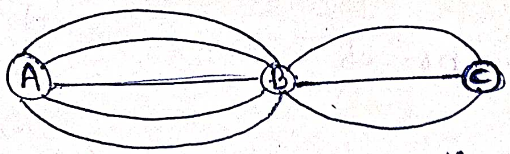


x ways y ways.

A to C = पहले B पर पहुँच जाओ और फिर C पर
= x * y

N = x*1 + y*1 because x से n-1 और y से n-2 तक जाने के लिए केवल एक ही ways है।

Another way

For N = 3
- 1 1 1
- 1 2
- 2 1

N = 2
1 1
2

N = 4

$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & \\ 2 & 1 & 1 & \end{bmatrix}$ 3 के लिए

$\begin{bmatrix} 1 & 1 & 2 \\ 2 & & 2 \end{bmatrix}$ 2 के लिए

So for N = 4 ways = N-1 + N-2 = 3 के लिए ways + 2 के लिए...

N=4 के लिए ways = 5 = (3 के 3 + 2 के 2) = 5

~~Hera~~

N = 3
1 1 1
1 2
2 1

N = 4
1 1 1 1
1 2 1
2 1 1
1 2 1
2 2

N = 5
1 1 1 2
1 2 2
2 1 2
] 3 वाले

1 1 1 1 1
1 2 1 1
2 1 1 1
1 2 1 1
2 2 1
] 4 वाले

total = 8

Iterative way

int dp[1000001] (Array represents no. of ways to reach nth stair)

int mod = 1e9 + 7;

void init()
{
    dp[1] = 1;
    dp[2] = 2;
    for(int i=3; i<=100000; i++)
        dp[i] = (dp[i-1] + dp[i-2]) % mod;
}

O(n) for preprocessing

0	1	2	3	4	5	6	7	8	9	10	11	12	14	15	16	17	18	19	20
0	1	2	3	5	8	13	21	34	55	89									

dp[3] = 1+2 = 3
dp[4] = 2+3 = 5
dp[5] = 5+3 = 8
dp[6] = 5+8 = 13
dp[7] = 13+8 = 21
dp[8] = 21+13 = 34
dp[9] = 34+21 = 55
dp[10] = 55+34 = 89

i=3 dp[3] = 1+2 = 3

i=5 dp[5] = dp[4] + dp[3] =

dp[5] = dp[i-2] + dp[i-3]

= 2x3 + 2x2 - 2 = 8

dp[7] = 2 x dp[5] + dp[4]

= 2x8 + 2x

n=1
n=2
n=3
4
5

int init(int n)
{
    if(n==1 || n==2 || n==3)
        return n;
    else return (n-1) + (n-2);
    return init(n-1) + init(n-2);
}

3+2

Staircase Problem Recursive Approach

```
int Stairs(int N)
{
    if (N==1 || N==2)
        return N;
    else
        return (Stairs(N-1) + Stairs(N-2));
}
```

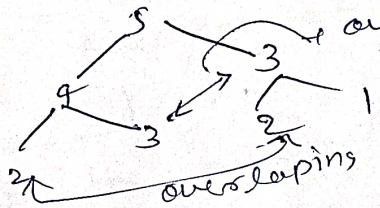
$O(2^n)$ This is not good.

jumping 1 jumping 2

we will make better this code by using back tracking and Dynamic programming.

Property of DP →

① overalping property.



② optimal substructure property.

Divide a big problem into small problems and using the Result of these subproblems calculate the result of big problems.

DP Solution →

```
int dp[1000001]
int Stairs(int N)
{
    for(int i=0; i<n; i++)
        dp[i] = -1;
```

we will do this in main bu

```
    if (n<=2)
        return N;
    if (dp[N] != -1)
        return dp[N];
    dp[N] = Stairs[N-1] + Stairs[N-2];
    return dp[N];
```

DP solution

```
int dp[1000001];
int Stairs(int N)
{
    if(N<=2)
        return N;

    if(dp[N]!=-1)
        return dp[N];

    dp[N] = Stairs[N-1] + Stairs(N-2);

    return dp[N];
}

int main()
{
    ~~for~~ maxN = 100000;
    int t, N;
    cin>>t>>
    for(int i=0; i<n; i++)
        dp[i] = -1;

    while(t--){
        cin>>N;
        cout<< Stairs[N] << endl;
    }
}
```

This is a Dynamic problem solution.

2 × N Tiling Problem

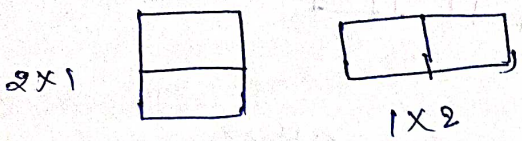
grid

using DP

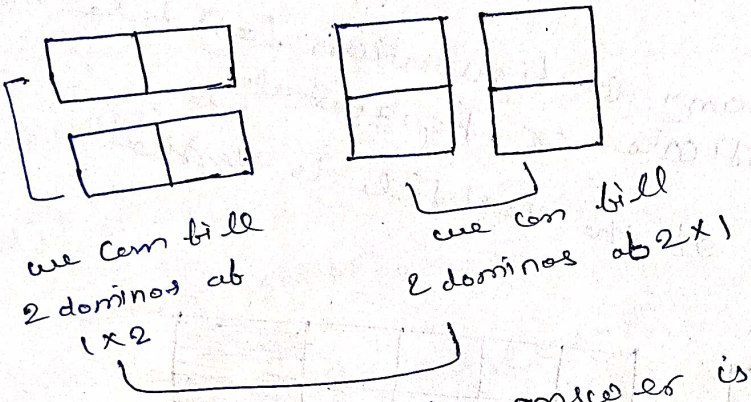
Q-1 Given a grid of 2 × N, find total number of ways to fill the grid using dominos of size 1×2 or 2×1

2×1 n=1

2×1



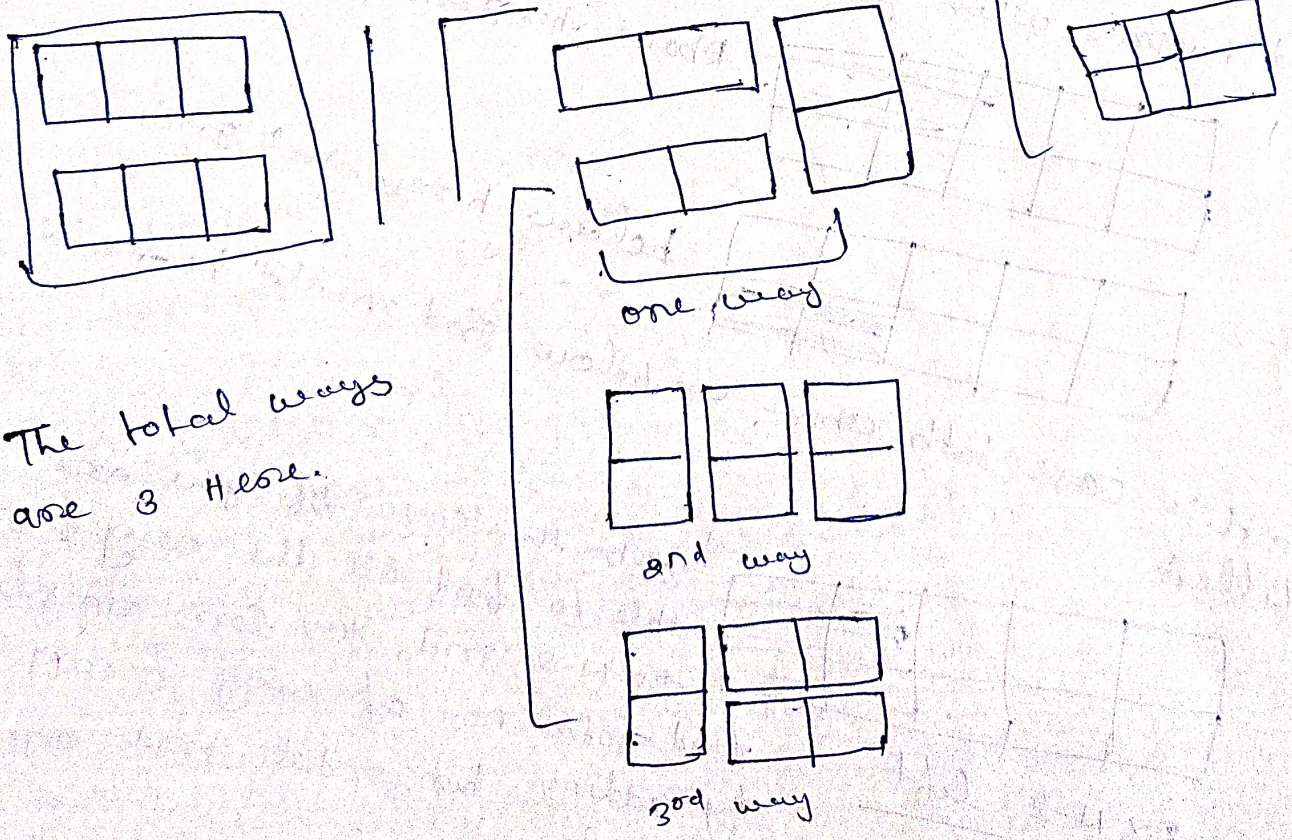
For N = 2 (2×2)



So for N = 2 the answer is 2.

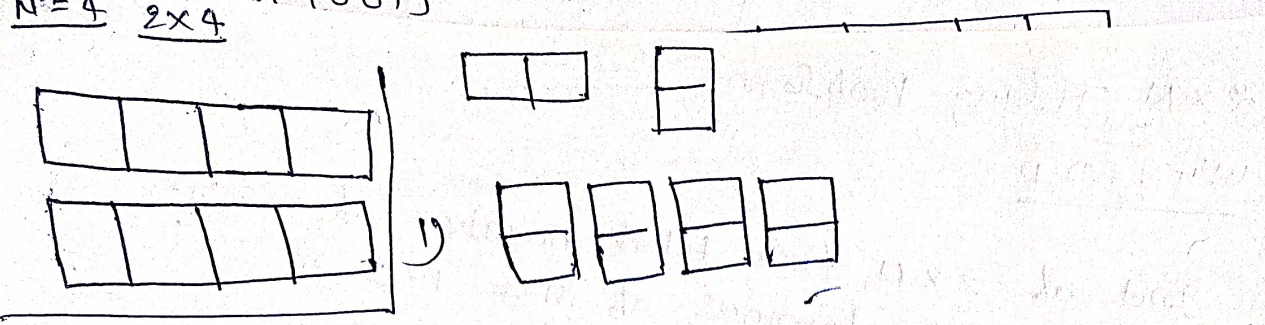
So

for N = 3 (2×3)



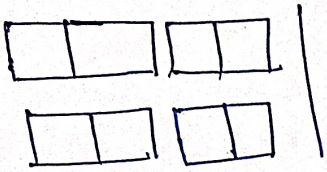
The total ways are 3 Here.

N=4 2×4

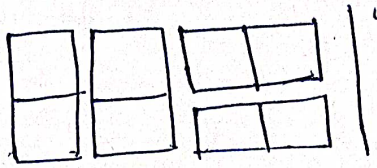


1)

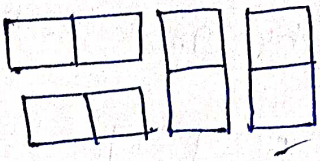
2)



3)



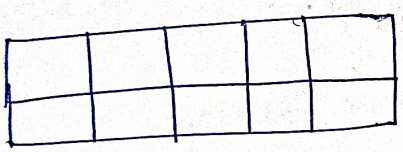
4)



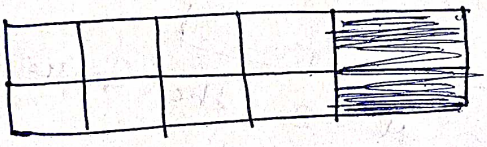
5)

Observation1 → In any configuration last tile is either vertical or horizontal.

ob-2 what happens if the last tile is vertical.



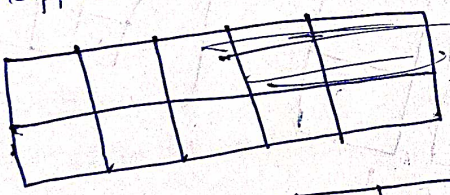
2×M grid



2×N grid left 2×N-1

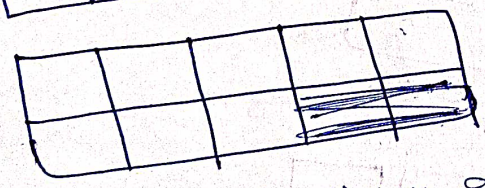
ob-3 what happens when you fill the last grid as horizontal grid.

ob3.1



Above horizontal grid

ob3-2



below horizontal grid.

In this case both above or below grid would be ~~filled~~ filled.

2×M-2 left

In this case if you are able to find all the way for 2×N-2 grid, you can simply find total no. of way for 2×N by adding horizontal grid one case.

① DP Problem solution

gb

# of Configuration of N-1 = x

N-2 = y

of N = x+y;

```
① int dp[1000001];
int ways(int N)
{
    if(N <= 2)
        return N;

    if(dp[N] != -1)
        return dp[N];

    dp[N] = ways(N-1) + ways(N-2);

    return dp[N];
}

int main()
{
    int t, N;
    int maxN = 1000000;
    for(int i=0; i<maxN; i++)
        dp[i] = -1;

    while(t--) {
        cin>>N;
        cout<<ways(N)<<endl;
    }
}.
```

Subset And BitMasking Problem → DP

Q-1 You are given an Array of size N; Print all 2^{rN} subsets;

Note → If there are n elements in a set then there will be 2^n subsets. or Cardinality of subset is 2^n

Set S = {A, B, C}

{ {}, {A}, {B}, {C}, {A, B}, {B, C}, {A, C}, {A, B, C} }
1 2 3 4 5 6 7 8

$2^3 = 8$

Set = {A, B, C}

{ }	0 0 0	0
{A}	0 0 1	1
{B}	0 1 0	2
{C}	1 0 0	3
{A, B}	0 1 1	4
{A, C}	1 0 1	5
{B, C}	1 1 0	6
{A, B, C}	1 1 1	7 (2^7-1)

Here what we can conclude is that for { }, bit configuration 0 0 0 means no element in the subset and no bit in configuration

for {A}, {B}, {C}, bit configuration (001, 010, 100) there is only one element on each subset and only one bit (1) in each configuration.

for {A, B}, {B, C}, {A, C}, we can conclude like that (011, 101, 110)

{A, B, C}, (111)

```
int main()
{
    int arr[] = {'A', 'B', 'C', 'D'};
    int t; n; cin>>t;
    while(t--)
    {
        cin>>n;
        int tot = 1<<n;
        for(int mask=0; mask<tot; mask++)
        {
            for(int i=0; i<n; i++)
            {
                int f = 1<<i;
                if((mask & f) != 0)
                    cout<<arr[i]<<" ";
            }
            cout<<endl;
        }
    }
}
```

n=2
m=2

```
for(i=0; i<=3; i--){
    for(int j=0; j<M; i++){
        f = 1<<i
```

0 0 0 0 0 0 0 0

1 [illegible]
i=0
(0,

arr={A, B, C, D}

n=3

tot = 1<<3 = 2^3 = 8

mask = 0
→ i=0
if 1<<0 = 1

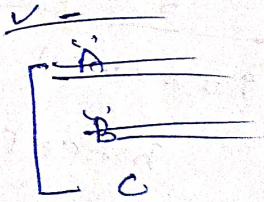
n=3
tot = 1<<3 = 8

mask=0 | mask=2
mask=1 | 1<<0=1
f=1<<1=2 | 1<<1=2
mask=3

0 0 0 0 0 0 0 1

mask=1
i=0
f = 1<<0=1
i=1
f=1<<1=2
i=2 =1<<2=4

mask=2 i=0 f=1



A
B

0 0 0 0 0 0 0 0 1

7	6	5	4	3	2	1	0
128	64	32	16	8	4	2	1
0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0

Subset sum

Constraints

$1 \le T \le 10$

$1 \le N \le 15$

$-10^6 \le A[i] \le 10^6$ for $0 \le i \le N$

we have to tell whether there exist any number of subsets such that their element sum would be equal to a number K.

Input

3 (test cases)

5 (element of array)

3 2 0 7 -1 (Array)

8 (The given number K)

1st test case → yes

T.C = $O(2^n * n)$

3

-1 3 3] 2nd → No

4

3

4 -5 1] 3rd → yes

5

```
int main()
{
    int ar[15];          // checking sum
    int n, t, sum, P;
    cin>>t;
    while(t--)
    {   cin>>n;
        for(int i=0; i<n; i++)
            cin>>ar[i];
```

cin>>P इस num. के बराबर sum we have to find and check.

```
    int flag = false;
    int tot = 1<<n;
```

```
for(int mask=0; mask<tot; mask++)
{   sum=0;
    for(int i=0; i<n; i++)
    {
        int f = 1<<i
        if((mask & i)!=0)
        {   sum+=ar[i];
        }
    }
}
if(sum==P)
    cout<<"YES"<<endl;
else
    cout<<"NO"<<endl;
```

```
if(sum==P)
{   flag=true;
    break;
}
```

(इस mask वाले loop से बाहर आ जाएंगे)

```
if(flag==true)
    cout<<"YES"<<endl;
else
    cout<<"NO"<<endl;
```

} out from while loop

Note → In this case the T.C = ($2^n \times n$) → for subset → for n.

But we can reduce the time complexity of this Code using Dynamic Programing

}

ar = 8 +2 -3 4 }

P = 9

8 -3 4 8-3+4 = 9

+9 yes

8
8 2
8 2 -3
8 2 -4

Chef Got Recipies (Code Chef) | Bit Masking

we अलग-2 dishes है (मतलब strings है) दो dishes को मिलाकर (मतलब two strings को concatenate करके) एक तीसरी dish इस प्रकार बनाती है जो सभी vowels को contains करे at least one time.

Approach→

we will represent each bit with 5 bit number where.

0th bit represent presence of 'a'

1st — 'e'

'i'

'o'

4th — 'u'

10101 में index right to left चलती है मतलब

101011 में index right to left चलती है

8 7 6 5 4 3 2 1 0
0 1 1 0 1 0 1 1 1

a e i o u

	u	o	i	e	a
a a o o a o a o o a	0	1	0	0	1
u i i e i e i i e i e u u u	1	0	1	1	0
a e i o o o e e i i a i e i	0	1	1	1	1

Understanding the problem

Example → Input = A B C ($2^3 = 8$)
0 1st 2nd

Here we are putting frequencies

	freq
000	
001	3
010	1
011	3
100	1
101	2
110	3
111	1

ABC OR 111

string	int mask	
	mask = 0 (0000) →	
AB	mask = 3 (0011) →	AB
AC	mask = 5 (0101)	AC
BC	mask = 6 (0110)	BC
BC	mask = 6 (0110)	BC
ABC	mask = 7 (0111)	ABC
AC	mask = 5 (0101)	AC
A	mask = 1 (0001)	A
C	mask = (0100)	C
A	mask = 1 (0001)	A
A	mask = 1 (0001)	A
B	mask = 2 (0010)	B
BC	mask = (0110)	BC

See: AB.

AB, AC, BC, BC, ABC, AC, A, C, A, B, BC

See here pairs BC are 3
and A ar 2 so the total number of ABC would be = 2×3 = 6

we will take OR of A and BC and multiply it by this frequencies (2×3) = 6. That will give us the total number of pairs of tasty dishes.

If we take 5 element the $2^5 = 32$ we will have to take the array of size 32. And we will put the frequency of each dishe.

- If (arr[i] OR arr[j]) == 31 then tasty dish.

So the number of tasty dishes are = frequency of arr[i] * fre arr[j]

```
int main()
{
    long long int F[32]; (for five set bit = 2^5 = 32)
    int t, n,;
    string dish(or st);
    cin>>t;
    while(t--)
    {
        cin >> n; → no. of dishes
        for(int i=0; i<32; i++) F[i]=0;
        for (int i=1; i<=n; i++)
        {
            cin>>st;
            int mask=0;
            for (char ch; st)
            {
                if (ch=='a') mask = mask|(1<<0);
                if(ch=='e') mask = mask|(1<<1);
                if(ch=='i') mask = mask|(1<<2);
                if(ch=='o') mask = mask|(1<<3);
                if ch=='u') mask = mask|(1<<4);
            }
            F[mask]++;
            mask++;
```

```
    F[mask]++;
}

long long int res = 0;
for(int i=0; i<32; i++)
{
    for(int j=i+1; j<32; j++)
    {
        if((i|j)==31)
        {
            res = res + F[i]*F[j];
        }
    }
}
for(ABC type dishes)
~~res = res + (F[3]*F[3]-1)~~
res = res + (F[3]*(F[3]-1))/2

cout<<res<<endl;
}
```

$\frac{3(3-1)}{2} = \frac{3\times2}{2} = 3$

$\frac{n(n-1)}{2}$

If total num of dish of type ABC the total number of dipo tasty dishes $= \frac{n(n-1)}{2}$

ABC 1 (1,2), (1,3)

ABC 2 (2,3)

ABC 3 $\frac{3(3-1)}{2} = \frac{3\times2}{2} = 3$

Coine change

Q-1 Given some coins of different denominations and a sum, Comput the fewest number of coins that you need to make up that ammount or tell if it is not possible.

Ex-1

Coins 1 2 5

Sum 11

output → 3 Coins (5+5+1)

Ex-2.

Coins : 2

Sum - 3

output -1 (no such coins arrangement)

Approach →

Let's take an array of size sum+1

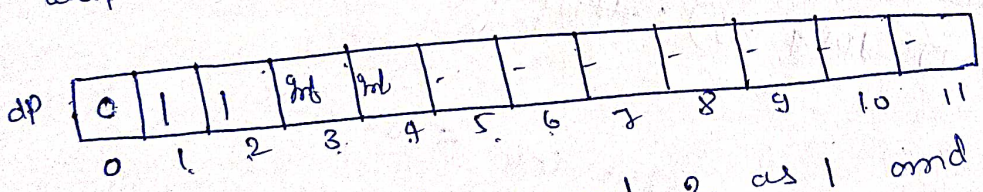
dp[sum+1]

where dp[i] = minimum coins to make total sum of i;

Ex→ Coinse: 1 2 5

Sum : 11

output : 3 (5+5+1)



Initialy we will put index 1 and 2 as 1 and rest of the index as inf

Now suppose there are K coins that can make sum as 3

$3 = C1 + C2 + C3 \cdots CK$

Since 5 > 3 so at the last coin 5 can not be there so, the possible cases

$3 = C1 + C2 + C3 - +1$

$3 = C1 + C2 + C3 - - +2$

$3 = C1 + C2 + C3 + \dots + 1$

$3 = C1 + C2 + C3 + \dots + 2$

Since 1 is for one coins so the remaining value $3-1 = 2$ will be calculated by $C1 + C2 + C3 \dots = 2$

Like wise $C1 + C2 + C3 + \dots + 2 = 3$

$C1 + C2 + C3 \dots = 1$

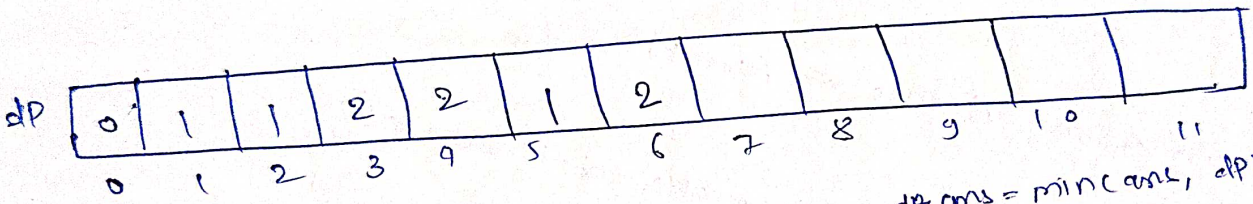
So to make this we need only one coin

$C1 + C2 + C3 \dots$

either 1 or 2 coins

So $\min(1, 1) = 1$

so to calculate the sum as 3 the total number of coins requires $= 1+1 = 2$ (1 we will add in each result)



$dP[3] = dP[3-1] + dP[3-2]$

$= \min(dP[3-1], dP[3-2]) = 1$

dp ans = min(ans, dp[i-C[j]])

= min(inf, dp[1-1] = 0

= min ans = min(0, dp[1-2])

Sum at 3 $dP[3] = 1+1 = 2$

for value 4 we can take coin 1 and 2 (but not 5)

$res = \min(dP[4-1], dP[4-2]) = dP[\min(2, 1) = 1$

$dP[4] = res + 1 = 2$

Like wise we can take here the coin 5 as well

$dP[5] = \min(dP[5-1], dP[5-2], dP[5-5])$

$res = \min(2, 2, 0) = 0$

$dP[5] = res + 1 = 1$

6 $res = dp \max(dP[6-1], dP[6-5], dP[6-2])$

$= \min(1, 1, 2) = 1$

$dP[6] = 1+1 = 2$

coins[0 0 1]

If the result does not exist. the.

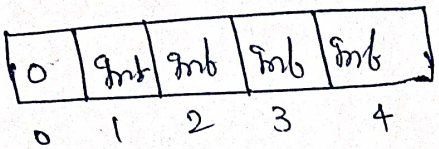
ex-1

Coins : 2

Sum : 3

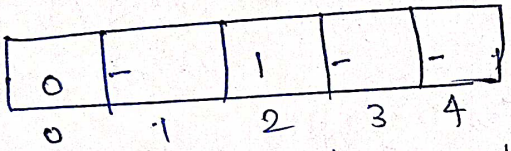
output -1

dp



because Here no coin has the value as 1 so we will not put 1 as 1. but we has one coin of 2 so we will put 1 at 2.

so dP



for 1 we can not find ans.

dP[1] = inf.

~~dP[2]~~ = min(dP[2-2], inf) = min(0, inf) = 1

dP[2] = res + 1 = 1

dP[3]

res = inf.

res = min(dP[3-2], inf) = min(dP[1], inf) = min(inf, inf) = inf.

∵ res > i(3) or res > sum or res ≠ sum.

so ans = -1

Problem → 322. Coin change on leet code.

```
int CoinChange(vector<int> &Coins, int amount)
{
    int CoinSize = Coins.size();
    int dP[amount+1];
    dP[0] = 0;
    int INF = 1000000000;
    for(int i=1; i<=amount; i++)
        dP[i] = INF;

    for(int i=1; i<=amount; i++)
    {
        int ans = INF;
        for(int j=0; j<CoinSize; j++)
            ans = min(ans, dP[i-Coins[j]]);

        if(ans==INF)
            dP[i] = INF;

        else
            dP[i] = ans+1;

    }

    if(dP[amount] == INF)
        return -1;

    return dP[amount];
}
};
```

Subset Sum →

Q-1 Given an array of integers and a number M, find whether there exists a subset whose sum is equal to M.

As A[] = {2, 5, 3, 9}, M = 12

$2^4 = 16$

A[] = {2, 5, 3}, M = 9 $2^3 = 8$ subsets

output = false

There are two different approaches using DP for this problem

1) Iterative

N = no. of element
M = sum.

1) T.C → O(N * M)

S.C → O(N * M)

2) T.C → O(N * M)

S.C → O(M)

(1) we will use a 2D array DP[N+1][M+1] where

DP[i][j] = true if you can make sum j using first i elements

{2} false otherwise.

A[] = {2, 5, 3, 4}

current {} we will start empty set (Can I make the some zero Ans yes so in ar[0][0] = 1

bool Array.

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	1	0	0	0	0	0	0	0	0	0	0
(2,5)	2												
(2,5,3)	3												
(2,5,3,9)	4												
		0	1	2	3	4	5	6	7	8	9	10	11	12

with empty set we can not create sum as 1, 2, 3 - - -
we can only make sum as zero only.
let element ab A[] = {2, 5, 3, 4} using first element 2
we can not make ~~sum~~ sum as 0 or 1 so we will assign
them 0, and 1 as false (means 0)

int dp[1000][1000]; Subset Sum →

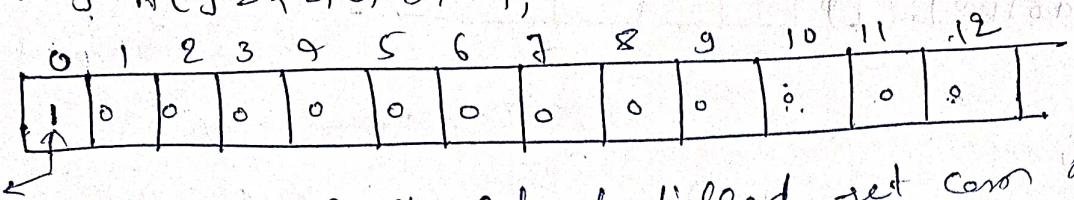
int main()
{
  int arr[100]; Array to store element ab set.
  int n, m; (no. ab element and sum)
  cin >> n >> m;
  for (int i=1; i<=n; i++)
    cin >> ar[i]; T.C O(n.m)

  int dp[n+1][m+1];
  dp[0][0] = 1;
  for (int i=1; i<=m; i++)
    dp[0][i] = 0;

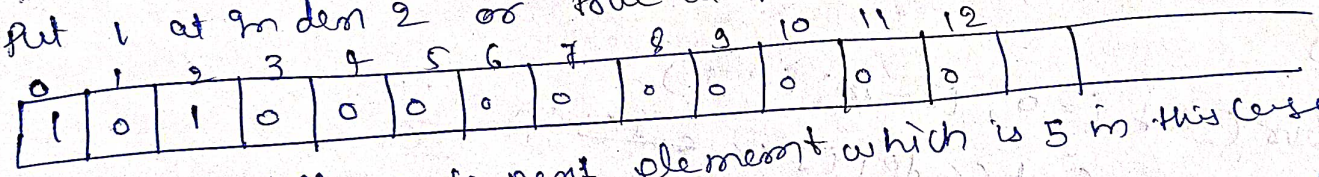
  for (int i=1; i<=n; i++)
  {
    for (int j=1; j<=m; j++)
    {
      if (j < ar[i])
        dp[i][j] = dp[i][j];
      else
      {
        int need = j - ar[i];
        if (dp[i-1][j]==1 || dp[i-1][need]==1)
          dp[0][j] = true;
        else dp[i][j] = false;
      }
    }
  }
  if int ans = dp[n][m];
  if (ans == true)
    cout << "YES"; << endl;
  else cout << "NO"; << endl;

Subset with Linear Search →

Array A[] = {2, 5, 3, 4}, M = 12, current set = {}



because every empty set of filled set can make sum as 0.
current element = 2 we will start from R→L and check
for 12 we will check if there is any element which has
the sum as 12-2 = 10 since index 10 has value as zero.
so we will put zero in 12
now we will come back to 11 and we will check if there
exist any index whose sum is (11-2=9) since index 9 is zero
so we will put 11 as 0.
Now we will go at 10 and we will check whether there exist
any number (i) whose sum is 10-2 = 8 since 8 is zero so will
come on 9 and check for 9-2 = 7 ∴ because 7 is also
0 . . like wise we will come on 2 and we will
check whether there exist any element whose sum is
2-2 = 0 because index 0 has value 1 (or true) so we will
put 1 at index 2 or true at index 2.

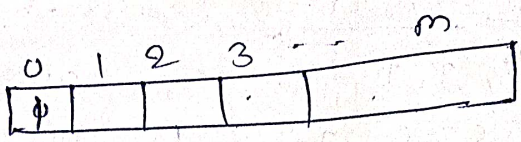


Now we will go to next element which is 5 in this case
again we will run the loop from 12 to 0
for 12 we will check whether there exist any number
whose sum is j - arr[i] = 12 - 5 = 7 since index 7 has 0 so
we will put 0 at 12 or we will continue and come on 11

11 - 5 = 6 6 has 0
10 - 5 = 5 5 has 0
9 - 5 = 4 has 0
8 - 5 = 3 0
7 - 5 = 2 1 since 2 has sum as 2 so we will put true
at index 5 because we can get sum as 7 by 2 and 5

The Time Complexity for

```
int ar[10000]
int main()
{
    int dp[1000]; ar[10];
    int n, m;
    cin >> n >> m;
    for(int i=0; i<m; i++)
        dp[i] = 0;

    dp[0] = 1;
    for(int i=1; i<=n; i++)
    {
        int current_ele = ar[i];
        for(int j=m; j>=current_ele; j--)
        {
            if(dp[j] || dp[j-current_ele] == 0)
                continue;
            else
                dp[j] = 1;
        }
    }
    for(int i=0; i<=m; i++)
        cout << dp[i] << endl;
}
```



2,

i=3 j=5

T.C = O(n x m)

S.C = O(n) ≈ O(1) space complexity has been reduced.

```
int main()
{
    int dp[100];  arr[100]
    int n, m;
    cin >> n >> m;
    for(int i=1; i<=m; i++)
            dp[i] = 0;

    dp[0] = 1;
    for (int i=1; i<=n; i++)
    {
        int curr_ele = arr dp[i];
        for (int j=m; j>=curr_ele; j--)
        {
            if(dp[j]!=0 || dp[j-
            if (pre[j] || pre[j-curr_ele] == 0)
                continue;

            else
                pre[j] = curr_ele;

        }
    }

    if (pre[m] == 0)
        cout<< -1 << endl;

    else
    {
        int curr = m;
        while (curr > 0)
        {
            cout<< pre[curr] << " ";
            curr -= pre[curr];
        }
    }
}
```

Min-Max Path on 2D Grid ⟹

Q-1 Given a grid of size N*M, find a path from cell (1, 1) to (N, M) which minimizes / maximizes the cost.

Note: You can go ~~write or~~ right or down only.

5	1	2	6
9	9	7	5
3	1	4	8

Q-2 Given a grid of N*M, find a path from any cell in first row to any cell in last row which minimizes / maximizes the cost.
You can go right or down, down-right or down-left.

① minimization →

5	1	2	6	inf
9	9	7	5	inf
3	1	4	8	0
inf	inf	inf	0	inf

what we will do here we will start from least cell (that is 8 in this case)

8	0
0	

now we will check which cell has minimum value of 8 (right or down)

and we will add min. value to 8 and put that in this cell = 0+8 = 8

27	22	21	19
25	22	19	13
16	13	12	8

Now we will check all cells before 8 (those are 5 and 4 in this case) for 5 we will check it's right and down cell and we will add min of those to 5 (

5	inf
8	

The min is 8 so we will put 8+5 = 13 at the index of 5 =

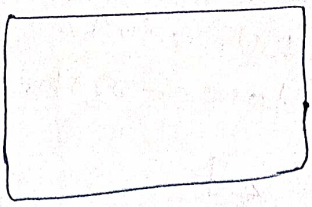
we we see for 4.

4	8
inf	

The min value is 8 so we will put 8+4 = 12 at the index of 4

Code for min path

```
int ar[50][50];
int dp[50][50];
int main()
{
    int n, m;
    cin >> n (no. of rows) >> m (no. of column);
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=m; j++)
            cin>>ar[i][j];
    }
```

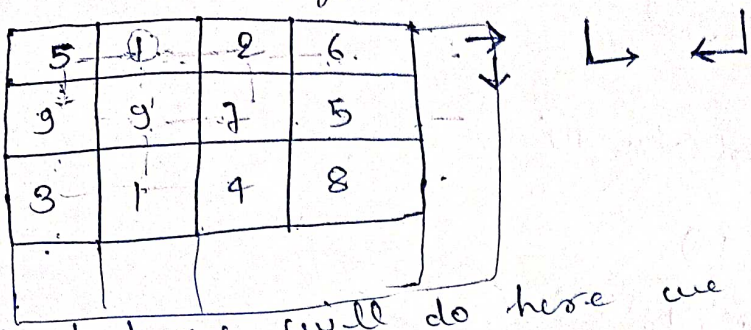



```
for(int i=1; i<=m; i++)
    dp[n+1][i] = INF;
```

```
for (int i=1; i<=m+1; i++)
    dp[n+1][i] = INF; (means 10^9)

for(int i=1; i<=n+1; i++)
    dp[n]
    dp[i][m+1] = INF;

dp[n+1][m] = dp[n][m+1] = 0;
for(int i=n; i>=1; i--)
{
    for(int j=m; j>=1; j--)
    {
        dp[i][j] = dp[i][j] + min(dp[i][j+1], dp[i+1][j]);
    }
}
for(int i=1; i<=n; i++)
{
    for (int j=1; j<=m; j++)
        cout<< dp[i][j] << " ";
    cout<< endl;
}
```

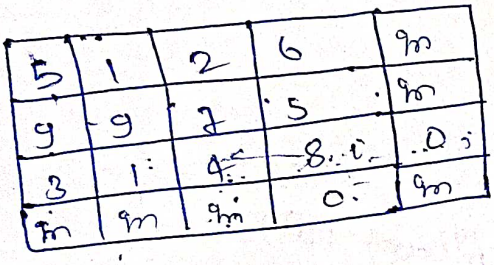
Grid and Path minimization →

Given a grid of size N×M, find a path from any cell in first row to any cell in last row which minimizes / maximizes the cost, You can go right or down, down-right or down-left.



Ans → what we will do here we will start from nth row and column.

See here we can start from any cell in the last row and from any cell in the first row.



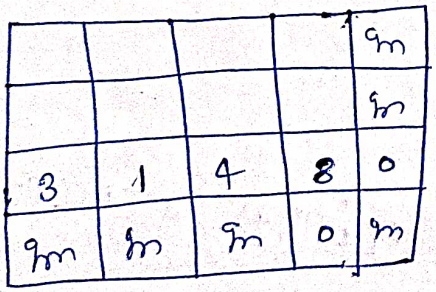
If 9 start from 8 so 9 can go to right, down and right-down

8 + min(right, down, right-down, down-left)

dp[i][j] + min(dp[i][j+1], dp[i+1][j], dp[i+1][j+1], dp[i+1][j-1])

8 + 0 = 8

Now we will go to 4,



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

= dp[i][j] + min(8, inf, 0, inf) = 4 + 0 = 4

Now we will go back to 1

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

= 1 + (min(4, inf, inf, inf)) = 1 + 4 = 5

In this problem we will not perform any operation on the last row because we have already achieved the goal.

So, we will start from second last row.

inf				
inf	10			
inf	3	1	4	8

for 9. we will see

m	9 → 9	
inf	3	1

$\min(9, 1, 3, inf) = 1$

$dp[i][j] = dp[i][j] + 1 = 9 + 1 = 10$

So we will start from $\boxed{dp[n-1][j=0]}$

```
for (i=n-1; i>=1 ; i--)
{
    for (int j=0; j<=m ; j++)
    {
        dp[i][j] = dp[i][j] + min(dp[i+1][j], dp[i][j+1],
                                  dp[i+1][j+1], dp[i+1][j-1]);
    }
}
```

⊙ Left Code

moves → down, down-left, down-right only.

m	3	1	2	inf
m	6	5	7	inf
m	8	9	2	inf

m	10	8	9	m
m	14	7	9	m
m	8	9	2	2

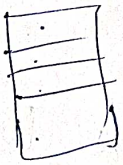
arr[]

we will see to 6. (m row-1)

~~dp[i][j] = arr[i][j] + min(arr[i+1][j-1], arr[i+1][j+1], arr[i+1][j-1])~~

dp[i][j] = arr[i][j] + min(dp[i+1][j], dp[i+1][j+1], dp[i+1][j-1])

```
int arr[50][50];
int dp[50][50];

int main()
{
    int n, m;
    cin >> n >> m;
    for(int i=1; i<=n; i++)
    {
        for(int j=1; j<=m; j++)
            cin >> arr[i][j];
    }

    for(int i=1;
    for(int i=1; i<=n+1; i++)
        dp[i][1] = inf;

    for(int i=1; i<=n+1; i++)
        dp[i][m+1] = INF;

    for(int i=n-1; i>=1; i--)
    {
        for(int j=1; j<=m; j++)
        {
            dp[i][j] = arr[i][j] + min(dp[i+1][j], dp[i+1][j-1], dp[i+1][j+1]);
            dp[i+1][j+1];
        }
    }

    for(int i=1; i<=n; i++)
    {
        for(int j=2; j<=m; j++)
            cout << dp[i][j]
```



```
for(int j=2; j<=m; j++)
    dp[n][i] = arr[n][i]
```

last वाली row के सभी elements में dp के last वाली row के सभी cells भरेंगे।

	1	2	3	4	5	6
1						
2						
3						
4						

3	1	2
6	5	7
8	9	2

m	10	8	9	m
m	14	7	9	m
m	8	9	2	m

dp for(int i=2; i<=m; i++)
dp[n][j] = ar[n][j-1];

dp[i][j] = ar[i][j-1] + min(dp[i+1][j], dp[i+1][j+1], dp[i+1][j-1]);

because we can start from any cell of first row. so we will start from the minimum cell. that is 8 in this case (8 means that, the cost is 8 to reach the goal means end cell)

```
int minFallingPathSum(Vector<Vector<int>> & ar) {
    int n = ar.size();
    int m = ar[0].size();
    int dp[n][m];
    for(int i=0; i<m; i++)
        dp[n-1][i] = ar[n-1][i];

    for(int i=n-2; i>=0; i--)
    {
        for(int j=0; j<m; j++)
        {
            dp[i][j] = ar[i][j];
            int mn = dp[i+1][j];
            if(j+1<m)
                mn = min(mn, dp[i+1][j+1]);
            if(j-1>=0)
                mn = min(mn, dp[i+1][j-1]);
            dp[i][j] += mn;
        }
    }
    int res = INT_MAX;
    for(int i=0; i<m; i++)
        res = min(res, dp[0][i]);
    return res;
}
```

Longest Palindromic Sub Array.

Input: 1 2 3 4 5 6 7 8
A B B C B B C A

what we will do is that we will create any 2D Array with size 8 x 8

row tells us the length of the sub Array and Column tells us the last index of sub array.

	0	1	2	3	4	5	6	7	8
0	1	1	1	1	1	1	1	1	1
1		1	1	1	1	1	1	1	1
2			0	1	0	0	1	0	0
3				0	0	1	0	0	0
4					0	0	0	1	0
5						0	1	0	0
6							0	0	0
7								0	0
8									0

① Array with 0 size is always be A palindromic Array.

And the next row length am we will start from Column 1 and row 1

One length array or string is also a palindromic string so we will put as 1 there.

AB for this first we will Chek whether the last and first element are same or not If they are same then 1 sub string may be palindrom (but not sure). After that we will chek for row no. N-2 and Column no. j-1 If this is 1 (It means the the sub Array is palindrom. because first and last element are same and sub array is also a palindrom the sub string will a palindrom.

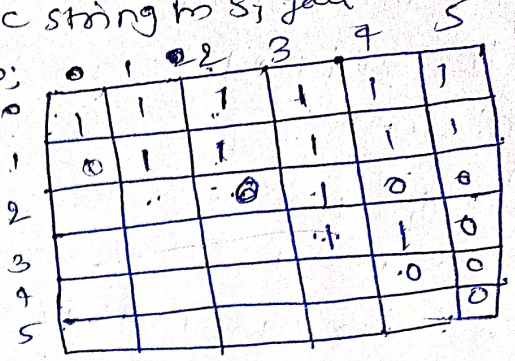
```
int dp[100][100];
int ar[100];
int main()
{
    int n, m;
    cin>>n;
    for(int i=1; i<=n; i++)
        cin>>ar[i];

    for(int i=0; i<=n; i++)
        dp[0][i] = dp[i][i] = 1;

    for( int i=2; i<=n; i++)
    {
        for(int j=i; j<=n; j++)
        {
            if(ar[j]==ar[j-i+1] && dp[i-2][j-1]==1)
                dp[i][j] = 1;
            else
                dp[i][j] = 0;
        }
    }

    for(int i=0; i<=n; i++)
    {
        for(int j=0; j<=n; j++)
            cout<< dp[i][j] << " ";
        cout<< endl;
    }
}
```

Leet Code

Q1 Given an string s, find the longest palindromic string in s, you may assume that the maximum length of s is 1000;

Input "babad"
output "bab" may be "aba"

Len 1 → 5



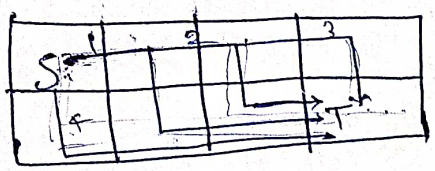
resE = 3
resL = 3
i = 3-3+1 = 1 3

```
String longestPalindrome(string s) {
    int N = s.size();
    int dp[N+1][N+1];
    int resE = 1, resL = 1;
    string res = "";

    if (N == 0)
        return res;

    for (int i=0; i<=n; i++)
        dp[0][i] = dp[i][i] = 1;

    for (int i=2; i<=n; i++)
    {
        for (int j=i; j<=n; j++)
        {
            if (s[j] == s[j-i+1] && s[i-2][j-1] == 1)
                dp[i][j] = 1, resE = j, resL = i;

            else
                dp[i][j] = 0;

3       3
3

    for (int i = resE-resL+1; i<=resE; i++)
        res.push_back(s[i-1]);

    return res;

3

3;
```

L14 : Uniaue ways to reach (M, M) from (1, 1)

Recursive + Iterative →

Given a grid of size N X M, find number of uniaue path from cell (1, 1) to (N, M) You can go right or down only.



Note → Find uniaue path from (1, 1) to (4, 4)

Building solution → Start

Start from N-1 row go till till first row, start filling values from 1 right most column to left most column.

Basic observation →

Since from (x, y) we can only go to (x+1, y) or (x, y+1)

Hence uniaue path from (x, y)

= uniaue path from (x+1, y) + uniaue path from (x, y+1);

20	10	4	1
10	6	3	1
4	3	2	1
1	1	1	1

we will fill all the cells of last row and last column by 1 because from these we can not go further this is the last step (means we can reach this pat in only one way).

for every cell

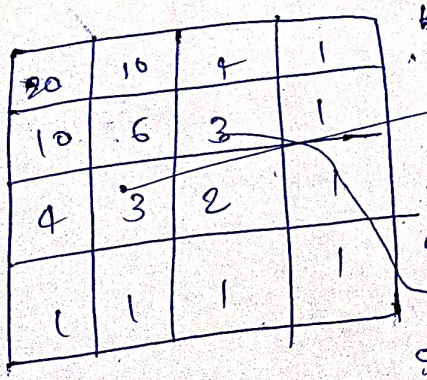
we will do dp[i][j] = right + down

= dp[i][j+1] + dp[i+1][j]

we will start from mm cell to 00 cell.

```
int dp[101][101];
int N, M;

int main()
{
    cin>>N>>M;
    for(int i=1; i<=M; i++) dp[N][i] = 1;
    for(int i=1; i<=N; i++) dp[i][M] = 1;

    for(i=M-1; i>=1; i--)
    {
        for(int j=M-1; j>=1; j--)
        {
            dp[i][j] = dp[i][j+1] + dp[i+1][j];
        }
    }

    for(int i=1; i<=N; i++)
    {
        for(int j=1; j<=M; j++)
        {
            cout<< dp[i][j] << " ";
        }
        cout<<endl;
    }
}
```

Recursive Approach



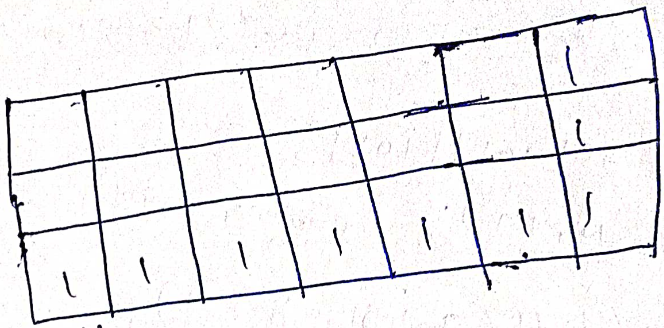
When we will go to calculate path for this 3 we will calculate value of 2 and 1 since we have calculated this value already so we will simply take this value from dp again when we will calculate the value of this 3 we will calculate value of 1 and 2 since we have calculated value of 2 so we will not calculate it again and we could simply get the value of 2. So these calculated value we will store in dp array.

we know for
ways from (x, y) = ways from (x+1, y) + ways from (x, y+1);

```
int dp[101][101];
int N, M);        starting point)
int fill(int x, int y)
{
    if (x>N || x<1 || y>M || y<1) return 0;
    if (x==N && y==M)
    {
        dp[x][y] = 1;
        return 1;
    }
    if (dp[x][y] != -1) return dp[x][y];
    dp[x][y] = fill(x+1, y) + fill(x, y+1);
    return dp[x][y];
}

int main()
{
    cin>>N>>M;
    memset(dp, -1, sizeof(dp));
    fill(1, 1);
    for(int i=1; i<=N; i++)
    {   for(int j=1; j<=M; j++)
            cout<<dp[i][j]<<" ";
        cout<<endl;
    }
}
```

LeetCode

moves → down or right

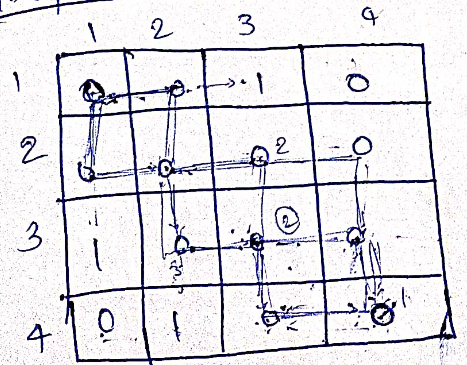


```
int uniquePaths(int m, int n){
    int dP[n+1][m+1];

    for(int i=1; i<=n; i++)  dP[n][i]=1;
    for(int i=1; i<=m; i++)  dP[i][m]=1;

    for(int i=N-1; i>=1; i--)
    {
        for(int j=m-1; j>=1; j--)
        {
            dP[i][j] = dP[i][j+1] + dP[i+1][j];
        }
    }
    return dP[1][1];
}
```

② moves down – right

An obstacle and empty space is marked as 1 and 0 respectively in the grid.

Preprocessing →



	1	2	3	4
1	10	5	0	1
2	5	5	3	1
3	0	2	2	1
4	0	0	1	1

nm

O(nm)

Unique paths with some obstacles (Leet code)

moves down, right

```
int uniquePathsWithObstacles(vector<vector<int>>& ar)
{
    int n = ar.size();
    int m = ar[0].size();
    long long int dP[n+1][m+1];
```

If means that if the last cell contains 1 (ob ar) that means obstacle we will bill that cell as zero (means xor ob 1) in that dP cell. In DP we will store ~~store~~ index from 1 to n+1, in ar it is 0 → n.

bro row

```
    for(int i=1; i<=n; i++)
    dP[n][m] = ar[n-1][m-1] ^ 1;
```

```
    for(int i=m-1; i>=1; i--)
    {
        if(ar[n-1][i-1]==1 || dP[n][i+1]==0)
                means obstacle
            dP[n][i] = 0;
        else
            dP[n][i] = 1;
    }
```

bro column

```
    for(int i=n-1; i>=1; i--)
    {
        if(ar[i-1][m-1]==1 || dP[i+1][m]==0)
                obstacle
            dP[i][m] = 0;
        else
            dP[i][m] = 1;
    }
```

```
    for(int i=n-1; i>=1; i--)
    {
        for(int j=m-1; j>=1; j--)
        {
            dP[i][j] = dP[i][j+1] + dP[i+1][j];
```

```
for(int i=n-1; i>=1; i--)
{
    for(int j=m-1; j>=1; j--)
    {
        if(arr[i-1][j-1] == 1)
            dp[i][j] = 0;
        else
            dp[i][j] = dp[i][j+1] + dp[i+1][j];
    }
}
return dp[1][1];
}
};
```

Rod Cutting Problem →

Q-1 Given a rod of length n and you want to cut up the rod and sell the pieces in a way that maximizes the total amount of money you get. A piece of length i is worth pi units of money.

Example

length i	1	2	3	4	5	6	7	8	9	10
Price Pi	1	5	8	9	10	17	17	20	24	30

For N = 3

1 + 1 + 1 = 3
1 + 2 = 1 + 5 = 6
2 + 1 = 5 + 1 = 6
3 = 8 ✓✓

N = 4

4 = 9
1 + 3 = 1 + 8 = 9
1 + 1 + 2 = 1 + 1 + 5 = 7
2 + 2 = 5 + 5 = 10

1 + 1 + 1 + 1 = 4
1 + 2 + 1 = 1 + 5 + 1 = 7
1 + 3 = 1 + 8 = 9
2 + 2 = 5 + 5 = 10
3 + 1 = 8 + 1 = 9

2 + 1 + 1 = 5 + 1 + 1 = 7

4 = 9

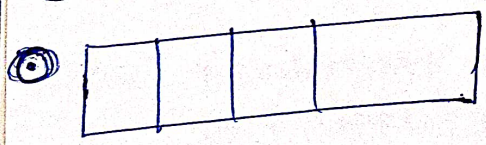
maximum profit = 10

Source

1) CLRS book

Problems to be discussed:

1) To find the maximum profit
2) To find and print the actual length of the cut part of the rod.



Yes =

(c) we will solve this problem, using

(1) Recursive (Top-Down) method

(2) Iterative (Bottom-up) method.)

Note → Since there are n-1 cut points, at each cut point you can decide whether to or not.

So total 2^{n-1} diff. combigurations are possible.

as n↑, no. of combiguration↑ so we can not use brute force approach here.

Bottom-up approach →

	1	2	3	4
P[ ]	0	1	5	8	9

P[i] : price of i length rod.

DP[ ]	0

DP[i]: maximum profit of length i rod.

base.

First step → Think about base case.

① for length n rod, we will sum n phase.

② In each phase, we will divide the rod in cut part and remaining part.

③ In ith phase: cut part length = i, remaining part length ≥ n-i.

```
for (int i=1; i<=n; i++)
{
    int profit = 0;
    for (int j=1; j<=i; j++)
    {
        profit = max (pro, P[j] + dp[i-j]);
        profit = max (profit, P[j] + dp[i-j]);
    }
    dp[i] = profit;
}
```

	0	1	2	3	4
P[ ]	0	1	5	8	9
dp[ ]	0	1	5	8	10

n=1

n=2

n=4

Phase	profit
i=1	1
i=2	2
i=2	5
i=3	6
i=3	6
i=3	8
i=4	9
i=4	10 →
i=4	9
i=4	9
i=4	9

Bottom-up approach →

	1	2	3	4
P[ ]	0	1	5	8	9

	0				
DP[ ]	0				

base

P[i] : price of i length rod.

DP[i]: maximum profit of length i rod.

First step → Think about base case.

1. for length n rod, we will run n phase.
2. In each phase, we will divide the rod in cut part and remaining part.
3. In ith phase: cut part length = i, remaining part length = n-i.

```
for(int i=1; i<=n; i++)
{
    int profit = 0;
    for(int j=1; j<=i; j++)
    {
        profit = max(profit, P[j] + dp[i-j]);
    }
    dp[i] = profit;
}
```

	0	1	2	3	4
P[ ]	0	1	5	8	9

	0	1	2	3	4
dp[ ]	0	1	5	8	10

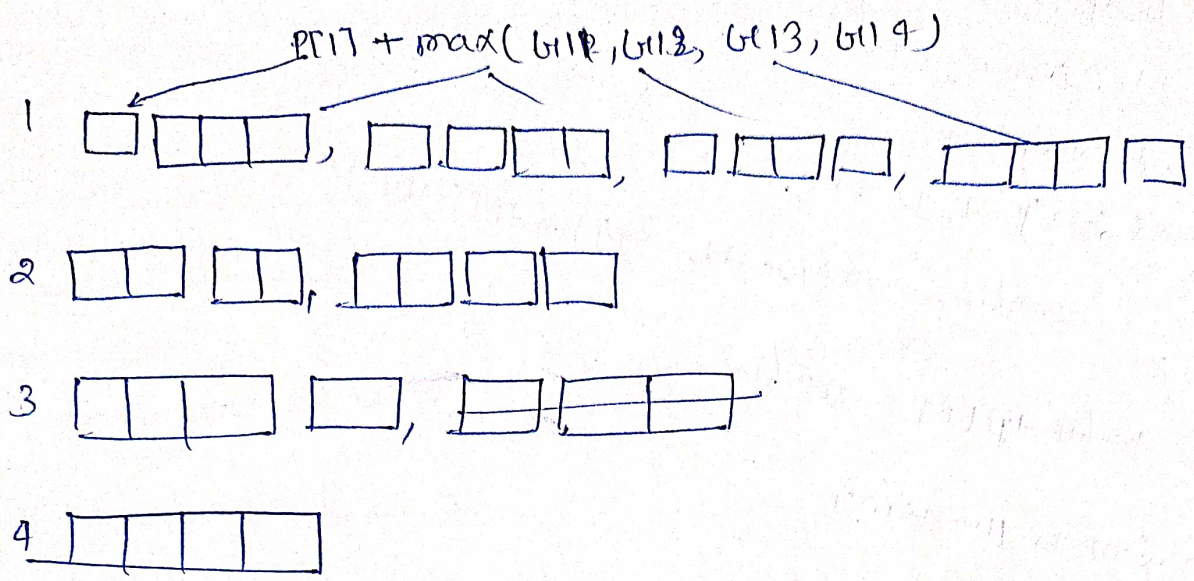
n=1

n=2

n=4

Phase	profit
i=1	1
i=2	2
i=2	5
i=3	6
i=3	6
i=3	8
i=4	9
i=4	10 →
i=4	9
i=4	9
i=4	9

Building the solution expression →



In first group it is common that first cut is of the size 1 so we will not include that. we will find remaining cut is 3 ($2^{3-1} = 2^2 = 4$ Configuration) so will find the maximum of all these 4 Configurations. we will find this by dP[3] = dP[n-1] = dP[4-1]

⊙ In the second group the first cut is of size 2.

P[2] + dP[4-2] = P[2] + dP[2]

In the third group

P[3] + dP[4-3] = dP[3] + dP[1]

In the 4th P[4] + dP[4-4] = dP[4] + dP[0]

It is already known that dP[0] will always be 0 so ~~for~~ for the n. first we will calculate dP[0] then dP[1] the dP[2] the dP[3] - - - dP[n-1] and in the end we will do **P[n] + dP[n-1]**

In this way we can find the maximum in O(1) time for n length of group. so T.C = O(N*1) = O(n)

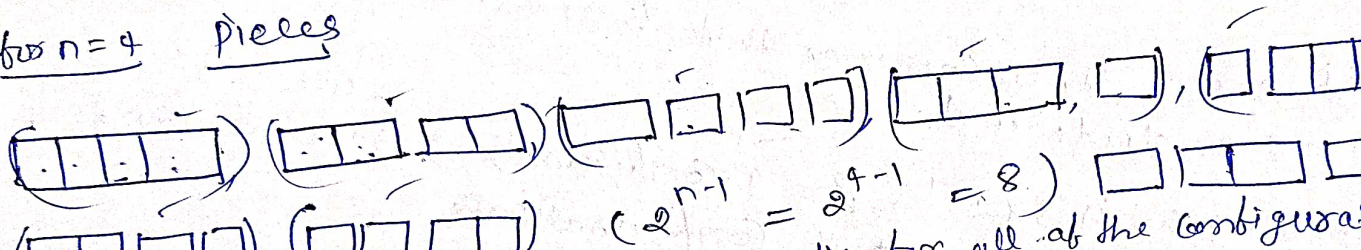
Building the solution expression →

How the loop

```
for (int i=1; i<=n; i++)
{
    int profit = 0;
    for (int j=1; j<=i; j++)
    {
        profit = max(profit, P[j] + dp[i-j]);
    }
    dp[i] = profit;
}
```

How this loop is working

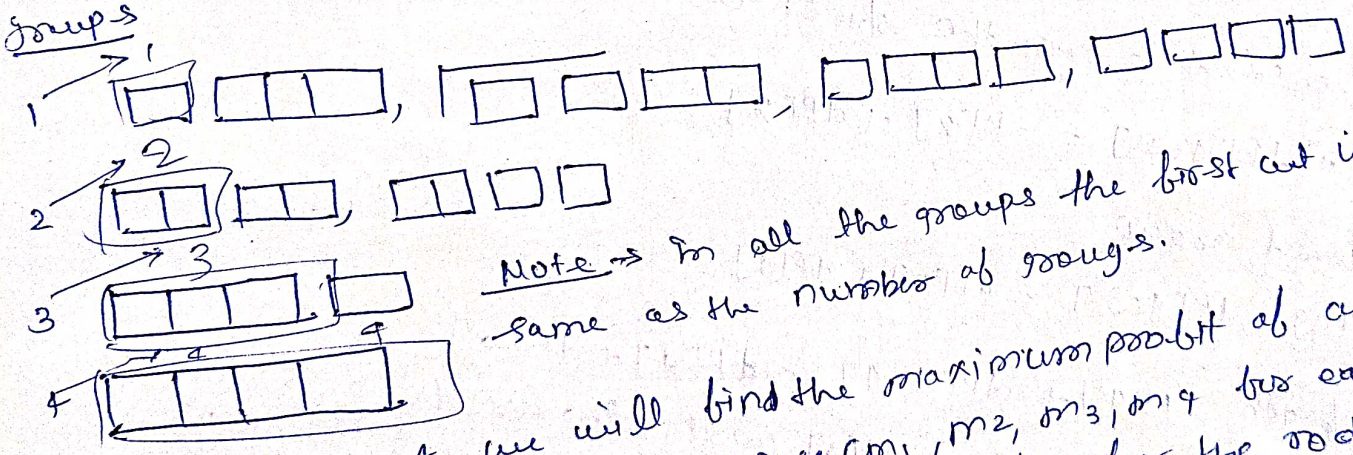
for n=4 pieces



$(2^{n-1} = 2^{4-1} = 8)$

so, we will check the maximum profit for all of the configurations.

groups



Note → In all the groups the first cut is same as the number of groups.

so, for the length 4 we will find the maximum profit of all the groups of length ~~and take the~~ suppose (m_1, m_2, m_3, m_4 for each groups respectively). so, the maximum profit for the rod of length 4 will be $\max(m_1, m_2, m_3, m_4)$

⊙ If I could find the maximum of each group in constant time then I would be able to find the solution for n groups in O(N×1) time Thus I can reduce the complexity

Kadane's Algorithm → for maximum subarray

Q-1 Given an integer array of size N, find the subarray with maximum sum.

As 8
-1 -4 [4 -2 0 1 4] -5

Ans 7 (4 -2 0 1 4)

DP Array
dP[i] = maximum subarray sum ending at position i

8
-1 -4 4 -2 0 1 4 -5

dP[4] = 2

-1 -4 4 -2 = -3
-4 4 -2 = -2
4 -2 = 2
-2 = -2

This will take $O(N^2)$

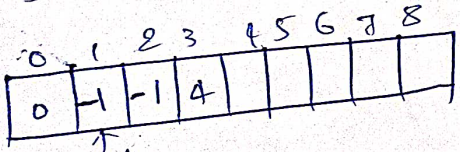
Q How to find max subarray sum ending at position i in O(1) time?

Hint 1: Can you utilize dP[i-1] somehow?

2: dP[i-1] is true, dP[i-1] is -ve.

dp →

ar = {-1, -4, 4, -2, 0, 1, 4, -5}



dP ar[i]

dP[4] we have to calculate

If we knows ~~or~~ zero length will contain maximum as 0

i length Contain maximum as the element itself

for index ending as 4

we will calculate dP[3] = dP[4-1]

for i = 3 we will Calculate dP[3-1] = dP[2]

for 2 we will Calculate dP[1] = dP[2-1]

for 1 we will Calculate dP[1-1] = dP[0]

dP[0] we already know

∴ dP[1] = ~~dP[i] + dP~~ dP[0] + ar[1] = 0 - 1 = -1

dP[2] = dP[1] + ~~dP~~ ar[2] = -1 + (-4) = -5

dP[2]
= max(dP[1], ar[2])
= max(-1, -4)
= -1

dP[3] = max(-1, 4)
= 4

dP[4]
= max(dP[3], ar[4])
= max(4, -2)
= 4

dP[i] = dP[i-1]
+ max(dP[i-1], ar[i])
= 0 + 0

{-1 -4 4 -2 0 1 4 -5} = ar

dP[4] = 2

Hint 1

-1 -4 +4 | -2 |
-4 4 | -2 |
4 | -2 |
| -2 |

-1-2 = -3
0-2 = -2
4-2 = 2 →
-2 = -2

ith position वाला element is coming in all of the sub arrays

| 0 | |

dP[1] =

Hint - 2

dP[i] = maximum subarray sum ending at position i

Hint 2 → dP[i-1] is +ve, or dP[i-1] is -ve.

&

-1 -4 4 -2 0 1 4 -5

-1 -4 4 2 2 3 7 2

If dP[i-1] is +ve we will add dP[i-1] in dP[i] and store that at the index i in dP[i]

If dP[i-1] is -ve we will simply ignore that.

ans 4 we will not add -4, and in +4 we will not add -1.

Code

```
int ar[100001];
int dP[100001];
int main()
{   int n;
    cin>>n;
    for(int i=1; i<=n; i++)
        cin>>ar[i];
    for(int i=1; i<=n; i++)
        dP[i] = max((dP[i-1], 0) + ar[i]
    cout<< * max_element(dP+1, dP+n+1);
                ↑ inbuilt function.
}
```

LNT - 1000000000

minimum no. of coins

CSES website →

Q-1 Given coins of n different values and an integer x, find the minimum numbers of coins to make total sum of x.

Ex 3 11

1	5	7

output 3 (5+5+1)

dp

0	1	2	3	4	5	6	7	8	9	10	11
0	1	2	0	0						0	0

N-1

1,1 (5-5) = 0

dp[1] = 1

dp[5-5] = 0 6-1

11 = 5 = c1 + c2 + c3 + ... + ck

dp[6-1] = 5 } 1

5 =

dp[6-5] = 1

N-1

dp[14-1] = dp[10]

(11) 11 = c1 + c2 + c3 + ... + ck (11)

1 = c1 + c2 + c3 + ... (1) 1-1

1 2 = c1 + c2 + c3 + ... + ck (2) 2-1

(3)

3 = (4)

4 =

5 =

for(int i = n; i >= 0; i--)

dp

0	1	2	3	4	5	6	7	8	9	10
0										

if x-ci sum can be made in y coins.

Then x sum can be made in y+1 coins.

dp[x] = min(dp[x-ci]) + 1

dp[1] = min(dp[0]) + 1 = 1

dp[2] = min(dp[1]) + 1 = 2

dp[3] = min(dp[2]) + 1 = 3

dp[4] = min(dp[3]

dp[5] = min(dp[4], dp[0]) + 1 = 1

minimum no. of coins

dp	0	1	2	3	4	1	2	3	2			
	0	1	2	3	4	5	6	7	8	9	10	11

ar = { 1, 5, 7 }

dp[4] = min(dp[4-1], dp[

we will run a loop on the coin array if the sum is less than coin we can not use that coin if the sum is greater than coin we can use that coin.

```
for (coin=1; coin<=no_of_coin; coin++)
{
    if (sum<coin)
    {
        dp[i]
for(sum=1; sum<=required_sum; sum++)
{
    result = inf.
    for (coin=1; coin<=no_of_coin; coin++)
    {
```

```
if (sum<coin)
{
    dp[sum] = min(dp[sum
}
if (sum > coin)
{
    result = min( for result, dp[sum-ar[coin]
```

④

```
if (sum >= coin)        2, 1, 0, 2
{
    result = min(result, dp[sum-ar[coin]);
}
dp[sum] = result+1;
}
}
```

2 ①
2 ②
2
re = 1

i=1

#define INF 1000000000 → minimum no. of Coins

int Coins[1001]

int dp[10000001];

dp	INF	INF	INF	INF	INF	INF

int main()
{

int n, x;

cin>>n>>x;

for(int i=1; i<=n; i++)

cin>> Coins[i];

for(int i=1; i<=x; i++)

dp[i] = INF;

for(int i=1; i<=x; i++)

{

for(int j=1; j<=n; j++)

{

if(i-Coins[j] >=0) or (if(Sum >= Coins[j])

dp[i] = min(dp[i], dp[i-Coins[j]]+1);

}

}

if(dp[x] == INF)

dp[x] = -1;

cout<< dp[x];

}

x=8

dp[8] = min(INF, dp[8-1]) +1 = min(INF, 1) +1 = 2

dp[9] = min(INF, 2) +1 = 3 (1 1 7)