Lecture-09

Dynamic Programming

Sharad Hasan

Lecturer

Department of Computer Science and Engineering

Sheikh Hasina University, Netrokona.

Those who cannot remember the past are condemned to repeat it.

- Dynamic Programming

What is Dynamic Programming?

- Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

- For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

Fibonacci Sequence

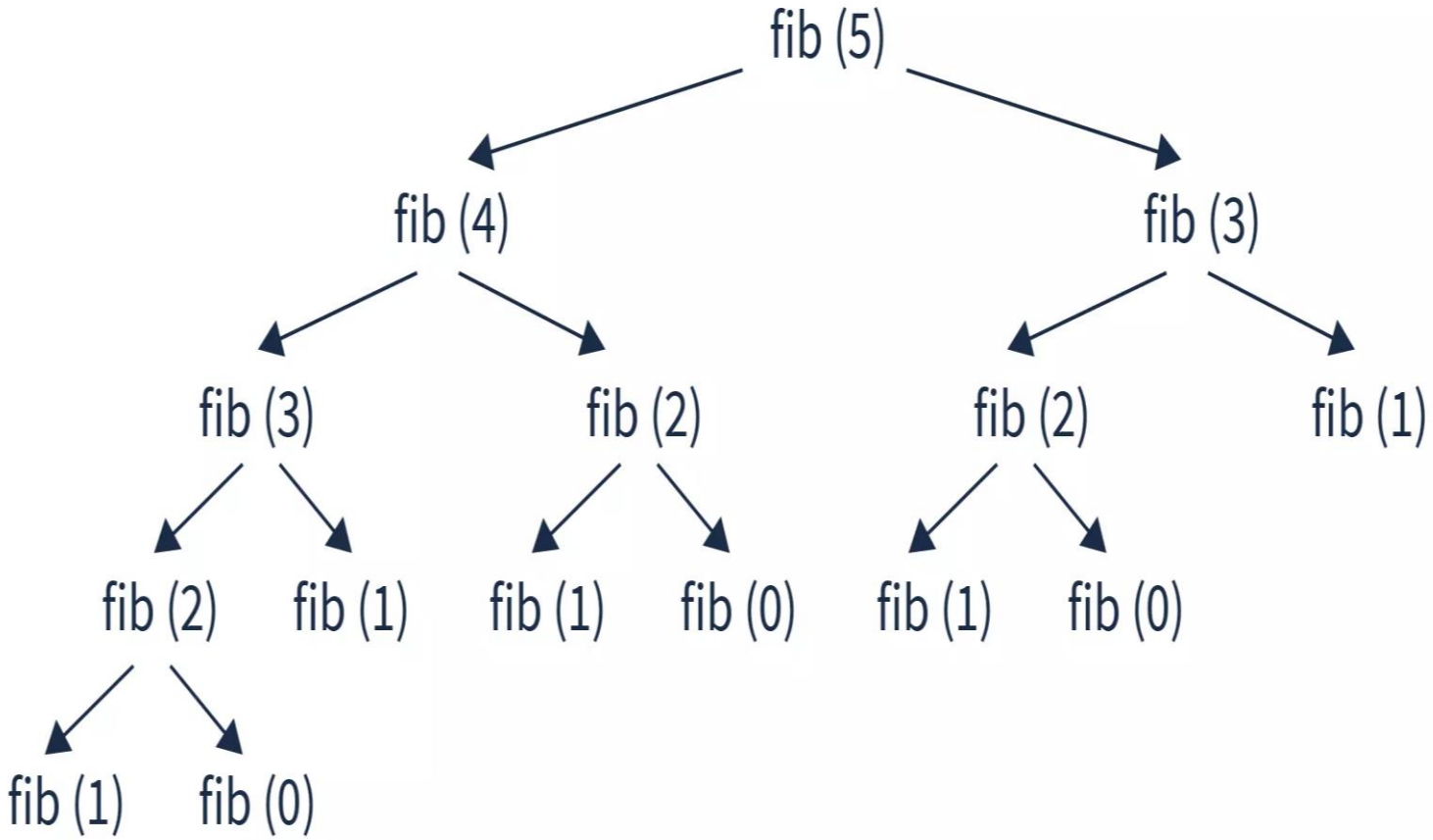
❖The Fibonacci sequence, also known as Fibonacci numbers, is defined as the sequence of numbers in which each number in the sequence is equal to the sum of two numbers before it. The Fibonacci Sequence is given as:

Fibonacci Sequence = 0, 1, 1, 2, 3, 5, 8, 13, 21, ….

❖ Now we will try to find n^{th} Fibonacci number using two different methods:

1. Plain Recursion
2. Dynamic Programming

N^{th} Fibonacci Number Using Plain Recursion



$$fib(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ fib(n-1) + fib(n-2), & if\ n > 1 \end{cases}$$

```
int fib (n) {
        if (n<1)
            return n;
        else
            return fib(n-1) + fib(n-2)
}
```

Calculating Time Complexity

From the pseudo code for n^{th} Fibonacci number using recursion, we can build a recurrence relation like this:

$$T(n) = T(n-1) + T(n-2) + 1$$

For sake of solving it, we take the relation as:

$$T(n) = T(n-1) + T(n-1) + 1$$

or, $T(n) = 2T(n-1) + 1$

Using Master Theorem for solving recurrence relation, we can calculate that, the time complexity of this algorithm is **$O(2^n)$**

This is exponential time complexity, which is very high. Can we do something to reduce the time complexity?

Let's observe the recurrence tree carefully…….

Some observations

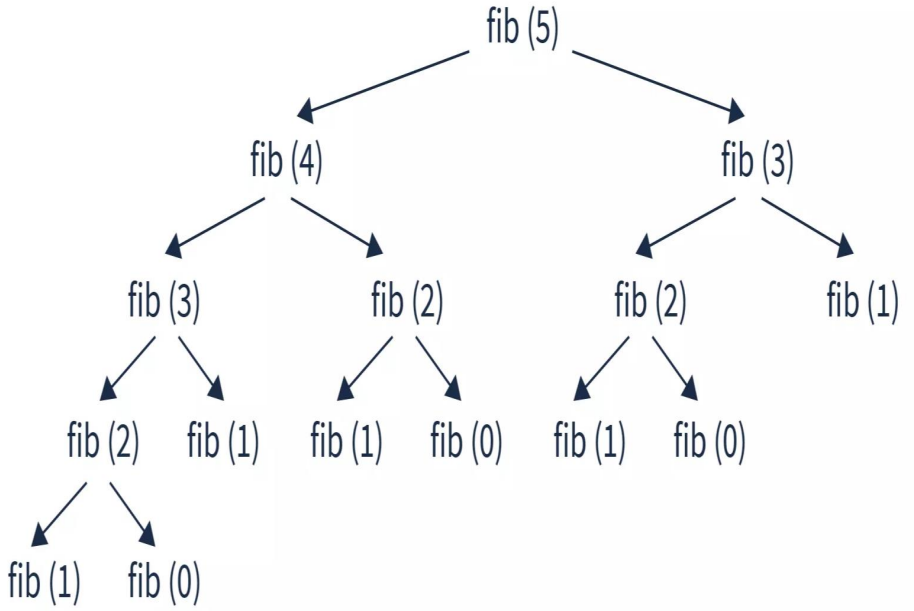
Total 15 function calls are happening here.

fib(0) is being calculated: 3 times

fib(1) is being calculated: 5 times

fib(2) is being calculated: 3 times

fib(3) is being calculated: 2 times



This type of situation is called:

❖ Overlapping subproblem

❖ Or, reoccurring subproblem

We can remove this situation using Dynamic Programming (DP). The idea is simple:

Store the results of intermediate calculations, so that we can use those results in future and don't have to calculate them again.

N^{th} Fibonacci Number Using DP

Now, we'll apply Dynamic Programming (DP) to find N^{th} Fibonacci number. DP can be applied using two techniques:

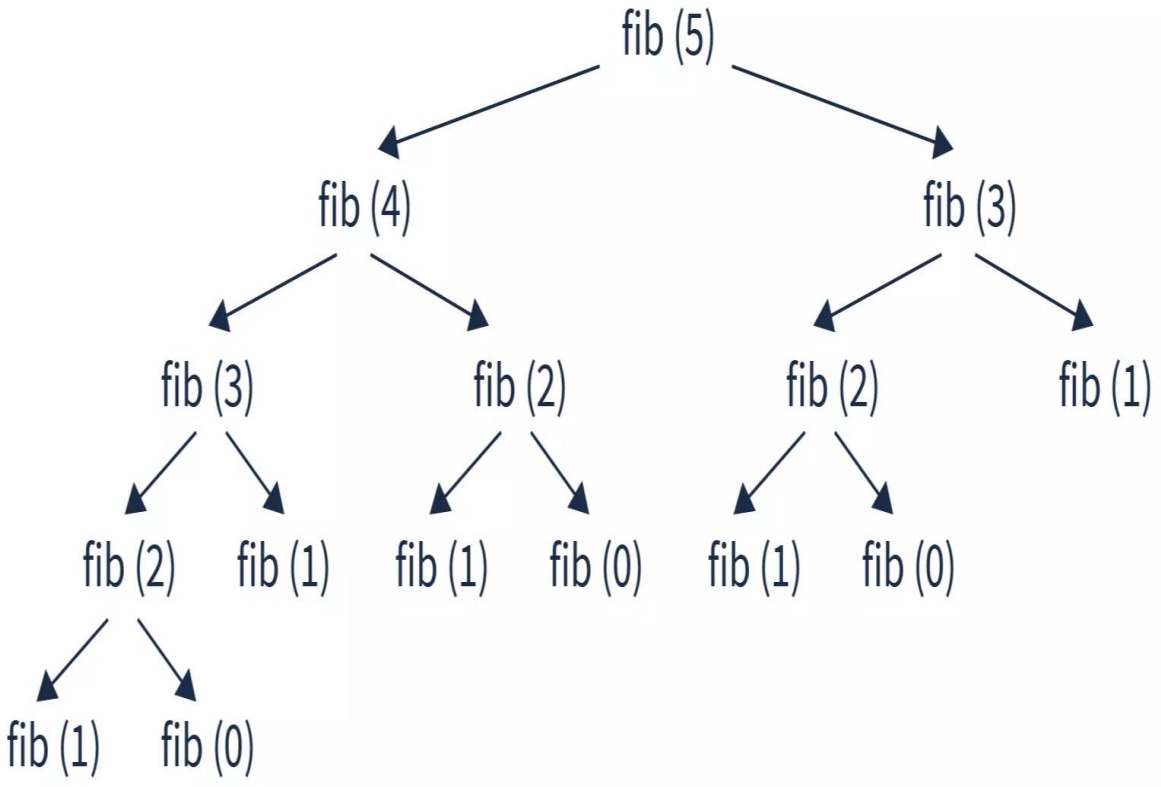
1. **Memoization (Top Down Approach)**
2. **Tabulation (Bottom Up Approach)**

We'll see the techniques one-by-one.

N^{th} Fibonacci Number Using DP (Memoization)

1. If we are to find N^{th} Fibonacci Number, we'll take an array of size (N+1) and initialize all the value as -1.

2. We'll traverse the recursion tree top to bottom and when we find an exact value for any function call, we store the value in the array at corresponding index.



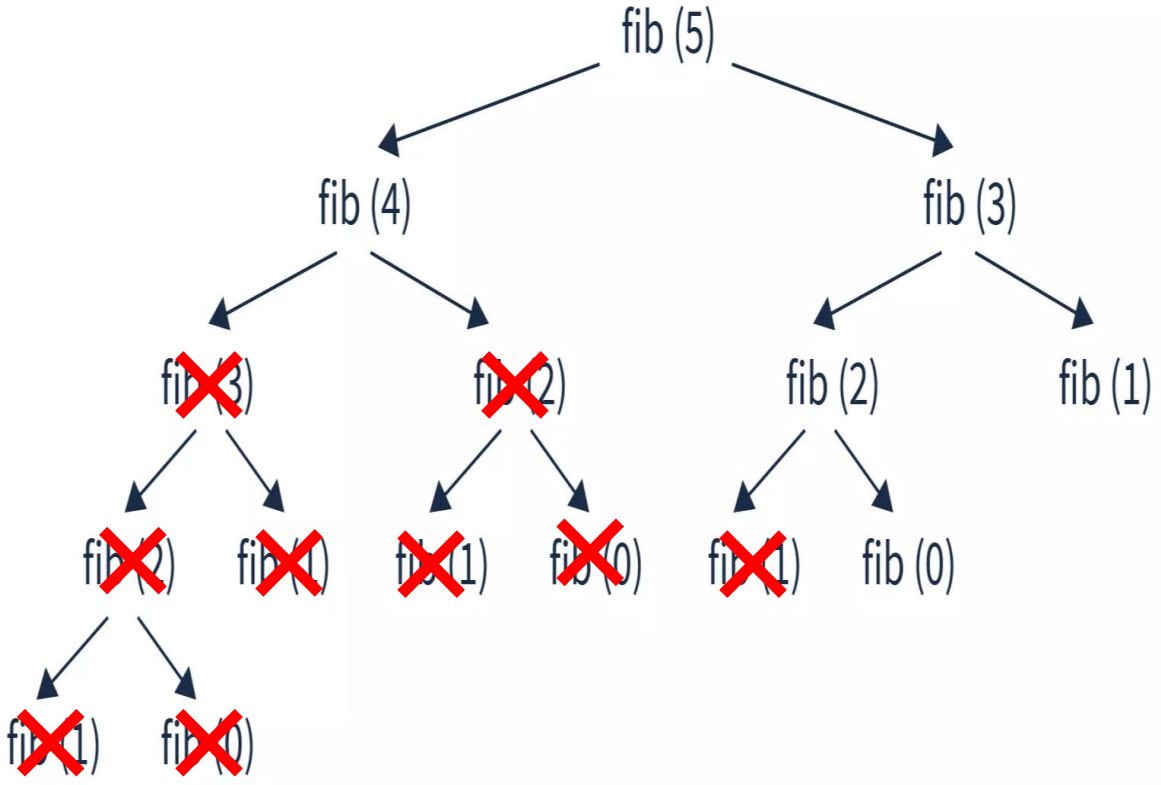
-1	-1	-1	-1	-1	-1
0	1	2	3	4	5

$$fib(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ fib(n-1) + fib(n-2), & if\ n > 1 \end{cases}$$

N^{th} Fibonacci Number Using DP (Memoization)

1. If we are to find N^{th} Fibonacci Number, we'll take an array of size (N+1) and initialize all the value as -1.

2. We'll traverse the recursion tree top to bottom and when we find an exact value for any function call, we store the value in the array at corresponding index.



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

$$fib(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ fib(n-1) + fib(n-2), & if\ n > 1 \end{cases}$$

Nth Fibonacci Number Using DP (Tabulation)

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

F	0	1	1	2	3	5
	0	1	2	3	4	5

$$fib(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ fib(n-1) + fib(n-2), & if\ n > 1 \end{cases}$$

N^{th} Fibonacci Number Using DP (Tabulation)

```
int fib(int n){
    if (n<1)
        return n;          → 1 unit of time
    F [0] = 0;             → 1 unit of time
    F [1] = 1;             → 1 unit of time
    for (int i = 2; i<=n; i++){   Loop will run (n-1) times
        F [i] = F [i-1] + F [i-2];   → 1 unit of time
    }
    return F [n];          → 1 unit of time
}
```

F	0	1	1	2	3	5
	0	1	2	3	4	5

$$fib(n) = \begin{cases} 0, & if\ n = 0 \\ 1, & if\ n = 1 \\ fib(n-1) + fib(n-2), & if\ n > 1 \end{cases}$$

$T(n) = 1 + 1 + 1 + 1 \times (n-1) + 1 = (n-1) + 4 = n+3$

This gives us time complexity: **O (n)**

Memoization and Tabulation

Tabulation and memoization are two techniques used in dynamic programming to optimize the execution of a function that has repeated and expensive computations. Although both techniques have similar goals, there are some differences between them.

1. Memoization

Memoization is a top-down approach where we cache the results of function calls and return the cached result if the function is called again with the same inputs. It is used when we can divide the problem into subproblems and the subproblems have overlapping subproblems. Memoization is typically implemented using recursion and is well-suited for problems that have a relatively small set of inputs.

2. Tabulation

Tabulation is a bottom-up approach where we store the results of the subproblems in a table and use these results to solve larger subproblems until we solve the entire problem. It is used when we can define the problem as a sequence of subproblems and the subproblems do not overlap. Tabulation is typically implemented using iteration and is well-suited for problems that have a large set of inputs.

Memoization vs Tabulation

	Tabulation	Memoization
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor	If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required
Table Entries	In Tabulated version, starting from the first entry, all entries are filled one by one	Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.

Steps to solve a Dynamic programming problem

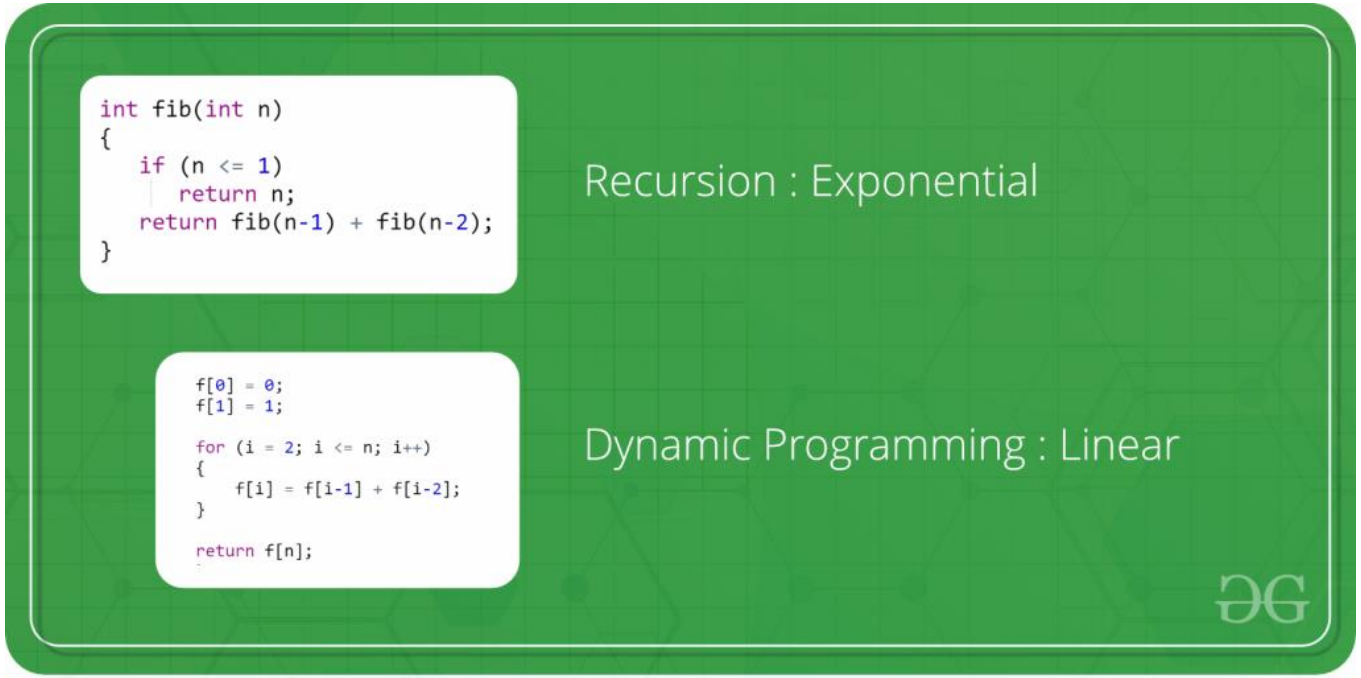
1. Identify if it is a Dynamic programming problem.
2. Decide a state expression with the Least parameters.
3. Formulate state and transition relationship.
4. Apply tabulation or memorization.

For details explanation of these steps, visit:

https://www.geeksforgeeks.org/solve-dynamic-programming-problem/

Recursion and Dynamic Programming

- Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence, most optimization problems require recursion and dynamic programming is used for optimization.
- But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.
- That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.



Recursion vs Dynamic Programming

Features	Recursion	Dynamic Programming
Definition	By breaking a difficulty down into smaller problems of the same problem, a function calls itself to solve the problem until a specific condition is met.	It is a technique by breaks them into smaller problems and stores the results of these subproblems to avoid repeated calculations.
Approach	Recursion frequently employs a top-down method in which the primary problem is broken down into more manageable subproblems.	Using a bottom-up methodology, dynamic programming starts by resolving the smallest subproblems before moving on to the primary issue.
Base Case	To avoid infinite loops, it is necessary to have a base case (termination condition) that stops the recursion when a certain condition is satisfied.	Although dynamic programming also needs a base case, it focuses mostly on iteratively addressing subproblems.
Performance	Recursion might be slower due to the overhead of function calls and redundant calculations.	Dynamic programming is often faster due to optimized subproblem solving and memoization.
Memory Usage	It does not require extra memory, only requires stack space	Dynamic programming require additional memory to record intermediate results.
Examples	It include computing factorials, using the Fibonacci sequence.	It include computing the nth term of a series using bottom-up methods, the knapsack problem

Greedy and Dynamic Programming

- Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.
- However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.
- Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

Greedy vs Dynamic Programming

Features	Greedy	Dynamic Programming
Feasibility	In a greedy Algorithm, we make whatever choice seems best at the moment in the hope that it will lead to global optimal solution	In Dynamic Programming we make decision at each step considering current problem and solution to previously solved sub problem to calculate optimal solution
Optimality	In Greedy Method, sometimes there is no such guarantee of getting Optimal Solution.	It is guaranteed that Dynamic Programming will generate an optimal solution as it generally considers all possible cases and then choose the best.
Recursion	A greedy method follows the problem solving heuristic of making the locally optimal choice at each stage.	A Dynamic programming is an algorithmic technique which is usually based on a recurrent formula that uses some previously calculated states
Memoization	It is more efficient in terms of memory as it never look back or revise previous choices.	It requires Dynamic Programming table for Memoization and it increases it's memory complexity.
Time complexity	Greedy methods are generally faster. For example, Dijkstra's shortest path algorithm takes O(ELogV + VLogV) time.	Dynamic Programming is generally slower. For example, Bellman Ford algorithm takes O(VE) time.
Fashion	The greedy method computes its solution by making its choices in a serial forward fashion, never looking back or revising previous choices.	Dynamic programming computes its solution bottom up or top down by synthesizing them from smaller optimal sub solutions.
Example	Fractional knapsack	0/1 knapsack problem

Divide and Conquer vs Dynamic Programming

Divide and Conquer	**Dynamic Programming**
Follows Top-down approach	Follows bottom-up approach
Used to solve decision problem	Used to solve optimization problem
Solution of subproblem is computed recursively more than once.	The solution of subproblems is computed once and stored in a table for later use.
It is used to obtain a solution to the given problem, it does not aim for the optimal solution	It always generates optimal solution
Recursive in nature	Not recursive in nature.
Less efficient and slower.	More efficient but slower than greedy.
Some memory is required.	More memory is required to store subproblems for later use.
Examples: Merge sort, Quick sort, Strassen's matrix multiplication.	Examples: 0/1 Knapsack, All pair shortest path, Matrix-chain multiplication.

When to use Dynamic Programming

- Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used.
- Mostly, these algorithms are used for optimization. An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.

Sample Questions

1. Define Dynamic Programming. In which situation we can use DP?
2. With suitable example show that, DP can reduce time complexity in comparison to plain recursion.
3. What is Memoization and Tabulation? Differentiate between them. Which one is mostly used in DP?
4. Discuss the relationship between Recursion and DP. What are the differences between them?
5. Differentiate between Greedy approach and DP.
6. Differentiate between Divide and Conquer and DP.

Popular problems solved by DP

1. 0/1 Knapsack problem
2. Coin change problem
    * Counting number of ways
    * Determining minimum number of coins
3. Longest common subsequence
4. Longest increasing subsequence
5. Matrix chain multiplication
6. Max Product Subarray
7. Maximal square
8. Minimum number of jumps to reach the end
9. Edit Distance
10. Cutting a Rod
11. Floyd Warshall Algorithm for shortest path etc……

0/1 Knapsack Problem

Problem Definition

Given N items where each item i has some weight w_i and profit p_i associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming

Example of 0/1 knapsack problem: Brute Force Approach

Consider the problem having weights and profits are:

Weights: {3, 4, 6, 5}

Profits: {2, 3, 1, 4}

The weight of the knapsack is 8 kg

The number of items is 4

We have to find out two things:

1. What will be the maximum profit?
2. Which items we have to include and which items we have to leave behind to gain max profit within the capacity?

The answer of this question is written as a set: $x_i = \{1, 0, 0, 1\}$

This means that: item-1 is included,

item-2 is not included

item-3 is not included

item-4 is included

Example of 0/1 knapsack problem: Brute Force Approach

The above problem can be solved by using the following method: We can try out all possible combinations and calculate the profit associated with each combination. Then we can choose the combination with profit.

$x_i = \{1, 0, 0, 1\} \longrightarrow$ profit: 2 +4 = 6

$x_i = \{0, 0, 0, 1\} \longrightarrow$ profit: 4

$x_i = \{0, 1, 0, 1\} \longrightarrow$ profit: 3 + 4 = 7

and so on…….

The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:

$2^4 = 16$; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

Another approach to solve the problem is dynamic programming approach.

Example of 0/1 knapsack problem: DP Approach

w	3	4	6	5
p	2	3	1	4

Capacity, C = 8
No. of items, n = 4

Example of 0/1 knapsack problem: DP Approach

w	3	4	6	5
p	2	3	1	4

Capacity, W = 8
No. of items, n = 4

		i \ *w* →	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
p_i	**w_i**	**0**									
2	**3**	**1**									
3	**4**	**2**									
4	**5**	**3**									
1	**6**	**4**									

Example of 0/1 knapsack problem: DP Approach

w	3	4	5	6
p	2	3	4	1

Capacity, C = 8
No. of items, n = 4

$$D[i][w] = max\begin{cases} D[i-1][w] \\ D[i-1][w-w[i]] + p[i] \end{cases}$$

		i \ w →	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\mathbf{p_i}$	$\mathbf{w_i}$										
		0	0	0	0	0	0	0	0	0	0
2	**3**	**1**	0								
3	**4**	**2**	0								
4	**5**	**3**	0								
1	**6**	**4**	0								

Example of 0/1 knapsack problem: DP Approach

w	3	4	5	6
p	2	3	4	1

Capacity, C = 8
No. of items, n = 4

$$D[i][w] = max \begin{cases} D[i-1][w] \\ D[i-1][w - w[i]] + p[i] \end{cases}$$

		i \ w →	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$\mathbf{p_i}$	$\mathbf{w_i}$										
		0	0	0	0	0	0	0	0	0	0
2	**3**	**1**	0	0	0	2	2	2	2	2	2
3	**4**	**2**	0	0	0	2	3	3	3	5	5
4	**5**	**3**	0	0	0	2	3	4	4	5	6
1	**6**	**4**	0	0	0	2	3	4	4	5	6

Example of 0/1 knapsack problem: DP Approach

w	3	4	5	6
p	2	3	4	1

$$D[i][w] = max \begin{cases} D[i-1][w] \\ D[i-1][w - w[i]] + p[i] \end{cases}$$

Capacity, C = 8
No. of items, n = 4

		i \ w →	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
p_i	**w_i**										
		0	0	0	0	0	0	0	0	0	0
2	**3**	**1**	0	0	0	2	2	2	2	2	2
3	**4**	**2**	0	0	0	2	3	3	3	5	5
4	**5**	**3**	0	0	0	2	3	4	4	5	6
1	**6**	**4**	0	0	0	2	3	4	4	5	6

X = {1, 0, 1, 0}

Time and Space Complexity

Time Complexity: O(N * W). where ‘N’ is the number of elements and ‘W’ is capacity.

Auxiliary Space: O(N * W). The use of a 2-D array of size ‘N*W’.

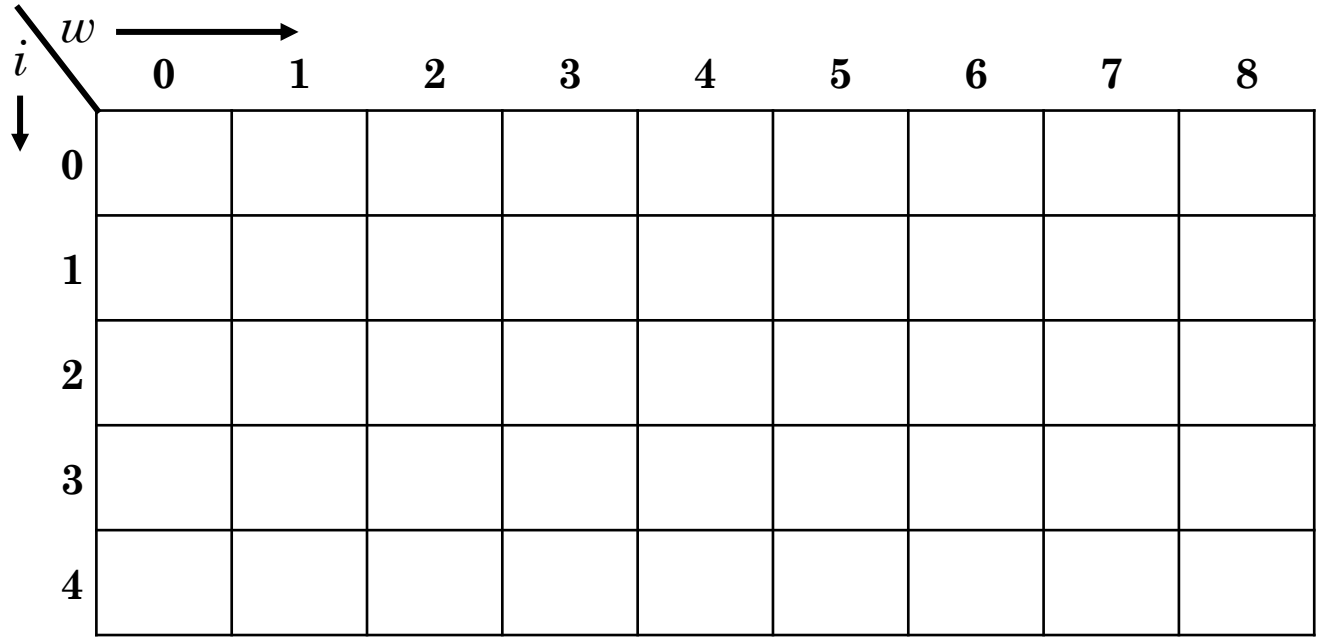
Real Life Example

- One application of this algorithm is Internet Download Manager. The data is broken into chunks. As per the maximum size of data that can be retrieved in one go , the server uses this algorithm and packs the chunks so as to utilize the full size limit.
- Airline carry-on: Your total weight must be under the airline's cabin baggage weight limit. For example, some airlines have a limit of 10kg, while others have a limit of 7kg.
- Picnic planning: You have a list of items you want to bring on a picnic, each with a weight. Your knapsack can carry no more than 15 pounds.
- Packing food in a knapsack for maximum nutrient value
- Home energy management
- Cognitive radio networks
- Resource management in software etc…….

p	0	1	2	5	6
	0	1	2	3	4

wt	0	2	3	4	5
	0	1	2	3	4

C = 8, n=4




```
int main(){
    int p[5]={0,1,2,5,6};
    int wt[5]={0,2,3,4,5};
    int c=8, n=4;
    int D[5][9];

    for(i=0;i<=ni++){
        for(w=0;w<=c;w++){
            if(i==0||w==0)
                D[i][w]=0;
            else if(wt[i]<=w)
                D[i][w]=max(D[i-1][w],
                    D[i-1][w-wt[i]+p[i]);
            else
                D[i][w]=D[i-1][w];
        }
    }
}
```

p	0	1	2	5	6
	0	1	2	3	4

wt	0	2	3	4	5
	0	1	2	3	4

C = 8, n=4

i \ *w*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

```
int main(){
    int p[5]={0,1,2,5,6};
    int wt[5]={0,2,3,4,5};
    int c=8, n=4;
    int D[5][9];

    for(i=0;i<=ni++){
        for(w=0;w<=c;w++){
            if(i==0||w==0)
                D[i][w]=0;
            else if(wt[i]<=w)
                D[i][w]=max(D[i-1][w],
                    D[i-1][w-wt[i]+p[i]);
            else
                D[i][w]=D[i-1][w];
        }
    }
}
```

p	0	1	2	5	6
	0	1	2	3	4

wt	0	2	3	4	5
	0	1	2	3	4

c = 8, n=4

i \ *w*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1
2	0	0	1	2	2	3	3	3	3
3	0	0	1	2	5	5	6	7	7
4	0	0	1	2	5	6	6	7	8

```
i=n, w=c
while(i>0 && w>0){
    if (D[i][w]==D[i-1][w]){
        cout<< i << "=0";
        i--;
    }
    else{
        cout<< i << "=1";
        i--;
        w=w-wt[i];
    }
}
```

Coin Change Problem

(Counting number of ways)

Problem Definition

Given an integer array of ***coins[ ]*** of size ***N*** representing different types of denominations and an integer ***amount***, the task is to count the number of ways the amount can be paid with the given denomination of coins.

Assume, there are infinite number of coins of each denomination.

Example

coins = [1, 3, 5]

amount = 8

Ways:

{1, 1, 1, 1, 1, 1, 1, 1}

{1, 1, 1, 1, 1, 3}

{3, 3, 1, 1}

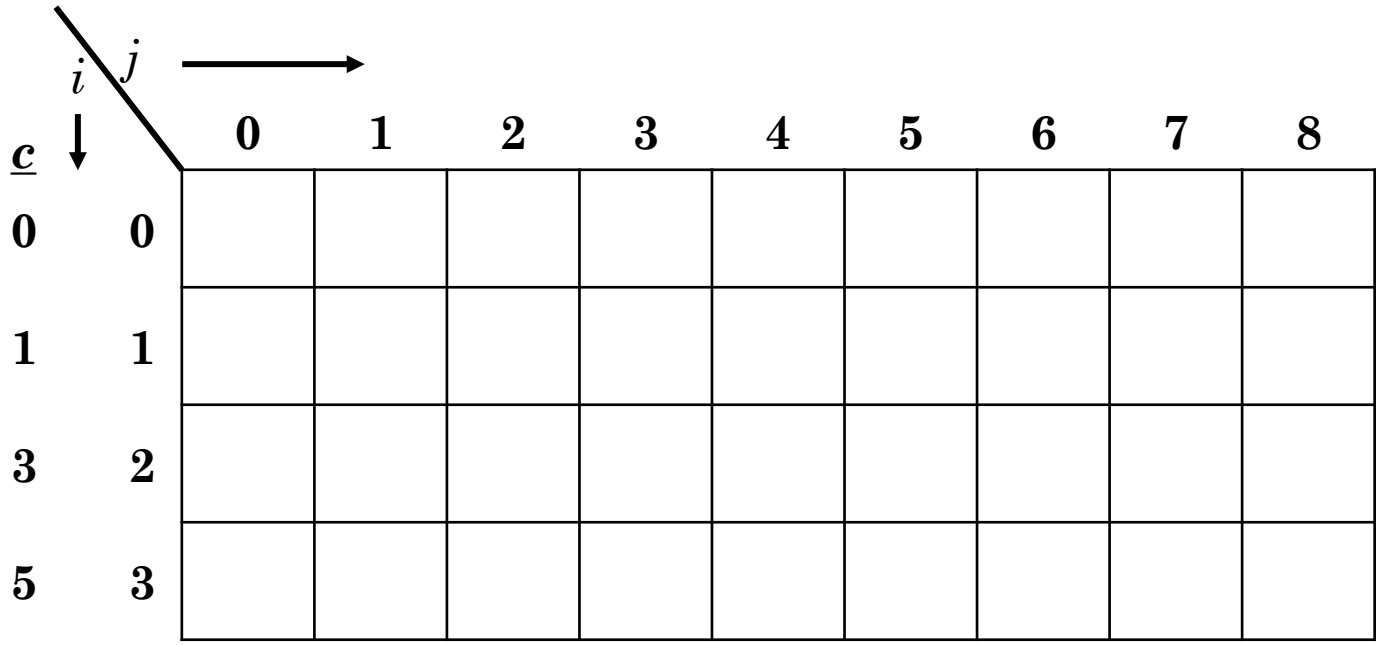
{5, 3}

{5, 1, 1, 1}

So, there are total *5 ways* to pay the amount 8 with the coins of given denominations.

c	0	1	3	5
	0	1	2	3

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



c	0	1	3	5
	0	1	2	3

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

i \ *j*		0	1	2	3	4	5	6	7	8
c										
0	**0**	1	0	0	0	0	0	0	0	0
1	**1**	1								
3	**2**	1								
5	**3**	1								

```
int nc = 3, amt = 8;
int c[4]={0,1,3,5};
int a[9]={0,1,2,3,4,5,6,7,8};
int D[4][9];
for(i=0; i<=nc; i++){
    D[i][0]=1;
}
for(j=1; j<=amt; j++){
    D[0][1]=0;
}
for(int i=1; i<=nc; i++){
    for(j=1; j<=amt; j++){
        if(c[i]>j)
            D[i][j]=D[i-1][j]
        else
            D[i][j]=D[i-1][j] + D[i][j-c[i];
    }
}
```

c	0	1	3	5
	0	1	2	3

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

i \ *j*		0	1	2	3	4	5	6	7	8
c										
0	**0**	1	0	0	0	0	0	0	0	0
1	**1**	1	1	1	1	1	1	1	1	1
3	**2**	1	1	1	2	2	2	3	3	3
5	**3**	1	1	1	2	2	3	4	4	5

```
int nc = 3, amt = 8;
int c[4]={0,1,3,5};
int a[9]={0,1,2,3,4,5,6,7,8};
int D[4][9];
for(i=0; i<=nc; i++){
    D[i][0]=1;
}
for(j=1; j<=amt; j++){
    D[0][1]=0;
}
for(int i=1; i<=nc; i++){
    for(j=1; j<=amt; j++){
        if(c[i]>j)
            D[i][j]=D[i-1][j]
        else
            D[i][j]=D[i-1][j] + D[i][j-c[i];
    }
}
```

Coin Change Problem

(Counting minimum number of coins)

Problem Definition

Given an integer array of ***coins[ ]*** of size ***N*** representing different types of denominations and an integer ***amount***, the task is to determining the minimum number of coins the amount can be paid with the given denomination of coins.

Assume, there are infinite number of coins of each denomination.

Example

coins = [1, 5, 7, 9]

amount = 12

Determine the minimum number of coins the amount can be paid with the given denomination of coins.

c	0	1	5	7	9
	0	1	2	3	4

a	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	1	2	3	4	5	6	7	8	9	10	11	12

i \ *j* →		0	1	2	3	4	5	6	7	8	9	10	11	12
c ↓														
0	**0**													
1	**1**													
5	**2**													
7	**3**													
9	**4**													

```
int nc = 4, amt = 12;
int c[5]={0, 1, 5, 7, 9};
int a[13]={0,1,2,3,4,5,6,7,8,9,10,11,12};
int D[5][13];
for(int i=0; i<=nc; i++){
    for(j=0; j<=amt; j++){
        if(i==0||w==0)
            D[i][w]=0;
        else if(c[i]>j)
            D[i][j]=D[i-1][j]
        else
            D[i][j]=MIN ( D[i-1][j],
                1+D[i][j-c[i]];
    }
}
```

c	0	1	5	7	9
	0	1	2	3	4

a	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	1	2	3	4	5	6	7	8	9	10	11	12

i \ *j* →		0	1	2	3	4	5	6	7	8	9	10	11	12
c ↓														
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0
1	**1**	0	1	2	3	4	5	6	7	8	9	10	11	12
5	**2**	0	1	2	3	4	1	2	3	4	5	2	3	4
7	**3**	0	1	2	3	4	1	2	1	2	3	2	3	2
9	**4**	0	1	2	3	4	1	2	1	2	1	2	3	2

```
int nc = 4, amt = 12;
int c[5]={0, 1, 5, 7, 9};
int a[13]={0,1,2,3,4,5,6,7,8,9,10,11,12};
int D[5][13];
for(int i=0; i<=nc; i++){
    for(j=0; j<=amt; j++){
        if(i==0||w==0)
            D[i][w]=0;
        else if(c[i]>j)
            D[i][j]=D[i-1][j]
        else
            D[i][j]=MIN ( D[i-1][j],
                1+D[i][j-c[i]];
    }
}
```

Longest Common Subsequence