CODING BLOCKS
Code Your Way To Success

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.

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);
}
```

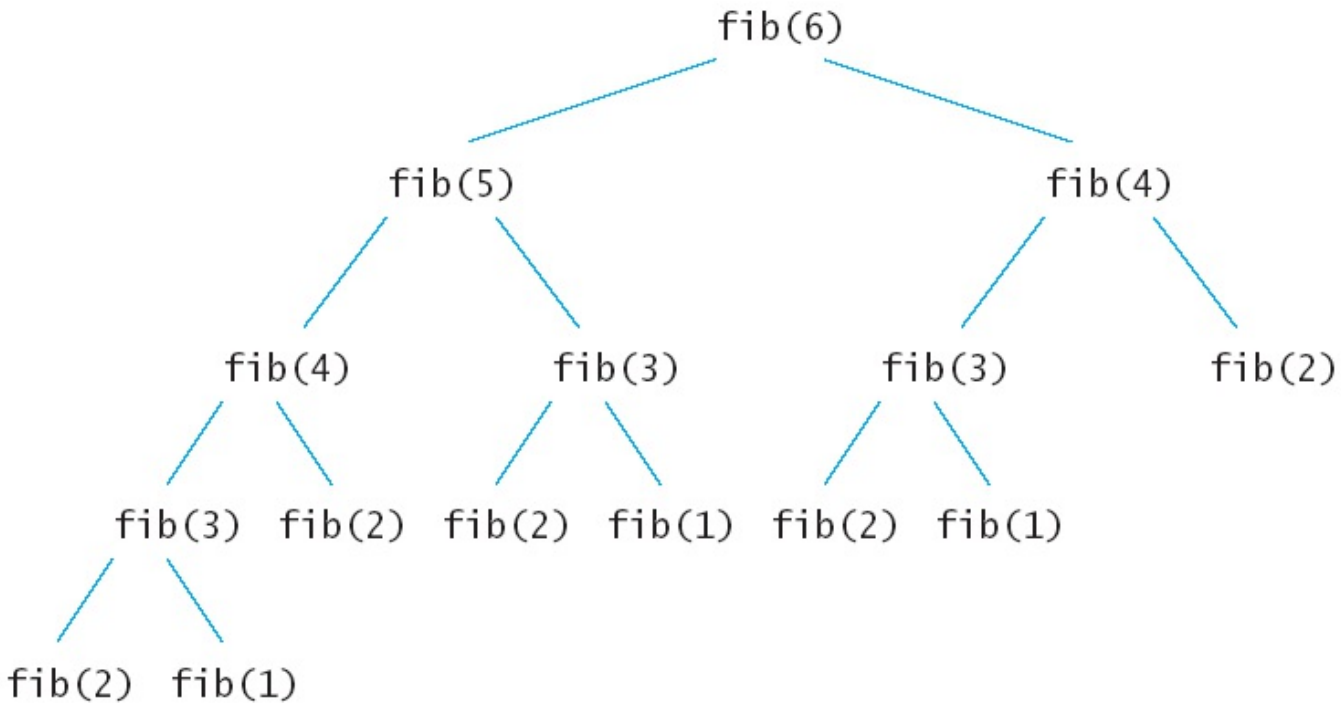
Using Dynamic Programming approach with memoization:

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

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

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 many 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.

Ques: 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 o
```

```
ne
    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 = Exponential (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 o
ne
    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.
```

Here ***ith state*** of dp: ***dp[i]*** denotes the most optimal answer till number N = ***i***.

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;
}
```

Ques. Find minimum number of coins that make a given value

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, recomputations 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 require to sum to ***i*** cents

Code

```
int minCoins(int N, int M)
{
    //Initializing all values to infinity i.e. minimum co
ins to make any
    //amout 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 u
pon 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 so
lution.
        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 includ
ed 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

Ques. Wine and Maximum Price

```
Imagine you have a collection of N wines placed next to e
ach other on a shelf. For simplicity, let's number the wi
nes 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 differ
ent).
```

```
Because the wines get better every year, supposing today
is the year 1, on year y the price of the ith wine will b
e 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. On
e 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 sh
elf (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, 4, 1

```
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 tha
n end.
    if(start > end)
        return 0;

    //Including the wine with starting index in our solut
ion
    int incStart = price[start] * year + maxPrice(price,
start + 1, end, year + 1);
    //Including the wine with ending index in our solutio
n
    int incEnd = price[end] * year + maxPrice(price, star
t, end-1, year + 1);

    //return the most optimal answer
    return max(incStart, incEnd);
```

```
}
```

Can we do better?

Yes we can! By crefully 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 memo
ized 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 solut
ion
    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 proble
m 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 sta
rt = 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:

```
MinCost(i,j) = min{ MinCost(i-1,j), MinCost(i,j-1) } + co
st[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-pr
oblems 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];
}
```

Ques: 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-pr
oblems 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 fur
ther
    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. f
rom 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-pr
oblems 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 re
ach
            //(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,j)
(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_sta
rt[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_s
tart[i+1][j], girl_start[i][j-1]);
        }
    }

    //building boy_end[][] table in bottomup fashion whic
h 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 bottomup fashion whi
ch specifies the journey from end to start
    //Here girl_end[i][j] --> the max calories that can b
e 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. b
etween (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

Ques. 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 seque
nce
    int cur_lis = LIS(n);

    //denotes the previous element printed
    //to print the LIS, previous element printed must alw
ays 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 th
e 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 fashi
on. 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)

Ques. To find the Shortest Common

Supersequence of two srings.

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

```
str1 = "geek",  str2 = "eke"
"geeke"

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 fashi
on. 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 sa
me, 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 o
f 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 characes of A and put the
```

```
m 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 the
m 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 ar
e equal
            //to the last value of LCS string
            ans += lcs_arr[l++];
            j--;
            i--;
        }
        //If we have exhausted all of our lcs_array and n
ow we
        //just have to merge the non-lcs characters of bo
th 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 numb
er of edits = 3
```

Recurrence relation

```
if str1[m] = str2[n]
    editDist(str1, str2, m, n) = editDist(str1, str2, m-1
, n-1)
otherwise
    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 --> Sat
ur
(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 transf
orm 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 second
string
            else if(i == 0)
```

```
                dp[i][j] = j;

            //If second string is empty, only option is t
o
            //remove all characters of first string
            //So number of edits is the length of first s
tring
            else if(j == 0)
                dp[i][j] = i;

            //If the last character of the two strings ar
e
            //same, ignore this character and recur for t
he
            //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 po
ssibilities
            //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

Ques. 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 its 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]) % 10
0;
                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];
}
```

Solution: http://pastebin.com/Wn685JaA

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 (excluding nth 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 t
he 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 inc
luded the item
                solve(n-1, W-weight[n]) //We have include
d 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 kna
psack's capacity is 0
            if(i == 0 || j == 0)
                dp[i][j] = 0;

             //When the wieght of the item to be consider
ed is more than the
            //knapsack's capacity, we will not include th
is 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 b
y the weight of that
                //item.

      val[i-1] + dp[i-1][j - wt[i-1]],
```

```
                //If we do not include this item, max val
ue will be the
                //solution obtained by taking objects upt
o index i-1, capacity
                //of knapsack will remain unchanged.
                                dp[i-1][j]

                                );
        }
    }
    return dp[N][W];
}
```

Time Complexity := O(N*W)*

***Space Complexity** := O(N*W)

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 kna
psack's capacity is 0
            if(i == 0 || j == 0)
                dp[1][j] = 0;

            //When the wieght of the item to be considere
d is more than the
            //knapsack's capacity, we will not include th
is 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 b
y the weight of that
```

```
                //item.

                            val[i-1] + dp[0][j - wt[i-1]]
,

                //If we do not include this item, max val
ue will be the
                //solution obtained by taking objects upt
o index i-1, capacity
                //of knapsack will remain unchanged.
                            dp[0][j]

                            );
        }
        //Here we are copying value of current row into t
he 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)

Solution: http://pastebin.com/V9fgGPxK

Created ßy Vishal Panwar