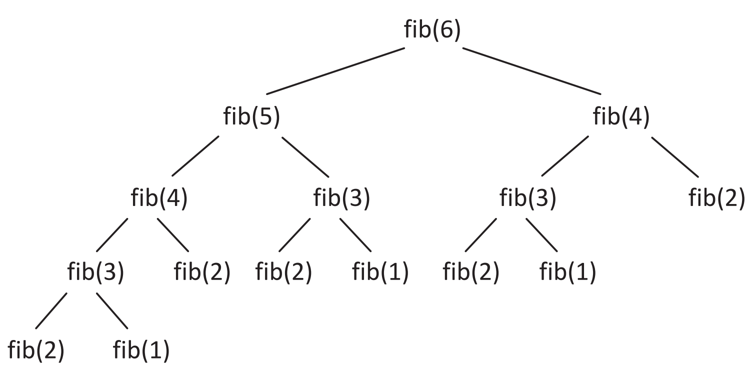
Using Dynamic Programming approach with memoization:

```
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 prevented using memoization.

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));}
```

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

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));
}
```

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.

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;
}
```

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;
}
```

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

❑ 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)},
```

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
```

```
  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);
```

```
            //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).

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.

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.

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
```

```
            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).

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
```

```
 //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];
```

```
    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;
```

```
  /* 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;
}
```

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

❑ LONGEST COMMON SUBSEQUENCE

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in 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 of 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
```

```
 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)

❑ 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
```

```
    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
```

```
    //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

❑ 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

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
```

```
                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
```

```
  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.

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
```

```
      //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).**

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)

❑ 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

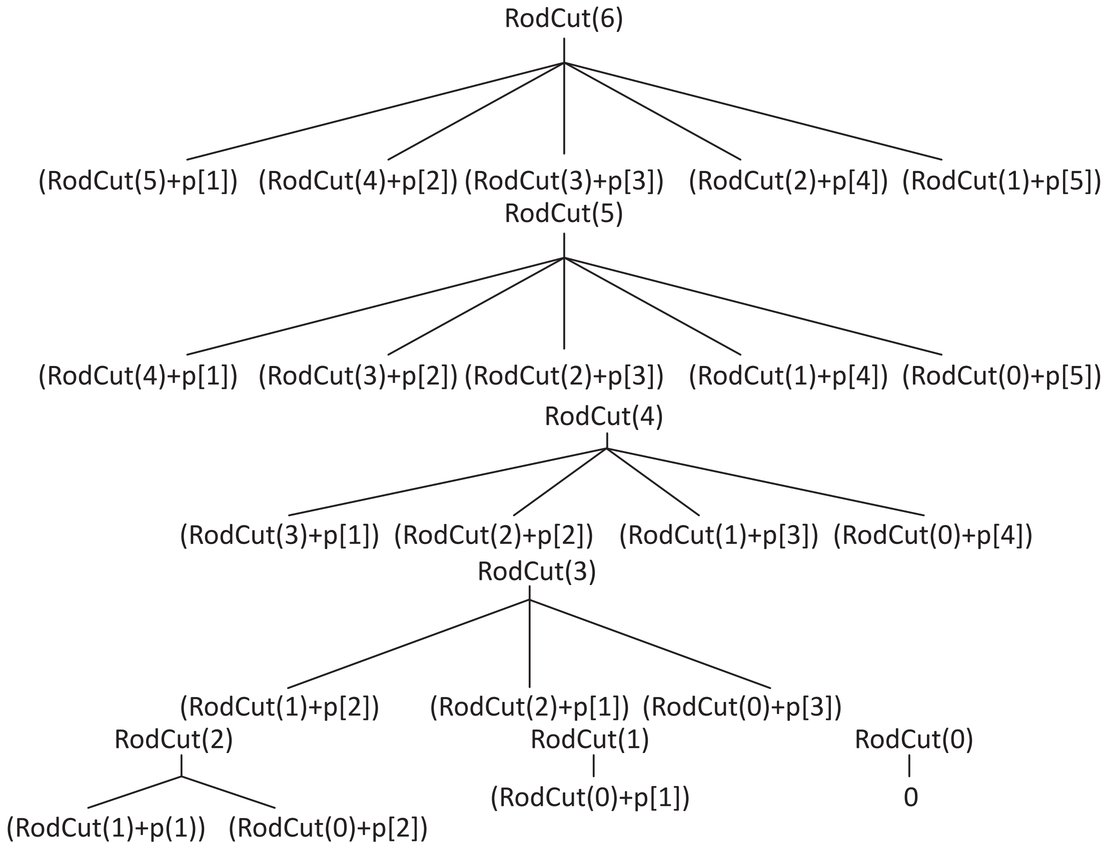
RodCut(n) = max[p(i) + RodCut(i)]] $\forall\ 1 \le i \le (n)$

Code :

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

```
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;
}
```

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;
```

```
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)
```

```
{
      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];
```

```
            }
            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()
```

```
{
        /* 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. $X = A_1A_2A_3 \ldots\ldots\ldots\ldots A_{n-1}\, A_n$

Recursive Relation :

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

Base Cases :

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

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

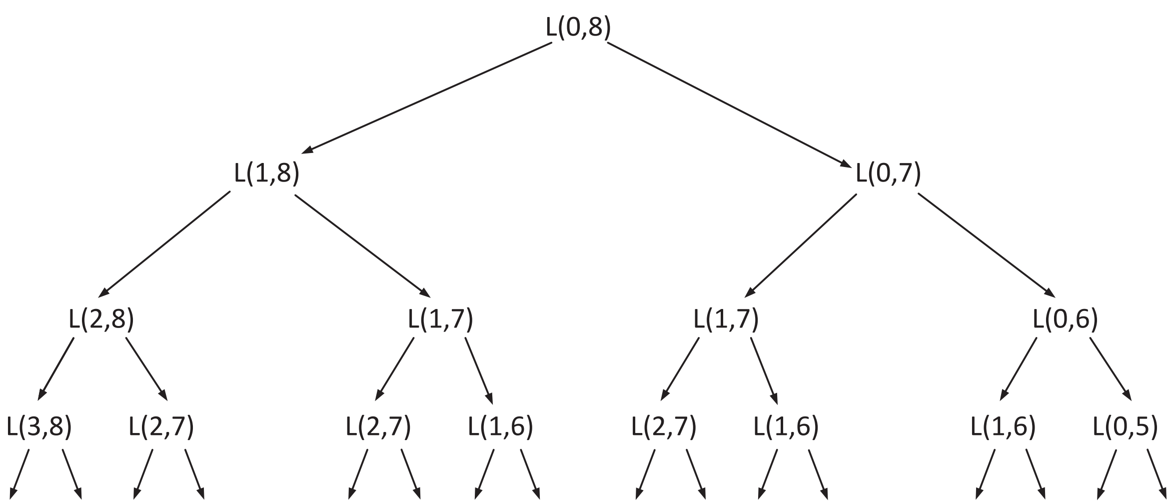
Code :

```
#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;
```

```
        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)
```

```
    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";
      }
}
```

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;
```

```
        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

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 ≤ i d ≤ 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. $A_i \cdot A_{j+1} \cdot A_{i+2} \cdots \cdots A_j$).

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 $A_{i......k}$ and $A_{k+1......j}$, plus the cost of multiplying these two matrices together.

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

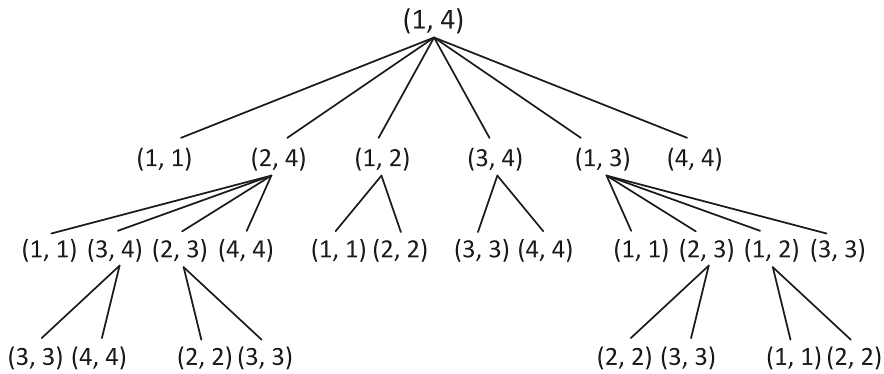
m[i,j] = min(m[i, k] + m[k + 1, j] + p[i – 1] * p[k] * p[j]), $\forall\, i \le k < j$

Code :

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

```
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";
   }
}
```

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++)
           {
```

```
                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";
   }
}
```

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;
```

```
       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";
   }
}
```

DO IT YOURSELVES

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

SELF STUDY NOTES

SELF STUDY NOTES

SELF STUDY NOTES

10 MO's Algorithm

Query SQRT Decomposition

Mo's algorithm is a generic idea. It applies to the following class of problems :

You are given array Arr of length N and Q queries. Each q uery is represented by two numbers L and R, and it asks you to compute some function with subarray Arr[L..R] as its argument.

Let's Start with a Simple Problem :

Given an array of size **N**. All elements of array <= **N**. You need to answer **M** queries. Each query is of the form **L, R**. You need to answer the count of values in range **[L, R]** which are repeated at least **3** times. Example: Let the array be {1, 2, 3, 1, 1, 2, 1, 2, 3, 1} (zero indexed)

Query: L = 0, R = 4**.** Answer = **1**. Values in the range [L, R] = {1, 2, 3, 1, 1} only 1 is repeated at least 3 times.

Query: L = 1, R = 8. Answer = **2**. Values in the range [L, R] = {2, 3, 1, 1, 2, 1, 2, 3} 1 is repeated 3 times and 2 is repeated 3 times.

Number of elements repeated at least 3 times = Answer = **2**.

Now explain a simple solution which takes O(N^2)

```
for each query:
answer = 0
count[] = 0
for i in {l..r}:
     count[array[i]]++
     if count[array[i]] == 3:
          answer++
```

Slight Modification to above algorithm. It still runs in O(N^2).

```
add(position):
  count[array[position]]++
  if count[array[position]] == 3:
    answer++

remove(position):
  count[array[position]]--
```

```
   if count[array[position]] == 2:
     answer–

currentL = 0
currentR = 0
answer = 0
count[] = 0
for each query:
   // currentL should go to L, currentR should go to R
   while currentL <= L:
      remove(currentL)
      currentL++
   while currentL >= L:
     add(currentL)
     currentL–
   while currentR <= R:
      add(currentR)
      currentR++
   while currentR >= R:
     remove(currentR)
     currentR–
   output answer
```

Initially we always looped from **L to R**, but now we are changing the positions from **previous query to adjust to current query.**

If previous query was **L=3, R=10**, then we will have **currentL=3** and **currentR=10** by the end of that query. Now if the next query is **L=5, R=7**, then we move the **currentL** to **5** and **currentR** to **7**.

add function means we are adding the element at position to our current set. And updating answer accordingly.

remove function means we are deleting the element at position from our current set. **And updating answer accordingly.**

MO’s algorithm is just an order in which we process the queries. We were given M queries, we will re-order the queries in a particular order and then process them. Clearly, this is an offline algorithm. Each query has L and R, we will call them opening and closing. Let us divide the given input array into Sqrt(N) blocks. Each block will be N / Sqrt(N) = Sqrt(N) size. Each opening has to fall in one of these blocks. Each closing has to fall in one of these blocks.

In this algorithm we will process the queries of **1st block**. Then we process the queries of **2nd block** and so on… finally **Sqrt(N)'th** block. We already have an ordering, queries are ordered in the ascending order of its block. There can be many queries that belong to the same block.

Let's focus on how we query and answer **block 1**. We will similarly do for all blocks. All of these queries have their opening in **block 1**, but their closing can be in any block including **block 1**. Now let us reorder these queries in ascending order of their **R value**. We do this for all the blocks.

How does the final order look like?

All the queries are first ordered in ascending order of their block number (block number is the block in which its opening falls). Ties are ordered in ascending order of their R value.

For example consider following queries and assume we have 3 blocks each of size 3.

{0, 3} {1, 7} {2, 8} {7, 8} {4, 8} {4, 4} {1, 2}

Let us re-order them based on their block number.

{0, 3} {1, 7} {2, 8} {1, 2} {4, 8} {4, 4} {7, 8}

Now let us re-order ties based on their R value.

{1, 2} {0, 3} {1, 7} {2, 8} {4, 4} {4, 8} {7, 8}

Now we use the same code stated in previous section and solve the problem. Above algorithm is correct as we did not do any changes but just reordered the queries.

Proof for complexity of above algorithm – O(Sqrt(N) * N) :

We are done with MO's algorithm, it is just an ordering.

It turns out that the O(N^2) code we wrote works in O(Sqrt(N) * N) time if we follow the above order.

With just reordering the queries we reduced the complexity from O(N^2) to O(Sqrt(N) * N), and that too with out any further modification to code.

Have a look at our code above, **the complexity over all queries is determined by the 4 while loops**. First 2 while loops can be stated as "**Amount moved by left pointer in total**", second 2 while loops can be stated as "**Amount moved by right pointer**". **Sum of these two will be the over all complexity.**

Let us talk about the right pointer first.

For each block, the queries are sorted in increasing order, so clearly the right pointer **(currentR)** moves in increasing order. During the start of next block the pointer possibly be at extreme end will move to least R in next block. That means for a given block, the amount moved by right pointer is **O(N)**. We have **O(Sqrt(N))** blocks, so the total is **O(N * Sqrt(N))**.

Let us see how the left pointer moves.

For each block, the left pointer of all the queries fall in the same block, as we move from query to query the left pointer might move but as previous L and current L fall in the same block, the moment is **O(Sqrt(N)) (Size of the block)**. In each block the amount left pointer movies is **O(Q * Sqrt(N))** where **Q** is number of queries falling in that block. Total complexity is **O(M * Sqrt(N))** for all blocks.

So, total complexity is **O((N + M) * Sqrt(N))** = **O(N * Sqrt(N)).**

This Algorithm is Offline :

That means we cannot use it when we are forced to stick to given order of queries. That also means we cannot use this when there are update operations. Not just that, there is one important possible limitation: We should be able to write the functions add and remove. There will be many cases where add is trivial but remove is not. One such example is where we want maximum in a range. As we add elements, we can keep track of maximum. But when we remove elements it is not trivial.

- **DQUERY (Spoj)**

Find number of distinct elements in a range l to r:

```
struct query {
     int l;
     int r;
     int in;
} q[200005];
```

How'll we sort query according to above defined way:

```
bool compare(query x, query y) {
    if (x.l / BLOCK != y.l / BLOCK) {
        // different blocks, so sort by block.
        return x.l / BLOCK < y.l / BLOCK;
    }
    // same block, so sort by R value
    return x.r < y.r;
}
```

Add and Remove :

```
void add(int cur) {
     cnt[arr[cur]]++;
     if (cnt[arr[cur]] == 1) {
         tans++;
    }
```

```
}

void remove(int cur) {
     cnt[arr[cur]]–;
     if (cnt[arr[cur]] == 0) {
          tans–;
     }
}
```

Main :

```
int main() {

     fastRead_int(n);
     loop(i, 0, n) {
          fastRead_int(arr[i]);
     }

     fastRead_int(t);
     loop(i, 0, t) {
          fastRead_int(a);
          fastRead_int(b);
          a–, b–;
          q[i].l = a, q[i].r = b, q[i].in = i;
    }

    //structure sorted
    sort(q, q + t, compare);
    int curL = 0, curR = 0;
    loop(i, 0, t) {

        while (curL < q[i].l) {
            remove(curL);
            curL++;
        }
        while (curL > q[i].l) {
            add(curL - 1);
            curL–;
        }
        while (curR <= q[i].r) {
```

```
            add(curR);
            curR++;
        }
        while (curR > q[i].r + 1) {
             remove(curR - 1);
             curR–;
        }
        ans[q[i].in] = tans;

    }
    for (int i = 0; i < t; ++i) {
        printf("%d\n", ans[i]);
    }
    return 0;
}
```

❑ POWERFUL ARRAY (CODE FORCES, 86-D)

An array of positive integers $a_1, a_2, \ldots, a_n$ is given. Let us consider its arbitrary subarray **al, al + 1 …, ar**, where **1** ≤ **l** ≤ **r** ≤ **n**. For every positive integer s denote by **Ks** the number of occurrences of s into the subarray. We call the power of the subarray the sum of products **Ks·Ks·s** for every positive integer **s**. The sum contains only finite number of nonzero summands as the number of different values in the array is indeed finite.

You should calculate the power of **t** given subarrays.

First line contains two integers n and t **(1 ≤ n, t ≤ 200000)** — the array length and the number of queries correspondingly.

Second line contains n positive integers ai **(1 ≤ ai ≤ 10^6)** — the elements of the array.

Next **t** lines contain two positive integers **l, r (1 ≤ l ≤ r ≤ n)** each — the indices of the left and the right ends of the corresponding subarray.

Code :

```
ll arr[200005];
ll ans[200005];
ll cnt[1000005];

ll tans;

struct query {
```

```
    int l;
    int r;
    int in;
} q[200005];

bool compare(query x, query y) {
   if (x.l / BLOCK != y.l / BLOCK) {
        // different blocks, so sort by block.
        return x.l / BLOCK < y.l / BLOCK;
    }
    // same block, so sort by R value
    return x.r < y.r;
}

void add(int cur) {
     cnt[arr[cur]]++;
     if(cnt[arr[cur]]>0)
          tans += arr[cur] * (2 * cnt[arr[cur]] - 1);
}

void remove(int cur) {
     cnt[arr[cur]]—;
     if(cnt[arr[cur]]>=0)
         tans -= arr[cur] * (2 * cnt[arr[cur]] + 1);
}

int main() {
     //ios_base::sync_with_stdio(false);
     //cin.tie(NULL);

     fastRead_int(n), fastRead_int(t);

     loop(i, 0, n) {
          fastRead_lint(arr[i]);
     }
     loop(i, 0, t) {
          fastRead_int(a), fastRead_int(b);
```

```
        a–, b–;
        q[i].l = a, q[i].r = b, q[i].in = i;
    }

    //structure sorted
    sort(q, q + t, compare);
    int curL = 0, curR = 0;
    loop(i, 0, t) {

        while (curL < q[i].l) {
            remove(curL);
            curL++;
        }
        while (curL > q[i].l) {
            add(curL-1);
            curL–;
        }
        while (curR <= q[i].r) {
            add(curR);
            curR++;
        }
        while (curR > q[i].r + 1) {
            remove(curR - 1);
            curR–;
        }
        ans[q[i].in] = tans;
    }

    for (int i = 0; i < t; ++i) {
        prl(ans[i]);
        nl;
    }

    return 0;
}
```

TIME TO TRY

Tree and Queries :

You have a rooted tree consisting of n vertices. Each vertex of the tree has some color. We will assume that the tree vertices are numbered by integers from 1 to n. Then we represent the color of vertex v as cv. The tree root is a vertex with number 1.

In this problem you need to answer to m queries. Each query is described by two integers vj,kj. The answer to query vj,kj is the number of such colors of vertices x, that the subtree of vertex vj contains at least kj vertices of color x.

INPUT :

8 5

1 2 2 3 3 2 3 3

1 2

1 5

2 3

2 4

5 6

5 7

5 8

1 2

1 3

1 4

2 3

5 3

OUTPUT :

2

2

1

0

1

SELF STUDY NOTES

SELF STUDY NOTES