DYNAMIC PROGRAMMING

Definition

Dynamic Programming (DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Imagine you are given a box of coins and you have to count the total number of coins in it. Once you have done this, you are provided with another box and now you have to calculate the total number of coins in both boxes. Obviously, you are not going to count the number of coins in the first box again. Instead, you would just count the total number of coins in the second box and add it to the number of coins in the first box you have already counted and stored in your mind. This is the exact idea behind dynamic programming.

Recording the result of a problem is only going to be helpful when we are going to use the result later i.e., the problem appears again. This means that dynamic programming is useful when a problem breaks into subproblems, the same subproblem appears more than once.

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have:

- overlapping subproblems
- optimal substructure property

Explanation

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

An Example of Dynamic Programming Approach

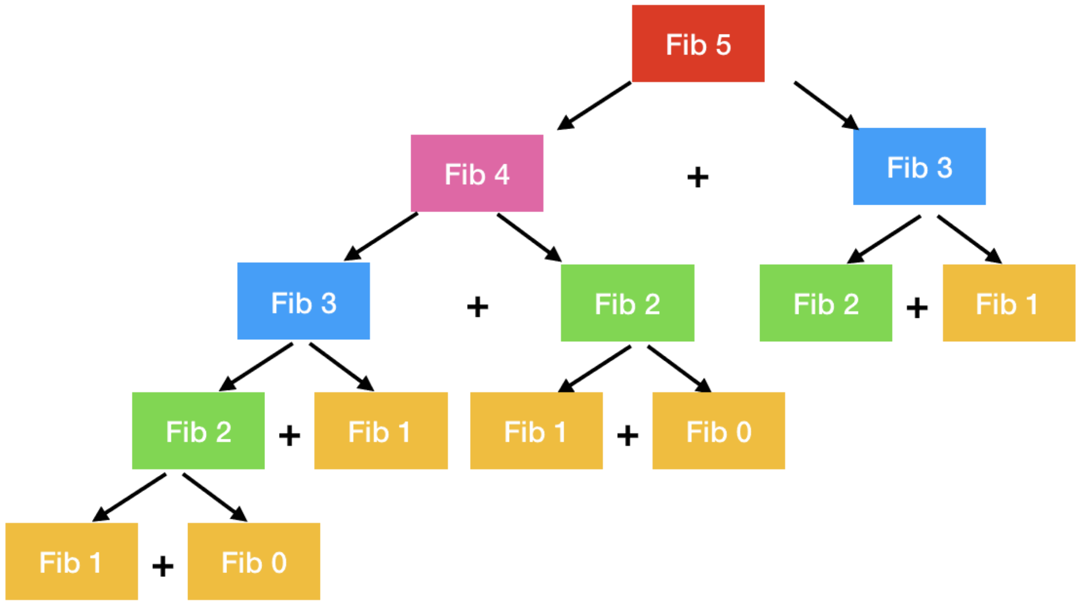
Let's take the example of the Fibonacci numbers. As we all know, Fibonacci numbers are a series of numbers in which each number is the sum of the two preceding numbers. The first few Fibonacci numbers are $0, 1, 1, 2, 3, 5, and\ 8$, and they continue on from there.

```
function fib(n)
    if n <= 1 return n
    return fib(n - 1) + fib(n - 2)
```

If we are asked to calculate the nth Fibonacci number, we can do that with the following equation,

$$Fib(n) = Fib(n-1) + Fib(n-2), for\ n > 1\ and\ F(0) = 0. F(1) = 1$$

As we can clearly see here, to solve the overall problem (i.e. Fib(n)), we broke it down into two smaller subproblems (which are Fib(n-1) and Fib(n-2)). This shows that we can use DP to solve this problem.



You can see here that to calculate the 5th term, the same subproblem appears more than once. For example, F(3) is occurring twice, F(1) is occurring 4 times, etc.

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

So, rather than calculating the result of the same problem, again and again, we can store the result once and use it again and again whenever needed.

Recursive Solution

Let’s see the non-DP recursive solution for finding the nth Fibonacci number:

```
FIBONACCI(n)

  if n==0

    return 0

  if n==1

    return 1

  return FIBONACCI(n-1) + FIBONACCI(n-2)
```

```
class Fib {
  public static int fibonacci(int n) {
    if (n < 2) {
      return n;
    }

    return fibonacci(n-1) + fibonacci(n-2);
  }

  public static void main(String[] args) {
    System.out.println(fibonacci(46));
  }
}
```

The code is simple. Since $F(0)$ and $F(1)$ are 0 and 1 respectively, we are handling those cases first. Otherwise, we are calculating the nth term is $fibonaci(n-1)\ +\ fibonacci(n-2)$ and we are returning that.

Dynamic Programming Solution

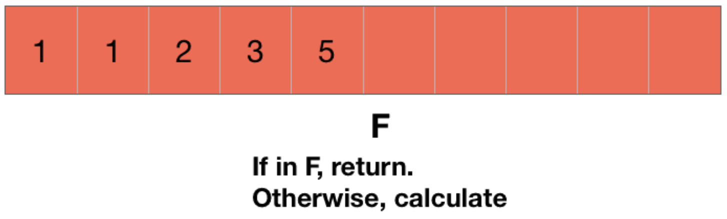
Let's write the same code but this time by storing the terms we have already calculated:

```
F = []
FIBONACCI(n)
  if F[n] == null
    if n==0
      F[n] = 0
    else if n==1
      F[n] = 1
    else
      F[n] = FIBONACCI(n-1) + FIBONACCI(n-2)
  return F[n]
```

```
class Fib {
 static int[] F = new int[50]; //array to store fibonacci terms

 public static void initF() {
  for(int i=0; i<50; i++) {
   F[i] = -1;
  }
 }

 public static int dynamicFibonacci(int n) {
  if (F[n] < 0) {
   if (n==0) {
    F[n] = 0;
   }
   else if (n == 1) {
    F[n] = 1;
   }
   else {
    F[n] = dynamicFibonacci(n-1) + dynamicFibonacci(n-2);
   }
  }
  return F[n];
 }
```

```
  public static void main(String[] args) {
    initF();
    System.out.println(dynamicFibonacci(46));
  }
}
```

Here, we are first checking if the result is already present in the array or not if ($F[n] \ == \ null$). If it is not, then we are calculating the result and then storing it in the array F and then returning it ($return \ F[n]$).



Running this code for the 100th term gave the result almost instantaneously and this is the power of dynamic programming.

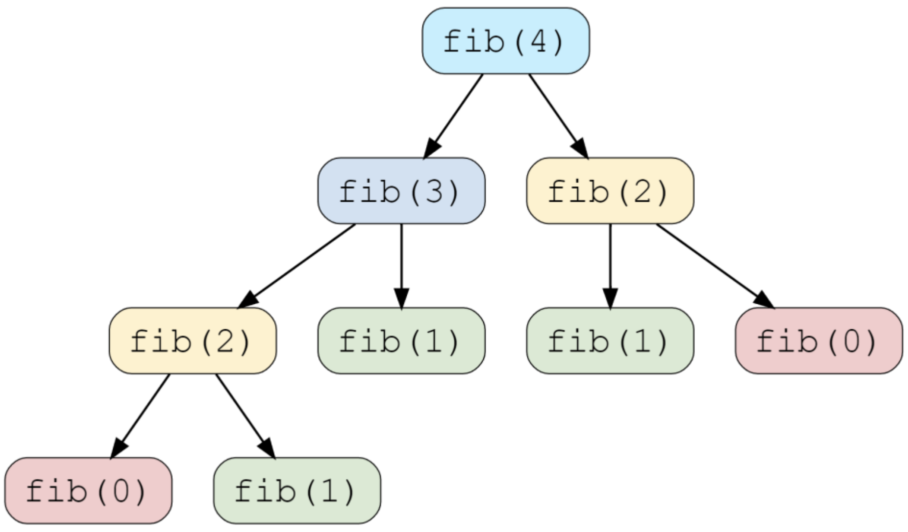
But are we sacrificing anything for the speed? Yes, memory. Dynamic programming basically trades time with memory. Thus, we should take care that not an excessive amount of memory is used while storing the solutions.

Characteristics of Dynamic Programming

Let’s first take a look at what are the characteristics of a problem that tells us that we can apply DP to solve it:

Overlapping Subproblems

Subproblems are smaller versions of the original problem. Any problem has overlapping sub-problems if finding its solution involves solving the same subproblem multiple times. Take the example of the Fibonacci numbers; to find the $fib(4)$, we need to break it down into the following sub-problems:



Recursion tree for calculating Fibonacci numbers

We can clearly see the overlapping subproblem pattern here, as $fib(2)$ has been evaluated twice and $fib(1)$ has been evaluated three times.

Optimal Substructure Property

Any problem has optimal substructure property if its overall optimal solution can be constructed from the optimal solutions of its subproblems. For Fibonacci numbers, as we know,

$$Fib(n) \ = \ Fib(n-1) \ + \ Fib(n-2)$$

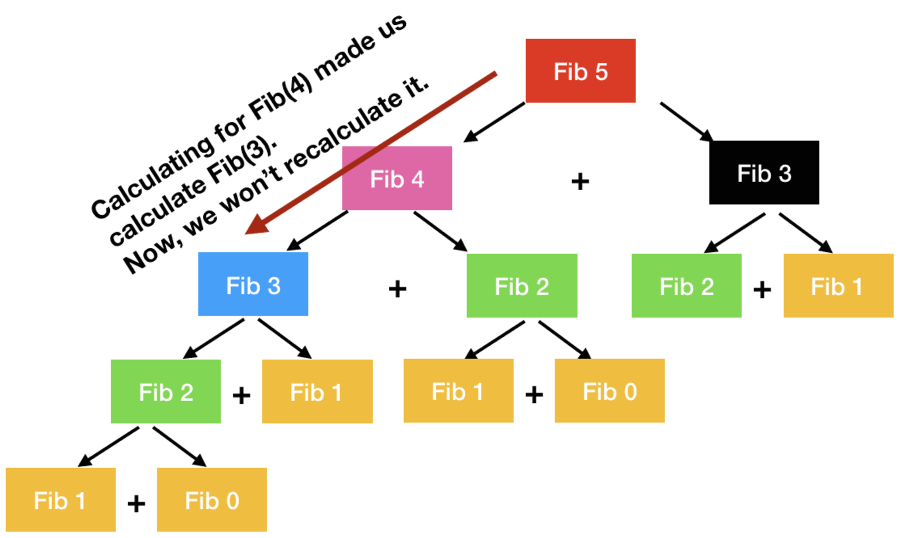
This clearly shows that a problem of size ‘n’ has been reduced to subproblems of size ‘$n-1$’ and ‘$n-2$’. Therefore, Fibonacci numbers have optimal substructure property.

Two Approaches of Dynamic Programming

There are two approaches of the dynamic programming. The first one is the top-down approach and the second is the bottom-up approach. Let's take a closer look at both the approaches:

Top-Down with Memoization

The way we solved the Fibonacci series was the top-down approach. We just start by solving the problem in a natural manner and stored the solutions of the subproblems along the way. We also use the term **memoization**, a word derived from memo for this.


Calculating for Fib(4) made us calculate Fib(3).
Now, we won't recalculate it.
Fib 5
Fib 4
+
Fib 3
Fib 3
+
Fib 2
Fib 2
+
Fib 1
Fib 2
+
Fib 1
Fib 1
+
Fib 0
Fib 1
+
Fib 0

In other terms, it can also be said that we just hit the problem in a natural manner and hope that the solutions for the subproblem are already calculated and if they are not calculated, then we calculate them on the way.

The dynamic programming way of solving the Fibonacci problem demonstrated above was a top-down approach (memorization).

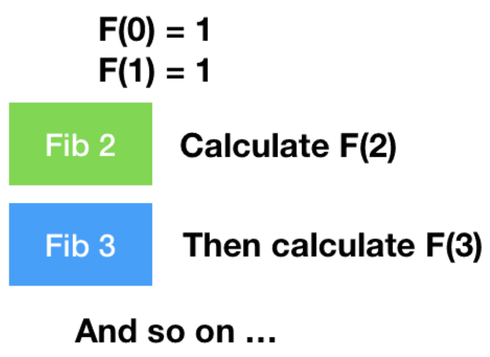
Bottom-up with Tabulation

Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem "bottom-up" (i.e., by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Tabulation is the opposite of Memoization, as in Memoization we solve the problem and maintain a map of already solved sub-problems. In other words, in memoization, we do it top-down in the sense that we solve the top problem first (which typically recurses down to solve the sub-problems).

Let's apply Tabulation to our example of Fibonacci numbers. Since we know that every Fibonacci number is the sum of the two preceding numbers, we can use this fact to populate our table.

The other way we could have solved the Fibonacci problem was by starting from the bottom i.e., start by calculating the 2nd term and then 3rd and so on and finally calculating the higher terms on the top of these i.e., by using these values.



We use a term tabulation for this process because it is like filling up a table from the start.

Here is the code for our bottom-up dynamic programming approach:

```
F = [] //new array
FIBONACCI-B-UP(n)
  F[0] = 0
  F[1] = 1

  for i in 2 to n
    F[i] = F[i-1] + F[i-2]

  return F[n]
```

```
class Fib {
 static int[] F = new int[50]; //array to store fibonacci terms

 public static int fibonacciBottomUp(int n) {
  F[n] = 0;
  F[1] = 1;
  for(int i=2; i<=n; i++) {
   F[i] = F[i-1] + F[i-2];
  }
  return F[n];
 }

 public static void main(String[] args) {
  System.out.println(fibonacciBottomUp(46));
 }
}
```

As said, we started calculating the Fibonacci terms from the starting and ended up using them to get the higher terms.

Also, the order for solving the problem can be flexible with the need of the problem and is not fixed. So, we can solve the problem in any needed order.

Generally, we need to solve the problem with the smallest size first. So, we start by sorting the elements with size and then solve them in that order.

Let's compare memoization and tabulation and see the pros and cons of both.

Memoization vs Tabulation

Memoization is indeed the natural way of solving a problem, so coding is easier in memoization when we deal with a complex problem. Coming up with a specific order while dealing with lot of conditions might be difficult in the tabulation.

Also think about a case when we don't need to find the solutions of all the subproblems. In that case, we would prefer to use the memoization instead.

However, when a lot of recursive calls are required, memoization may cause memory problems because it might have stacked the recursive calls to find the solution of the deeper recursive call but we won't deal with this problem in tabulation.

Generally, memoization is also slower than tabulation because of the large recursive calls.

Recursion vs 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.

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.

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.

Greedy Algorithms vs Dynamic Programming

Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.

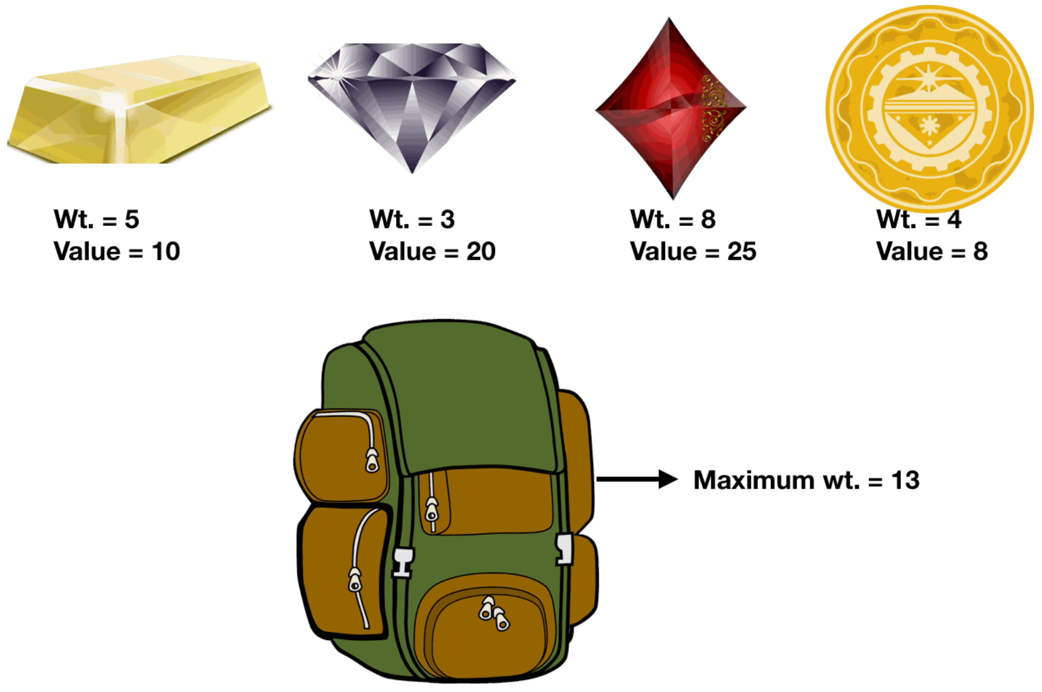
However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

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.

Examples of Dynamic Programming

0/1 Knapsack Problem

Suppose you woke up on some mysterious island and there are different precious items on it. Each item has a different value and weight. You are also provided with a bag to take some of the items along with you but your bag has a limitation of the maximum weight you can put in it. So, you need to choose items to put in your bag such that the overall value of items in your bag maximizes.



This problem is commonly known as the knapsack or the rucksack problem. There are different kinds of items (i) and each item 'i' has a weight (w_i) and value (v_i) associated with it. x_i is the number of 'i' kind of items we have picked. And the bag has a limitation of maximum weight (W).

So, our main task is to maximize the value i.e., summation of the *number of items taken * its value*, such that the weight of all the items should be less than the maximum weight.

In $0/1\ knapsack\ problem$, there is only one item of each kind (or we can pick only one). So, we are available with only two options for each item, either pick it (1) or leave it (0) i.e., $x_i \in \{0,1\}$.

Let's start by taking an example:

Given the weights and profits of 'N' items, we are asked to put these items in a knapsack that has a capacity 'C'. The goal is to get the maximum profit from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

Let's take Mary's example, who wants to carry some fruits in the knapsack to get maximum profit. Here are the weights and profits of the fruits:

Items: $\{\ Apple, Orange, Banana, Melon\ \}$
Weights: $\{\ 2, 3, 1, 4\ \}$
Profits: $\{\ 4, 5, 3, 7\ \}$
Knapsack capacity: 5

Since it is a $0/1\ knapsack\ problem$, it means that we can pick a maximum of 1 item for each kind. Also, the problem is not a fractional knapsack problem but an integer one i.e., we can't break the items and we must pick the entire item or leave it.

Let's try to put different combinations of fruits in the knapsack, such that their total weight is not more than 5:

$$\begin{aligned}&Apple\ +\ Orange\ (total\ weight\ 5)\ => \ 9\ profit\\&Apple\ +\ Banana\ (total\ weight\ 3)\ => \ 7\ profit\\&Orange\ +\ Banana\ (total\ weight\ 4)\ => \ 8\ profit\\&Banana\ +\ Melon\ (total\ weight\ 5)\ => \ 10\ profit\end{aligned}$$

This shows that $\boldsymbol{Banana\ +\ Melon}$ is the best combination, as it gives us the maximum profit and the total weight does not exceed the capacity**.**

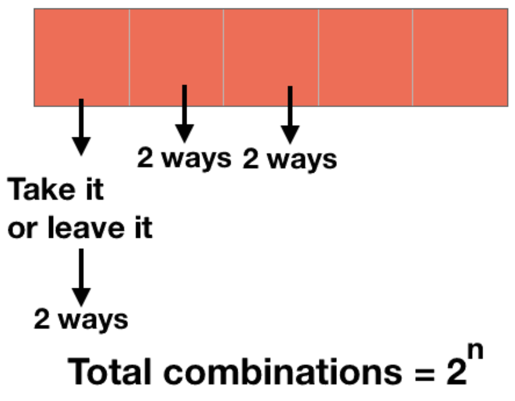
Problem Statement: Given two integer arrays to represent weights and profits of ‘N’ items, we need to find a subset of these items which will give us maximum profit such that their cumulative weight is not more than a given number ‘C’. Write a function that returns the maximum profit. Each item can only be selected once, which means either we put an item in the knapsack or skip it.

Brute Force Solution:

First take a case of solving the problem using brute force i.e., checking each possibility. As mentioned above, we have two options for each item i.e., either we can take it or leave it.

So, there are two ways for each item and a total of n items and hence, the total possible combinations are:

$$\underbrace{2 * 2 * 2 * \ldots * 2}_{n \text{ times}} = 2^n \text{ combinations}$$

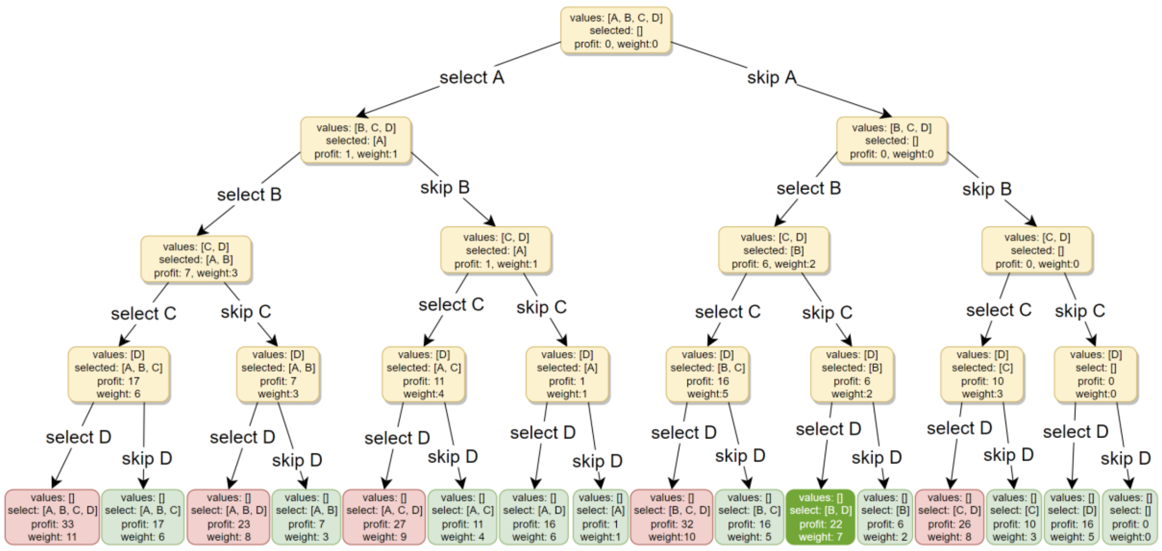


```
for each item 'i'
  create a new set which INCLUDES item 'i' if the total weight does not exceed the capacity, and
      recursively process the remaining capacity and items
  create a new set WITHOUT item 'i', and recursively process the remaining items
return the set from the above two sets with higher profit
```

Here is a visual representation of our algorithm:

items	A	B	C	D
profit	1	6	10	16
weight	1	2	3	5

Capacity: 7



values: [A, B, C, D] selected: [] profit: 0, weight:0
select A
skip A
values: [B, C, D] selected: [A] profit: 1, weight:1
values: [B, C, D] selected: [] profit: 0, weight:0
select B
skip B
select B
skip B
values: [C, D] selected: [A, B] profit: 7, weight:3
values: [C, D] selected: [A] profit: 1, weight:1
values: [C, D] selected: [B] profit: 6, weight:2
values: [C, D] selected: [] profit: 0, weight:0
select C
skip C
select C
skip C
select C
skip C
select C
skip C
values: [D] selected: [A, B, C] profit: 17 weight: 6
values: [D] selected: [A, B] profit: 7 weight:3
values: [D] selected: [A, C] profit: 11 weight:4
values: [D] selected: [A] profit: 1 weight:1
values: [D] selected: [B, C] profit: 16 weight:5
values: [D] selected: [B] profit: 6 weight:2
values: [D] selected: [C] profit: 10 weight:3
values: [D] select: [] profit: 0 weight:0
select D
skip D
select D
skip D
select D
skip D
select D
skip D
select D
skip D
select D
skip D
select D
skip D
select D
skip D
values: [] select: [A, B, C, D] profit: 33 weight: 11
values: [] select: [A, B, C] profit: 17 weight: 6
values: [] select: [A, B, D] profit: 23 weight: 8
values: [] select: [A, B] profit: 7 weight: 3
values: [] select: [A, C, D] profit: 27 weight: 9
values: [] select: [A, C] profit: 11 weight: 4
values: [] select: [A, D] profit: 16 weight: 6
values: [] select: [A] profit: 1 weight:1
values: [] select: [B, C, D] profit: 32 weight:10
values: [] select: [B, C] profit: 16 weight: 5
values: [] select: [B, D] profit: 22 weight: 7
values: [] select: [B] profit: 6 weight: 2
values: [] select: [C, D] profit: 26 weight: 8
values: [] select: [C] profit: 10 weight: 3
values: [] select: [D] profit: 16 weight: 5
values: [] select: [] profit: 0 weight:0

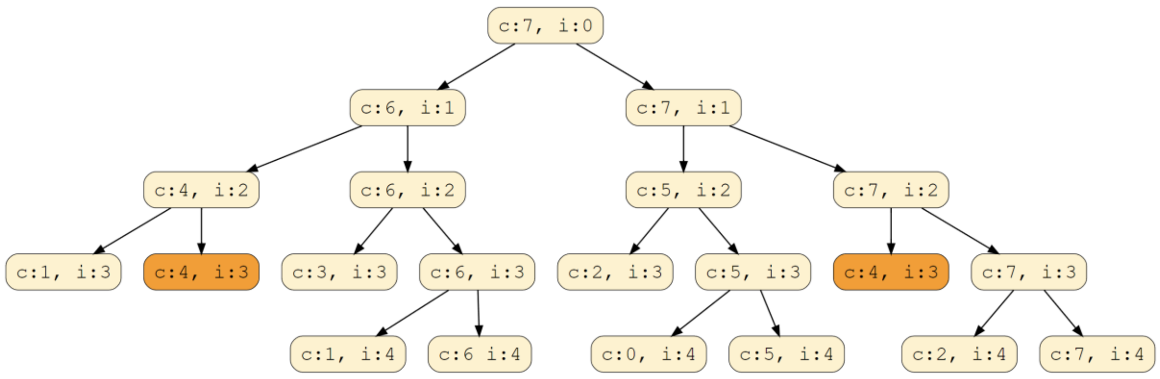
All green boxes have a total weight that is less than or equal to the capacity (7), and all the red ones have a weight that is more than 7. The best solution we have is with items [B, D] having a total profit of 22 and a total weight of 7.

The above algorithm's time complexity is exponential $O(2^n)$, where 'n' represents the total number of items. This can also be confirmed from the above recursion tree. As we can see that we will have a total of '31' recursive calls – calculated through $(2n) + (2n) - 1$, which is asymptotically equivalent to $O(2^n)$.

The space complexity is $O(n)$. This space will be used to store the recursion stack. Since our recursive algorithm works in a depth-first fashion, which means, we can't have more than 'n' recursive calls on the call stack at any time.

One can also think of a solution of always taking the item with the highest $\frac{value}{weight}$ ratio first (known as greedy algorithm) but it is also not going to help here. As mentioned above, it could have helped in the case of the fractional knapsack problem.

Let's visually draw the recursive calls to see if there are any overlapping sub-problems. As we can see, in each recursive call, profits and weights arrays remain constant, and only capacity and currentIndex change. For simplicity, let's denote capacity with 'c' and currentIndex with 'i':



We can clearly see that **'c:4, i=3'** has been called twice; hence we have an overlapping sub-problems pattern. As we discussed above, overlapping sub-problems can be solved through Memoization.

Top-down Dynamic Programming with Memoization:

We can use memoization to overcome the overlapping sub-problems. To reiterate, memoization is when we store the results of all the previously solved sub-problems and return the results from memory if we encounter a problem that has already been solved.

Since we have two changing values ($capacity$ and $currentIndex$) in our recursive function $knapsackRecursive()$, we can use a two-dimensional array to store the results of all the solved sub-problems. As mentioned above, we need to store results for every sub-array (i.e., for every possible index 'i') and for every possible capacity 'c'.

```
class Knapsack {
  public int solveKnapsack(int[] profits, int[] weights, int capacity) {
    Integer[][] dp = new Integer[profits.length][capacity + 1];
    return this.knapsackRecursive(dp, profits, weights, capacity, 0);
  }

  private int knapsackRecursive(Integer[][] dp, int[] profits, int[] weights, int
capacity,
      int currentIndex) {

    // base checks
    if (capacity <= 0 || currentIndex >= profits.length)
      return 0;
    // if we have already solved a similar problem, return the result from
memory
    if(dp[currentIndex][capacity] != null)
      return dp[currentIndex][capacity];

    // recursive call after choosing the element at the currentIndex
    // if the weight of the element at currentIndex exceeds the capacity, we
shouldn't process this
    int profit1 = 0;
    if( weights[currentIndex] <= capacity )
        profit1 = profits[currentIndex] + knapsackRecursive(dp, profits, weights,
                capacity - weights[currentIndex], currentIndex + 1);

    // recursive call after excluding the element at the currentIndex
    int profit2 = knapsackRecursive(dp, profits, weights, capacity, currentIndex +
1);

    dp[currentIndex][capacity] = Math.max(profit1, profit2);
    return dp[currentIndex][capacity];
  }

  public static void main(String[] args) {
    Knapsack ks = new Knapsack();
    int[] profits = {1, 6, 10, 16};
    int[] weights = {1, 2, 3, 5};
    int maxProfit = ks.solveKnapsack(profits, weights, 7);
    System.out.println("Total knapsack profit ---> " + maxProfit);
    maxProfit = ks.solveKnapsack(profits, weights, 6);
    System.out.println("Total knapsack profit ---> " + maxProfit);
  }
}
```

What is the time and space complexity of the above solution?

Since our memoization array $dp[profits.length][capacity + 1]$ stores the results for all the subproblems, we can conclude that we will not have more than $N * C$ subproblems (where 'N' is the number of items and 'C' is the knapsack capacity). This means that our time complexity will be $O(N * C)$.

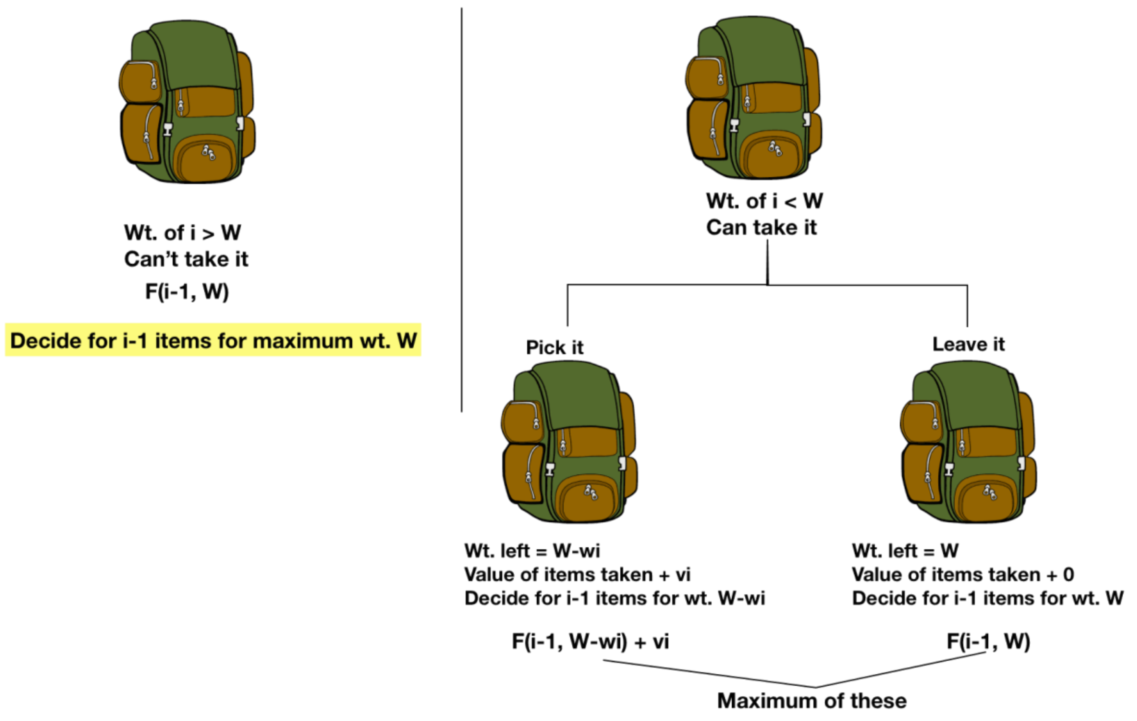
The above algorithm will be using $O(N * C)$ space for the memoization array. Other than that, we will use $O(N)$ space for the recursion call-stack. So the total space complexity will be $O(N * C \; + \; N)$, which is asymptotically equivalent to $O(N * C)$.

Bottom-up Dynamic Programming:

Let’s try to populate our $dp[][]$ array from the above solution, working in a bottom-up fashion. Essentially, we want to find the maximum profit for every sub-array and for every possible capacity. This means, $dp[i][c]$ will represent the maximum knapsack profit for capacity ‘c’ calculated from the first ‘i’ items.

Basically, we are developing a bottom-up approach to solve the problem. Let's say that we must make a decision for the first i elements to get the optimized value. Now, there can be two cases: the first that the weight of the i^{th} item is greater than the weight limit i.e., $w_i > W$, in this case, we can't take this item at all, so we are left with first $i - 1$ items and with the weight limit of W; the second case, when $w_i \leq W$, in this case, we will either take it or leave it.

Let's talk about the second case. If we pick the item, we will add the value of the i^{th} item (v_i) to the total value and then we are left with a total of first $i - 1$ items and the total weight limit will reduce to $W - w_i$ for rest of the items. And if we don't pick it, even then we are left with the first $i - 1$ items but the weight limit will be still W. And then we will choose the maximum of these two to get an optimized solution.



So, for each item at index 'i' ($0 <= i < items.length$) and capacity 'c' ($0 <= c <= capacity$), we have two options:

1. Exclude the item at index 'i'. In this case, we will take whatever profit we get from the sub-array excluding this $item => dp[i-1][c]$
2. Include the item at index 'i' if its weight is not more than the capacity. In this case, we include its profit plus whatever profit we get from the remaining capacity and from remaining $items => profits[i] + dp[i-1][c-weights[i]]$

Finally, our optimal solution will be maximum of the above two values:

$$dp[i][c] = max\ (dp[i-1][c], profits[i] + dp[i-1][c-weights[i]])$$

Here is the code for our bottom-up dynamic programming approach:

```
class Knapsack {
  public int solveKnapsack(int[] profits, int[] weights, int capacity) {
    // basic checks
    if (capacity <= 0 || profits.length == 0 || weights.length != profits.length)
      return 0;

    int n = profits.length;
    int[][] dp = new int[n][capacity + 1];

    // populate the capacity=0 columns, with '0' capacity we have '0' profit
    for(int i=0; i < n; i++)
      dp[i][0] = 0;

    // if we have only one weight, we will take it if it is not more than the
capacity
    for(int c=0; c <= capacity; c++) {
      if(weights[0] <= c)
        dp[0][c] = profits[0];
    }

    // process all sub-arrays for all the capacities
    for(int i=1; i < n; i++) {
      for(int c=1; c <= capacity; c++) {
        int profit1= 0, profit2 = 0;
        // include the item, if it is not more than the capacity
      if(weights[i] <= c)
        profit1 = profits[i] + dp[i-1][c-weights[i]];
      // exclude the item
      profit2 = dp[i-1][c];
      // take maximum
      dp[i][c] = Math.max(profit1, profit2);
     }
    }

    // maximum profit will be at the bottom-right corner.
    return dp[n-1][capacity];
  }

  public static void main(String[] args) {
    Knapsack ks = new Knapsack();
    int[] profits = {1, 6, 10, 16};
    int[] weights = {1, 2, 3, 5};
    int maxProfit = ks.solveKnapsack(profits, weights, 7);
    System.out.println("Total knapsack profit ---> " + maxProfit);
    maxProfit = ks.solveKnapsack(profits, weights, 6);
    System.out.println("Total knapsack profit ---> " + maxProfit);
  }
}
```

The above solution has a time and space complexity of $O(N * C)$ where 'N' represents total items, and 'C' is the maximum capacity.

How to find the selected items?

As we know that the final profit is at the bottom-right corner; therefore, we will start from there to find the items that will be going in the knapsack.

As you remember, at every step, we had two options: include an item or skip it. If we skip an item, then we take the profit from the remaining items (i.e., from the cell right above it); if we include the item, then we jump to the remaining profit to find more items.

Let's write a function to print the set of items included in the knapsack:

```
private void printSelectedElements(int dp[][], int[] weights, int[] profits, int
capacity){
  System.out.print("Selected weights:");
  int totalProfit = dp[weights.length-1][capacity];
  for(int i=weights.length-1; i > 0; i--) {
    if(totalProfit != dp[i-1][capacity]) {
      System.out.print(" " + weights[i]);
      capacity -= weights[i];
      totalProfit -= profits[i];
    }
  }

  if(totalProfit != 0)
    System.out.print(" " + weights[0]);
  System.out.println("");
 }
```

Space-optimized Solution

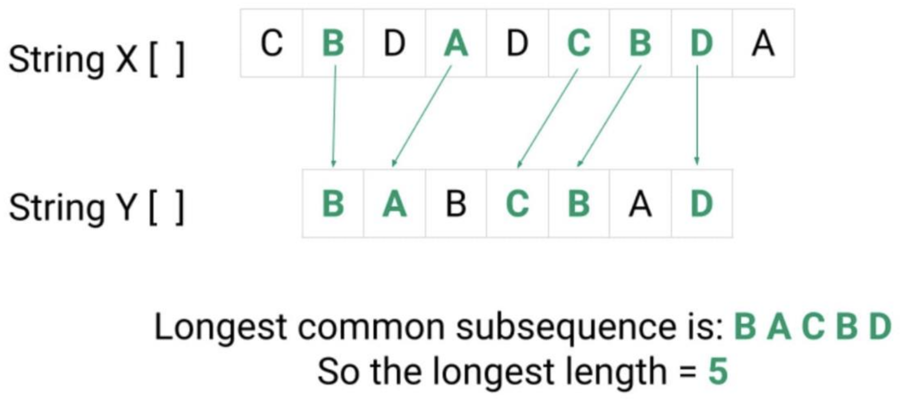
```
class Knapsack {
  static int solveKnapsack(int[] profits, int[] weights, int capacity) {
    // basic checks
    if (capacity <= 0 || profits.length == 0 || weights.length != profits.length)
      return 0;

    int n = profits.length;
    // we only need one previous row to find the optimal solution, overall we
need '2' rows
    // the above solution is similar to the previous solution, the only difference is
that
    // we use `i%2` instead if `i` and `(i-1)%2` instead if `i-1`
    int[][] dp = new int[2][capacity+1];

    // if we have only one weight, we will take it if it is not more than the
capacity
    for(int c=0; c <= capacity; c++) {
      if(weights[0] <= c)
        dp[0][c] = dp[1][c] = profits[0];
    }

    // process all sub-arrays for all the capacities
    for(int i=1; i < n; i++) {
      for(int c=0; c <= capacity; c++) {
        int profit1= 0, profit2 = 0;
        // include the item, if it is not more than the capacity
        if(weights[i] <= c)
          profit1 = profits[i] + dp[(i-1)%2][c-weights[i]];
        // exclude the item
        profit2 = dp[(i-1)%2][c];
        // take maximum
        dp[i%2][c] = Math.max(profit1, profit2);
      }
    }

    return dp[(n-1)%2][capacity];
  }
}
```

The above solution is similar to the previous solution; the only difference is that we use $i\%2$ instead of i and $(i-1)\%2$ instead of $i-1$. This solution has a space complexity of $O(2*C) = O(C)$, where 'C' is the knapsack's maximum capacity.

Longest Common Subsequence

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

If $S1$ and $S2$ are the two given sequences then, Z is the common subsequence of $S1$ and $S2$ if Z is a subsequence of both $S1$ and $S2$. Furthermore, Z must be a **strictly increasing sequence** of the indices of both $S1$ and $S2$.


String X [ ]
C B D A D C B D A
String Y [ ]
B A B C B A D
Longest common subsequence is: B A C B D
So the longest length = 5

Problem Statement:

Given two strings $text1$ and $text2$, return *the length of their longest* ***common subsequence****.* If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "$abcde$".

A **common subsequence** of two strings is a subsequence that is common to both strings.

Constraints:

- 1 <= text1.length, text2.length <= 1000
- text1 and text2 consist of only lowercase English characters.

Examples:

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length
is 3.
```

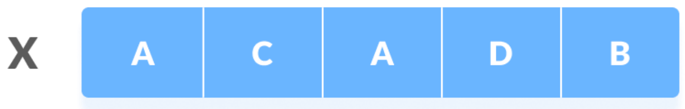
Example 2:

```
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length
is 3.
```

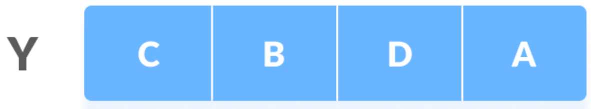
Example 3:

```
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is
0.
```

Using Dynamic Programming to find the LCS:



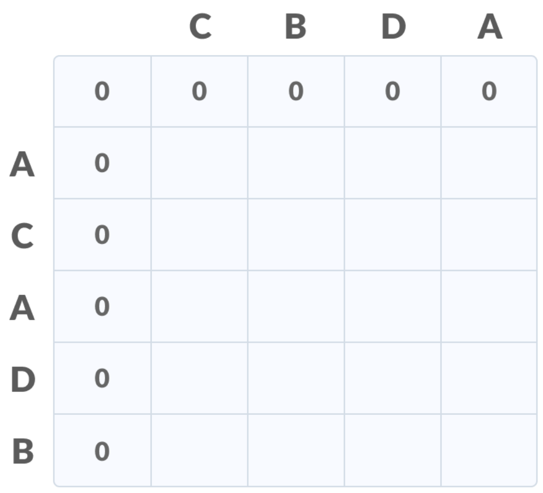
The first sequence



The second sequence

The following steps are followed for finding the longest common subsequence.

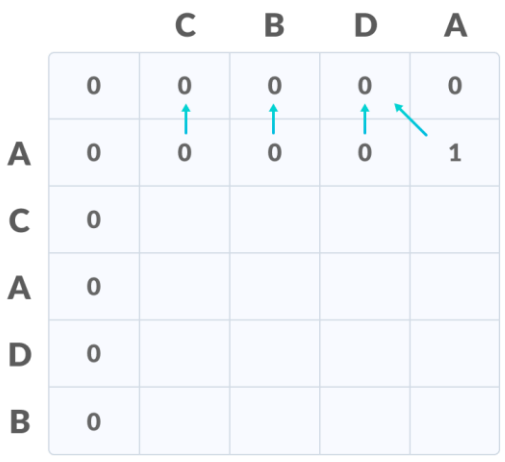
1. Create a table of dimension $n + 1 * m + 1$ where n and m are the lengths of X and Y respectively. The first row and the first column are filled with zeros.



		C	B	D	A
	0	0	0	0	0
A	0				
C	0				
A	0				
D	0				
B	0				

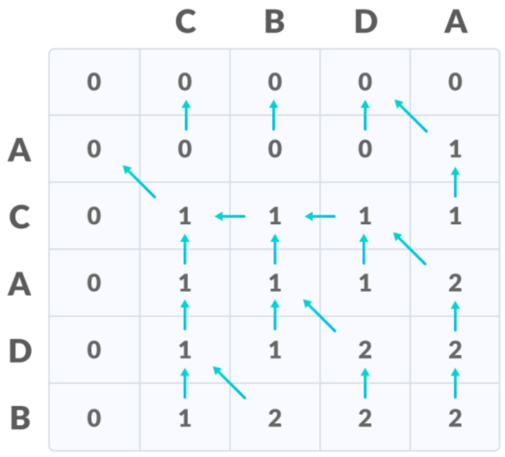
Initialise a table

2. Fill each cell of the table using the following logic.
3. If the character corresponding to the current row and current column are matching, then fill the current cell by adding one to the diagonal element. Point an arrow to the diagonal cell.
4. Else take the maximum value from the previous column and previous row element for filling the current cell. Point an arrow to the cell with maximum value. If they are equal, point to any of them.



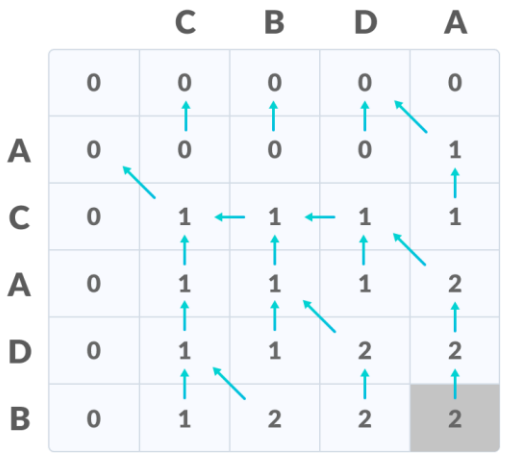
Fill the values

5. **Step 2** is repeated until the table is filled.



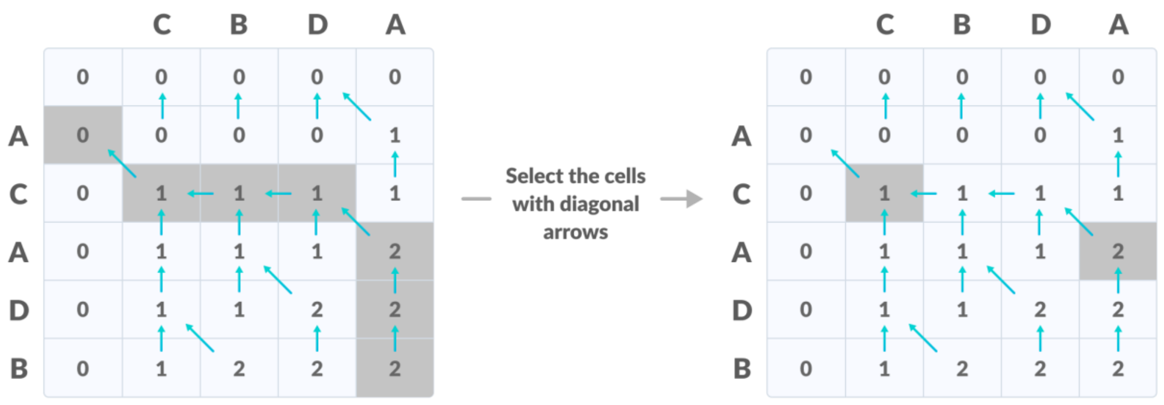
Fill the values

6. The value in the last row and the last column is the length of the longest common subsequence.



The bottom right corner is the length of the LCS

7. In order to find the longest common subsequence, start from the last element and follow the direction of the arrow. The elements corresponding to () symbol form the longest common subsequence.



Create a path according to the arrows

8. Thus, the longest common subsequence is CA.

LCS

How is a dynamic programming algorithm more efficient than the recursive algorithm while solving an LCS problem?

The method of dynamic programming reduces the number of function calls. It stores the result of each function call so that it can be used in future calls without the need for redundant calls.

In the above dynamic algorithm, the results obtained from each comparison between elements of X and the elements of Y are stored in a table so that they can be used in future computations.

So, the time taken by a dynamic approach is the time taken to fill the table (i.e., $O(mn)$). Whereas the recursion algorithm has the complexity of $2^{\max(m,n)}$.

Algorithm:

```
X and Y be two given sequences
Initialize a table LCS of dimension X.length * Y.length
X.label = X
Y.label = Y
```

```
LCS[0][] = 0
LCS[][0] = 0
Start from LCS[1][1]
Compare X[i] and Y[j]
    If X[i] = Y[j]
        LCS[i][j] = 1 + LCS[i-1, j-1]
        Point an arrow to LCS[i][j]
    Else
        LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])
        Point an arrow to max(LCS[i-1][j], LCS[i][j-1])
```

Implementation:

```
class LCS_ALGO {
  static void lcs(String S1, String S2, int m, int n) {
    int[][] LCS_table = new int[m + 1][n + 1];

    // Building the mtrix in bottom-up way
    for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
        if (i == 0 || j == 0)
          LCS_table[i][j] = 0;
        else if (S1.charAt(i - 1) == S2.charAt(j - 1))
          LCS_table[i][j] = LCS_table[i - 1][j - 1] + 1;
        else
          LCS_table[i][j] = Math.max(LCS_table[i - 1][j], LCS_table[i][j - 1]);
      }
    }

    int index = LCS_table[m][n];
    int temp = index;

    char[] lcs = new char[index + 1];
    lcs[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
      if (S1.charAt(i - 1) == S2.charAt(j - 1)) {
        lcs[index - 1] = S1.charAt(i - 1);
```

```
          i--;
          j--;
          index--;
        }

        else if (LCS_table[i - 1][j] > LCS_table[i][j - 1])
          i--;
        else
          j--;
      }

      // Printing the sub sequences
      System.out.print("S1 : " + S1 + "\nS2 : " + S2 + "\nLCS: ");
      for (int k = 0; k <= temp; k++)
        System.out.print(lcs[k]);
      System.out.println("");
    }

    public static void main(String[] args) {
      String S1 = "ACADB";
      String S2 = "CBDA";
      int m = S1.length();
      int n = S2.length();
      lcs(S1, S2, m, n);
    }
  }
```

Applications:

- in compressing genome resequencing data
- to authenticate users within their mobile phone through in-air signatures

Problem Statements

Coin Change Problem

Explanation:

In the coin change problem, we are basically provided with coins with different denominations like 1¢, 5¢ and 10¢. Now, we have to make an amount by using these coins such that a minimum number of coins are used.

Let's take a case of making 10¢ using these coins, we can do it in the following ways:

1. Using 1 coin of 10¢
2. Using two coins of 5¢
3. Using one coin of 5¢ and 5 coins of 1¢
4. Using 10 coins of 1¢

Out of these 4 ways of making 10¢, we can see that the first way of using only one coin of 10¢ requires the least number of coins and thus it is our solution.

So in a coin change problem, we are provided with different denominations of coins:

$$1 = d_1 < d_2 < d_3 < \cdots < d_k$$

$d_1 = 1$ ensures that we can make any amount using these coins.

Now, we have to make change for the value nn using these coins and we need to find out the minimum number of coins required to make this change.

Problem Statement:

Given a value N, if we want to make change for N cents, and we have infinite supply of each of $S = \{ S_1, S_2, \dots , S_m\}$valued coins, how many ways can we make the change? The order of coins doesn't matter.

Example:

For $N = 4$ and $S = \{1,2,3\}$, there are four solutions:

$$\{1,1,1,1\}$$

$$\{1,1,2\}$$

$$\{2,2\}$$

$$\{1,3\}.$$

So, the output should be 4.

For $N = 10$ and $S = \{2, 5, 3, 6\}$, there are five solutions:

$$\{2,2,2,2,2\}$$

$$\{2,2,3,3\}$$

$$\{2,2,6\}$$

$$\{2,3,5\}$$

$$\{5,5\}.$$

So, the output should be 5.

Equal Subset Sum Partition

Problem Statement:

Given a set of positive numbers, find if we can partition it into two subsets such that the sum of elements in both the subsets is equal.

Examples:

Example 1:

```
Input: {1, 2, 3, 4}
Output: True
Explanation: The given set can be partitioned into two subsets wit
h equal sum: {1, 4} & {2, 3}
```

Example 2:

```
Input: {1, 1, 3, 4, 7}
Output: True
Explanation: The given set can be partitioned into two subsets wit
h equal sum: {1, 3, 4} & {1, 7}
```

Example 3:

```
Input: {2, 3, 4, 6}
Output: False
Explanation: The given set cannot be partitioned into two subsets wi
th equal sum.
```

Longest Increasing Subsequence (LIS)

Explanation:

The Longest Increasing Subsequence (LIS) problem is to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

For example, the length of LIS for $\{10, 22, 9, 33, 21, 50, 41, 60, 80\}$ is 6 and LIS is $\{10, 22, 33, 50, 60, 80\}$.

arr[]	10	22	9	33	21	50	41	60	80
LIS	1	2		3		4		5	6

Examples:

```
Input: arr[] = {3, 10, 2, 1, 20}
Output: Length of LIS = 3. The longest increasing subsequence is {3,
10, 20}.

Input: arr[] = {3, 2}
Output: Length of LIS = 1. The longest increasing subsequence are {3}
and {2}.
```

Binomial Coefficients

Explanation:

Following figure shows the General formula to expand the algebraic equations by using Binomial Theorem,

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Where,

$\sum$ = Known as “Sigma Notation” used to sum all the terms in expansion from k = 0 to k = n

n = positive integer power of algebraic equation

$\binom{n}{k}$ = read as “n choose k”

According to theorem, expansion goes as following for any of the algebric equation containing any positive power,

$$(x+y)^n = \binom{n}{0}x^n y^0 + \binom{n}{1}x^{n-1}y^1 + \binom{n}{2}x^{n-2}y^2 + \cdots + \binom{n}{n-1}x^1 y^{n-1} + \binom{n}{n}x^0 y^n$$

As after getting done by expansion one task remains pending that is to calculate the Binomial Coefficients $\left({}^{n}_{k}C\right)$.

A binomial coefficient $C(n,k)$ can be defined as the coefficient of x^k in the expansion of $(1 + x)^n$.

A binomial coefficient $C(n,k)$ also gives the number of ways, disregarding order, that k objects can be chosen from among n objects more formally, the number of k-element subsets (or k-combinations) of a n-element set.

Problem Statement:

Write a function that takes two parameters n and k and returns the value of Binomial Coefficient $C(n,k)$.

Example:

Your function should return 6 for $n = 4$ and $k = 2$, and it should return 10 for n = 5 and k = 2.

Input:

```
Enter the value of n: 4
Enter the value of k: 2
```

Output:

```
Value of C(4, 2) : 6
```

Assessment

1) Which of the following is/are property/properties of a dynamic programming problem?
   a) Optimal substructure
   b) Overlapping subproblems
   c) Greedy approach
   d) Both optimal substructure and overlapping subproblems
2) If an optimal solution can be created for a problem by constructing optimal solutions for its subproblems, the problem possesses ____________ property.
   a) Overlapping subproblems
   b) Optimal substructure
   c) Memoization
   d) Greedy
3) If a problem can be broken into subproblems which are reused several times, the problem possesses ____________ property.
   a) Overlapping subproblems
   b) Optimal substructure
   c) Memoization
   d) Greedy
4) In dynamic programming, the technique of storing the previously calculated values is called ___________
   a) Saving value property
   b) Storing value property
   c) Memoization
   d) Mapping
5) A greedy algorithm can be used to solve all the dynamic programming problems.
   a) True
   b) False
6) When a top-down approach of dynamic programming is applied to a problem, it usually _____________
   a) Decreases both, the time complexity and the space complexity
   b) Decreases the time complexity and increases the space complexity
   c) Increases the time complexity and decreases the space complexity
   d) Increases both, the time complexity and the space complexity

7) Which of the following problems should be solved using dynamic programming?
   a) Mergesort
   b) Binary search
   c) Longest common subsequence
   d) Quicksort
8) What is the time complexity of the recursive implementation used to find the nth Fibonacci term?
   a) O(1)
   b) $O(n^2)$
   c) O(n!)
   d) Exponential
9) Suppose you have coins of denominations 1, 3 and 4. You use a greedy algorithm, in which you choose the largest denomination coin which is not greater than the remaining sum. For which of the following sums, will the algorithm NOT produce an optimal answer?
   a) 20
   b) 12
   c) 6
   d) 5
10) You are given infinite coins of denominations 1, 3, 4. What is the minimum number of coins required to achieve a sum of 7?
   a) 1
   b) 2
   c) 3
   d) 4