Dynamic Programming

We have seen in the Greedy Algorithm, that we always choose the local optimal solution but in dynamic programming, we solve all the subproblems and choose the best one which gives us the global optimal solution.

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. It is a method used to optimize recursive algorithms by ensuring that each subproblem is solved only once. Instead of solving the same subproblems repeatedly, which is common in naive recursive solutions, dynamic programming stores the results of subproblems in a table as they are solved. This way, if a solution to a subproblem is needed again, it can simply be looked up rather than recomputed, trading off memory for time savings. This trade-off can significantly improve efficiency, as it usually reduces the time complexity from exponential to polynomial.

Dynamic programming is applicable when there is:

- **Optimal Substructure**: A problem has optimal substructure if the optimal solution of the whole problem can be built from the optimal solutions of its subproblems.

- **Overlapping Subproblems:** A problem has overlapping subproblems if the problem can be broken down into subproblems that are reused multiple times.

Dynamic programming typically applies to optimization problems. Such problems can have many possible solutions. Each solution has a value, and you want to find a solution with the optimal (minimum or maximum) value.

There are two main approaches to implementing dynamic programming:

- **Top-down with memoization:**

In the top-down approach, the algorithm starts solving the problem from the "top" (the original, larger problem) and breaks it down into smaller subproblems as needed. It begins with the main problem and recursively breaks it down into its constituent subproblems, solving each one as it encounters it. If the solution to a subproblem has already been computed (and stored), it is reused instead of recomputed. This approach is called "top-down" because it starts from the main problem and works its way "down" to the smaller subproblems.

- **Bottom-up:**

In the bottom-up approach, the algorithm starts solving from the "bottom" (the smallest or simplest subproblems) and builds up to the larger problem. It begins by solving the smallest subproblems first, then combines their solutions to solve progressively larger subproblems, until it finally arrives at the solution to the main problem. This approach is called "bottom-up" because it starts with the smallest subproblems and works its way "up" to solve the entire problem.

These two approaches yield algorithms with the same asymptotic running time, except in unusual circumstances where the top-down approach does not actually recurse to examine all possible subproblems. The bottom-up approach often has much better constant factors, since it has lower overhead for procedure calls.

Let's see an example through a popular concept: Fibonacci numbers

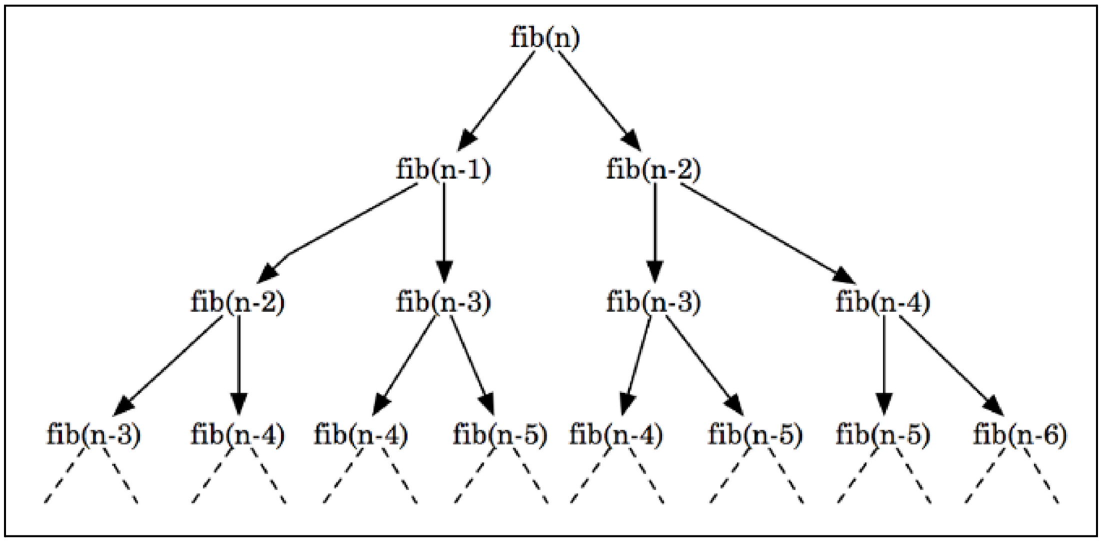
The Fibonacci numbers are given by the following sequence:

0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89, . . .

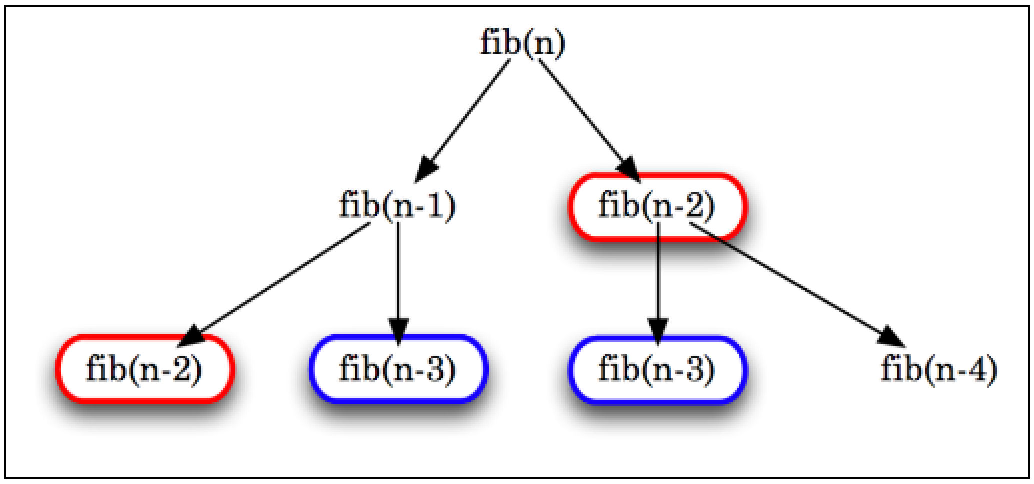
Recursive algorithm of Fibonacci (Top-down):

```
FIBONACCI($n$)
1  if $n = 0$ or $n = 1$
2      then return $n$
3  else return FIBONACCI($n - 1$) + FIBONACCI($n - 2$)
```

Recursive Tree:

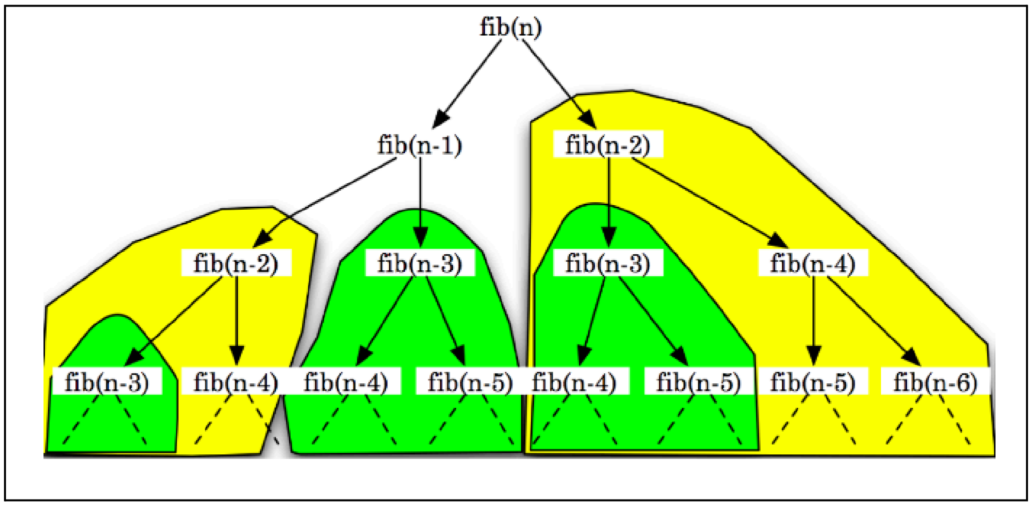


Time Complexity: The total number of nodes is $2^0 + 2^1 + 2^2 + .. + 2^n = O(2^n)$, so $T(n) = O(2^n)$
We can see some redundant computations:



Note how fib(n-2) and fib(n-3) are each being computed twice.

In fact, computing fib(n-2) involves computing a whole subtree. Likewise for computing fib(n-3).



What if we compute and save the results of fib(n-2) and fib(n-3) the first time, and then reuse them each time afterward? This is called Memoization.

Memoization:

The process of saving solutions to subproblems that can be reused later without redundant computations.

Typically, the solutions to subproblems are saved in a global array, which are later looked up and reused as needed.

1. At each step of computation, first see if the solution to the subproblem has already been found and saved.
2. If so, simply return the solution.
3. If not, compute the solution and save it before returning the solution.

Memoized Recursive algorithm of Fibonacci (Top-down):

```
global $F = [0 \,..\, n]$

M-FIBONACCI($n$)
if $n = 0$ or $n = 1$
    then return $n$
if $F[n]$ is empty
    then $F[n] \leftarrow$ M-FIBONACCI($n - 1$) + M-FIBONACCI($n - 2$)
return $F[n]$
```

What is global array F? It is a global array that stores the results of the subproblems. Initially, it must be all empty.

```
FIBONACCI($n$)
for $i \leftarrow 0$ to $n$
        do $F[i] \leftarrow -1$
return M-FIBONACCI($F, n$)

M-FIBONACCI($F, n$)
if $n \leq 1$
    then return $n$
if $F[n] = -1$
    then $F[n] \leftarrow$ M-FIBONACCI($F, n - 1$) + M-FIBONACCI($F, n - 2$)
return $F[n]$
```

Time Complexity: Each element F[2] . . . F[n] is filled in just once in O(1) time, so T(n) = O(n).

The idea is to reuse saved solutions, trading off space for time. Any recursive algorithm can be memoized, but only helps if there is redundancy in computing solutions to subproblems (if there are overlapping subproblems). This is often called Top-down Dynamic Programming.

Dynamic Programming algorithm of Fibonacci (Bottom-Up):

```
global $F = [0 \,..\, n]$

FIBONACCI($n$)
1  $F[0] \leftarrow 0$
2  $F[1] \leftarrow 1$
3  for $i \leftarrow 2$ to $n$
4          do $F[i] \leftarrow F[i-1] + F[i-2]$
5  return $F[n]$
```

Idea: Why not build up the solution bottom-up, starting from the base case(s) all the way to n?
This is often called Bottom-up Dynamic Programming.

Time Complexity: Each element F[0] . . . F[n] is filled in just once in O(1) time, so T(n) = O(n).

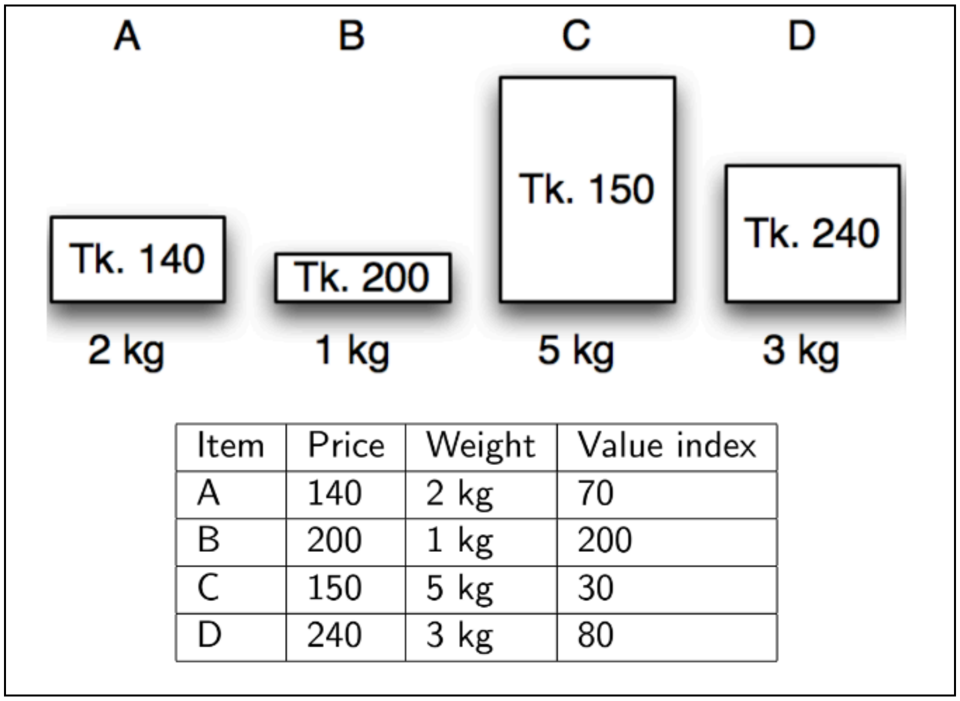
Knapsack Problem:

- **Fractional Knapsack:**

Given N items, each item has some weight and price or profit associated with it, and a bag with capacity W is also given, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag so that the sum of price/profits associated with them is the maximum possible. In fractional knapsack, we can take fractional quantities, for example, if an item is 2 kg, we can take 0.5 kg from it if needed.

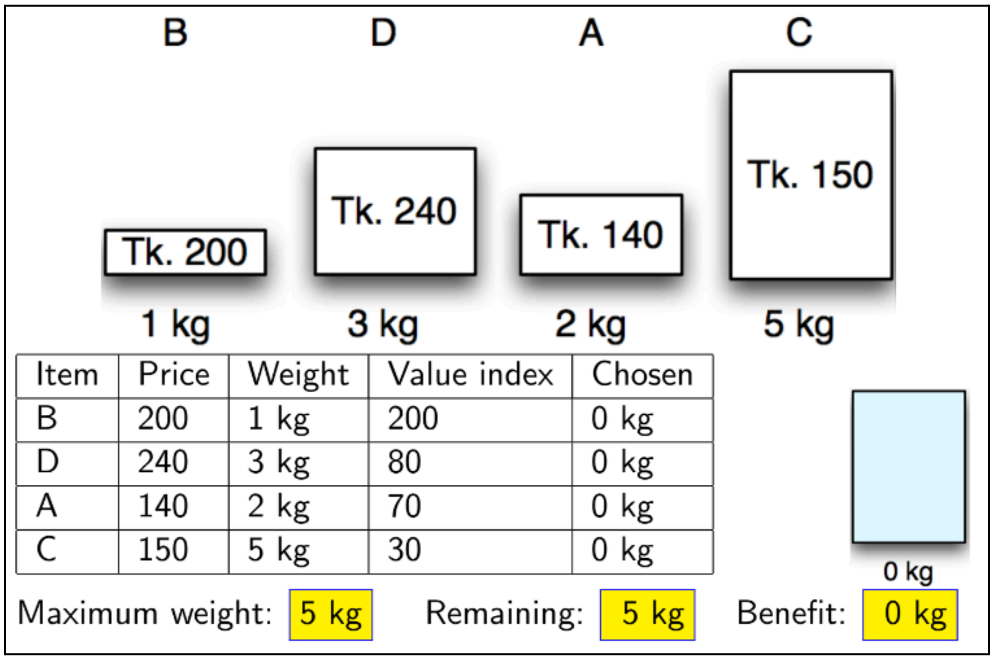
Greedy Approach:

For example, if we have the following items with the price and weight. The maximum weight is 5 kg.



Item	Price	Weight	Value index
A	140	2 kg	70
B	200	1 kg	200
C	150	5 kg	30
D	240	3 kg	80

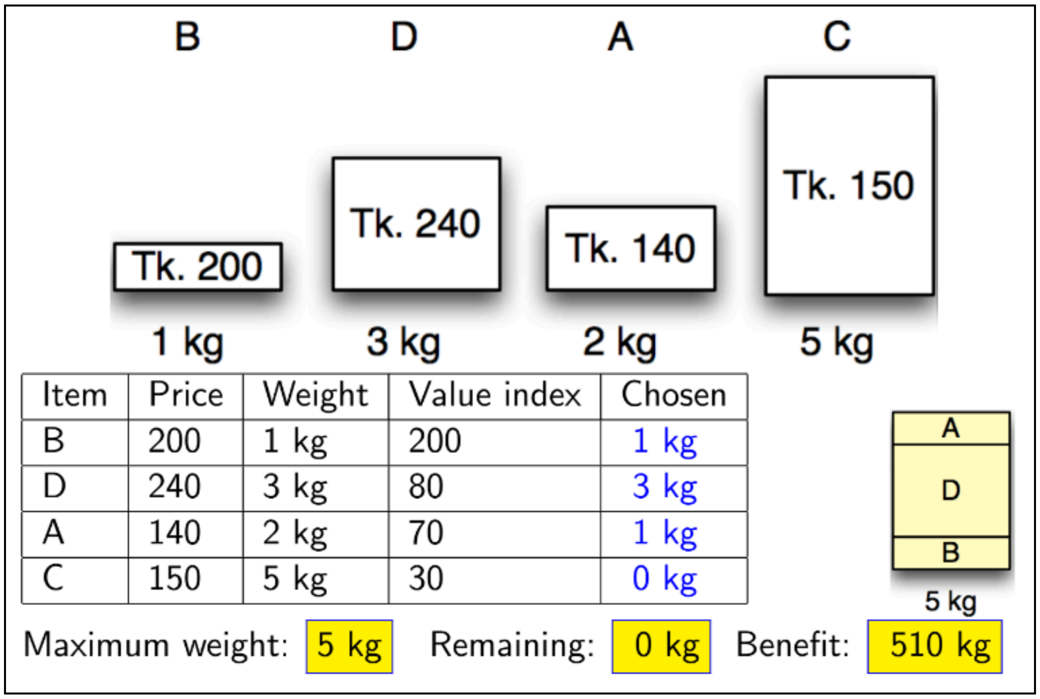
For a greedy approach, we can first calculate the unit price or value index that is price/weight. As a higher value index indicates more price or profit per weight, we can sort the items by decreasing value index.



Item	Price	Weight	Value index	Chosen
B	200	1 kg	200	0 kg
D	240	3 kg	80	0 kg
A	140	2 kg	70	0 kg
C	150	5 kg	30	0 kg

Maximum weight: 5 kg Remaining: 5 kg Benefit: 0 kg

We can then keep picking the items with the highest value index until the maximum weight is hit. (The maximum weight in this example is 5 kg). In fractional knapsack, we can take fractional quantities, for example, if an item is 2 kg, we can take 0.5 kg from it if needed.



Item	Price	Weight	Value index	Chosen
B	200	1 kg	200	1 kg
D	240	3 kg	80	3 kg
A	140	2 kg	70	1 kg
C	150	5 kg	30	0 kg

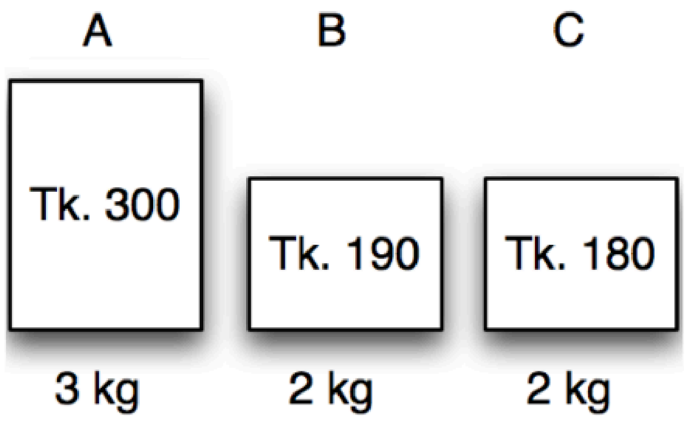
Maximum weight: 5 kg Remaining: 0 kg Benefit: 510 kg

- **0/1 or Unbounded Knapsack:**

In contrast to the fractional knapsack, we cannot take fractional quantities; for example, if an item is 2 kg, we either have to take the whole item that is 2 kg or we will not take the item that is 0 kg.

It is not possible to solve the 0/1 Knapsack using the Greedy Approach.

For example, if we have the following items with the price and weight. The maximum weight is 4 kg.



Item	Price	Weight	Value index
A	300	3 kg	100
B	190	2 kg	95
C	180	2 kg	90

Maximum weight: 4 kg

Greedy solution: item A Benefit: 300

Optimal solution: items B and C Benefit: 370

Recursive algorithm of 0/1 Knapsack:

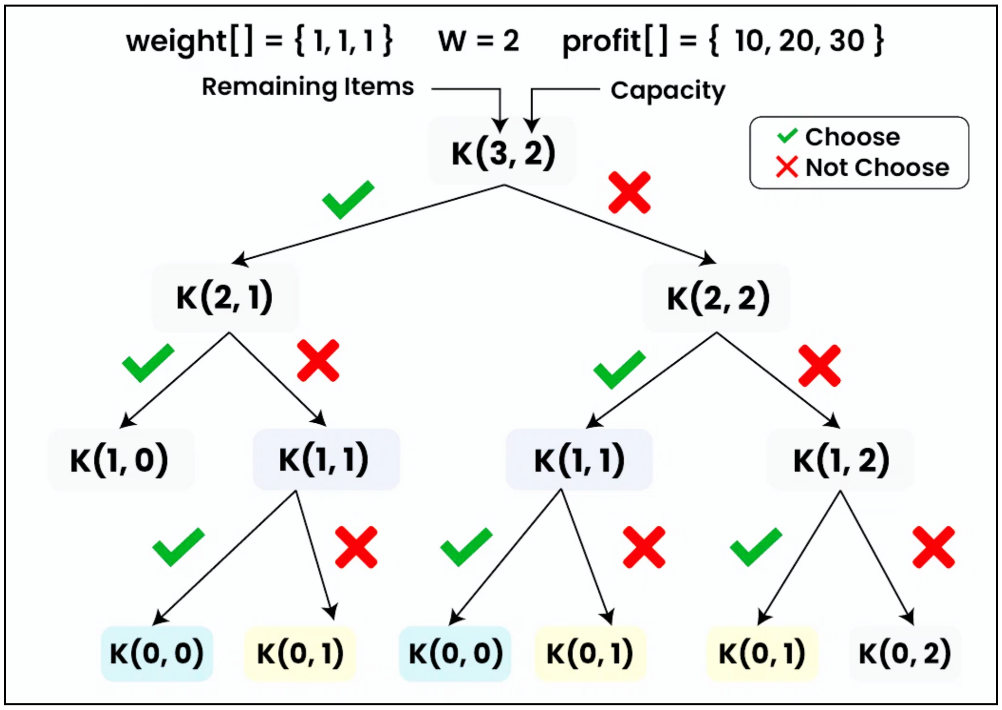
The 0/1 knapsack problem is divided into smaller sub-problems, where we take each item and check if we should include it in our knapsack. If the weight of an item (w_j) is smaller than or equal to the weight of the knapsack (W), we have two choices: either to include the item in the knapsack or to avoid it. We will include the item if it maximizes the total price or profit.

The base case is when there is no item in our list (n = 0) of items or the weight of the knapsack is equal to 0 (W = 0). It returns 0 because the price is 0 when n = 0 or W = 0.

Next, we need to make choices; if we decide to include the item in our knapsack, we should add the price of the current item v_j and reduce the weight of the item from the total weight the knapsack can hold (W - w_j). Regardless of whether we include our item in the knapsack, we should remove that item from our list while making the next recursive call, hence, we should decrement the value of n (n - 1).

```
KNAPSACK($n$, $W$)
1  if $n = 0$ or $W = 0$
2      then return 0
3  elif $w_j > W$
4      then return KNAPSACK($n - 1, W$))
5  else return MAX($v_j$ + KNAPSACK($n - 1, W - w_j$),
                        KNAPSACK($n - 1, W$))
```

Recursive Tree:



The initial call is Knapsack(n, W), where n is the number of items and W is the maximum weight or weight of the knapsack.

The solution involves trying all possible combinations of items to find the maximum profit. For every item, it has a binary choice: include or exclude. If there are n items, there will be 2^n combinations.

Time Complexity: $T(n) = O(2^n)$.

Memoized Recursive algorithm of 0/1 Knapsack (Top-down):

From the above recursive tree, we can see that there are some overlapping subproblems. K(1,1) is called twice. This increases time complexity. We can use memoization to make the solution efficient.

```
M-KNAPSACK$(n, W)$
if $n = 0$ or $W = 0$
    then return 0
elif $M[n, W]$ is empty
    if $w_j > W$
        then $M[n, W] \leftarrow$ KNAPSACK$(n-1, W))$
    else $M[n, W] \leftarrow \text{MAX}(v_j +$ M-KNAPSACK$(n-1, W-w_j)$,
                                M-KNAPSACK$(n-1, W))$
else return $M[n, W]$
```

Here, M is a 2D array or a matrix that saves the results of the recursion calls and can be used later.

Time Complexity: Each entry in M[j,w] gets filled in only once at O(1) time, and there are (n + 1) × (W + 1) entries, so M-Knapsack(n, W) takes O(nW) time.

Dynamic Programming algorithm of 0/1 Knapsack (Bottom-up):

```
KNAPSACK$(n, W)$
for $i \leftarrow 0$ to $n$
    do $M[i, 0] \leftarrow 0$
for $w \leftarrow 0$ to $W$
    do $M[0, w] \leftarrow 0$
for $j \leftarrow 1$ to $n$
    do for $w \leftarrow 1$ to $W$
        do if $w_j > w$
            then $M[j, w] = M[j-1, w]$
            else $M[j, w] \leftarrow \text{MAX}(v_j + M[j-1, w-w_j]$,
                                   $M[j-1, w])$
return $M[n, W]$
```

For example,

x_1 x_2 x_3 x_4
Given, W = 8, w = [2, 3, 5, 4],
v = [1, 2, 6, 5],

Initially:

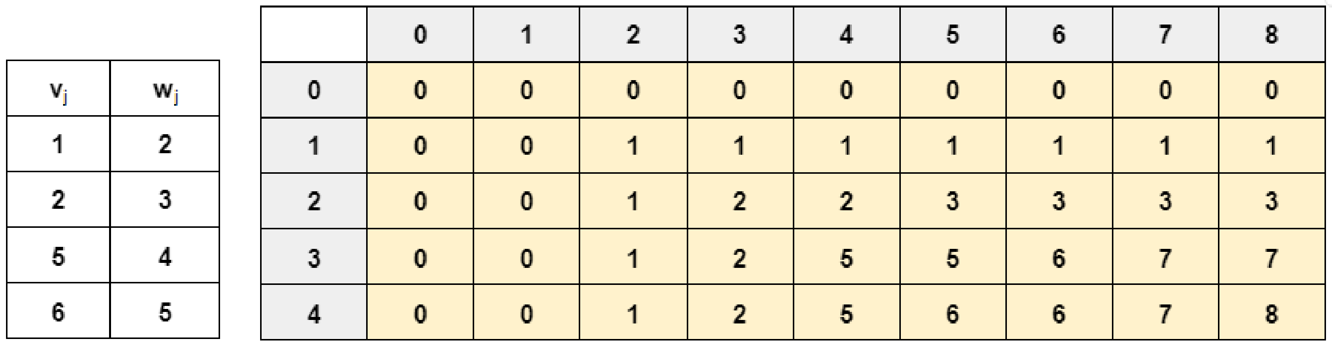
Create a table with (n + 1) × (W + 1), where n is the number of items and W is the capacity.
Fill the base cases (n = 0 or W = 0) with 0's.

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

Final Table:

The items are sorted in ascending order according to their weights and then the table is filled.
After sorting, the items are x_1 x_2 x_4 x_3.

If $w_j > w$, copy from the upper cell, else take max(v_j + upper row's $w - w_j$, upper cell)



v_j	w_j
1	2
2	3
5	4
6	5

	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

Time Complexity: Each entry in M[j,w] gets filled in only once at O(1) time, and there are (n + 1) × (W + 1) entries, so it takes O(nW) time.
Backtracking:

Items: x_1 x_2 x_4 x_3

Row Count	Item Count
4th row	Extract M[n,W] = 8 Since, 8 is not in the previous row, include 4th row (x_3) Items: x_1 x_2 x_4 x_3 (1) Profit of 4th row (x_3) is 6, Remaining = 8 - 6 = 2
3rd row	Remaining = 2 Since, 2 is in the previous row, do not include 3rd row (x_4) Items: x_1 x_2 x_4 (0) x_3 (1)
2nd row	Remaining = 2 Since, 2 is not in the previous row, include 2nd row (x_2) Items: x_1 x_2 (1) x_4 (0) x_3 (1) Profit of 2nd row (x_2) is 2, Remaining = 2 - 2 = 0
1st row	Remaining = 0 Since, 0 is in the previous row, do not include 1st row (x_2) Items: x_1 (0) x_2 (1) x_4 (0) x_3 (1)

Final sequence in original order = x_1 (0) x_2 (1) x_3 (1) x_4 (0)
Total Weight = 8
Total Profit = 8

Longest Common Subsequence (LCS):

The Longest Common Subsequence (LCS) problem involves finding the longest subsequence that is common to two sequences. A subsequence is a sequence that appears in the same relative order but not necessarily contiguously. A subsequence of a sequence is a new sequence derived from the original sequence by deleting some or no elements without changing the order of the remaining elements.

For example:

String 1: "abcdefghij"
String 2: "cadgi"

The LCS of these two sequences is "cdgi" as it is the longest sequence that appears in both sequences in the same order.

Recursive algorithm of LCS:

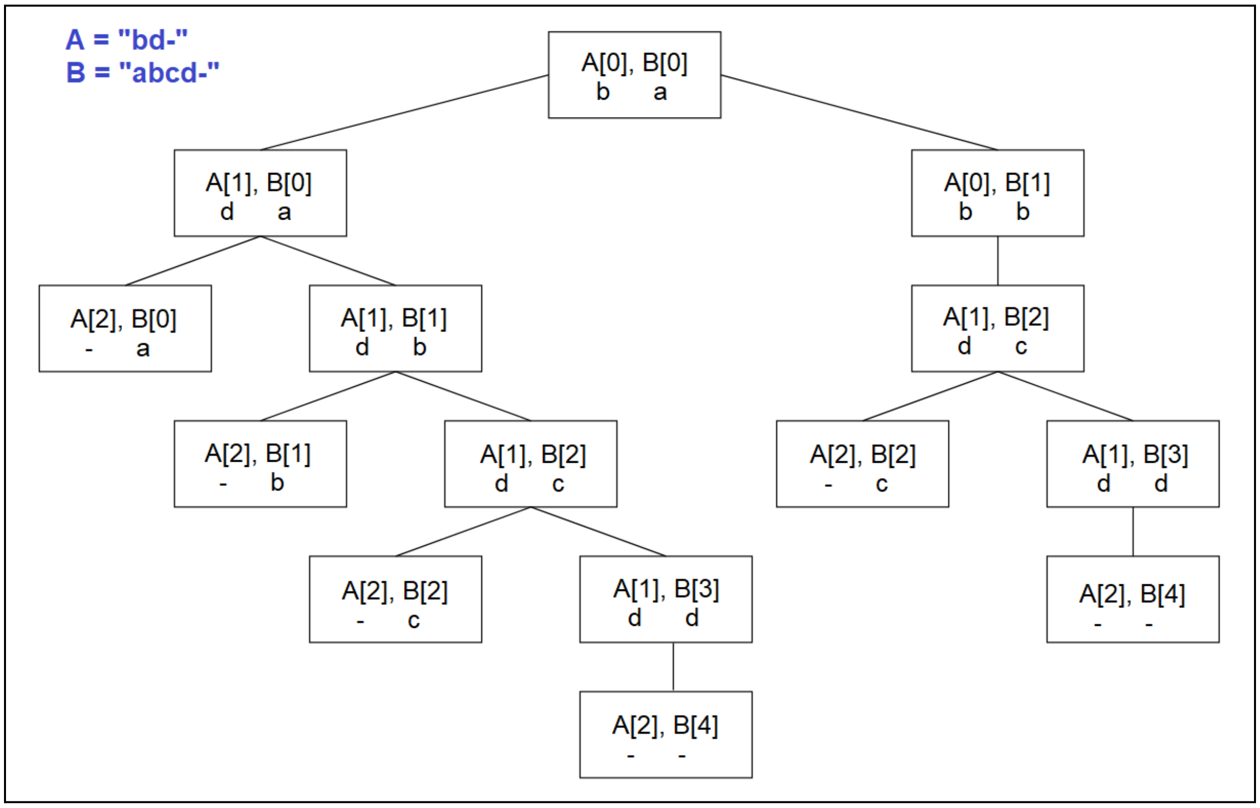
The LCS problem is divided into smaller sub-problems, where we take each character and check if we should include it or not. We will include the character if it maximizes the length of the common subsequence.

The base case is when there is no character left in sequence A or in sequence B. We will add "-" at the end of both sequences to indicate the end of the sequence. It returns 0 when the index i or the index j reaches "-".

Next, we need to make choices; if we decide to include the character, we should add 1 to keep count and increment both the index (i + 1 and j + 1) while making the next recursion to check the next character. If the characters do not match, we will increment both indexes separately while making the recursion calls and take the maximum one to reach the longest common subsequence.

```
LCS($i, j$)
if A[$i$] == "-" or B[$j$] == "-"
    then return 0
elif A[$i$] == B[$j$]
    then return 1 + LCS($i + 1, j + 1$)
else return MAX(LCS($i + 1, j$),
                LCS($i, j + 1$))
```

Recursive Tree:



The initial call is LCS(i, j) where i is the index that iterates the sequence A and j is the index that iterates the sequence B.

The solution involves trying all possible subsequences of the input sequences to determine the longest common subsequence. In the worst case, every character has a binary choice: include or exclude. If there are n characters in A and m characters in B, there will be $2^n * 2^m = 2^{n+m}$ combinations.

Time Complexity: $T(n) = O(2^{n+m})$.

Memoized Recursive algorithm of LCS (Top-down):

There are many overlapping subproblems, so the obvious choice is to memoize the recursion.

```
M-LCS(i, j)
if A[i] == "-" or B[j] == "-"
    then return 0
elif M[i, j] is empty
    do if A[i] == B[j]
        then M[i, j] ← 1 + M-LCS(i + 1, j + 1)
        else M[i, j] ← MAX(M-LCS(i + 1, j),
                           M-LCS(i, j + 1))
return M[i, j]
```

Here, M is a 2D array or a matrix that saves the results of the recursion calls and can be used later.

Time Complexity: Each entry in M[i,j] gets filled in only once at O(1) time, and there are (n + 1) × (m + 1) entries, where n is the number of characters in sequence A and m is the number of characters in sequence B. So M-LCS(i, j) takes O(nm) time.

Dynamic Programming algorithm of LCS (Bottom-up):

```
LCS(n, m)
for i ← 0 to n
    do M[i, 0] ← 0
for j ← 0 to m
    do M[0, j] ← 0
for i ← 1 to n
    do for j ← 1 to m
        do if A[i] == B[j]
            then M[i, j] ← 1 + M[i - 1, j - 1]
            else M[i, j] ← MAX(M[i - 1, j],
                               M[i, j - 1])
return M[n, m]
```

Here, the parameters of LCS are n and m where n is the number of characters in sequence A and m is the number of characters in sequence B.

For example,

Given, A = "longest" and B = "stone"

Initially:

Create a table with (n + 1) × (m + 1), where n is the length of A and m is the length of B.
Fill the base cases (n = 0 or m = 0) with 0's.

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

Final Table:

If characters match, 1 + leftmost diagonal cell, else take max(left cell, upper cell)

			l	o	n	g	e	s	t
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
s	1	0	0	0	0	0	0	1	1
t	2	0	0	0	0	0	0	1	2
o	3	0	0	1	1	1	1	1	2
n	4	0	0	1	2	2	2	2	2
e	5	0	0	1	2	2	3	3	3

Time Complexity: Each entry in M[i,j] gets filled in only once at O(1) time, and there are (n + 1) × (m + 1) entries, where n is the number of characters in sequence A and m is the number of characters in sequence B. So LCS(n, m) takes O(nm) time.

Backtracking:

A: "longest"

Column Count	**Item Count**
7th col	Extract M[n,m] = 3 Since, 3 is achieved from left cell M[5,6], do not include 7th col Subsequence: "longest (0)" Move to cell M[5,6]
6th col	Current cell M[5,6] = 3 Since, 3 is achieved from left cell M[5,5], do not include 6th col Subsequence: "longes (0) t (0)" Move to cell M[5,5]
5th col	Current cell M[5,5] = 3 Since, 3 is achieved from diagonal cell M[4,4], include 5th col Subsequence: "longe (1) s (0) t (0)" Move to cell M[4,4]
4th col	Current cell M[4,4] = 2 Since, 2 is achieved from left cell M[4,3], do not include 4th col Subsequence: "long (0) e (1) s (0) t (0)" Move to cell M[4,3]
3rd col	Current cell M[4,3] = 2 Since, 2 is achieved from diagonal cell M[3,2], include 3rd col Subsequence: "lon (1) g (0) e (1) s (0) t (0)" Move to cell M[3,2]
2nd col	Current cell M[3,2] = 1 Since, 1 is achieved from diagonal cell M[2,1], include 2nd col Subsequence: "lo (1) n (1) g (0) e (1) s (0) t (0)" Move to cell M[2,1]
1st col	Current cell M[2,1] = 0 Since, 2 is achieved from left/upper cell, do not include 1st col Subsequence: "l (0) o (1) n (1) g (0) e (1) s (0) t (0)"

Longest common subsequence = "one"
Total Length = 3