DP LECTURE

Dynamic Programming : a very powerful method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions/results.

There are two ways of doing this.

1.) Top-Down : Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it **and save the answer**. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***. *(We will go from higher states towards lower states).*

2.) Bottom-Up : Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is the **Iterative Approach.** *(We will go from lower states towards higher states).*

Iterative approach can be implemented in 2 ways, depending on the problem and what you find more intuitive.

a) **Push dp style** : *Pushing* the value of ith state to higher states to make suitable changes.
b) **Pull dp style** : computing the current state by *pulling* the values from lower states.

Types of problems that can be solved using Dp/Motivation behind studying the topic :

- If the problems asks us to minimize/maximize something or asks us to find the # of ways of doing something , then it’s a big hint towards dp.
- If you are able to form a recurrence relation ,then dp can be applied.
- If greedy approach isn't working then dp might just work.

Let’s take an example of finding the nth fibonacci term .

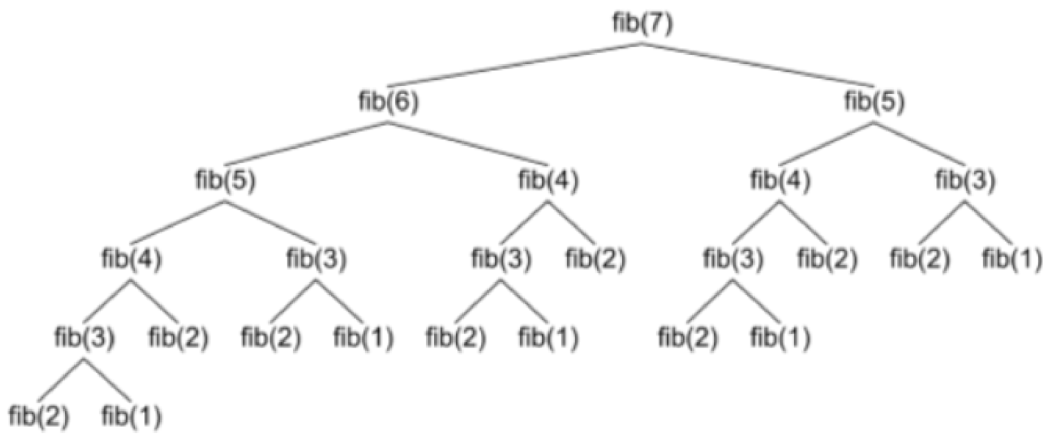
1) Method 1: (Recursive Solution)

Just naively writing down a recursive function :

Top down approach but without memoization:

```
ll fibo_simple(ll x) {
      if (x <= 2)return 1;
      return fibo_simple(x - 1) + fibo_simple(x - 2);
}
```

Problem with above approach :



For n=7, fibo(5) being calculated twice , fibo(4) being calculated thrice and so on……

Time complexity : T(n) = T(n-1) + T(n-2)

So to calculate the nth term, we will spend fibo(n) time.

Recursive solution (but with memoization) :

```
vector<ll>dp(N,garbage_val);
ll fibo_with_memo(ll x) {

      if (dp[x] > 0)return dp[x]; // since we already know it’s value.

      if (x <= 2) {
            dp[x] = 1;
      }
      else {
            dp[x] = fibo_with_memo(x - 1) +
                    fibo_with_memo(x - 2);
      }

      return dp[x];
}
```

Time complexity : O(n) Linear !!

Saving/Storing the result saved us a lot of time and computation.

Above was the example of *Top down Dp/Memoization*

Let's look at the *bottom up/iterative approach* : (pull dp)

```
vector<ll>fibo(1005);

      fibo[1] = 1, fibo[2] = 1;

      for (ll i = 3; i <= 100; i++) {
            fibo[i] = fibo[i - 1] + fibo[i - 2];
      }
```

- **The way you define your dp is super important.**

In the above case, fibo[i] denotes the ith term which seems very trivial and obvious but it is always not the case.

Problem: Find the number of different binary sequences of length N such that no 2 1's are adjacent to each other.

Solution :

Let dp[i] denotes the no. of such sequences of length i.

Now,consider a sequence of length i.
2 cases :

- ith element is 0. In this case (n-1)th element can be 0 or 1. So no. of ways will be dp[i-1]

[...........] 0

- ith element is 1. (i-1)th element must be 0. Then, (i-2)th element can either be 1 or 0.
  So # of ways will be dp[i-2]

…………...1

[………] 01

Hence we get the relation :

dp[i] =dp[i-1] +dp[i-2].

PROBLEM : https://atcoder.jp/contests/dp/tasks/dp_b

Solution : let dp[i] denote the minimum cost to reach the ith Stone:

Dp[i] =min (dp[j]+abs(h[i]-h[j])   (denotes the transition from jth to ith state)

where j=[i-1,i-2…..i-k]

Push Dp Implementation:

(from ith state ,make suitable changes to (i+1)th, (i+2)th …..(i+k)th state…)

```
        ll n, k;

        ll INF = 1e18;

        cin >> n >> k;

        vector<ll> h(n + 5);
```

```
for (ll i = 1; i <= n; i++)cin >> h[i];

vl dp(n + 5, INF);

dp[1] = 0; // important line.

for (ll i = 1; i <= n; i++) {
        for (ll j = i + 1; j <= min(n, i + k); j++) {
                dp[j] = min(dp[j], abs(h[i] - h[j]) + dp[i]);
        }
}

cout << dp[n] << endl;
```

It’s called push dp because we are “pushing” the ith state value to make changes to higher states.

Some Standard Dp Problems :

(will help you understand different ways in which we may define our dp)

- **0-1 Knapsack**
- **Coin change Problem**
- **Longest common subsequence (LCS)**

- **Longest Increasing Sequence (LIS)**
- **Solving Recursive relations for large n (order of 1e9,1e18) using matrix exponentiation**

0-1 Knapsack Problem

Problem: There are N items ,each of them having some weight w[i] and .cost[i] associated with them.You also have a Knapsack with capacity S
You need to put a subset of items in the Knapsack such that their total weight
does not exceed S and their total cost is Maximized.

You cannot break an item i.e either pick it completely or don’t pick it.
(Hence the name 0-1 Knapsack)

1st method (Brute force):

Consider all possibilities using bitmasks.

```
ll n, S;

        cin >> n >> S;

        vector<ll> w(n), c(n);

        for (ll i = 0; i < n; i++) {
                cin >> w[i] >> c[i];
        }

        ll ans = 0;

        for (ll msk = 0; msk < (1ll << n); msk++) {
                ll ret = 0;
                ll tw = 0;
                for (ll j = 0; j < n; j++) {
                        if ((1ll << j)&msk) {
                                ret += c[j];
                                tw += w[j];
                        }
                }
                if (tw <= S) {
                        ans = max(ans,ret);
                }
        }
        cout<< ans<< endl;
```

Time Complexity: O(n* 2^n) Huge !!!!!!   (works only for n<=20)

Let’s try to solve a simpler version of the problem.

What if you just need to tell ,what is the maximum weight <=S that you can achieve by choosing a subset of the items.

2nd Method (using DP):

Let us make a 2D array dp in which dp[i][j] is 1 if it is possible to make weight =‘j’ from the first ‘i’ items
Else dp[i][j] is 0.

Dp[i][j] ---> Dp[i-1][j] ,we didn’t choose the ith item.

Dp[i][j] ---> Dp[i-1][j-w[i]] , we chose the ith item.

If atleast one of the states is 1, then dp[i][j] will be 1.

In the end, simply report the maximum ‘x’ st dp[n][x] =1.

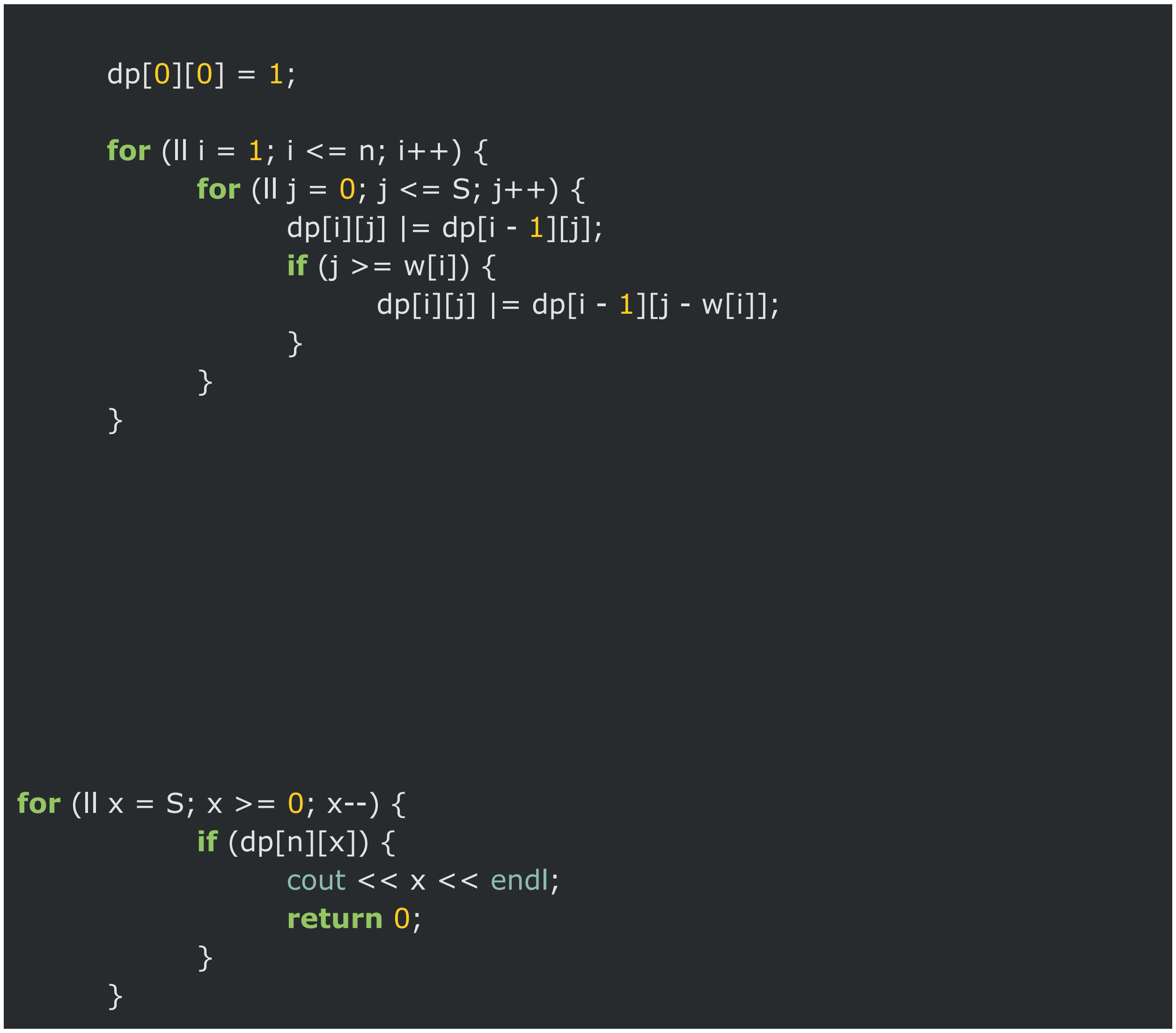
```
ll n, S;

cin >> n >> S;


vector<vector<ll>>dp(n + 5, vector<ll>(S + 5));

vector<ll>w(n + 5);

for (ll i = 1; i <= n; i++)cin >> w[i];
```



```
    dp[0][0] = 1;

    for (ll i = 1; i <= n; i++) {
        for (ll j = 0; j <= S; j++) {
            dp[i][j] |= dp[i - 1][j];
            if (j >= w[i]) {
                dp[i][j] |= dp[i - 1][j - w[i]];
            }
        }
    }









for (ll x = S; x >= 0; x--) {
        if (dp[n][x]) {
            cout << x << endl;
            return 0;
        }
    }
```

Time complexity : O(N*S)

Space complexity : O(N*S)

(However ,it can implemented in O(S) space)

So if N*S <=1e7-1e8 then do consider knapsack.

Now let’s come back to our original problem:

Let dp[i][j] denote the maximum cost that we can get from first ‘i’ items and total weight j.

Dp[i][j] --> Dp[i-1][j]   (Don’t pick the ith item)

Dp[i][j] ----> Dp[i-1][j-w[i]] + cost[i]   (pick it)

Note : Check whether these transitions can take place or not.
(for this ,we can use a 2d ‘vis’ array).

vis[i][j] is 1 if we can reach this state, else it is 0.

```
ll n, S;
        cin >> n >> S;

        vector<vector<ll>>dp(n + 5, vector<ll>(S + 5, -1e9));
        vector<vector<ll>>vis(n + 5, vector<ll>(S + 5, 0));
        vector<ll>w(n + 5);
        vector<ll>c(n + 5);
        for (ll i = 1; i <= n; i++)cin >> w[i] >> c[i];
        vis[0][0] = 1;

        for (ll i = 1; i <= n; i++) {
                for (ll j = 0; j <= S; j++) {

                        if (vis[i - 1][j]) {

                                dp[i][j] = max(dp[i][j], dp[i - 1][j]);

                                vis[i][j] = 1;
                        }
                        if (j >= w[i] && vis[i - 1][j - w[i]]) {

                dp[i][j] = max(dp[i][j], dp[i - 1][j - w[i]] + c[i]);

                                vis[i][j] = 1;
                        }
                }
        }


ll ans = 0;

        for (ll x = S; x >= 0; x--) {
                ans = max(ans, dp[n][x]);
        }

        cout << ans << endl;
```

Note: We can implement the above in O(S) space .(This memory optimisation is useful for avoiding MLE).

(Will discuss this optimization later)

- ## Coin Change Problem:

Problem: You have an infinite supply of n coins with values V1,V2,....Vn resp. and a target sum ‘S’.
Find the number of ways to make sum S. (**order doesn’t matter** i.e
For S=4 and V={1,2,5}, [1 1 2] is same as [1 2 1])

Solution : Let f[i][j] denote the no. of ways to get sum j using first i coins only.

Let the value of ith coin be x (i.e v[i] =x)

Then f[i][j] = f[i-1][j] + f[i-1][j-x] +f[i-1][j-2*x]..........

Note: f[i-1][j-m*x] denotes that we have chosen ith coin exactly m times.

ith row denotes that we are considering only first i coins.

jth column denotes sum=j.

F[i][j] : Value in the ith row and jth column.

Sum		0	1	2	3	4	5	6	7	8	9	10
{}		1	0	0	0	0	0	0	0	0	0	0
{1}		1	1	1	1	1	1	1	1	1	1	1
{1,2}		1	1	2	2	3	3	4	4	5	5	6
{1,2, 5}		1	1	2	2	3	4	5	6	7	8	10

Time complexity : bigger than O(n*m)

Reason: for v[i] =x , f[i][j] takes “some” time to compute and there are total n*m states.

Let’s try to compute f[i][j] in O(1).

f[i][j] = f[i-1][j] + f[i-1][j-x] +f[i-1][j-2*x]..........

F[i-1][j-x] +f[i-1][j-2*x]+........

is nothing but f[i][j-x]

So we can write : **f[i][j] = f[i-1][j] +f[i][j-x]**

Where x= V[i]

Time complexity and Space complexity O(N*M)

- **Finding minimum number of coins that make a given value/sum S.**

https://cses.fi/problemset/task/1634/

Similar solution as above .

We will get the following recurrence relation:

f[i][j] = min(f[i-1][j] ,1+f[i][j-x])

Where x= V[i]

(Try to derive it yourself)

Memory Efficient implementation (using 2 vectors).

```
ll n, x;

 cin >> n >> x;

 vl a(n + 1);
 for (ll i = 1; i <= n; i++) {
   cin >> a[i];
 }

 vector<ll> dp(x + 5, 1e9);


 dp[0] = 0;


 for (ll i = 1; i <= n; i++) {
   vector<ll> new_dp(x+5, 1e9);

   /*
            for ith iteration :
                       dp[j] refers to table[i-1][j]

                       new_dp[j] refers to table[i][j]
   */

   for (ll j = 0; j <= x; j++) {
     if (j >= a[i])
       new_dp[j] = min(new_dp[j], new_dp[j - a[i]] + 1);
```

```
    new_dp[j] = min(new_dp[j], dp[j]);

   }

   dp = new_dp;

 }

 if (dp[x] >= 1e9)dp[x] = -1;

 cout << dp[x] << endl;
```

• Longest Increasing Subsequence

Problem: Given an array A of length n, find the length of the longest increasing subsequence of A.

Example-
N=7
A=[2,1,8,5,3,6,9]
1,3,6,9
Ans=4

Solution :
Let dp[i] denote length of Longest Increasing Subsequence ending at i.

Then for every i in 1 to n,we see every j from 1 to i-1. If arr[i]>arr[j], then arr[i] is a possible contender for the last element of the LIS and we can include arr[i] in the LIS ending at j. We take the maximum of all the values of dp[i] to get the answer.(This is because LIS can end at any index and not necessarily n)

Code: (O(N*N) implementation)

```
        cin >> n;
        int arr[n+1];
        for(i=1;i<=n;i++)
            cin>>arr[i];
        int dp[n+1];
        //dp[i] denotes length of LIS ending at i
        for(i=1;i<=n;i++)
        {
            dp[i]=1;
            for(j=1;j<=(i-1);j++)
            {
                if(arr[i]>arr[j])
                {
                    dp[i]=std::max(dp[i],dp[j]+1);

                    //Taking maximum as we want largest possible
                    //length of LIS ending at i
                }
            }
            ans=std::max(ans,dp[i]);

        }
        cout<<ans;
```

For NlogN implementation -

https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/

Practice Task -

https://codeforces.com/problemset/problem/269/B
(Try to implement in NlogN as well)

Extra Task-

Try to print the Longest Increasing Subsequence

• Longest Common Subsequence

Given two string s1 and s2 of size n and m respectively, find the length of their Longest Common Subsequence(LCS)

Example -
4 5
KXBY
AXCCY

Answer =2
(XY)

Solution:

Let dp[i][j] denote length of Longest Common subsequence such that we consider first i characters of s1 and j characters of s2.Thus, the answer will be dp[n][m]. When i or j is 0, dp[i][j] will be 0. When characters at ith position of s1 and jth position of s2 are same(Here s1[i-1] and s2[j-1] due to 0-based indexing), we can include them in the LCS and dp[i][j]=1+dp[i-1][j-1]. If characters are different , we cannot include them in the LCS.So, dp[i][j] will be the maximum of dp[i-1][j](When we don't consider ith character of s1) and dp[i][j-1].(When we don't consider jth character of s2)

Code:

```
        cin >> n >> m;
        cin>>s1>>s2;
        int dp[n+1][m+1];
        //dp[i][j] denotes length of Longest Common Subsequence
        //when we consider first i characters of s1 and j characters
        //of s2
        for(i=0;i<=n;i++)
        {
            for(j=0;j<=m;j++)
            {
                if((i==0)||(j==0))
                {
                    dp[i][j]=0;
                }
                else
                {
                    if(s1[i-1]==s2[j-1])
                        dp[i][j]=1+dp[i-1][j-1];
                    else
                    {
                        dp[i][j]=std::max(dp[i-1][j],dp[i][j-1]);
                    }
                }

            }
        }
        int ans=dp[n][m];
        cout<<ans;
```

Practice Task :
https://cses.fi/problemset/task/1639

Extra Task:
Try to print the LCS of 2 strings.

Bitmask DP:

Problem:

There are N tasks and N people to complete those tasks. Given an N x N matrix where ith row and jth column is the cost of the jth person completing the ith task.

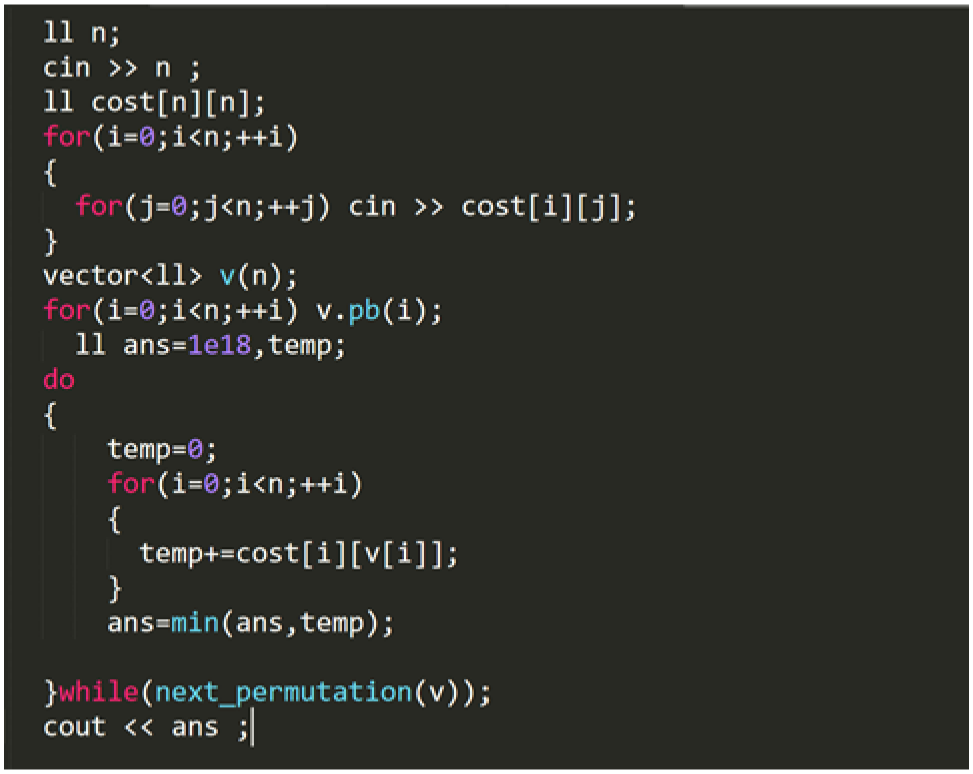
Find the assignment order such that total cost is minimum and one person can be assigned only one task.

Solution:

Brute Force :

Calculate the cost of all possible combinations to find the minimum.

Time complexity : O(N*(N!))



```
ll n;
cin >> n ;
ll cost[n][n];
for(i=0;i<n;++i)
{
  for(j=0;j<n;++j) cin >> cost[i][j];
}
vector<ll> v(n);
for(i=0;i<n;++i) v.pb(i);
  ll ans=1e18,temp;
do
{
    temp=0;
    for(i=0;i<n;++i)
    {
      temp+=cost[i][v[i]];
    }
    ans=min(ans,temp);

}while(next_permutation(v));
cout << ans ;
```

https://www.geeksforgeeks.org/stdnext_permutation-prev_permutation-c/

Bitmask dp solution:

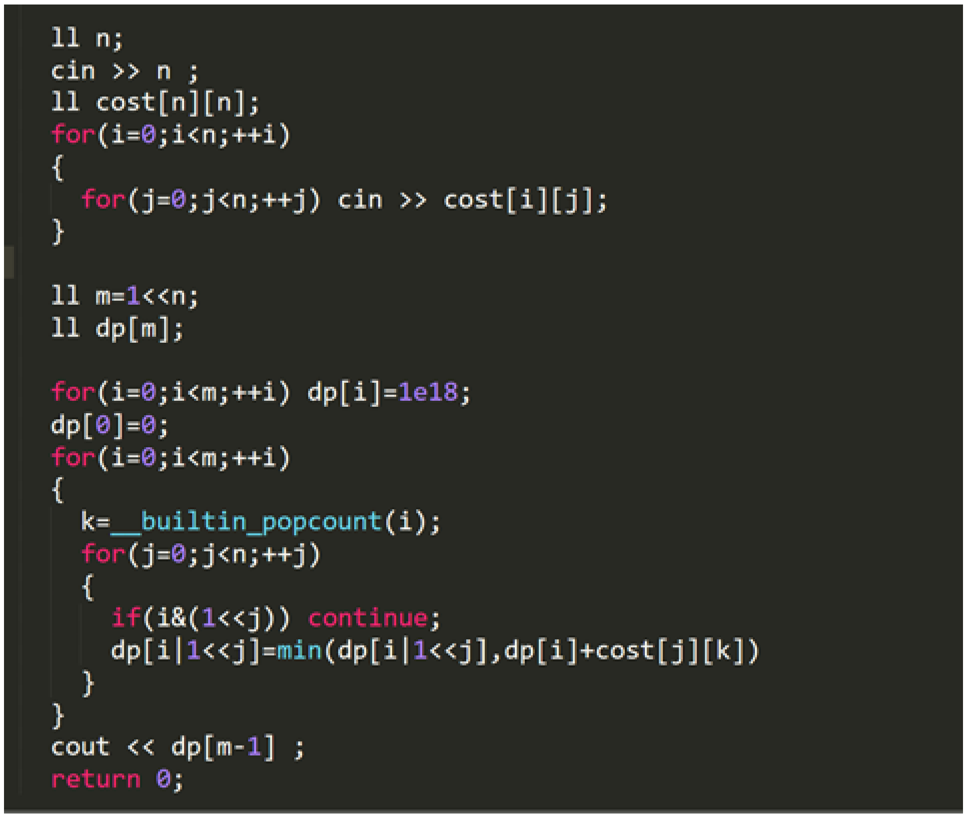
A binary number of length n is used to represent the tasks that have been completed. If the i-th bit is set 1 then it means that the i-th task has been completed.

Here,dp[x] represents the minimum cost of completing the tasks whose corresponding bit is set 1 using the first j-th people. Where j is the number of bits in x that are set 1.

For some jth task if its bit is not set 1 in x then :

(2^j)&x should be zero

Note: Given below is the “push” dp implementation. For a particular mask, all the set bits represents that those tasks have already been assigned. If exactly ‘k’ bits are set in the mask ‘i’ ,then the first k tasks have already been assigned. Now will try to assign (k+1)th task .



```
ll n;
cin >> n ;
ll cost[n][n];
for(i=0;i<n;++i)
{
  for(j=0;j<n;++j) cin >> cost[i][j];
}

ll m=1<<n;
ll dp[m];

for(i=0;i<m;++i) dp[i]=1e18;
dp[0]=0;
for(i=0;i<m;++i)
{
  k=__builtin_popcount(i);
  for(j=0;j<n;++j)
  {
    if(i&(1<<j)) continue;
    dp[i|1<<j]=min(dp[i|1<<j],dp[i]+cost[j][k])
  }
}
cout << dp[m-1] ;
return 0;
```

- Popcount is an built-in function in the gcc compiler used to find the # of set bits in the binary representation of a number.

Time complexity: O(N * 2^N)

Space complexity: O(2^N)

Resources :

1) Intuition Behind Dp
2) Knapsack
3) Coin Change
4) LIS/LCS
5) BitMask Dp
6) Matrix Exponentiation

Contests for practice :

1) ***** AtCoder Educational DP
2) ***** CSES
3) Codechef DSA Dp

Problems for practice :

1) Lecture Sleep
2) Mashmokh and ACM
3) Orac and Models
4) XOR with Subset
5) Find the Spruce

6) Dessert Wizard
7) Maximum Sum of Products
8) Baby Ehab Partitions Again
9) Good Subarrays
10) Almost Identity Permutations
11) Booking System
12) Bargain
13) Buns
14) Treats for the Cows
15) Buses
16) Mysterious Present
17) Ilya and Escalator
18) Multiple of 2019
19) The Empty One
20) Caesar's Legion
21) Wonderful Randomized Sum
22) Flowers
23) Maximum Score
24) Kefa and Dishes
25) Fish
26) Let's Go Rolling
27) LionAge II
28) Longest Regular Bracket Sequence
29) Shooting Gallery
30) Education Reform
31) Two out of Three
32) Traveler
33) Petr#
34) The least round way
35) Checkout assistant
36) Looking for Order
37) Petya and spiders
38) Games with rectangles
39) Pawn
40) Comb
41) Big Maximum sum
42) Knapsack for All Segments
43) Dot