Dynamic Programming

Dynamic Programming is Optimization of Recursion

→ If subtree is calculated, then store it and utilize it. (repetitive tree)

Solve Problems in 3 ways → Recursion
Memoization
Tabulation

1. Fibonacci-dp

Input	Output	Constraints
10	55	$0 <= n <= 45$

Steps:

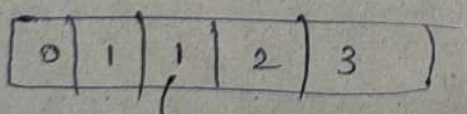
Recursion: Use the recursive function

Memoization: ① Take An Array of size n+1

② Store the resultant of recursive function in dp(n).

③ Before performing the Recursive calls, check whether dp(n) contains any value, if Yes return the value.

Tabulation:

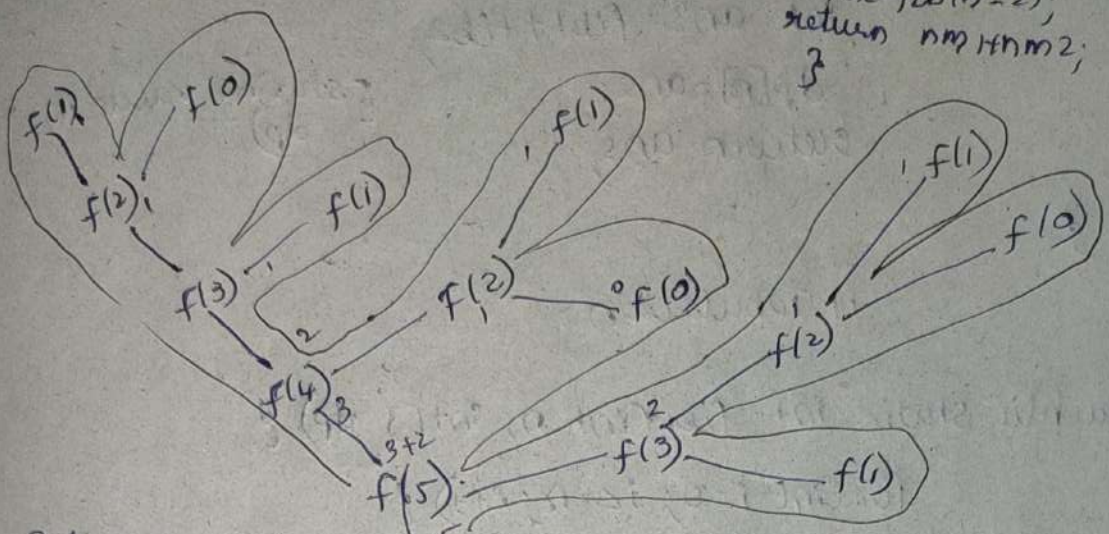


value is equal to sum of previous two indices of Array

At last return dp[n] → Gives the Fibonacci Number

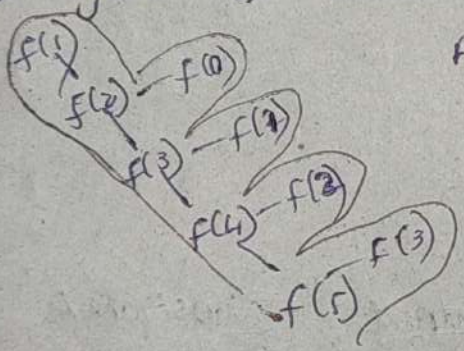
```
fib(int n){
    if(n==0||n==1) return n;
int nm1=fib(n-1);
int nm2=fib(n-2);
return nm1+nm2;
}
```

Dry Run:

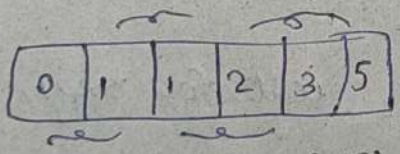


But we notice, that ~~f(1)~~, f(2), f(3) are calculated many times, Instead of calculating many times, we calculate once, and store it, we ~~calls~~ can use again the same.

After modifying.



Tabulation: n=5, size=6



Program: Memoitization

```
int[] dp=new int[n+1];
public static int fib(int n, int[] dp){
    if(n==0||n==1)
        return n;
    if(dp[n]!=0)
        return dp[n];
```

[checking whether subtree is calculated or not before calling Recursive function calls]

```
int fib1 = fib(n-1, dp);
int fib2 = fib(n-2, dp);
int ans = fib1 + fib2;
dp[n] = ans;                    [storing value in dp]
return ans;
}
```

// Tabulation

```
public static int fibi(int n, int[] dp) {
    for(int i=0; i<=n; i++) {
        if(i==0 || i==1)
            dp[i] = i;
        else
            dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

2. Climb Stairs

Sample Input
4

Sample Output
7

Constraints
$0 <= n <= 20$

Steps:

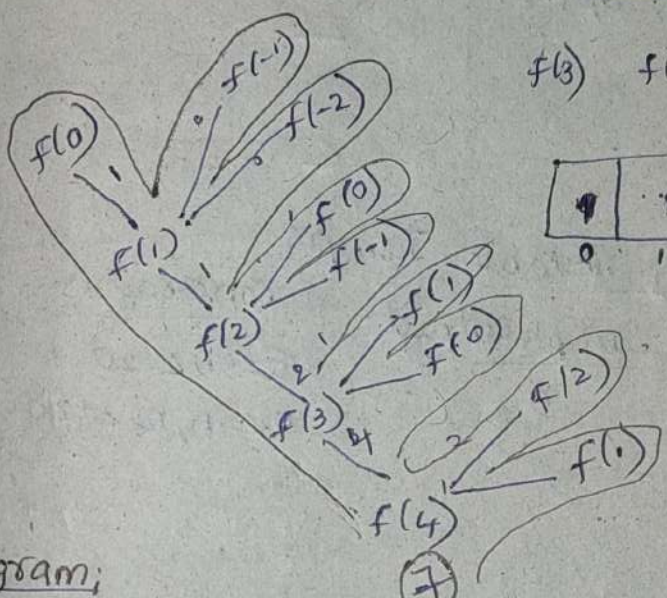
1. Make three Recurrive calls for step1, step2, step3
2. Create an dp array with n+1 size.
3. Store the result of recursive calls in dp at respective index.
4. Before calling Recursive function calls, check whether the dp array contains value or not, If it contains value Return the value.

Tabulation;

→ By understanding the recursive tree we may observe that, no of ways to reach step depends on (i-1)+(i-2)+(i-3) steps.

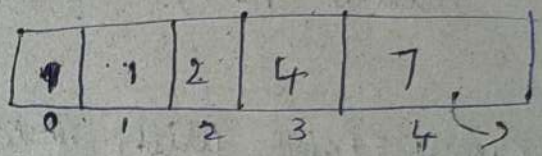
→ Return the value for n^{th} position of dp array which gives the possible no of ways to reach n^{th} step.

Dry Run;



Tabulation; For f(4) depends on sum of f(2) & f(1)—

f(3) f(4)=f(4-2)+f(4-3)+f(4-1)



Program;

```
int[] dp = new int[n+1];
public static int stair(int n, int[] dp) {
        if (n==0)
            return 1;
        if (n<0)
            return 0;
        if (dp[n]!=0)
            return dp[n];
        int x= stair(n-1, dp);
        int y = stair(n-2, dp);
        int z = stair(n-3, dp);
        int ans= x+y+z;
        dp[n]=ans;
        return ans;
}
```

```
// Tabulation
public static int stairTab(int n, int[] dp) {
    dp[0] = 1;
    for (int i = 1; i <= n; i++) {
        dp[i] = dp[i-1];
        if (i-2 >= 0)
            dp[i] += dp[i-2];
        if (i-3 >= 0)
            dp[i] += dp[i-3];
    }
    return dp[n];
}
```

3. Climb Stairs With Variable Jumps

Sample Input	Sample Output	Constraints
10 -> no of stairs 3 3 0 2 1 2 4 2 0 0	5	0 <= n <= 20 0 <= n1, n2 <= 20

Steps;

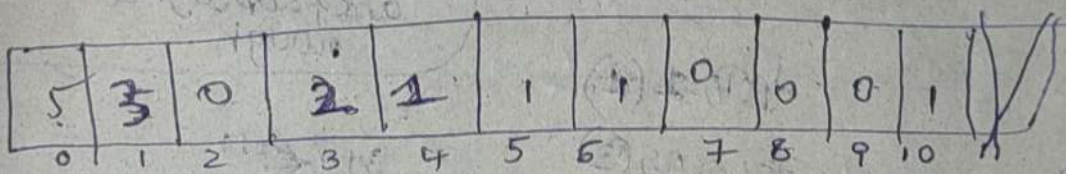
1. We can make 1 jump to <= stairs value.
2. We will add all the possible values to reach from source to destination.
3. If you reach the end of the stairs core return 1.
4. Use Dp Array for Memoization.

Tabulation;

1) Take an dp Array of greater than no of elements of array.
2) dp(i) → stores the → Total No of ways to reach from source to destination.
3) Apply the condition for boundary.
4) In Dp Array we traverse from right to left

Array:

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



→ dp[10] ⇒ Source to Destination - 1 jump

⇒ dp[9] → jumps=1 to <= arr[9] & ⇒ dp(9) = 0

→ dp[8] = jumps=1 to <= arr(8) & → dp(8) = 0

→ dp[7] = jumps: 1 to <= arr(7) , 7+1 < 11 ✓
ans+ = dp(8) = 0

jump2; 7+2 < 11 ✓
ans+ = d(9) = 0

d(7) = 0

→ dp[6] = jumps = 1 to <= arr(6) , 6+1 < 11 ✓
ans+ = dp(7) = 0

6+2 < 11 ✓
ans+ = dp(8) = 0

6+3 < 11 ✓
ans+ = dp(9) = 0

6+4 < 11 ✓ ans+ = dp(10) = 1

∴ dp[6] = 1

dp[5] = jumps=1 to <= arr[5]
(2 jumps)

5+1 <= 11
ans += dp[6] = 1

5+2 <= 11
ans += dp[7] = ...

dp[5] = 1

dp[4] = jumps=1 to <= arr[4]
(1 jump)

4+1 <= 11
ans += dp[5] = 1

4+2 <= 11
ans += dp[6] = 1+1 = 2

4+3 <= 11
ans += dp[7] = 0+0 = 2

4+4 <= 11
ans += dp[8] = 2
rough

dp[4] = ~~1~~ (1)

dp[3] = jumps=1 to <= arr[3]
(2 jumps)

3+1 <= 11
ans += dp[4] = ...

3+2 <= 11
ans += dp[5] = 0+1 = ~~1~~

dp[3] = ~~2~~

dp[2] = jumps=1 to <= arr[2]
(0 jumps)

dp[2] = 0

dp[1] = jumps=1 to <= arr[1]
(3 jumps)

1+1 <= 11
ans += dp[2] = 0

1+2 <= 11
ans += dp[3] = 2

1+3 <= 11
ans += dp[4] = ~~3~~

dp[1] = ~~3~~

dp[0] = jumps=1 to <= arr[0]
(3 jumps)

0+1 <= 11
ans += dp[1] = ~~3~~

0+2 <= 11
ans += dp[2] = 3+0 = ~~3~~

0+3 <= 11
ans += dp[3] = 3+2 = 5

Program;

```
// Recursion-Memoization
stairs(a, 0, dp)   dp[] = new int[a.length+1];

public static int stairs(int[] a, int idx, int[] dp) {
        if (idx == a.length)
            return 1;
        if (idx > a.length)
            return 0;
        if (dp[idx] != 0)
            return dp[idx];
        int ans = 0;
        for (int jumps = 1; jumps <= a[idx]; jumps++)
            ans += stairs(a, idx + jumps, dp);
        dp[idx] = ans;
        return ans;
}
```

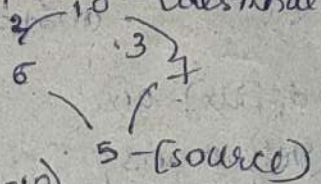
```
// Tabulation
public static int stairsTab(int[] a, int[] dp) {
        dp[dp.length-1] = 1;
    for (int i = dp.length-2; i >= 0; i--) {
            int ans = 0;
        for (int jumps = 1; jumps <= a[i].length; jumps++)
          {
          if (i + jumps < dp.length) {
              ans += dp[jumps + i];
          }
        }
        dp[i] = ans;
    }
    return dp[0];
}
```

4. Climb Stairs with Minimum Moves

Sample Input	Sample Output	Constraints
10 3 3 0 2 1 2 4 2 0 0	4	$0 <= n <= 20$ $0 <= n1, n2 \ldots <= 20$

Steps:

1. General Thinking: From 5 we can take 2 moves



From source to Destination,
it takes , 1+ Min of (6-10, 7-10)
= 1+ min(2, 3)
= 1+2 = 3 → (3 Min Moves from source to Destination) Why to Add 1? It takes 1 more from 5 to 6/7.

Recursive Approach with Memoisation

→ Call the Recursive function with the jumps w.r.t arr(i) by adding 1 and getting Min value.

→ If idx, reaches the scopes, return 0, if it equals to end of Array return 1.

Tabulation:

1. Take a dp Array of length [ar.length+1]
2. Go in the Reverse Direction, from last point to destination at it takes 0 mores
3. Go By Jumps wise, By checking boundary condition, store the min mores +1.

dp(i) → stores minimum No of mores to reach destination stairs from ith stair

Dry Run:

Array:

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

dp Array:

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

dp[10] = 0 [Min mores from ss to destination]

dp[9] = min=Int.Max-1, jumps=1 to <= ar[9] (0 jumps)

dp[9] = 1+min = Maxvalue [when you add Integer.max-1 gives -0]

dp[8] = min.Int.Max-1, jumps=1 to <= ar[8] (0 jumps)

dp[8] = 1+min = Maxvalue

dp[7] = min.Int.Max-1, jumps=1 to <= ar[7]

1+7 < 11, dp=

min=Int.Max, dp(8)

min=Max

2+7 < 11

min=1, dp[9]=1

min=1, 1

min=Map

dp[7] = 1+Max

dp(6) = min = Integer.Max-1 jumps = 1 to <= arr(6)
(4 jumps)

6+1<11, dp(7) = Max
min = (max, dp(7))
min = Max

6+2<11 dp(8) = Max
min = (Max, Max) = Max
min = Max

6+3<11, dp(9) = Max
min = (Max, Max) = Max

6+4<11, dp(10) = 0
min = (Max, 0) = 0

dp(6) = 1+0 = 1

dp(5) = min = Integer.Max-1, jumps = 1 to <= arr(5)
(2 jumps)

5+1<11, dp(6) = 1
min = (max, 1)
min = 1

5+2<11, dp(7) = Max
min = (1, Max) = 1

dp(5) = 1+1 = 2

dp(4) = min = Integer.Max-1, jumps = 1 to <= arr(4)
(1 jumps)

4+1<11, dp(5) = 2
min = (max, 2)
min = 2

dp(4) = 1+2 = 3

dp(3) = min = Integer.Max-1, jumps = 1 to <= arr(3)
(2, jumps)

3+1<11, dp(4) = 3
min = (max, 3)
min = 3

3+2<11, dp(5) = 2
min = (3, 2) = 2

dp(3) = 1+2 = 3

dp(2) = min = Integer.Max-1, jumps = 1 to <= arr(2)
(0 jumps)

dp(2) = Max

dp(1) = min=Integer.Max-1 , jumps=3

1+1 < 11 , dp(2)=max

min=(max, max) = max

1+2 < 11 , dp(3) = 3

min=(max, 3)=3

1+3 < 11, dp[4]=3.

min=(3,3)=3

dp(1)=1+3=4

dp(0)= min=Integer.Max-1, jumps=3

0+1 < 11, dp(1)=4

min=(max, 4)=4

0+2 < 11, dp(2)=max

min=(4, max)=4

0+3 < 11, dp[3]=3

min=(4,3)=3

dp(0)=1+3=4

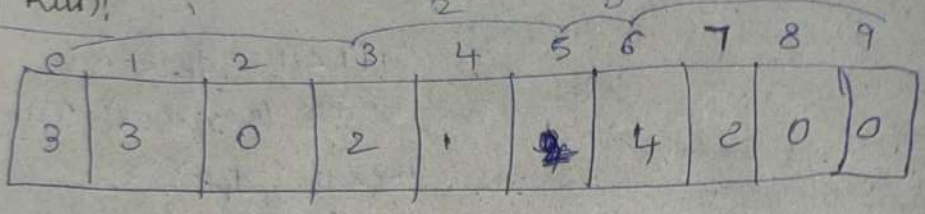
Using Greedy Approach;

-> Go in the Forward Direction, If array contains 0, we can't reach destination return Integer.MAX-VALUE.

-> Get the max index and max(by going through jumps which takes you to max position increment the steps, iterate the loop through max-index

-> If the jumps and position reaches to the end, return steps.

Dry Run:



max=0, maxidx=0 steps=0

jumps=1 to <=a[i].

(3) j=0+1=1 [1!=9)✓

1+a[1] >= max

→ maxidx=1

max=4

j=0+2=2 (2!=9)

2+a[2] >= max [2>=4)✓

j=0+3=3 (3!=9)

3+a[3] >= max [5>=4)

max=5, maxidx=3, steps=1

jumps=1 to a[i] [1 to a[maxidx]=1 to a[3]

[2 jumps)

j=

j=3+1=4 [4!=9)

4+a[4] >= max [5>=4)

max=5, maxidx=4

j=3+2=5 (5!=9)

5+a[5] >= max (6>=5) steps++

max=6, maxidx=5, steps=2

jumps=1 to a[i] [1 to a[maxidx]=1 to a[5]

[2 jumps)

j=5+1=6, (6!=9)

6+a[6] >= max, (10>=6)

max=10, maxidx=6

j=5+2=7, (7!=9)

7+a[7] >= max (9>=10)✓

max=10, maxidx=6, steps=3

jumps = 1 to a[i] (i to max (idx) = i to a[6])

(4 jumps)

j = 6+1 = 7, (7 != 9)

7 + a[7] = (9 >= 10) ×
max

j = 6+2 = 8 (8 != 9)

8 + a(8), (8 >= 10) ×

j = 6+3 = 9, (9 == 9) → steps++

steps = 4

Program;

Recursive Memoization

dp = new int[a.length+1]

Stairs(a, 0, dp)

```
public static int stairsR(int[] a, int idx, int[] dp) {
    if(idx >= a.length)
        return 0;
    if (idx == a.length-1)
        return 1;
    if (dp[idx] != 0)
        return dp[idx];
    int ans = Integer.MAX_VALUE-1;
    for (int jumps=1; jumps <= a[idx]; jumps++) {
        ans = Math.min(ans, 1+stairsR(a, idx+jumps, dp));
    }
    dp[idx] = ans;
    return ans;
}
```

```
//Tabulation
public static int staisTab(int[] a){
        int[] dp=new int[a.length+1];
        dp[dp.length-1]=0;
    for(int i=dp.length-2; i>=0; i--){
        int min=Integer.MAX_VALUE-1;
    for(int jumps=1; jumps<=a[i]; jumps++){
        if(i+jumps<dp.length)
            min=Math.min(min, dp[i+jumps]);
    }
    dp[i]=1+min;
}
return dp[0];
}
                //Greedy Approach
public static int staisOpt(int[] a){
        int steps=0;
        if(a[i]==0){
            break;
        }
        int max=0;
        int maxidx=0;
        for(int jumps=1; jumps<=a[i]; jumps++){
            int j=i+jumps;
            if(j==a.length-1){
                steps++;
                return steps;
            }
            if(j+a[j]>=max){
                maxidx=j;
                max=j+a[j];
            }
        }
    i=maxidx;
    steps++;
}
return Integer.MAX_VALUE;
}
```

5. MinCost in Maze Traversal

Sample Input

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

Sample Output

23

Constraints

$1 <= n <= 10^2$

$1 <= m <= 10^2$

$0 <= e_1, e_2, \ldots$ n*m elements $<= 1000$

Steps;

→ Using Recursion - Memoization

1) From the starting cell, we can go either horizontal or vertically [choose the minimum one after applying recursive calls] and add the current cell.
2) If sr exceeds Maze length (or) sc exceeds maze width, we can't reach destination, so return Integer.MAX_VALUE.
3) If we reach the destination, return the destination value. 4) Apply Memoization

Using Tabulation;

1) We observe that rows and columns are varying, so we take a dp array of 2D.
2) dp[i][j] mincost to reach destination starting from i, j

3) Totally 4 cases:

i) Store the destination value of Array directly into Dp Array.

ii) If we are in the last row, we can move only vertically.

iii) If we are in the last column, we can move only in the horizontally.

iv) Rest in all, get the Min value from (horizontally, vertically). Return dp[0][0].

Dry Run:

Original Array

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

Ans

Dp Array

	0	1	2	3	4	5
0	23	23	24	20	21	19
1	24	22	23	18	13	17
2	20	19	17	13	13	13
3	21	19	19	12	9	7
4	23	20	19	16	7	5
5	23	21	14	14	6	1

-> We will fill from back, i=5, j=5

from ①, dp[i][j] = arr[i][j] = 1

i=5, j=4, [last row] dp[i][j] = dp[i][j+1] + arr[i][j]
= 1+5 = 6

i=5, j=3, dp[i][j] = 8+6 = 14

i=5, j=2, dp[i][j] = 0+14 = 14

i=5, j=1, dp[i][j] = 7+14 = 21

i=5, j=0, dp[i][j] = 2+21 = 23

i--,

i=4, j=5 [from ③] [last column]

dp[i][j] = dp[i+1][j] + arr[i][j] = 4+1 = 5

i=4, j=4 [from ④]

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

= min(5,6) + 2 = 7

i=4, j=3 (from ④)

dp[i][j] = min(7, 14) + 9

= 16

i=4, j=2 (from ④)

dp[i][j] = min(16, 14) + 5

= 19

i=4, j=1 (from ④)

dp[i][j] = min(19, 21) + 1

= 20

i=4, j=0 (from ④)

dp[i][j] = min(20, 23) + 3

= 23

i--

i=3, j=5 (from ③), i=3, j=4 (from 4)

dp[i][j] = 5 + 2 = 7

dp[i][j] = min(7, 7) + 2 = 9

i=3, j=3 (from 4)

dp[i][j] = min(9, 16) + 3 = 12

i=3, j=2 (from 4)

dp[i][j] = min(12, 19) + 7 = 19

i=3, j=1 (from 4)

dp[i][j] = min(19, 20) + 0 = 19

i=3, j=0 (from 4)

dp[i][j] = min(19, 23) + 2 = 21

i--

i=2, j=5 (from 3)

dp[i][j] = 7 + 6 = 13

i=2, j=4 (from 4)

dp[i][j] = min(9, 7) + 4 = 13

i=2, j=3 (from 4)

dp[i][j] = min(12, 13) + 1 = 13

i=2, j=2 (from 4)

dp[i][j] = min(13, 19) + 4 = 17

i=2, j=1 (from 4)

dp[i][j] = min(17, 19) + 2 = 19

i=2, j=0 (from 4)

dp[i][j] = min(19, 21) + 1 = 20

i--

i=1, j=5 (from 3)

dp[i][j] = 4 + 13 = 17

i=1, j=4 (from 4)

dp[i][j] = min(13, 17) + 0

= 13

i=1, j=3 (from 4)

dp[i][j] = min(13, 13) + 5 = 18

i=1, j=2 (from 4)

dp[i][j] = min(18, 17) + 6 = 23

i=1, j=1 (from 4)

dp[i][j] = min(23, 19) + 3

= 22

i=1, j=0 (from 4)

dp[i][j] = min(20, 22) + 4

= 24

i--, i=0, j=5 (from 3)

dp[i][j] = 2 + 17 = 19

i=0, j=4 (from 4)

dp[i][j] = min(13, 19) + 8

= 21

i=0, j=3 (from 4)

dp[i][j] = min(18, 21) + 2

= 23

i=0, j=2 (from 4)

dp[i][j] = min(23, 20) + 4

= 24

i=0, j=1 (from 4)

dp[i][j] = min(22, 24) + 1

= 23

i=0, j=0 (from 4)

dp[i][j] = min(23, 24) + 0

= 23

dp[0][0] = 23

Program:

// Recursion - Memoization

minCost(arr, 0, 0, new int[n][m]); // 0 → source row, 0 → source column

```
public static int minCost(int[][] arr, int sr, int sc, int[][] dp) {
    if (sr >= arr.length || sc >= arr[0].length) {
        return Integer.MAX_VALUE;
    }
    if (sr == arr.length-1 && sc == arr[0].length-1) {
        return arr[sr][sc];
    }
    if (dp[sr][sc] != 0)
        return dp[sr][sc];
    int f1 = minCost(arr, sr, sc+1, dp);
    int f2 = minCost(arr, sr+1, sc, dp);
    int ans = Math.min(f1, f2) + arr[sr][sc];
    dp[sr][sc] = ans;
    return ans;
}
```

// Tabulation

```
public static int minTabCost(int[][] arr) {
    int[][] dp = new int[arr.length][arr[0].length];
    for (int i = arr.length-1; i >= 0; i--) {
        for (int j = arr[0].length-1; j >= 0; j--) {
            // 1st case lastrow && last col
            if (i == arr.length-1 && j == arr[0].length-1) {
                dp[i][j] = arr[i][j];
            }
            // 2nd case, last row
            if (i == arr.length-1) {
                dp[i][j] = dp[i][j+1] + arr[i][j];
            }
```

```
// 3rd case last col
else if (j == arr[0].length-1) {
    dp[i][j] = dp[i+1][j] + arr[i][j];
}
// 4th case - Both directions
else {
    dp[i][j] = Math.min(dp[i+1][j-1], dp[i+1][j]) + arr[i][j];
}
return dp[0][0];
}
```

6. GoldMine

Sample Input:

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

left → right

Output
33

Constraints
1 <= n <= 10^2
1 <= m <= 10^2
0 <= e1, e2 ... n*m elements <= 1000

Steps:
1) According to problem statement, we will be standing on the left wall (first col), we need to reach to right wall (last col) by going 1-cell up, 1 cell right, 1-cell right down.
2) We should get maximum amount of gold.

Recursion - Memoization

1) Apply three recursive calls for the three directions get the max value and add current cell.
2) If sr, sc reaches boundary, return 0, else if we reach the destination return the cell value.
3) Apply Memoization.

Tabulation:

1. We observe that rows and columns are varying, so we take a dp array 2D.
2. dp[i][j] - stores max amount of gold collected from i, j to end last column.
3. Totally 4 cases
   i) If we are in the last col, simply store the arr[i][j] into dp[i][j].
   ii) If we are in the first row, we can go up, only, we can go (right or rightdown) → get the max value and add current cell.
   iii) If we are in the last row, we can go only (right up or right) get the max value and add the current cell.
   iv) When all possible 3 directions are possible get the max value and ~~return~~ add the current cell.
4. Now take the max value in first col and return the value.

Dry Run;

Original Array

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

DP Array

26	24	21	14	12	2
31	26	23	17	5	4
28	27	21	11	10	6
29	25	25	13	8	2
33	26	23	18	6	4
32	30	18	17	9	1

j = col, i = row

j = 5, j = 0 → [From i) → last col

i = 0, dp[0][5] = arr[0][5] = 2
i = 1, dp[1][5] = arr[1][5] = 4
i = 2, dp[2][5] = arr[2][5] = 6
i = 3, dp[3][5] = arr[3][5] = 2
i = 4, dp[4][5] = arr[4][5] = 4
i = 5, dp[5][5] = arr[5][5] = 1

j = 4, i = 0 - [From ii) → first row

dp[0][4] = max(2, 4) + 8 = 12

i = 1, [From iv) → dp[1][3] = max(2, 4, 6) + 0 = 6

i = 2, [From iv) → dp[2][4] = max(4, 6, 2) + 4 = 10

i = 3, [From iv) → dp[3][4] = max(6, 2, 4) + 2 = 8

i = 4, [From iv) → dp[4][4] = max(2, 4, 1) = 4 + 2 = 6

i = 5, [From iii) → dp[5][4] = max(4, 1) + 5 = 9

Ny fill for all cols

→ Now in the first call get the max value → 33

Program: Recursion // Memorization

```
public static int gold(int[][] arr, int sr, int sc, int[][] dp){
  if (sr >= arr.length || sc >= arr[0].length){
      return 0;
  }
  if (sc == arr[0].length-1){
      return dp[sr][sc];
  }
  if (dp[sr][sc] != 0){
      return dp[sr][sc];
  }
  int f1 = 0;
  if (sr != 0)
    f1 = gold(arr, sr-1, sc+1, dp);
  int f2 = gold(arr, sr, sc+1, dp);
  int f3 = gold(arr, sr+1, sc+1, dp);
  int ans = Math.max(f1, Math.max(f2,f3)) + arr[sr][sc];
  dp[sr][sc] = ans;
  return ans;
}
```

Tabulation

```
public static int goldTab(int[][] arr){
    int[][] dp = new int[arr.length][arr[0].length];
  for(int j = arr[0].length-1; j >= 0; j--){
      for(int i=0; i< arr.length; i++){
        // last col
        if (j == arr[0].length-1){
            dp[i][j] = arr[i][j];
        }
```

```
// first row right and right down
else if (i == 0){
    dp[i][j] = Math.max(dp[i][j+1], dp[i+1][j+1]) +
               arr[i][j];
}
// last row, right and right up
else if (i == arr.length-1) {
    dp[i][j] = Math.max(dp[i][j+1], dp[i-1][j+1]) +
               arr[i][j];
}
// all cases
else {
dp[i][j] = arr[i][j] + Math.max(dp[i-1][j+1],
        Math.max(dp[i][j+1], dp[i+1][j+1]));
}
}
}
// max value in first column
int max = 0;
for(int i=0; i< arr.length; i++) {
        max = Math.max(max, dp[i][0]);
}
return max;
}
```

7. Target sum subsets - DP

Sample Input:

5 — n
4
2
7
1
3
10 — target

Sample Output:

true

Constraints

$1 <= n <= 30$

$0 <= n1, n2 \cdots n$ elements $<= 20$

$0 <= tar <= 50$

Steps:

Recursion/Memotization

1. Make two recursive calls one for the element included or not.
2. If the target reaches, 0 return
3. If the idx reaches, boundary (or) tar < 0, then return false.
4. Take the Logical OR of two cases and Return
5. Apply Memotization.

Tabulation

1) From Recursion, we observe index and target both are varying take a 2d array of arr.length+1 = rows and target+1 = columns.
2) As target 0 is always possible mark the 0th column true. [Initially, boolean Array will have false values].
3) dp[i][j] → consider i no of elements, j = target possible? ~~if yes, no~~
4) Two cases;
   → dp[i][j] = dp[i-1][j]
   (2) If dp[i][j] == false, j = arr[i-1] > 0
   dp[i][j] = dp[i-1][j - arr[i-1]]
5) ~~Stop~~ Return the last value

Dry Run:

target 0 is always possible, dp(i)(0) = true

→ For i=1 (element 4)

j=1, dp(i)(j) = dp(i-1)(j) = false, && 1-(4) >= 0 ✗

j=2, dp(i)(j) = dp(i-1)(j) = false, && 2-(4) >= 0 ✗

j=3, dp(i)(j) = dp(i-1)(j) = false && 3-(4) >= 0 ✗

j=4 dp(i)(j) = dp(i-1)(j) = false && 4-4 >= 0 ✓

dp(i)(j) = dp(i-1)(j - arr(i-1))

= dp(0)(4-4) = dp(0)(0)

= true

j=5, dp(i)(j) = false, 5-4 = 1 >= 0 ✓

dp(i)(j) = dp(i-1)(j - arr(i-1))

= dp(0)(1) = false

silly up to j=10, we get dp(i)(j) = false

For i=2 (element 4, 2)

j=1, dp(i)(j) = dp(i-1)(j) = false, && 1 - arr(i)

= 1-2 >= 0 ✗

dp(i)(j) = false

j=2, dp(i)(j) = dp(i-1)(j) = false && 2 - arr(i) = 2-2 >= 0 ✓

dp(i)(j) = dp(i)(j - arr(i-1)) = dp(i)(0) = true

silly Apply the procedure for rest of elements → Ans will be stored in the last cell

dp(i)(j) → stores up to i no of elements, does the target achieved.

Program:

Recursion / Memoization

```
Boolean[][] dp = new Boolean[n][tar+1];
                      index  target
target(ars, 0, tar, dp)
```

Why? Boolean() By default all the values in cell contains null values.

```
public static boolean target(int[] ar, int idx, int tar,
    Boolean[][] dp){
        if(tar==0)
            return true;
        if(idx==ar.length || tar<0)
            return false;
        if(dp[idx][tar]!=null)
            return dp[idx][tar];
    boolean f1 = target(ar, idx+1, tar, dp);      // no
    boolean f2 = target(ar, idx+1, tar-ar[idx], dp);  // yes
     dp[idx][tar] = f1 || f2;
     return f1 || f2;
}
```

Tabulation

```
public static boolean tarTab(int[] ar, int tar){
    boolean[][] dp = new boolean[ar.length+1][tar+1];
    // 0th column -> true (target 0 is always possible)
    for(int i=1; i<dp.length; i++){
        for(int j=1; j<dp[0].length; j++){
            dp[i][j] = dp[i-1][j];
            if(dp[i-1][j]==false && j-ar[i-1]>=0)
                dp[i][j] = dp[i-1][j-ar[i-1]];
        }
    }
    return dp[dp.length-1][dp[0].length-1];
}
```

8. Zero-One-KnapSack

Input

5 → count
→ values
15 14 10 45 30
2 5 1 3 4 → weights
7 → capacity of bag

Output

75

Constraints

$0 <= V_1, V_2 .. n$ elements $<= 50$
$0 <= W_1, W_2 .. n$ elements $<= 50$
$0 <= cap <= 10$

Steps:

According to Problem statement, we need to fill the bag without overflowing capacity having maximum value.

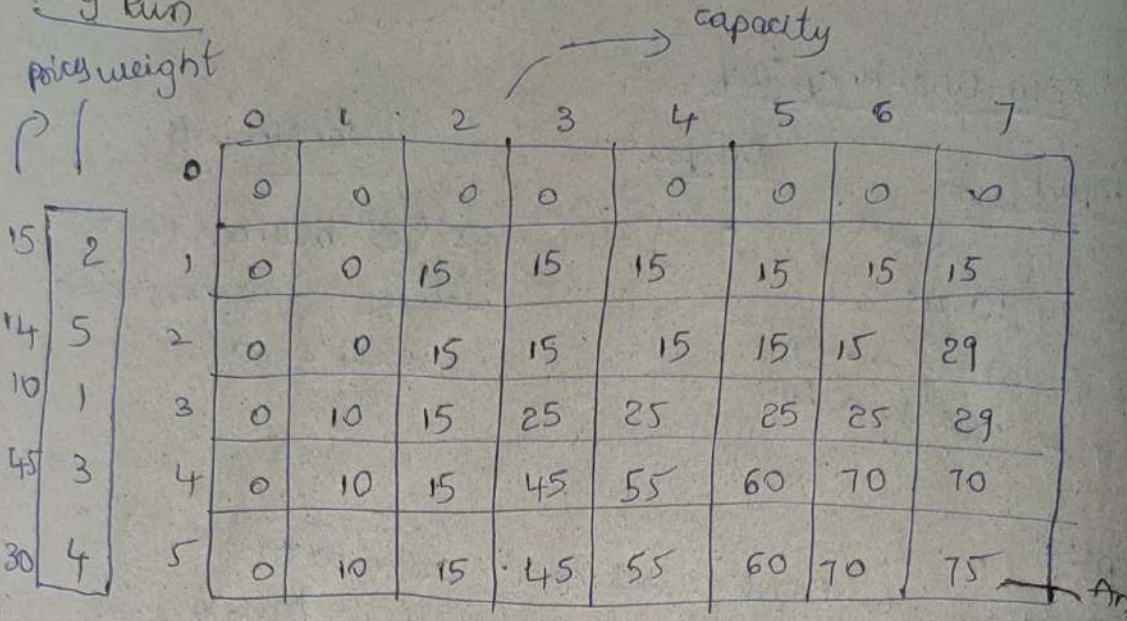
Recursion / Memotization:

1. Include the weight in the bag, with the value.
2. Don't include the weight, take the value as 0.
3. From the two recursive calls get the maximum value.
4. If the cap reaches, negative return MIN value
5. Continue till it covers all the values.

Tabulation

1. From Recursion, we observe index and capacity are varying, take a 2D array of price.length+1 = rows and capacity+1 = columns
2. dp[i][j] → i No of items considering ~~gives~~ with capacity j gives Profit?
3. Two cases: i) dp[i][j] = dp[i-1][j]
   ii) Getting the max values of dp[i][j], price[i-1] + dp[i-1][j-weights[i-1]]
4) Return the last value of Array

Dry Run



For 0 weight, and 0 capacity → Nothing will be there

For $i=1$, $j=1$ $dp[i][j] = dp[i-1][j] = 0$,

j-weights $(i-1) = 1-2 >= 0$ ✗

$dp[i][j] = 0$

$j=2$, $dp[i][j] = dp[i-1][j] = 0$,

j-weights $[i-1] = 2-2 >= 0$ ✓

$dp[i][j] = \max(0, 15 + dp[0][0]) = 15$

$j=3$, $dp[i][j] = dp[i-1][j] = 0$

j-weights $(i-1) = 3-2 >= 1$ ✓

$dp[i][j] = \max(0, 15 + dp[0][1]) = 15$

$j=4$ to $j=7$, continues, $dp[i][j] = 15$

For $i=2$, $j=1$ $dp[i][j] = dp[i-1][j] = 0$

j-weights $(i-1) = 1-5 >= 0$ ✗

$dp[i][j] = 0$

$j=2$, $dp[i][j] = dp[i-1][j] = 15$

j-weights $(i-1) = 2-5 >= 0$ ✗

$dp[i][j] = 15$

$j=3$ to $j=6$ continues

$j=7$, $dp[i][j] = 15$

j-weights $(i-1) = 7-5 = 2$

$dp[i][j] = \max(15, 14 + dp[1][2]) = \max(15, 29) = 29$

Silly the process continues

Return the last value of array

Programs:

Recursion Tabulation / Memoization

knapSack(prices, weights, 0, cap) (idx, capacity)

```
public static int knapSack(int[] prices, int[] weights, int idx,
  int cap){
      if(cap<0)
          return Integer.MIN_VALUE;

      if(idx == prices.length)
          return 0;

int f1 = 0 + knapSack(prices, weights, idx+1, cap);   // weight Not considered
int f2 = prices[idx] + knapSack(prices, weights, idx+1,
      cap - weights[idx]);                            // weight included

int ans = Math.max(f1, f2);
return ans;
}
```

Tabulation

```
public static int knapTab(int[] prices, int[] weights, int cap){
    int[][] dp = new int[prices.length+1][cap+1];

    for(int i=1; i<dp.length; i++){
        for(int j=1; j<dp[0].length; j++){
            dp[i][j] = dp[i-1][j];
            if(j - weights[i-1] >= 0){
                dp[i][j] = Math.max(dp[i][j],
                    prices[i-1] + dp[i-1][j - weights
                        [i-1]]);
            }
        }
    }

    return dp[dp.length-1][dp[0].length-1];
}
```

9. Coin Change Combination

Sample Input

4 → n
2
3
5
6
7 → amount

Sample Output

2

Constraints

- $1 <= n <= 30$

$0 <= n1, n2, \ldots n\text{elements} <= 20$

$0 <= amt <= 50$

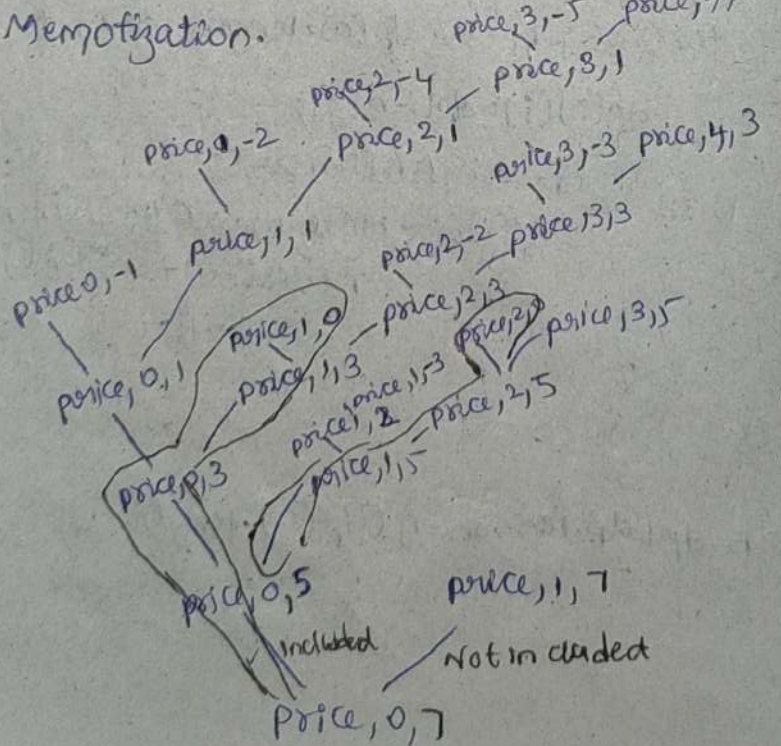
Steps;

According to the problem statement, we can have combination of coins (can be used more than once) that make up the total Amount in How Many number of ways.

Recursion / Memoization

1. The coin can be included, modify the capacity/total amount
2. The current coin cannot be included
3. Count the total number of ways by adding the returns of two recursive calls.
4. Apply Memoization.

DryRun;



Only Part of the two solutions possible

2 2 3
2 5

Tabulation

Memoization [2D]

1) As index and cap are varying we take 2D array of price.length+1 = rows, cap.length+1 = columns

2) dp[i][j] → ; considering the upto the number of coins, i with coins = ? Gives the total Amount

3) Occurs two cases;

i) dp[i][j] = dp[i-1][j]

ii) If j - price[i-1] >= 0 then

dp[i][j] = dp[i-1][j] + dp[i][j - price[i-1]];

4) Return the last value of Array

Dry Run:

→ Amount (capacity)

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

Denomination 0 can be made only 1 way irrespective of denominations with 0 capacity, so dp[i][0] = 1

for i = 1, j = 1 dp[i][j] = dp[i-1][j] = 0

1-2 >= 0 ✗

j = 2, dp[i][j] = 0,

2-2 >= 0 ✓ dp[i][j] = 0 + dp[i][0] = 1

j = 3, dp[i][j] = 0

3-2 >= 1 ✓ dp[i][j] = 0 + dp[i][1] = 0

j = 4, dp[i][j] = 0

4-2 >= 2, dp[i][j] = 0 + dp[i][2] = 1

j = 5, dp[i][j] = 0

5-2 >= 3, dp[i][j] = 0 + dp[i][3] = 0

j = 6, dp[i][j] = 0

6-2 >= 4, dp[i][j] = 0 + dp[i][4] = 1

j=7, dp(i)(j)=0
7-2>=0, dp(i)(j)=0+dp(i)(5)=0

i=2, j=1, dp[i](j)=0
1-3>=0✗

j=2, dp(i)(j)=0
2-3>=0

j=3, dp(i)(j)=0
3-3>=0
dp(i)(j)=dp[2](0)
=1

j=4, dp(i)(j)=1
4-3>=1, dp(i)(j)=0+1
=1

j=5, dp(i)(j)=0
5-3>=2, dp(i)(j)=dp[2](2)+0
=1

j=6, dp(i)(j)=1
6-3>=3, dp(i)(j)=1+1=2

j=7, dp(i)(j)=0
7-3>=4, dp(i)(j)=0+1=1

i=3, j=1, dp(i)(j)=0
1-5>=0✗

j=2, dp(i)(j)=1
2-5>=0✗

j=3, dp(i)(j)=1
3-5>=0✗

j=4, dp(i)(j)=1
4-5>=0✗

j=5, dp(i)(j)=1
5-5>=0✓ dp(i)(j)=1+1
=2

j=6, dp(i)(j)=2
6-5>=1
dp(i)(j)=2+0=2

j=7, dp(i)(j)=1
7-5>=0, dp(i)(j)=1+1=2

i=4, j=1, dp(i)(j)=0
1-6>=0✗

j=2, dp(i)(j)=1
2-6>=0✗

j=3, dp(i)(j)=1
3-6>=0✗

j=4, dp(i)(j)=1
4-6>=0✗

j=5, dp(i)(j)=2
5-6>=0✗

j=6, dp(i)(j)=2
6-6>=0, dp(i)(j)=2+1=3

j=7, dp(i)(j)=2
7-6>=0, dp(i)(j)=2+0=2

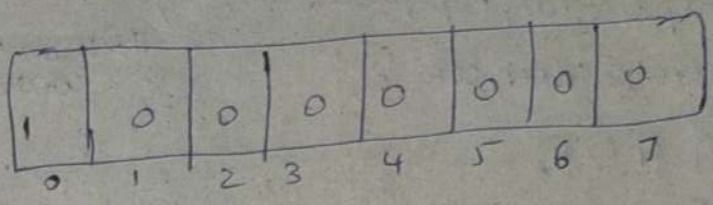
(2-ways)

Optimization of 2D-1D

1) Take an array of cap+1. and dp(0)=1 [denomination is possible with cap 0 in 1 way]
2) Go through the prices array
3) Get the price value, iterate through dp and assign dp(j) = dp(j)+dp[j-price(i)]
4) Return the last value.

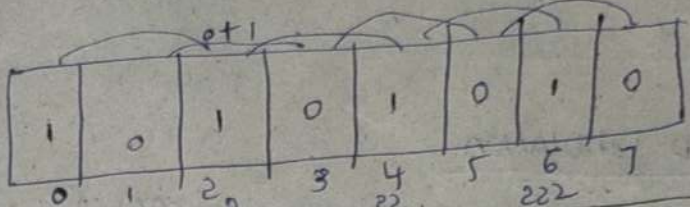
cap = 7, price [2,3,5,6]

Step 1;

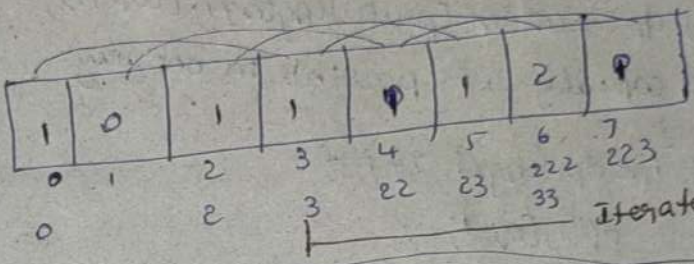


and assign dp[0] = 1

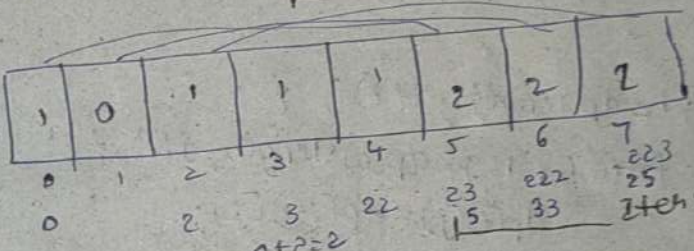
Iterate through price 2



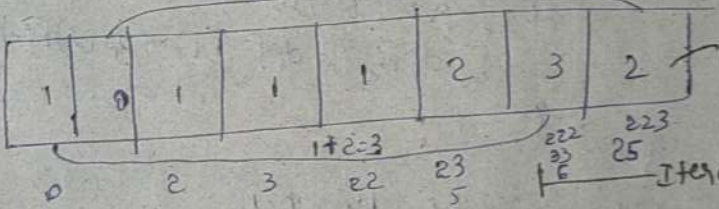
Iterate through price 3



Iterate through price 5



Iterate through price 6



Programs;

Recursion + Memoization

```
coinR(price, 0, cap, new int[price.length+1][cap+1]);
                 index
public static int coinR(int[] price, int idx, int cap, int[][] dp) {
    if (cap < 0)
        return 0;
    if (cap == 0)
        return 1;
    if (idx == price.length)
        return 0;
    if (dp[idx][cap] != 0)
        return dp[idx][cap];
```

```
int fi = coinR(price, idx, cap-price[idx], dp);   // Denom considered
int fe = coinR(price, idx+1, cap, dp);            // Denom Not considered

dp[idx][cap] = fi + fe;
return fi + fe;
}
```

2D Tabulation

```
public static int coin2d(int[] price, int cap) {
    int[][] dp = new int[price.length+1][cap+1];
    // capacity 0 is possible in one way

    for(int i=0; i<dp.length; i++) {
        dp[i][0] = 1;
    }
    for(int i=1; i<dp.length; i++) {
        for(int j=1; j<dp[0].length; j++) {
            dp[i][j] = dp[i-1][j];
            if(j-price[i-1] >= 0) {
                dp[i][j] = dp[i-1][j] + dp[i][j-price[i-1]];
            }
        }
    }
    return dp[dp.length-1][dp[0].length-1];
}
```

1D Tabulation (optimization)

```
public static int coin(int[] price, int cap) {
    int[] dp = new int[cap+1];
    dp[0] = 1;                    // capacity 0 is possible in one way
    for(int i=0; i<price.length; i++) {
        int coin = price[i];
        for(int j=coin; j<dp.length; j++) {
            dp[j] = dp[j] + dp[j-coin];
        }
    }
    return dp[cap];
}
```

10. Coin Change Permutations

Sample Input	Sample Output	Constraints
4 → n	5	1 <= n <= 20
2		0 <= n1, n2, ... n <= 30
3		0 <= amount <= 30
5		
6		
7 → amount		

Steps;

According to the problem statement, we can have permutations of coins (can be used more than once) that make up the Total Amount in how many number of ways?

Recursion, Memoization

1) As permutations are allowed, we should go through every denominations to get all the posibilities.

2) If cap < 0 return 0, cap == 0 return 1

3) For Memoization, we can use, Integer Array which stores null values by default (why? it reduces stack calls for value 0 :).

Tabulation (i/p)

1) Take an array of cap+1 and dp[0]=1 (denomination is possible with cap 0 in one way.

2) Go through the dp Array.

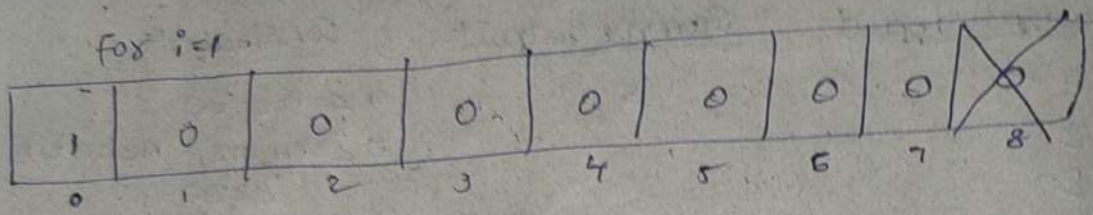
3) Go through the price Array. and assign the value, dp[i] = dp[i] + dp[i - price[j]];

4) Return the last value.

~~Retu~~

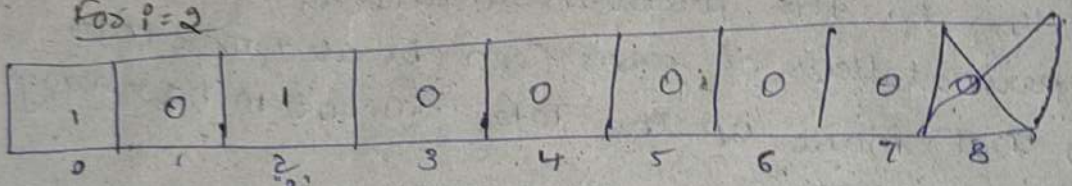
Dry Run: cap = 7, price = 8, 3, 5, 6

for i = 1



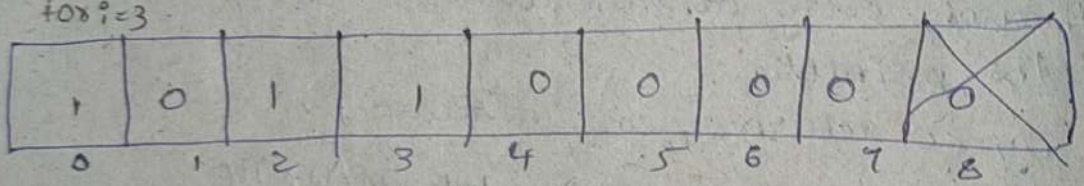
j = 0 < 4, 1-2 >= 0 ✗
1-3 >= 0 ✗
1-5 >= 0 ✗
1-6 >= 0 ✗

for i = 2



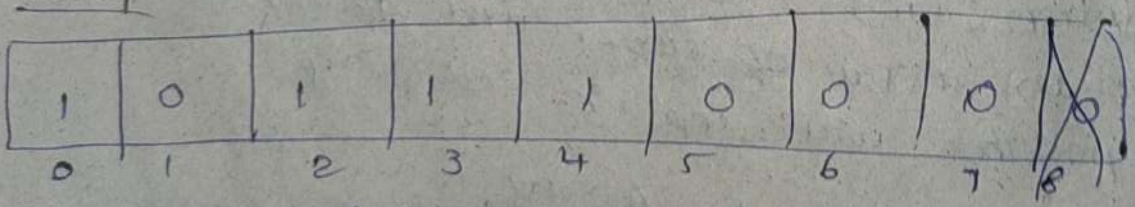
j = 0 < 4, 2-2 >= 0 ✓ dp[i] = dp[0] = dp[0] = 1
2-3 >= 0 ✗
2-5 >= 0 ✗
2-6 >= 0 ✗

for i = 3



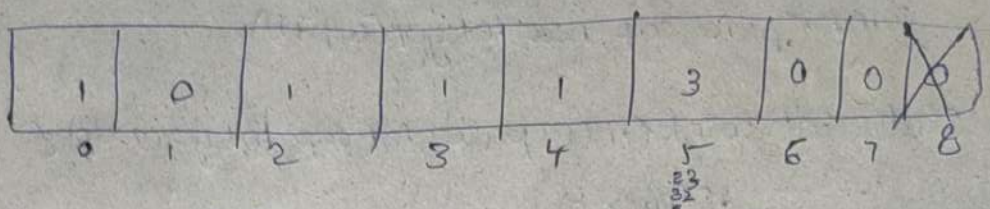
j = 0 < 4, 3-2 >= 1 ✓ dp(i) = dp(1) = 0
3-3 >= 0 ✓, dp[i] = 0 + 1 = 1
3-5 >= 0 ✗
3-6 >= 0 ✗

for i = 4



j = 0 < 4, 4-2 >= 2, dp(2) = 1
4-3 >= 1, dp[4] = 1 + dp(1) = 1
4-5 >= 0 ✗
4-6 >= 0 ✗

For i = 5



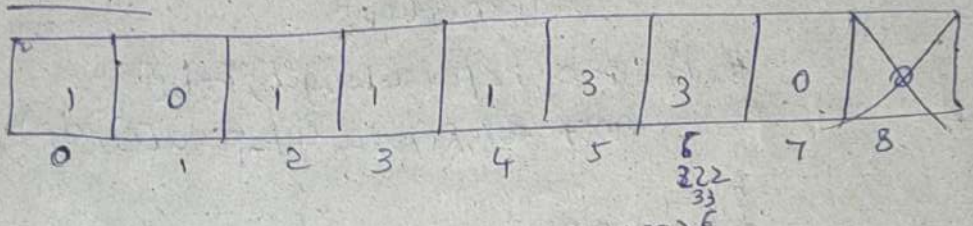
j = 0 < 4, 5-2 >= 0 ✓ dp(5) = dp(3) = 1

5-3 >= 2 ✓ dp(5) = 1 + dp(2) = 2

5-5 >= 0, dp(5) = 2 + dp(0) = 2+1 = 3

5-6 >= 0 ✗

For i = 6



j = 0 < 4, 6-2 >= 0, dp(6) = dp[4] = 1

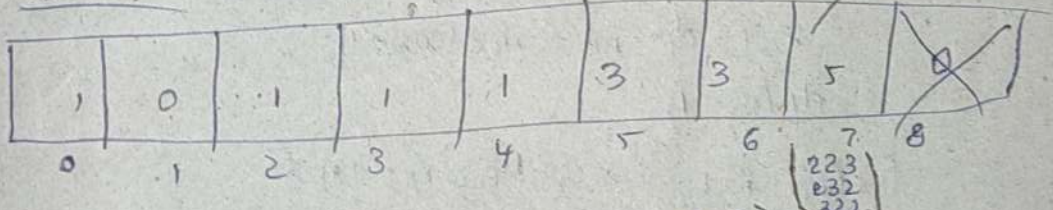
6-3 >= 0, dp(6) = 1 + dp(3) = 2

6-5 >= 0, dp[6] = 2 + dp[1] = 2 + 0 = 2

6-6 >= 0, dp(6) = 2 + dp(0) = 2+1 = 3

For i = 7

Solution



j = 0 < 4, 7-2 >= 0, dp(7) = dp(5) = 3

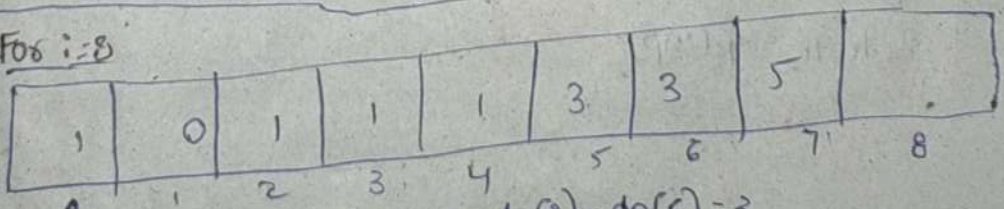
223
232
322
52
25

7-3 >= 0, dp(7) = 3 + dp(4) = 3+1 = 4

7-5 >= 0, dp(7) = 4 + dp(2) = 4+1 = 5

7-6 >= 0, dp(7) = 5 + dp(0) = 5

For i = 8



j = 0 < 4, 8-2 >= 0, dp(8) = dp(6) = 3

8-3 >= 0, dp(8) = 3 + dp(5) =

Rough work

Program; //Recursion-Memoization

```
coinR(price, cap, new Integer[cap+1]);

Public static int coinR(int[] price, int cap, Integer[] dp) {
        if(cap<0)
            return 0;
        if(cap==0)
            return 1;
        int ans=0;
        if(dp[cap]!=null)
            return dp[cap];
        for(int i=0; i<price.length; i++){
            ans+=coinR(price, cap-price[i], dp);
        }
        dp[cap]=ans;
        return ans;
}
```

//Tabulation

```
public static int coin(int[] price, int cap) {
        int[] dp=new int[cap+1];
        dp[0]=1;
    for(int i=1; i<dp.length; i++) {
            for(int j=0; j<price.length; j++) {
                if(i-price[j]>=0)
                    dp[i]+=dp[i-price[j]];
            }
    }
    return dp[cap];
}
```

11. UnBounded KnapSack

Input	Output	Constraints
5 → n	100	1 <= n <= 20
15 14 10 45 30 → values		0 <= v1, v2 .. n elements <= 50
2 5 1 3 4 → weights		0 < w1, w2 .. n elements <= 10
~~100~~ 7 → cap		0 < cap <= 10

Steps:

According to Problem statement, we need to fill the bag without overflowing capacity, (same weight can be taken multiple times), should Return the Maximum Value.

Recursion/Memoization:

1. The coin can be included, modify the cap with corresponding weight, include the price.
2. The coin cannot be included.
3. Return the Max of Two Recursive calls.
4. Apply Memoization.

Tabulation ID:

1. Iterate through ~~through~~ the prices, get corresponding weight.
2. Iterate through the dp from that weight.
3. Get the Max value of current price including, and not,
4. Store the max value at respective position in dp Array.

Dry Run:

prices: 15 14 10 45 30

weights: 2 5 1 3 4

cap: 7

dp[i][j]:
stores with price i, weight j
Gives: max

i=0,

ip = price[0] = 15

iw = weight[0] = 2

0	0	15	15	30	30	45	45
0	1	2	3	4	5	6	7

from iw=2 < 8, j=2

dp[j] = (0, 15+dp[2-2]) = 15

j=3 < 8, j=3

dp[j] = (0, 15+dp[3-2]) = 15

j=4,

dp[j] = (0, 15+dp[4-2]) = 15+15 = 30

j=5

dp[j] = (0, 15+dp[5-2]) = 15+15 = 30

j=6,

dp[j] = (0, 15+dp[4]) = 45

j=7

dp[j] = (0, 15+dp[5]) = 15+30 = 45

i=1, ip = price[1] = 14

iw = weight[1] = 5

0	0	15	15	30	30	45	45
0	1	2	3	4	5	6	7

from, iw=5, j=5

j=5 < 8, dp[j] = max(30, 14+dp[5-5]):

= max(30, 14) = 30

j=6 < 8, dp[j] = max(45, 14+dp[1]):

= 45

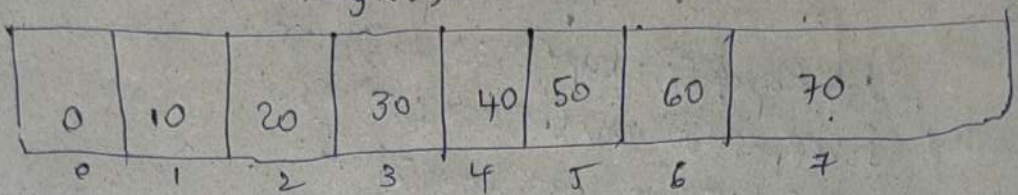
$j = 7 < 8$, $dp(j) = \max(45, 14 + dp[2]) =$

$= 45$

$i = 2$,

$ip = price(2) = 10$

$iw = weight(2) = 1$



$j = 1 < 8$,

$dp(j) = \max(0, dp(0) + 10) = 10$

$j = 2 < 8$,

$dp(j) = \max(15, 10 + dp[1]) = 20$

$j = 3 < 8$, $dp(j) = \max(15, 10 + dp(2)) = 30$

$j = 4 < 8$, $dp[j] = \max(30, 10 + dp(3)) = 40$

$j = 5 < 8$, $dp(j) = \max(30, 10 + dp(4)) = 50$

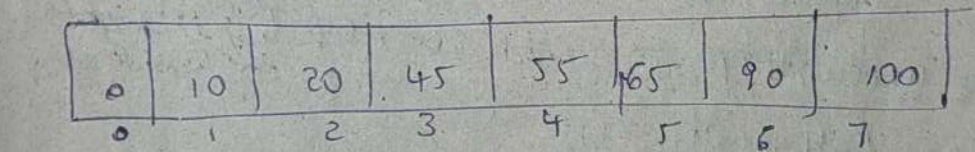
$j = 6 < 8$, $dp(j) = \max(45, 10 + dp[5]) = 60$

$j = 7 < 8$, $dp(j) = \max(45, 10 + dp[6]) = 70$

$i = 3$,

$ip = price(3) = 45$

$iw = weight(3) = 3$



$j = 3 < 8$, $dp(j) = \max(30, 45 + dp[0]) = 45$

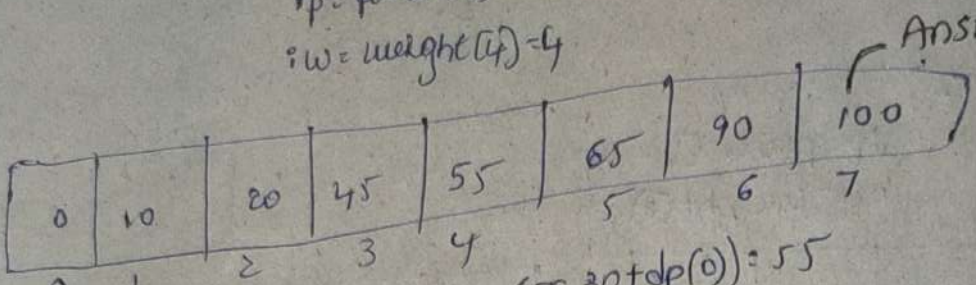
$j = 4 < 8$, $dp(j) = \max(40, 45 + dp(1)) = 55$

$j = 5 < 8$, $dp(j) = \max(50, 45 + dp(2)) = 65$

$j = 6 < 8$, $dp(j) = \max(60, 45 + dp(3)) = 90$

$j = 7 < 8$, $dp[j] = \max(70, 45 + dp(4)) = 100$

i = 4,
ip = price[4] = 30
iw = weight[4] = 4



j = 4 < 8, dp[j] = max(55, 30 + dp(0)) = 55
j = 5 < 8, dp[j] = max(65, 30 + dp(1)) = 65
j = 6 < 8, dp[j] = max(90, 30 + dp[2]) = 90
j = 7 < 8, dp[j] = max(100, 30 + dp[3]) = 100

Program:

Recursion/Memotization

```
public static int unboundR(int[] prices, int[] weights,
    int idx, int cap, int[][] dp) {          // dp = new int[n+1][cap+1];
        if(cap < 0)
            return Integer.MIN_VALUE;
        if(idx == prices.length)
            return 0;
        if(dp[idx][cap] != 0){
            return dp[idx][cap];
    int f1 = unboundR(prices, weights, idx, cap-weights[idx], dp)
            + prices[idx]                    // weight considered
    int f2 = unboundR(prices, weights, idx+1, cap, dp); // weights
                                                 Not considered
     int ans = Math.max(f1, f2);
     dp[idx][cap] = ans;
     return ans;
}
```

// Tabulation

```
public static int unboundTab(int[] prices, int[] weights, int cap){
    int[] dp = new int[cap+1];
    for(int i=0; i<price.length; i++){
        int ip=prices[i];
        int iw= weights[i];
        for(int j=iw; j<dp.length; j++){
            dp[j]= Math.max(dp[j], ip+dp[j-iw]);
        }
    }
return dp[cap];
}
```

18 Count Binary Strings

Sample Input: 6

Output: 21

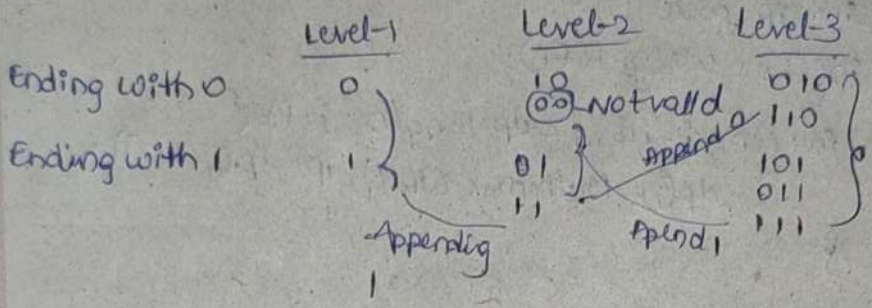
Constraints: $0 <= n <= 45$

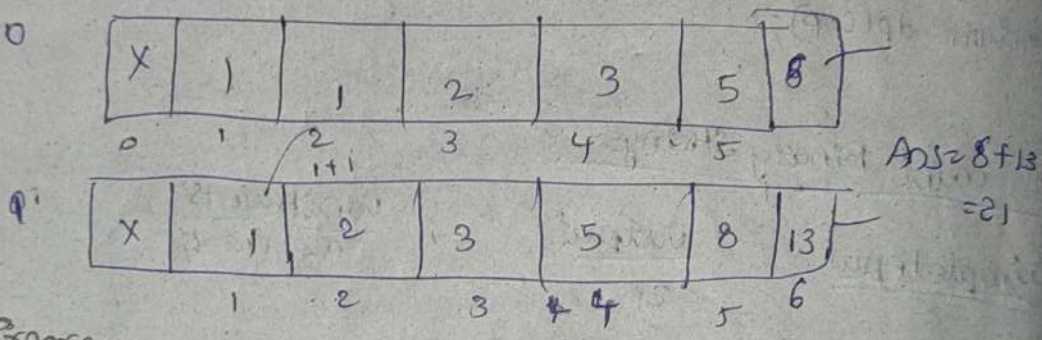
Steps; Tabulation ~~Memo~~;

1. Take 2 dp arrays of n+1.
2) One dp Array Represents Binary strings ending with 0 and Another dp Array Represents Binary Strings Ending with 1.
3) Store dp1[0]=1 [strings ending with 0 = 1
   dp2[0]=1 strings ending with 1 = 1]
4) Store dp1[i]=dp2[i-1];
   dp2[i]= dp1[i-1]+dp2[i-1];
5) Return the sum of last two values of 2 dp Arrays.

Recursion/Memoization: From Tabulation, we can observe the Fibonacci(n+3) gives the solution.

Dry Run: Tabulation





Programs:

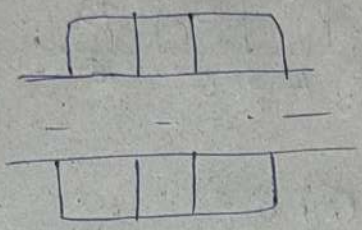
```
public static int count(int n) {
    int[] dp1 = new int[n+1];
    int[] dp2 = new int[n+1];

    dp1[1] = 1;
    dp2[1] = 1;
    for(int i = 2; i < dp1.length; i++){
        dp1[i] = dp2[i-1];
        dp2[i] = dp1[i-1]+dp2[i-1];
    }
    return dp1[n]+dp2[n];
}
```

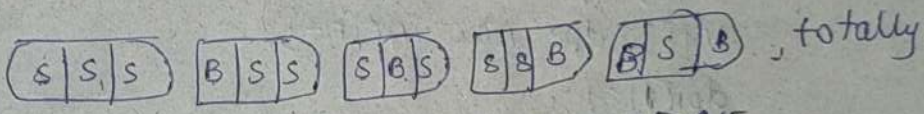
13. Arrange Buildings

Sample Input	Sample Output	Constraints
6	441	$0 <= n <= 45$

Steps:



For n=3,
on one side of the road there can 5 combinations



, totally on both sides of the road = 5x5 = 25

a is ~~sing~~ same as Count Binary Strings.
(sum of last two values of Ending with 0 and Ending with 1).

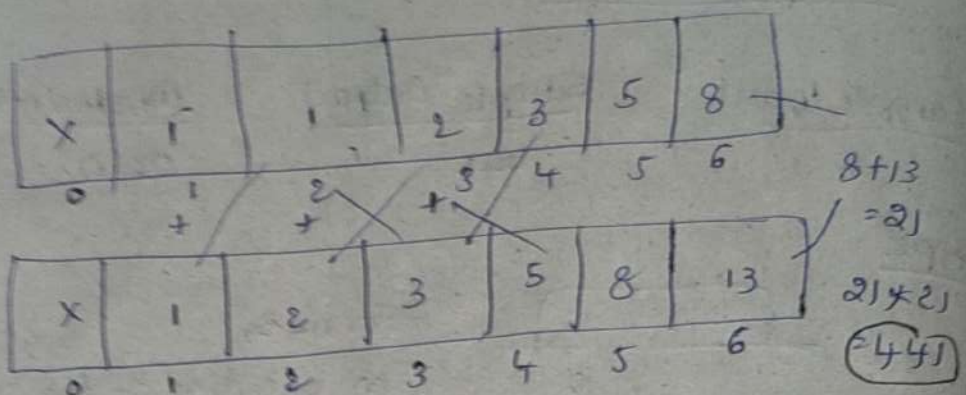
Tabulation:

1. Take 2 dp Arrays of n+1.
2. One DP Array represents Block with Building and another DP Array Represents Block with space.
3. Store dp1(0) = 1 (blocks ending with building
   dp2(0) = 1 (blocks ending with space)
4. Store dp1(i) = dp2(i-1)
   Store dp2(i) = dp1(i-1) + dp2(i-1)
5. (Return the sum of two values of Arrays) * a
6. Return a * a

Recursion / Memoization

From Tabulation, we can observe that (Fibonacci(n+3) * (Fibonacci(n+3)) gives the solution.

Dry Runs:



Program:

```
public static int count(int n){
    int[] dp1 = new int[n+1];
    int[] dp2 = new int[n+1];
    dp1[1] = 1;
    dp2[1] = 1;
    for(int i=2; i<dp1.length; i++){
        dp1[i] = dp2[i-1];
        dp2[i] = dp1[i-1] + dp2[i-1];
    }
    int a = dp1[n] + dp2[n];
    return a*a;
}
```

// take long instead of int for handling bigger values

14. Count Encodings

Sample Input	Sample Output	Constraints
123	3	0<Str.length<=10

Steps:

Recursion / Memoization:

1. Consider the current character of a string, apply Recursive call.
2. If the length of the string is greater than current idx+1, and the two characters less than 26, Apply the Recursive call.
3. Return the sum of two recursive calls.

4. If the starting character is 0, then return 0.

5. If we reach end of the string, return 1.

Tabulation:

1. Take a 1D array of length = s.length() + 1

2. Assign the first two values of dp Array = 1
   [Single character will give 1 encoding] & Empty is possible gives 1 encoding

3. Take two variables current term depends on i-1
   previous term on i-2

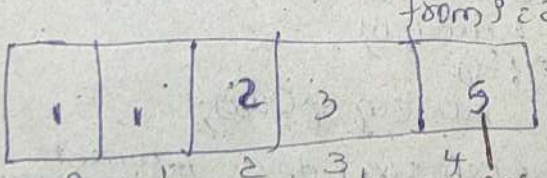
4. Apply 4 cases: i) both pc and cc = 0, then dp(i) = 0
   ii) if pc == 0, then dp(i) = dp(i-1)
   iii) if cc = 0, then check the two characters lies in range of 26, then dp(i) = dp(i-2)
   iv) Assign dp(i) = dp(i-1), check the last two characters in the range of 26, if modify dp(i) += dp(i-2);

5. Return the dp[dp.length] dp(i) → count of encodings from 1 to i-1.

Dry Run:

Str → 1 1 2 3

①



from i = 2,
cc = (i-1) - '0' = 1
pc = (i-2) - '0' = 1

so 4th case;
dp(i) = dp(i-1) = 1
10 + 1 < = 26 ✓
dp(i) = 1 + 1 = 2

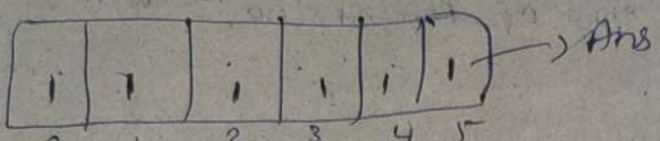
i = 3 ✓ cc = 2
pc = 1
4th case; 1 x 10 + 2 = 12 < = 26,
dp(i) = 2 + 1 = 3

i = 4, cc = 3
pc = 2 4th case, dp(i) = 3
2 x 10 + 3 < = 26
dp(i) = 3 + 2 = 5

(2)

0	1	2	3	4
1	0	e	0	3

str :



for i=2, cc=0, pc=1 — 3th case
dp(i) = 1 × 10+0 <= 26
dp(i) = dp[i-2] = 1

for i=3, cc=2, pc=0 — 2nd case
dp(i) = dp(i-1) = 1

i=4, cc=0, pc=2 — 3rd case
20+0 <= 26
dp(i) = dp[i-2] = 1

i=5, cc=3, pc=0 — 2nd case
dp(i) = dp(i-1) = 1

Program;

Recursion / Memoization

```
count(str, 0, new int[str.length()+1])

public static int count(String str, int idx, int[] dp){
    if(idx == str.length())
        return 1;
    if(str.charAt(idx) == '0')
        return 0;
    if(dp[idx] != 0)
        return dp[idx];
    int f1 = count(str, idx+1, dp);   // current character included
    int f2 = 0;
    if(idx+1 < str.length() && Integer.parseInt(str.
        substring(idx, idx+2)) <= 26){   // considering next two characters
        f2 = count(str, idx+2, dp);
    }
    int ans = f1+f2;
    dp[idx] = ans;
    return ans;
}
```

Tabulation

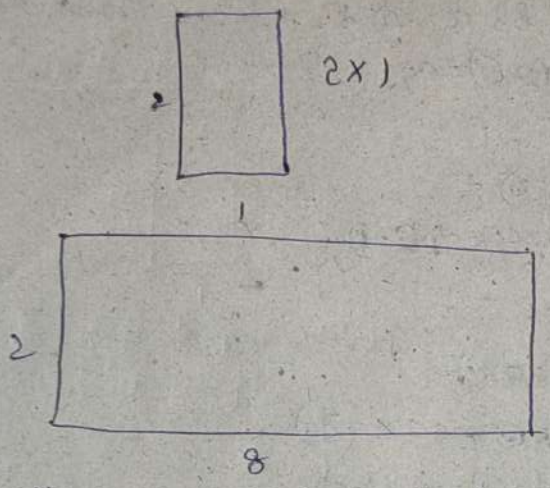
```
public static int countTab(String str){
    int[] dp = new int[str.length()+1];
    dp[0] = dp[1] = 1;
    for(int i=2; i<dp.length; i++){
        int cc = str.charAt(i-1) - '0';
        int pc = str.charAt(i-2) - '0';
        if(pc==0 && cc==0){
            dp[i] = 0;
        }
        else if(pc==0){
            dp[i] = dp[i-1];
        }
        else if(cc==0){
            if((pc*10)+cc<=26)
                dp[i] = dp[i-2];
        }
        else{
            dp[i] = dp[i-1];
            if((pc*10)+cc<=26)
                dp[i] += dp[i-2];
        }
    }
}
```

15. Tiling with 2x1 tiles

Sample Input	Sample Output	Constraints
8	34	1 <= n <= 100

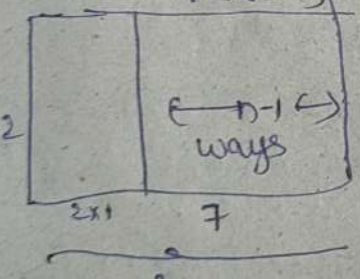
Steps:

According to problem statement, 2*n board should be filled with 2x1 tiles in how many ways?

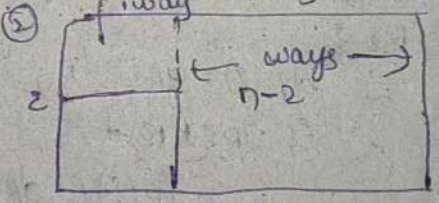


The 2x1 tile can be filled in two way

① vertically



② filled in horizontal way



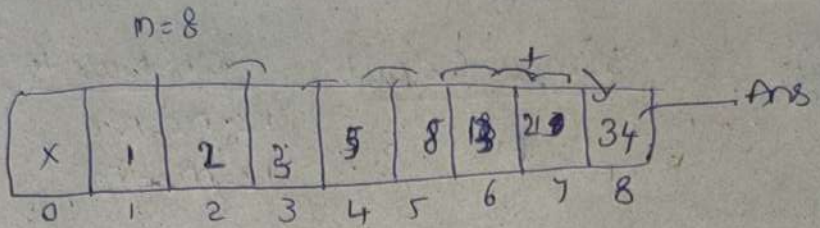
Recursion + Memoization:

1) Make two recursive calls of n-1 and n-2, Add the sum.
2) If n==1, only one way can fill n, n==2, can fill in 2 ways, n<=0 return 0
3) Apply Memoization

Tabulation:

1) Take a dp Array and fill dp[1]=1, dp[2]=2
2) dp[i]=dp[i-1]+dp[i-2]
3) Return dp[n].

Dry Run:



Program: Recursion-Memoization

```
tileR(int n, new int(n+1]);
public static int tileR(int n, int[] dp) {
    if (n<=0)
        return 0;
    if (n==1)
        return 1;
    if (n==2) return 2;
    if (dp[n]!=0)
        return dp[n];
int ans= tileR(n-1, dp) + tileR(n-2, dp);
dp[n] = ans;
return ans;
}
```

Tabulation

```
public static int tiler(int n, int[] dp) {
    dp[1]=1;
    dp[2] = 2;
for (int i=3; i<=n; i++)
    dp[i]=dp[i-1]+dp(i-2);

return dp[n];
}
```

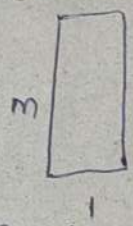
16. Tiling with m*1 Tiles

Sample Input:	Sample Output	Constraints
39 – n	61	1 <= n <= 100
16 – m		1 <= m <= 50

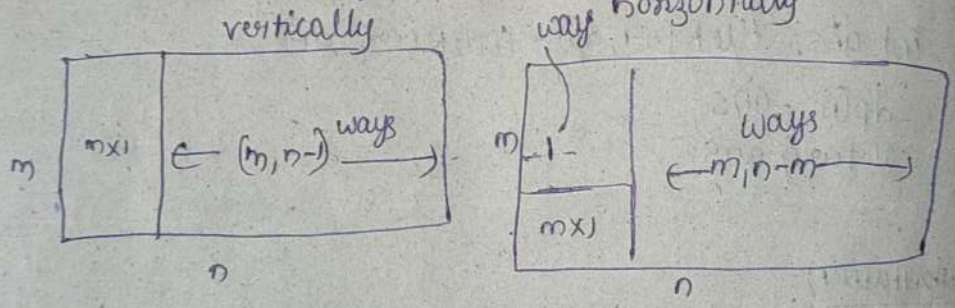
Steps:

According to Problem statement, m*n Floor should be filled with m*1 tiles; in how many ways?

Recursion / Memoization:

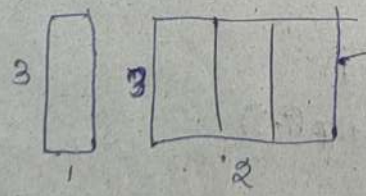


m×1 can be filled in 2 ways



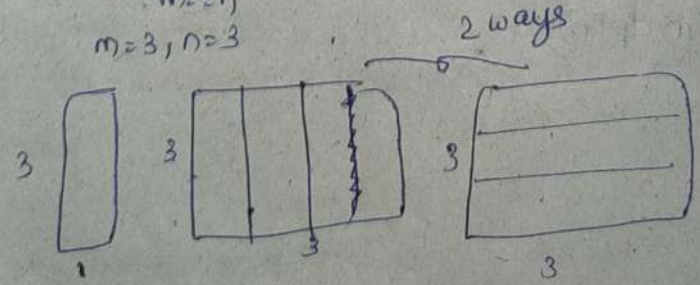
Spl cases

n < m, n = 2, m = 3



m = n

m = 3, n = 3



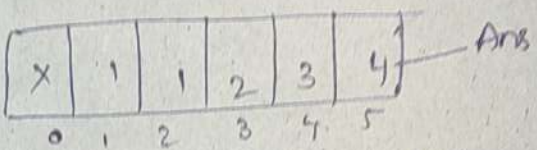
1) Apply Recursive calls, and add the sum
2) The Recursive calls are (m, n-1) & (m, n-m)
3) write the conditions for spl cases
4) Apply Memoization.

Tabulation:

1. Iterate through the dp Array from i=1
2. Check for the spl cases else store dp(i) = dp[i-1] + dp[i-m];
3. Return dp(n);

Dry Run:

n=5
m=3



i=1, 1<m, dp(i)=1
i=2, 2<m, dp(i)=1
i=3, 3≥m, dp(i)=2
i=4, dp(i)=2+dp(1)=3
i=5, dp[i]=3+dp(2)=1

Program; Recursion/Memoization

```
tilesR(m,n, new int[n+1])
public static int tilesR(int m, int n, int[] dp) {
    if(n<=0)
        return 0;
    // spl cases
    if(n<m)
        return 1;
    if(n==m)
        return 2;
    if(dp[n]!=0)
        return dp(n);
```

```
int ans = tilesR(m, n-1, dp) + tilesR(m, n-m, dp);
    dp[n] = ans;
    return ans;
}

// Tabulation
public static int tilesT(int m, int n, int[] dp) {
    for (int i = 1; i <= n; i++) {
        if (i < m)
            dp[i] = 1;
        else if (i == m)
            dp[i] = 2;
        else
            dp[i] = dp[i-1] + dp[i-m];
    }
    return dp[n];
}
```

17. Paint House

Sample Input

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

Sample Output

```
8
```

Constraints

All costs are positive integers.

Steps; Tabulation

1) According the problem statements, houses should be painted with any of three colors, but adjacent houses should not be colored with same color.

2) First house can be coloured with 3 colors so, copy the same in dp Array.

3) From second house, color with other than consecutive color, take the min value and add the cost.

4) Continue the process

5) Last Row Minimum value returns the minimum no of colors Required.

Dry Run:

	Red	Blue	Green
H1	1	5	7
H2	5	8	4
H3	3	2	9
H4	1	2	4

dp Array

		min
1	5	7
5+1=10	9	5
8	7	18
8	10	11

dp[i][j] → Min cost to paint i. numbers of houses & no two adjacent have same color and the house is painted with jth color

dp[i][0] = cost[i][0] + Math.min(dp[i-1][1], dp[i-1][2])

Silly continues

last row: 8 10 11 → Min = 8 → Ans

Program:

```
public static int minCost(int[][] costs) {
    int[][] dp = new int[costs.length][costs[0].length];
    dp[0][0] = costs[0][0];
    dp[0][1] = costs[0][1];
    dp[0][2] = costs[0][2];
    for (int i = 1; i < costs.length; i++) {
        dp[i][0] = costs[i][0] + Math.min(dp[i-1][1],
                   dp[i-1][2]);
        dp[i][1] = costs[i][1] + Math.min(dp[i-1][0],
                   dp[i-1][2]);
        dp[i][2] = costs[i][2] + Math.min(dp[i-1][0],
                   dp[i-1][1]);
    }
}
```

return Math.min(dp[dp.length-1][0], Math.min(dp[dp.length-1][1], dp[dp.length-1][2]));

}

}

18. Maximum Sum Non Adjacent Elements

Sample Input	Sample Output	Constraints
6 → n 5 10 10 100 5 6	116	1<=n<=1000 -1000<=n1, n2... — n elements <=1000

Steps;

Recursion | Memoization

1. Make two recursive calls
   a) Don't consider current Element
   b) Consider current element and make Recursive call with idx+2.
2. Take the max of two Recursive calls and Apply Memoization.

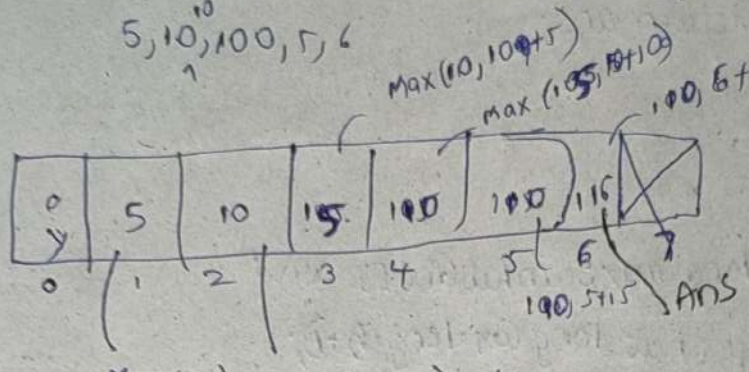
Tabulation;

1. Take a dp Array of arr.length+1.
2. Store the max of arr[0], 0 in dp[1] →
   Why? (If negative element occurs: so considering 0)
3. From i=2, dp[i] = Store max of dp[i-1], arr[i-1]+dp[i-2]
4. Return the last index

Dry Run:

dp[i] → Maximum sum of non-adjacent elements till ith index.

5, 10, 100, 5, 6

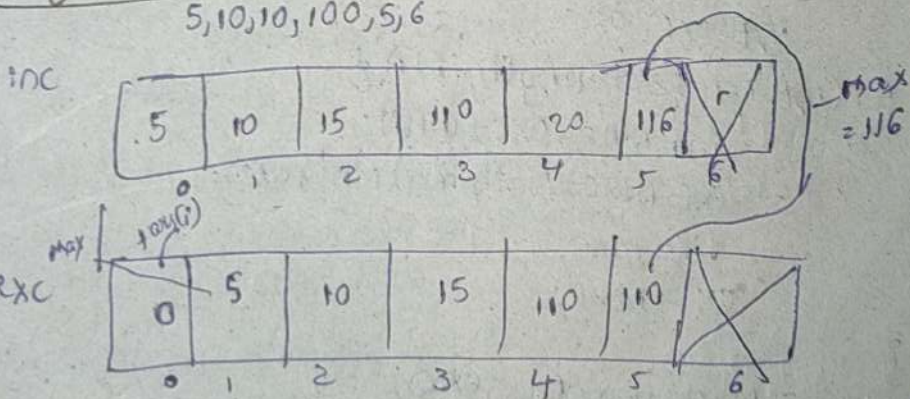


2nd Approach Tabulation

1. Take 2 variables including and excluding
   Initially inc = arr[0]
   exc = 0
2. Modify the inc, exc variables with inc = exc + arr[i]
   exc = Max of (previous inc, exc)
3. At last, return the max value of inc, exc.

Dry Run; understand with 2 Inc and Exc Arrays

5, 10, 10, 100, 5, 6



Programs; Recursion | Memoization

```
public static int maxSumR(int[] arr, int idx, int[] dp) {
        if (idx >= arr.length)
            return 0;
        if (dp[idx] != 0)
            return dp[idx];
        int f1 = 0 + maxSumR(arr, idx+1, dp);   // current element not considered
        int f2 = arr[idx] + maxSumR(arr, idx+2, dp);   // current element considered
        int max = Math.max(f1, f2);
```

new int[arr.length+1]

```
        dp[idx] = ans;
        return ans;
}
```

Tabulation-1

```
public static long maxSumTab(int[] ar) {
    long[] dp = new long[ar.length+1];
    dp[1] = Math.max(ar[0], 0);

    for(int i=2; i<dp.length; i++) {
        dp[i] = Math.max(dp[i-1], ar[i-1] +
                dp[i-2]);
    }
    return dp[dp.length-1];
}
```

Tabulation-2

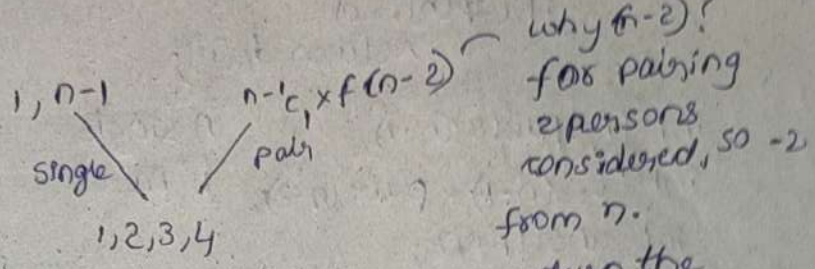
```
public static int maxSumTab2(int[] ar) {
    int inc = ar[0];
    int exc = 0;
    for(int i=1; i<ar.length; i++) {
        int ninc = exc + ar[i];
        int nexc = Math.max(inc, exc);
        inc = ninc;
        exc = nexc;
    }
    return Math.max(inc, exc);
}
```

19. Friends Pairing

Sample Input	Sample Output	Constraints
4	10	$0 < n < 20$

Steps:

1. Recursion / Memoization

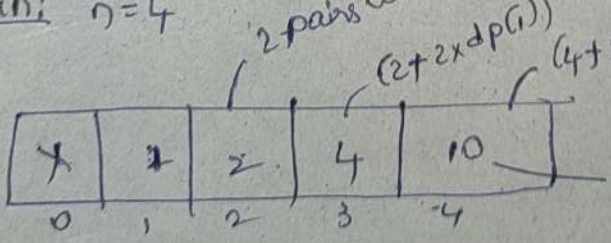


1. Apply the two recursive calls and return the sum.
2. Apply Memoization.

Tabulation:

1) From Recursion, we observe n is varying so take dp Array of n+1.
2) dp[1] = 1 [1 person will be only single]
   dp[2] = 2 [either they can pair up or they can stay alone]
3) dp[i] = ((i-1) * dp[i-2]) + dp[i-1];

Dry Run: n = 4



Program;

Recursion, Memoization

```
public static int pair(int n, int[] dp) {
    if(n==0)
        return 1;
    if(n<0)
        return 0;
    if(dp[n]!=0)
        return dp[n];

int f1 = pairs(n-1);          // single
int f2 = (n-1) * pair(n-2);   // pairs
int ans = f1+f2;
  dp[n] = ans;
return ans;
}
```

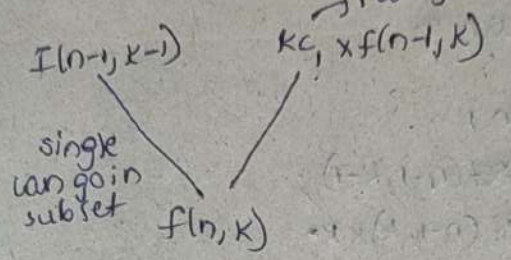
Tabulation

```
public static int pairs(int n) {
    int[] dp = new int[n+1];
    dp[1] = 1;
    dp[2] = 2;
    for(int i=3; i<=n; i++) {
        dp[i] = dp[i-1] + (i-1) * dp[i-2];
    }
return dp[n];
}
```

20. Partition into Subsets

Sample Input	Sample Output	Constraints
4 - n	6	0 <= n <= 20
3 - K		0 <= K <= n

Steps:



Recursion:

1) Apply the Recursive calls (n-1, k-1), (n-1, k) * k and add the result
2) Return

Tabulation:

1) Take a 2D Array as, n and k are varying
2. Go through dp, j to i, i<=k, if i == j ; store 1
else dp[i][j] = dp[i-1][j-1] + j * dp[i-1][j];
3. Return dp[n][k];

Dry Run;

n=4, k=3

	0	1 j	2	3
0	0	0	0	0
1 i	0	1	0	0
2	0	1	1	0
3	0	1	3	1
4	0	1	7	6

+ 3*2

i == j = 1

dp(1)(2) = dp[0][1] + 2 * dp[0][2]
= 0 + 2 * 0 = 0

dp[2][1] = dp[1][0] + 1 * dp[1][1]

dp[2][3] = dp[1][3] + 3 * dp[2][3]
= 0 + 3 * 1

dp[3][1] = dp[2][0] + 1 * dp[2][1]

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

Solution

dp[4][2] = 1 + 2 * 3 = 7

dp[4][3] = 3 * 3 * 1 = 6

Programs:

Recursion

```
Public static long partitionKSubset(int n, int k) {
        if(k>n)
            return 0;
        if(n==0 || k<=0)
            return 0;
        if(n==k)
            return 1;
    long f1 = partitionKSubset(n-1,k-1);
    long f2 = partitionKSubset(n-1, k)*k;
    long ans = f1+f2;
    return ans;
}
```

Tabulation

```
public static long partitionKSubsetT(int n, int k) {
    long dp[][] = new long[n+1][k+1];
    for(int i=1; i<dp.length; i++) {
        for(int j=1; j<=i && j<=k; j++) {
            if(i==j)
                dp[i][j]=1;
            else
                dp[i][j]=dp[i-1][j-1]+j*dp[i-1][j];
        }
    }
    return dp[n][k];
}
```

81. Paint Fence

Sample Input	Sample Output	Constraints
8	3672	1 <= n <= 10
3		1 <= K <= 10

Steps: (Not more than 2 consecutive fences with the same color).

n=1: r, g, b

n=2: rr (Not valid), rb, rg, gr, gg (Not valid), gb, br, bg, bb (Not valid)

Intially same[0] = 0
diff[0] = K
i = 1 to n,
same[i] = diff[i-1]
diff[i] = (same[i-1] + diff[i-1]) * (k-1);
Return the sum of last values of two Arrays

Dry Run: same[i] - last 2 fences having same color
diff[i] - last 2 fences having different colors.

	n=1	n=2
same	0	rr, bb, gg (3)
diff	r, b, g	rg, rb, gr, gb, br, bg (6)

n=4, K=3

	0	1	2	3	4
same	-	3	6	18	48
diff	3	6	18	48	132

(0+3)*2 = (3+6)*2 (6+18)*2 (18+48)*2

Ans: 48+132 = 180

Program:

```
public static long fence(int n, int k) {
    long same[] = new long[n];
    long diff[] = new long[n];
    diff[0] = k;
    for(int i=1; i<n; i++){
        same[i] = diff[i-1];
        diff[i] = (same[i-1] + diff[i-1]) * k-1;
    }
    return same[n-1] + diff[n-1];
}
```

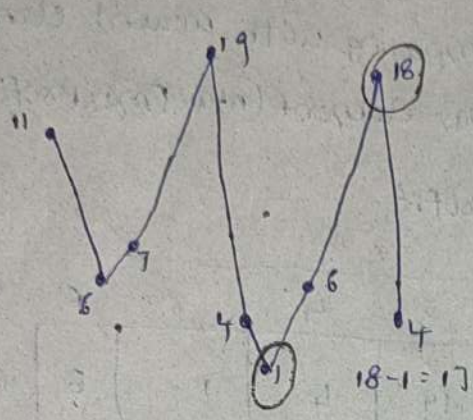
22. Buy And Sell Stocks - One Transaction Allowed

Sample Input

9 → elements
11
6
7
19
4
1
6
18
4

Sample Output

17



18 - 1 = 17

Constraints

$0 <= n1, n2 \ldots <= 100$

one transaction allowed
Buy at Minum Rate,
Sell at Maximum Rate

Max Profit

Steps;

Two Approaches;

1) Right-left Approach

a) Assign maxSofar = arr(n-1), maxProfit = 0

b) Go through i = n-2; i > 0; i--, if the Array element is greater than current maxSofar, modify

c) Modify the maxProfit with maxProfit, maxSofar - arr(i);

Dry Run;

0	1	2	3	4	5	6	7	8
11	6	7	19	4	1	6	18	4

maxSofar = arr(n-1) = 4
maxProfit = 0

i = 7
18 > 4, maxsofar = 18
maxprofit = max(0,0) = 0

i = 6
6 > 18 ✗, mf = 18
mp = max(0,12) = 12

i = 5
1 > 18 ✗ mf = 18
mp = max(12,17) = 17

i = 4
4 > 18 ✗, mf = 18
mp = max(17,14) = 17

i = 3
19 > 18, mf = 19
mp = max(17,1) = 17

i = 2
7 > 19 ✗ mf = 19
mp = max(17,12) = 17

i = 1
6 > 18 ✗ mf = 19
mp = max(17,13) = 17

i = 0
11 > 18 ✗, mf = 19
mp = max(17,8) = 17

maxProfit = 17

2nd Method: From left to Right

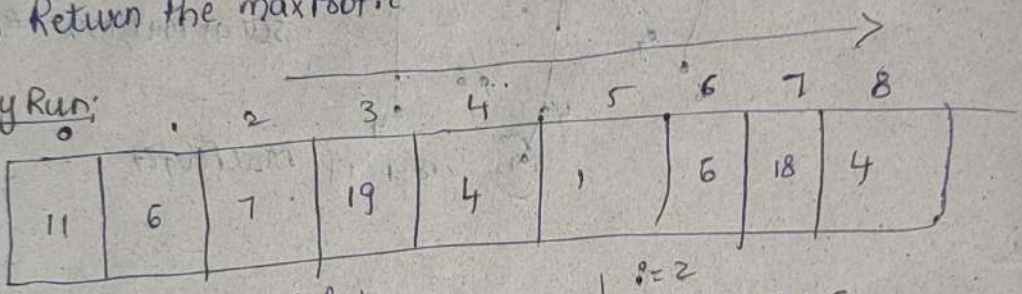
a) Assign minSofar = ar(0)

b) MaxProfit = 0

c) Go from i = 0 to end of Array, modify the minsofar by comparing with curent element, modify the max element (max(maxProfit, ar(i) - minsofar))

d) Return the maxProfit

Dry Run:



minsofar = ar(0) = 11
mp = 0

i = 1
mf = min(6,11) = 6
mp = max(0,6) = 6

i = 2
mf = min(6,7) = 6
mp = max(6,1) = 6

i = 3
mf = min(19,6) = 6
mp = max(6,13) = 13

i = 4
mf = min(4,6) = 4
mp = max(13,0) = 13

i = 5
mf = min(4,1) = 1
mp = max(13,0) = 13

i = 6
mf = min(1,6) = 1
mp = max(13,5) = 13

i = 7
mf = min(1,8) = 1
mp = max(13,17) = 17

i = 8
mf = min(1,4) = 1
mp = max(17,3) = 17 ← Ans

Program:

```
// Right to left
public static int rightStack(int[] ar) {
        int maxSofar = ar[n-1];
        int maxProfit = 0;
    for(int i = n-2; i >= 0; i--) {
            if(ar[i] > maxsofar)
                maxsofar = ar[i];
            maxProfit = Math.max(maxProfit,
                                 maxSofar - ar[i]);
    }
    println(maxProfit);
}
```

// left stock

```
pubalic static int leftStock (int[] ar) {
        int minSofar=ar[0];
        int maxProfit=0;
        for(int i=0; i<ary.length; i++) {
            minSofar=Math.min(minSofar, ar[i]);
            maxProfit=Math.max(maxProfit, ar[i]-minsofar);
        }
        return maxProfit;
}
```

23. Buy And Sell Stocks-Two Transactions Allowed

Sample Input	Sample Output	Constraints
9 11 6 7 19 4 1 6 18 4	30	0 <= n <= 20 0 <= n1, n2 ... <= 100

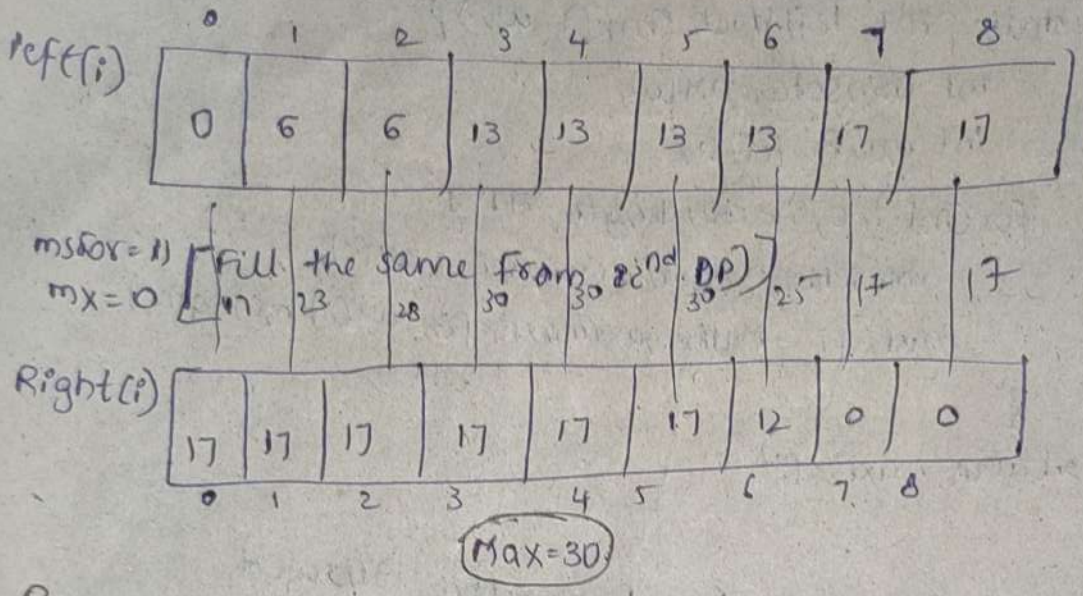
Steps:

1. Two Transactions are Allowed .. One should Be Sold, Before Buying Another.
2. Take Two Arrays. One; left stock max
   Two; right stock max
3. Get the max value of left(i)+right(i).
4. Return the max value.

left(i) -> Max Profit till ith by completing atmast 1st Transaction

right(i) -> i to end. at most profit by completing atmost 1st Transaction.

Dry Run; Array: 11, 6, 7, 19, 4, 1, 6, 18, 4



Program;

```
int[] left = new int(n);
int minSofar = ar(0);
int max1 = 0;
for(int i=0; i<n; i++){
        minSofar = Math.min(minSofar, ar(i));
        max1 = Math.max(max1, ar(i)-minSofar);
        left(i) = max1;
}
int[] right = new int(n);
int maxSofar = ar(n-1);
int max2 = 0;
for(int i = n-1; i>=0; i--){
        maxSofar = Math.max(maxSofar, ar(i));
        max2 = Math.max(max2, maxSofar - ar(i));
        right(i) = max2;
}
int ans = 0;
for(int i=0; i<n; i++){
        ans = Math.max(ans, left(i)+right(i));
}
println(max);
}
```

24. Buy and sell stocks - K Transactions Allowed

Sample Input	Sample Output	Constraints
6 - n	5	0 <= n <= 20
9		0 <= n1, n2.. <= 10
6		0 < K <= n/2
7		
6		
3		
8		
, - K		

Steps:

1. Take a 2D Array of length, rows = K+1, cols = arr.length
2. Go through the dp Array, Get max Element initially for each row, dp[i][0] - arr[0]
3. Assign dp[i][j] = Math.max(dp[i][j-1], max + arr[j]);
4. Modify max = Math.max(max, dp[i-1][j] - arr[j])
5. Return last value of 2D Array.

dp[i][j]: i no of transactions, j th index

i : no of transactions (i-1) th day

j th transaction will be completed today : (i-1) no of transactions on K th day?

Dry Run; Array: [9 6 7 6 3 8], K = 3 → price of stock

No of transactions ↓	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	1	1	1	5
2	0	0	1	1	1	6
3	0	0	1	1	1	6

Ans

For i = 1 : max = dp[i][0] - arr[0] = 0 - 9 = -9

j = 1 < 6, dp[i][j] = max(dp[i][j-1], max + arr[j])

= max(0, -9 + 6) = 0

Modify max = max(-9, 0 - 6) = -6

j = 2 < 6, dp[i][j] = max(0, -6 + 7) = 1

max = max(-6, -7) = -6

$j=3<6$, $dp(i)(j) = \max(1, -6+6) = 1$

$\max = \max(-6, -6) = -6$

$j=4<6$, $dp(i)(j) = \max(1, -3) = 1$

$\max = \max(-6, -3) = -3$

$j=5<6$, $dp(i)(j) = \max(1, 5) = 5$

$\max = \max(-3, -8) = -3$

For $i=2$, $\max = 0-9 = -9$

$j=1<5$, $dp(i)(j) = \max(0, -9+6) = -3$

$\max = \max(-9, -6) = -6$

$j=2<6$, $dp(i)(j) = \max(0, 1) = 1$

$\max = \max(-6, -6) = -6$

$j=3<6$, $dp(i)(j) = \max(1, 0) = 1$

$\max = \max(-6, -5) = -5$

$j=4<5$, $dp(i)(j) = \max(1, -2) = 1$

$\max = \max(-5, -2) = -2$

$j=5<6$, $dp(i)(j) = \max(1, 6) = 6$

$\max = \max(-2, -3) = -2$

For $i=3$, $\max = 0-9 = -9$

$j=1<6$, $dp(i)(j) = \max(0, -9+6) = 0$

$\max = \max(-9, -6) = -6$

$j=2<6$, $dp(i)(j) = \max(0, 1) = 1$

$\max = \max(-6, -6) = -6$

$j=3<6$, $dp(i)(j) = \max(1, 0) = 1$

$\max = \max(-6, -5) = -5$

$j=4<6$, $dp(i)(j) = \max(1, -2) = 1$

$\max = \max(-5, -2) = -2$

$j=5<6$, $dp(i)(j) = \max(1, 6) = 6$

$\max = \max(-2, -2) = -2$

Program:

```
public static int maxK(int[] ar, int k){
    int[][] dp = new int[k+1][ar.length];

    for(int i=1; i<dp.length; i++){
        int max = dp[i][0] - ar[0];

        for(int j=1; j<dp[0].length; j++){
            dp[i][j] = Math.max(dp[i][j-1], max+ar[j]);
            max = Math.max(max, dp[i-1][j] - ar[j]);
        }
    }
```

25. Buy and sell Stocks - Infinite Transactions Allowed

Sample Input	Sample Output	Constraints
9 → days	30	0 <= n <= 20
11		0 <= n1, n2 ... <= 100
6		
7		
19		
4		
1		
6		
18		
4		

Steps:

Greedy Approach:

1. Initially Assign profit = 0
2. Iterate through Array : if(i+1) > i, then
   modify profit: ar(i+1) - ar(i)
3. Return Profit.

Dry Run:

0	1	2	3	4	5	6	7	8
11	6	7	19	4	1	6	18	4

mp=0, i=0, 6 > 11 ✗
i=1, 7 > 6, mp=0+7-6=1
i=2, 19 > 7, mp=1+19-7=13
i=3, 4 > 19 ✗
i=4, 1 > 4 ✗
i=5, 6 > 1, mp=13+5=18
i=6, 18 > 6, mp=18+12=30
4 > 18 ✗

Ans

Bought state - Sold state

1. Take two arrays of size n [Buy and sell]
2. Initially Assign buy(0) = -arr(0) , sell(0) = 0
3. Modify the sell(i), buy(i) by iterating through loop.

sell(i) = Math.max(sell(i-1), buy(i-1) + arr(i)); [Max profit till ith idx while remaining in sold state]

buy(i) = Math.max(buy(i-1), sell(i-1) - arr(i)); [max profit till ith idx, while remaining in buy state]

4. Return last value of sell Assay.

Dry Run; Assay: 11 6 7 19 4 1 6 18 4

	0	1	2	3	4	5	6	7	8
buy	-11	-6	-6	-6	9	12	12	12	26
sell	0	0	1	13	13	13	18	30	30

Ans

Program: //Greedy Approach

```
public static int greed(int[] arr) {
    int profit = 0;
    for(int i=0; i < arr.length-1; i++) {
        if(arr(i+1) > arr(i))
            profit += arr(i+1) - arr(i);
    }
    return profit;
}
```

//Buy And sell State

```
public static int State(int[] arr) {
    int[] buy = new int(arr.length);
    int[] sell = new int(arr.length);
    buy(0) = -arr(0);
    sell(0) = 0;
```

```
for (int i = 1; i < arr.length; i++) {
    sell[i] = Math.max(sell[i-1], buy[i-1] + arr[i]);
    buy[i] = Math.max(buy[i-1], sell[i-1] - arr[i]);
}
return sell[sell.length-1];
}
```

29. Buy and Sell Stocks with Transaction Fee - Infinite Transaction Allowed

Sample Input	Sample Output	Constraints
12 - days 10 15 17 20 16 18 22 20 23 25 8 - fee	13	0 <= n <= 20 0 <= n1, n2 ... <= 100 0 <= fee <= 5

Steps:

1. Similar to 25th Problem, when you sell the transaction Modify by deducting fee

```
sell[i] = Math.max(sell[i-1], buy[i-1] + arr[i] - fee);
```

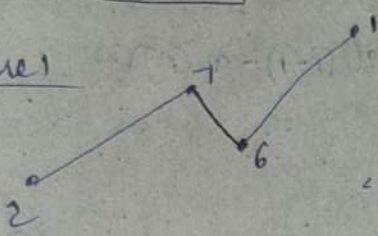
Dry Run: Array: 11 6 7 19 4 1 6 18 4 , fee = 2

	0	1	2	3	4	5	6	7	8
buy	-11	-6	-6	-6	7	10	10	10	22
sell	0	0	0	11	11	11	14	26	26

(Ans

Why Not Greedy?

1) Case1



with fee = 2

(7-2-2) + (11-6-2) = 6
(individual transactions)

singletrans: 11-2-2=7 (7≠6)

2) Case

(4-3-2) = -1 ∝ not useful profit

Profit:

```
public static int state(int[] ar, int fee) {
    int[] buy = new int[ar.length];
    int[] sell = new int[ar.length];
    buy[0] = -ar[0];
    sell[0] = 0;
    for(int i=1; i<ar.length; i++) {
        sell[i] = Math.max(sell[i-1], buy[i-1] +
                           ar[i] - fee);
        buy[i] = Math.max(buy[i-1], sell[i-1] - ar[i]);
    }
    return sell[sell.length-1];
}
```

27. Buy and sell Stocks with Cooldown - Infinite Transaction Allowed

Sample Input	Sample Output
12 10 15 17 20 16 18 22 20 23 25	19

Constraints

0 <= n <= 20

0 <= n1, n2, ... <= 100

Steps:

1. Same as 25th Problem, Modify the sell(i)

if 1 cool down : sell(i-2) - arr(i)

2 cool down : sell(i-3) - arr(i)

kth cool down : sell(i-k-1) - arr(i)

In Problem state, cooldown = 1

Dry Run:

Array: 18 6 19 4 1 6 18 4

	0	1	2	3	4	5	6	7	8
buy	-18	-6	-6	-6	-3	12	12	12	14
sell	0	0	1	13	13	13	18	30	30

Ans

Program:

```
public static int state(int[] arr) {
    int[] buy = new int[arr.length];
    int[] sell = new int[arr.length];
    buy[0] = arr[0];
    sell[0] = 0;
    for(int i=1; i<arr.length; i++) {
        sell[i] = Math.max(sell[i-1], buy[i-1] + arr[i]);
        if(i-2 >= 0)
            buy[i] = Math.max(buy[i-1], sell[i-2] - arr[i]);
        else
            buy[i] = Math.max(buy[i-1], 0 - arr[i]);
    }
    return sell[sell.length-1];
}
}
```

28. Longest Increasing Subsequence

Sample Input	Sample Output	Constraints
10 10 22 9 33 21 50 41 60 80	6	$0 <= n <= 20$ $0 <= n1, n2 \ldots <= 100$

Steps:

Recursion-1

1. Use, :lse, index of last selected element ~~in Longest sequence~~
   Initially -1
2. If the ~~current~~ next element is not considered make a recursive call.
3. If the next element is considered, make a Recursive call. Before I check whether it is greater than the :lse)
4. Get max from Two Recursive calls.
5. Apply Memoization

Recursion-2:

1. Iterate in Reverse Order. and assign max=0
2. Check ars(i) < ars(idx)
3. If yes, make a Recursive call, Math.max(max, 1+stR(ars, i));
4. Return 1+max.

Tabulation:

1. Take a dp Array of length = ar.length
2. Make dp[0] = 1 As single Element sequence is possible of length 1.
3. Iterate through array with max = 0
4. Initially Assign dp[i] = 1 [since every element will have one sequence ending with that element.
5. Now iterate through j = i-1 to j >= 0,
   compare ar[j] < ar[i], if yes modify
   Max; max = max (max, dp[j])
6. Get the Max Value of dp Array → length of longest sequence.

dp[i] → stores ~~longest~~ increasing sequence ending with ar[i] element

Array:

0	1	2	3	4	5	6	7	8	9
10	22	9	33	21	50	41	60	80	1

DpArray

1	2	1	3	2	4	4	5	6	1

dp[0] = 1 [10 is possible)

for i = 1, max = 0 [initially)

dp[1] = 1

$\begin{bmatrix} -22 \\ 10 & 22 \end{bmatrix}$

j = 0 >= 0,

10 < 22 ✓

max = max (0, 1) = 1

dp[1] = 1 + 1 = 2

For i=2, max=0 initially

dp(2)=1

1>=0

22<9 ✗

0>=0

10<9 ✗

dp[2] = 1+0 = 1

(9)

For i=3, max=0

dp(3)=1

2>=0

9<33 ✓

max=(0,1)=1

1>=0 ✓

22<33 ✓

max=(1,2)=2

0>=0

10<33 ✓

max=(2,1)=2

dp(3)=1+2=3

$\begin{bmatrix} 10 \\ 22 \\ 33 \end{bmatrix}$

For i=4, max=0 initially

dp(4)=1

3>=0

33<21 ✗

2>=0 9<21 ✓

max=max(0,1)=1

22<21 ✗

10<21 ✓

max=max(1,1)=1

dp(4)=1+1=2

$\begin{bmatrix} 10 \\ 21 \end{bmatrix}$

For i=5, max=0 initially

dp(5)=1

4>=0 21<50

max(0,1)=2

3>=0, 33<50

max(2,3)=3

2>=0, 9<50

max(3,1)=3

1>=0, 22<50

max(3,2)=3

0>=0, 10<50

max(3,1)=3

dp(5)=1+3=4

$\begin{bmatrix} 10 \\ 22 \\ 33 \\ 50 \end{bmatrix}$

For i=6

dp(6) = 1

5 >= 0, 50 < 41 ✗

4 >= 0, 21 < 41 ✓

max(0, 2) = 2

3 >= 0, 33 < 41

max(2, 3) = 3

2 >= 0, 9 < 41

max(3, 1) = 3

1 >= 0, 22 < 41

max(3, 2) = 3

0 >= 0, 10 < 41

max(3, 1) = 3

dp(6) = 1 + 3 = 4 [10, 22, 33, 41]

For i=7

dp(7) = 1

6 >= 0, 41 < 60

max(0, 4) = 4

5 >= 0, 50 < 60

max(4, 4) = 4

4 >= 0 21 < 60

max(4, 2) = 4

3 >= 0, 33 < 60

max(4, 3) = 4

2 >= 0, 9 < 60

max(4, 1) = 4

1 >= 0, 22 < 60

max(4, 2) = 4

0 >= 0, 10 < 60

max(4, 1) = 4

dp(7) = 1 + 4 = 5 [10, 22, 33, 50, 60]

For i=8

dp[8] = 1

7 >= 0, 60 < 80

max(0, 5) = 5

6 >= 0, 41 < 80

max(5, 4) = 4

5 >= 0, 50 < 80

max(5, 4) = 4

4 >= 0, 21 < 80

max(5, 2) = 5

3 >= 0, 33 < 80

max(5, 3) = 5

2 >= 0, 9 < 80

max(5, 1) = 5

1 >= 0, 22 < 80

max(5, 2) = 5

0 >= 0, 10 < 80

max(5, 1) = 5

dp[8] = 1 + 5 = 6

For i=9

1 is smaller

dp(9) = 1 + 0 = 1

Max Element in dp Array = 6

1

Ans

Program:

```
lisER(ar1, 0, -1)
public static int lisER(int[] ar, int idx, int prv) {
    if(idx == ar.length)
        return 0;

    int f1 = 0 + lisER(ar, idx+1, prv);
    int f2 = 0;
    if (prv == -1 || ar[idx] > ar[prv])
        f2 = 1 + lisR(ar, idx+1, idx);

    int ans = Math.max(f1, f2);
    return ans;
}
```

Recursion-2

```
int ans = 0;
for(int i=0; i<ar.length; i++) {
    ans = Math.max(ans, lisR1(ar, i));
}
println(ans);
public static int lisR1(int[] ar, int idx) {
    int max = 0;
    for(int i=idx-1; i>=0; i--) {
        if (ar[i] < ar[idx]) {
            max = Math.max(max, lisR1(ar, i));
        }
    }
    return 1+max;
}
```

Tabulation

```
public static int lengthT(int[] nums) {
    int[] dp = new int[nums.length];
    dp[0] = 1;
    for(int i=1; i<dp.length; i++) {
        int max=0;
        dp[i]=1;
        for(int j=i-1; j>=0; j--) {
            if(nums[j]<nums[i])
                max=Math.max(max, dp[j]);
        }
        dp[i]+=max;
    }
    // Max Element in dp Array
    int ans=0;
    for(int i=0; i<dp.length; i++) {
        ans=Math.max(ans, dp[i]);
    }
    return ans;
}
```

29. Paint House Many colors

Sample Input	Sample Output	Constraints
4 3 1 5 7 5 8 4 3 2 9 1 2 4	8	0<=n<=1000 0<=k<=10 0<=n-0th...<=1000

Program Steps

1. Fill the dp Array, First Row with same as Houses Array.
2. Get the first least and second least from the first row.
3. Now Fill from the second house, if the corresponding adjacent house is least, paint with second least; else paint with first least
4. Get the new least, second least and modify
5. Return the least After visiting.

Dry Run:

4 3

1 5 7

5 8 4

3 2 9

1 2 4

dp

1	5	7
10	9	5
8	7	18
8	10	11

Initially dp[0][i] = arr[0][i], least = 1, sleast = 5

From i=1, dp[1][0] = least, so paint with sleast

dp[1][0] = 5+5 = 10, ~~modify~~ :

10 <= max 10 <= 5 ✗, least == 10 } Modify
nsec = max sleast = max
nleast = 10

dp[2][0] = least ✗, paint with 1st least

8+1=9

} continue the process

Program:

```
int[][] dp = new int[n][m];

int least = Integer.MAX_VALUE;
int sleast = Integer.MAX_VALUE;

for (int i=0; i<dp[0].length; i++) {
        dp[0][i] = ar[0][i]  //filling 1st Row

    if (ar[0][i] <= least) {
        sleast = least;
        least = ar[0][i];
    }
    else if (ar[0][i] <= sleast) {
        sleast = ar[0][i];
    }
}

for (int i=1; i<dp.length; i++) {
    int nleast = Integer.MAX_VALUE;
    int nsleast = Integer.MAX_VALUE;
    for (int j=0; j<dp[0].length; j++) {
        if (least == dp[i-1][j])
            dp[i][j] = sleast + ar[i][j];
        else
            dp[i][j] = least + ar[i][j];

        if (dp[i][j] <= nleast) {
            nsleast = nleast;
            nleast = dp[i][j];
        }
        else if (dp[i][j] <= nsleast) {
            nsleast = dp[i][j];
        }
    }
}
```

Getting 1st least, 2nd least from 1st row

Getting least, and least from ith row

```
        least = nleast;
        sleast = nsleast;
    }
    println(least);
```

} Modifying Values