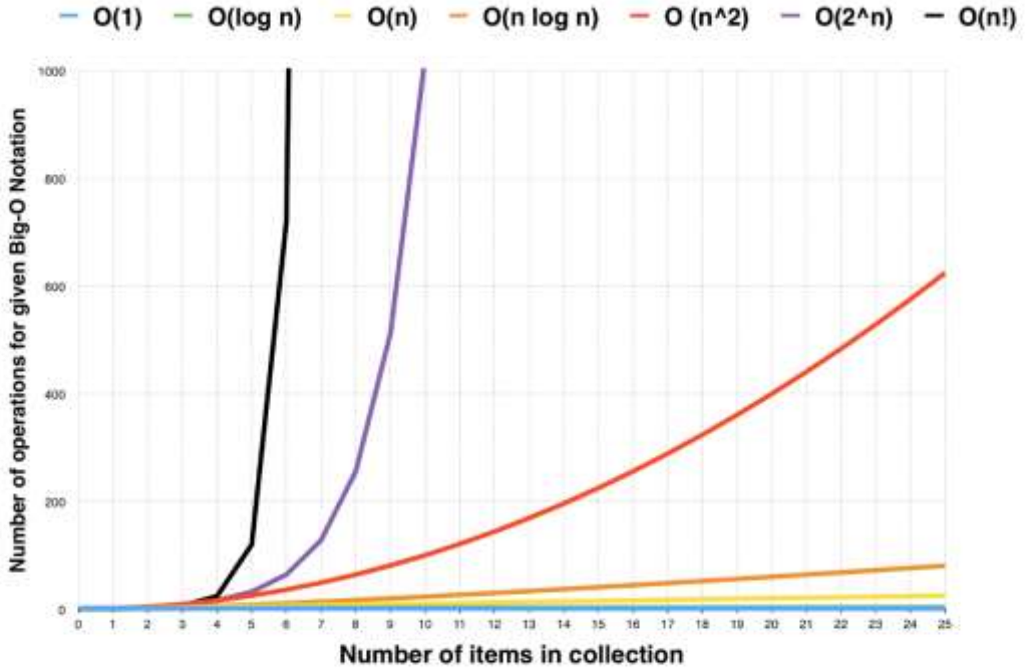
# Dynamic Programming

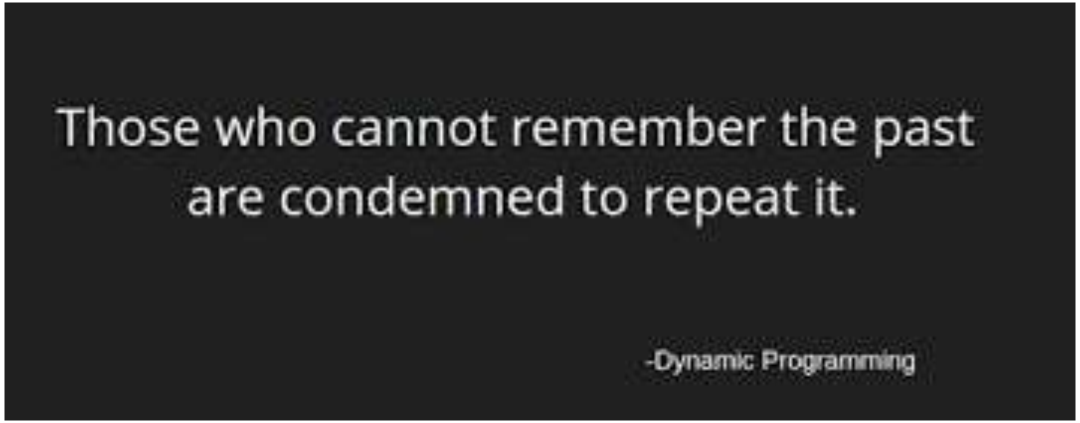
Volodymyr Synytskyi, software developer at ElifTech

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.



Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure



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

-Dynamic Programming

## Overlapping Subproblems

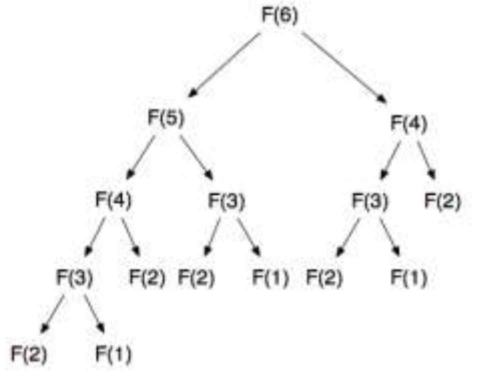
Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems.

Dynamic Programming is mainly used when solutions of same subproblems are needed again and again.

In dynamic programming, computed solutions to subproblems are stored in a table so that these don’t have to recomputed.

So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if t (Binary search)

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



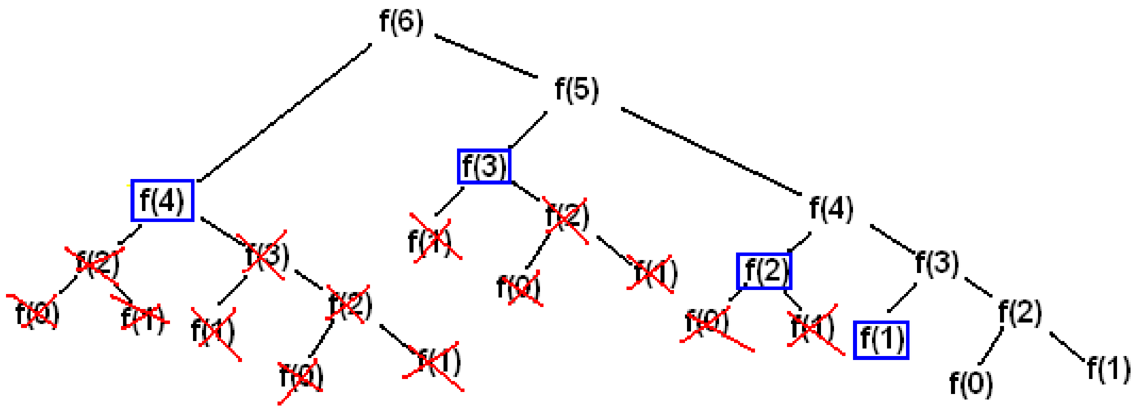
# Memoization

There are following two different ways to store the values so that these values can be reused

a) Memoization (Top Down)

b) Tabulation (Bottom Up)

```
/* function for nth Fibonacci number */
int fib(int n)
{
  if (lookup[n] == NIL)
  {
    if (n <= 1)
      lookup[n] = n;
    else
      lookup[n] = fib(n-1) + fib(n-2);
  }
```



## Tabulation

The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table.

```
int fib(int n)
{
  int f[n+1];
  int i;
  f[0] = 0;   f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}
```

```
def fib(x: Int): BigInt = {
        @tailrec def fibHelper(x: Int, prev: BigInt = 0, next: BigInt
= 1): BigInt = x match {
            case 0 => prev
            case 1 => next
            case _ => fibHelper(x - 1, next, (next + prev))
        }
        fibHelper(x)
    }
```

## 40th fibonachi - 102334155

Recursion - Time Taken 0.831763

Memoization - Time Taken 0.000014

Tabulation - Time Taken 0.000015

Both Tabulated and Memoized store the solutions of subproblems.

Following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

1) Overlapping Subproblems
2) Optimal Substructure

## Optimal Substructure Property

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

For example, the Shortest Path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v.

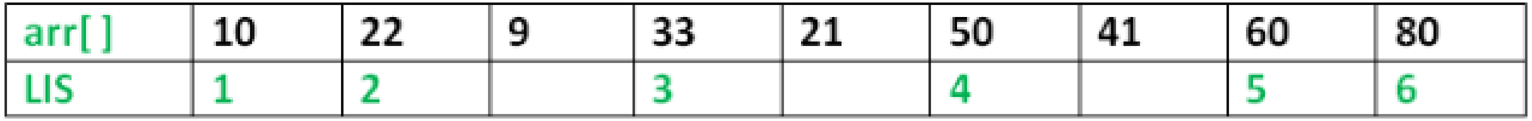
The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. (NP-complete)

First of all we need to find a state for which an optimal solution is found and with the

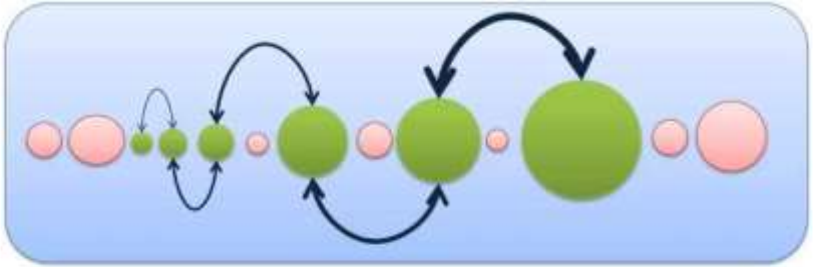
## Longest Increasing Subsequence

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 |

Input : arr[] = {50, 3, 10, 7, 40, 80}
Output : Length of LIS = 4
The longest increasing subsequence is {3, 7, 40, 80}



## Optimal Substructure

Let arr[0..n-1] be the input array and L(i) be the length of the LIS ending at index i such that arr[i] is the last element of the LIS.

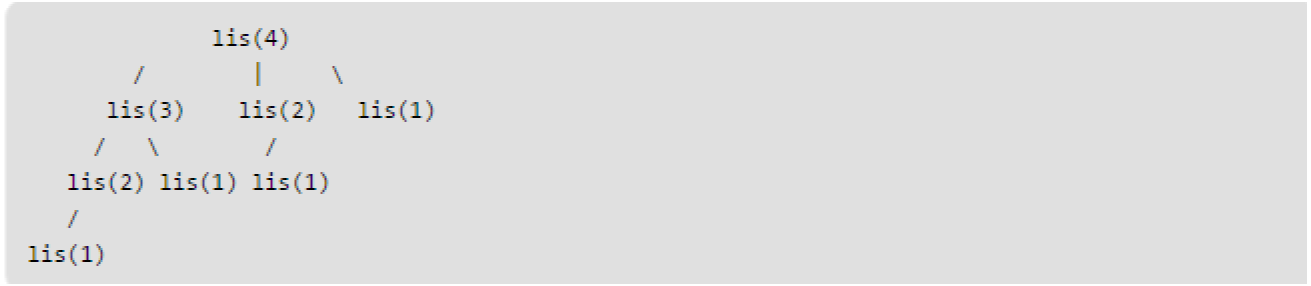
Then, L(i) can be recursively written as:
L(i) = 1 + max( L(j) ) where 0 < j < i and arr[j] < arr[i]; or
L(i) = 1, if no such j exists.

To find the LIS for a given array, we need to return max(L(i)) where 0 < i < n.

Thus, we see the LIS problem satisfies the optimal substructure property as the main problem can be solved using solutions to subproblems.

## Overlapping Subproblems

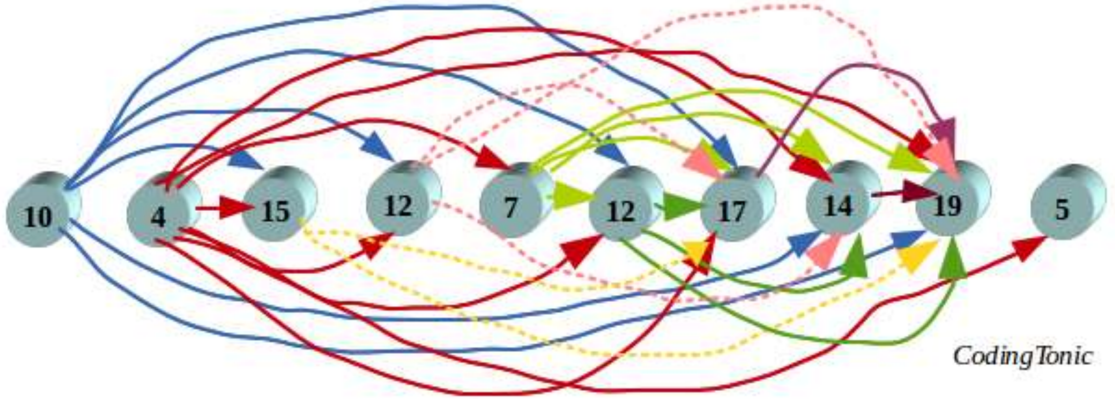
Considering the above implementation, following is recursion tree for an array of size 4. lis(n) gives us the length of LIS for arr[].



```
             lis(4)
        /       |     \
      lis(3)    lis(2)   lis(1)
     /   \        /
   lis(2) lis(1) lis(1)
   /
lis(1)
```

We can see that there are many subproblems which are solved again and again. So this problem has Overlapping Substructure property and recomputation of same subproblems can be avoided by either using Memoization or Tabulation. Following is a tabluated implementation for the LIS problem.

```
/* Initialize LIS values for all indexes */
    for (i = 0; i < n; i++ )
        lis[i] = 1;

    /* Compute optimized LIS values in bottom up manner */
    for (i = 1; i < n; i++ )
        for (j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;

    /* Pick maximum of all LIS values */
    for (i = 0; i < n; i++ )
        if (max < lis[i])
            max = lis[i];
```



## Longest Common Subsequence

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

It is a classic computer science problem, the basis of diff (a file comparison program that outputs the differences between two files), and has applications in bioinformatics.

**Examples:**

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

## Optimal Substructure

Let the input sequences be X[0..m-1] and Y[0..n-1] of lengths m and n respectively. And let L(X[0..m-1], Y[0..n-1]) be the length of LCS of the two sequences X and Y. Following is the recursive definition of L(X[0..m-1], Y[0..n-1]).

If last characters of both sequences match (or X[m-1] == Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = 1 + L(X[0..m-2], Y[0..n-2])

If last characters of both sequences do not match (or X[m-1] != Y[n-1]) then
L(X[0..m-1], Y[0..n-1]) = MAX ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2])

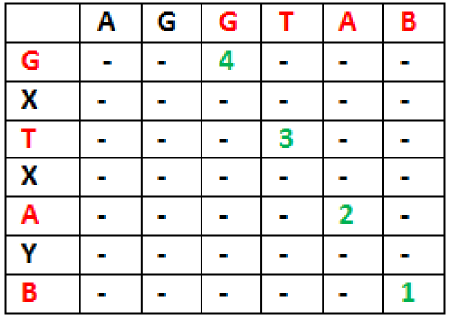
Examples:

1) Consider the input strings “AGGTAB” and “GXTXAYB”. Last characters match for the strings. So length of LCS can be written as:

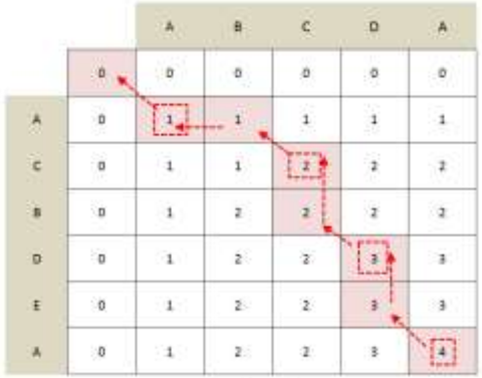
L(“AGGTAB”, “GXTXAYB”) = 1 + L(“AGGTA”, “GXTXAY”)

2) Consider the input strings “ABCDGH” and “AEDFHR. Last characters do not match for the strings. So length of LCS can be written as:

L(“ABCDGH”, “AEDFHR”) = MAX ( L(“ABCDG”, “AEDFHR”), L(“ABCDGH”, “AEDFH”) )



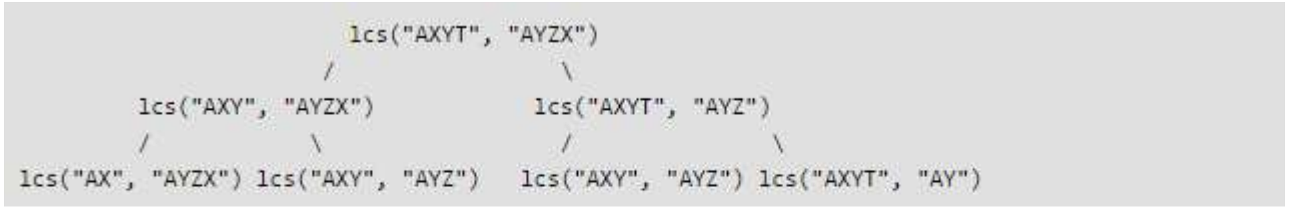
| | A | G | G | T | A | B |
|---|---|---|---|---|---|---|
| G | - | - | 4 | - | - | - |
| X | - | - | - | - | - | - |
| T | - | - | - | 3 | - | - |
| X | - | - | - | - | - | - |
| A | - | - | - | - | 2 | - |
| Y | - | - | - | - | - | - |
| B | - | - | - | - | - | 1 |



| | | A | B | C | D | A |
|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 |
| C | 0 | 1 | 1 | 2 | 2 | 2 |
| B | 0 | 1 | 2 | 2 | 2 | 2 |
| D | 0 | 1 | 2 | 2 | 3 | 3 |
| E | 0 | 1 | 2 | 2 | 3 | 3 |
| A | 0 | 1 | 2 | 2 | 3 | 4 |

LCS - "ACDA"

Time complexity of the above naive recursive approach is O(2^n) in worst case and worst case happens when all characters of X and Y mismatch i.e., length of LCS is 0.
Considering the above implementation, following is a partial recursion tree for input strings “AXYT” and “AYZX”

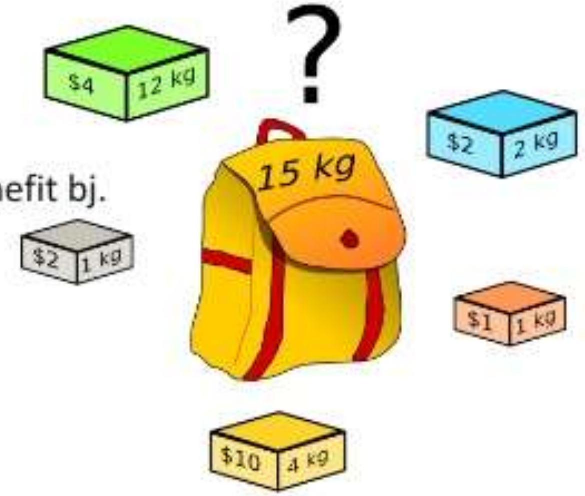


```
                     lcs("AXYT", "AYZX")
                   /                 \
         lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
         /              \                /               \
lcs("AX", "AYZX") lcs("AXY", "AYZ")   lcs("AXY", "AYZ") lcs("AXYT", "AY")
```

```
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
   int L[m+1][n+1];
   int i, j;

   /* Following steps build L[m+1][n+1] in bottom up fashion. Note
      that L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1] */
   for (i=0; i<=m; i++)
   {
     for (j=0; j<=n; j++)
     {
       if (i == 0 || j == 0)
         L[i][j] = 0;

       else if (X[i-1] == Y[j-1])
         L[i][j] = L[i-1][j-1] + 1;

       else
         L[i][j] = max(L[i-1][j], L[i][j-1]);
     }
   }

   /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
   return L[m][n];
}
```

## Knapsack Problem

- You have a knapsack that has capacity (weight) W.
- You have several items I1,…,In.
- Each item Ij has a weight wj and a benefit bj.
- You want to place a certain number of copies of each item Ij in the knapsack so that:
    - The knapsack weight capacity is not exceeded and
    - The total benefit is maximal.

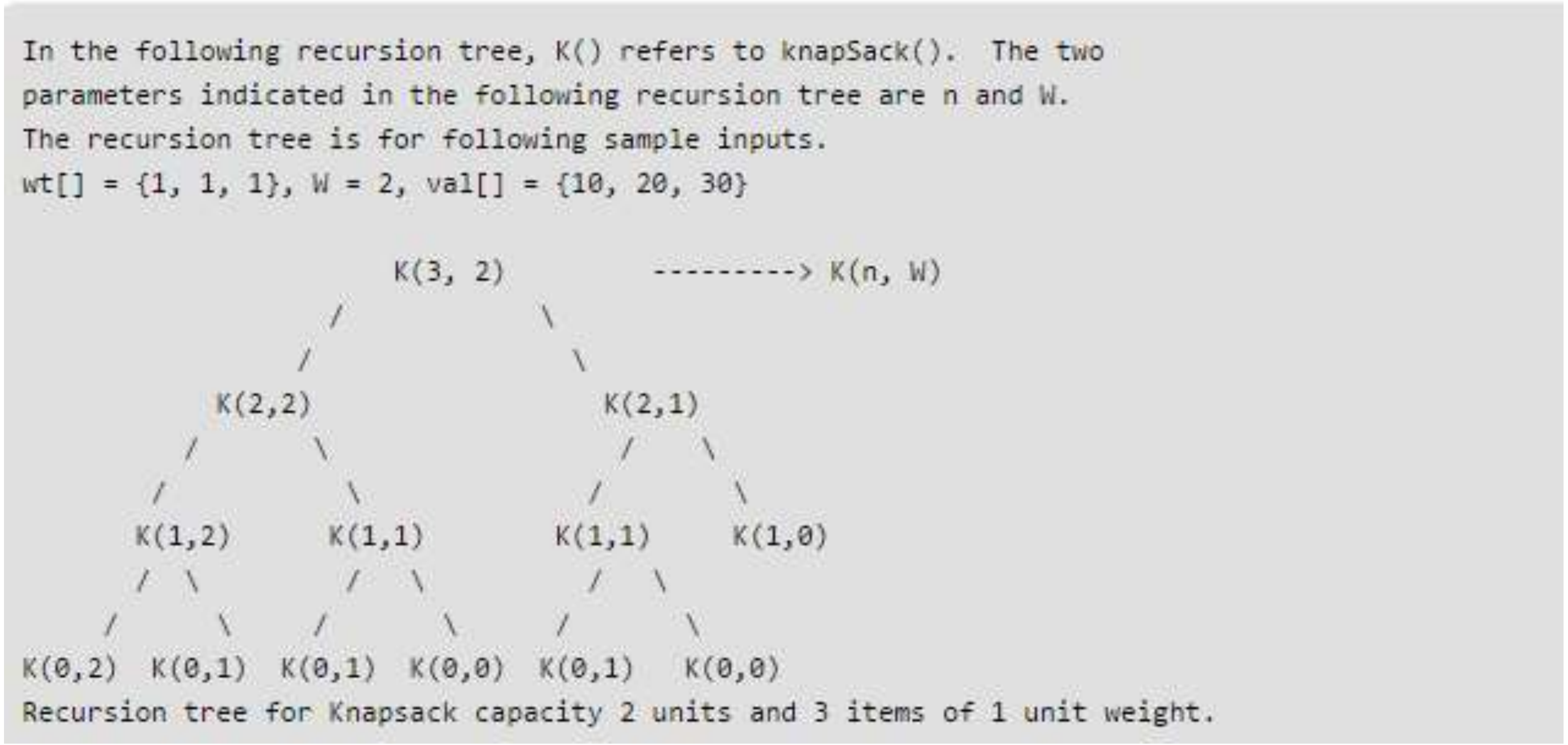


A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the maximum value subset.

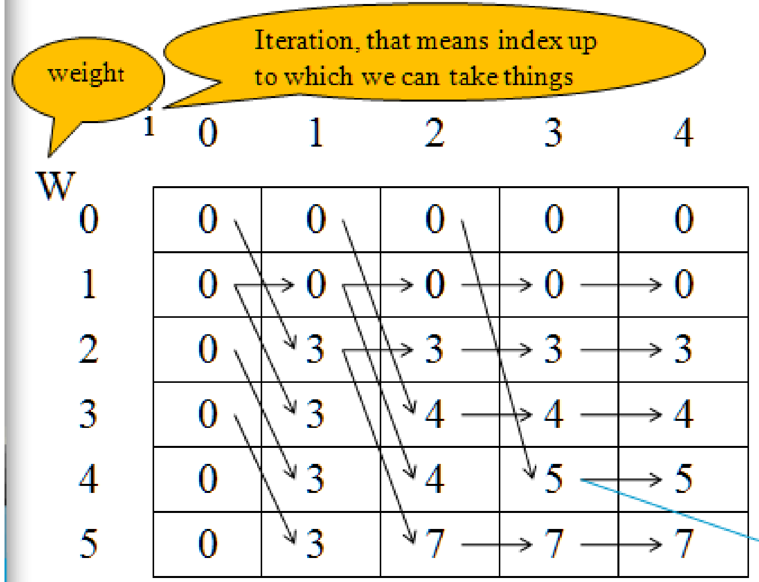
## Optimal Substructure

To consider all subsets of items, there can be two cases for every item: (1) the item is included in the optimal subset, (2) not included in the optimal set.
Therefore, the maximum value that can be obtained from n items is max of following two values.
1) Maximum value obtained by n-1 items and W weight (excluding nth item).
2) Value of nth item plus maximum value obtained by n-1 items and W minus weight of the nth item (including nth item).

If weight of nth item is greater than W, then the nth item cannot be included and case 1 is the only possibility.



```
In the following recursion tree, K() refers to knapSack().  The two
parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
wt[] = {1, 1, 1}, W = 2, val[] = {10, 20, 30}

                       K(3, 2)         ---------> K(n, W)
                   /            \
                 /                \
            K(2,2)                  K(2,1)
          /       \                 /    \
        /           \             /        \
       K(1,2)      K(1,1)        K(1,1)     K(1,0)
       /  \         /   \          /   \
     /      \     /       \      /       \
K(0,2)  K(0,1)  K(0,1)  K(0,0)  K(0,1)   K(0,0)
Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.
```



| W \ i | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 3 | 3 | 3 | 3 |
| 3 | 0 | 3 | 4 | 4 | 4 |
| 4 | 0 | 3 | 4 | 5 | 5 |
| 5 | 0 | 3 | 7 | 7 | 7 |

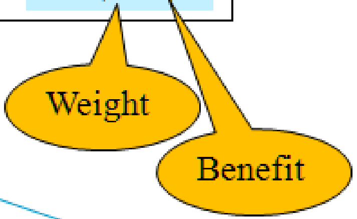
Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)



```
if $w_i$ <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i, w] = b_i + B[i-1,w- w_i]$
    else
        $B[i, w] = B[i-1,w]$
else $B[i, w] = B[i-1,w]$  // $w_i > w$
```

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w- w_i=1$

```
// Returns the maximum value that can be put in a knapsack of capacity W
int knapSack(int W, int wt[], int val[], int n)
{
   int i, w;
   int K[n+1][W+1];

   // Build table K[][] in bottom up manner
   for (i = 0; i <= n; i++)
   {
       for (w = 0; w <= W; w++)
       {
           if (i==0 || w==0)
               K[i][w] = 0;
           else if (wt[i-1] <= w)
                 K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w]);
           else
                 K[i][w] = K[i-1][w];
       }
   }

   return K[n][W];
}
```

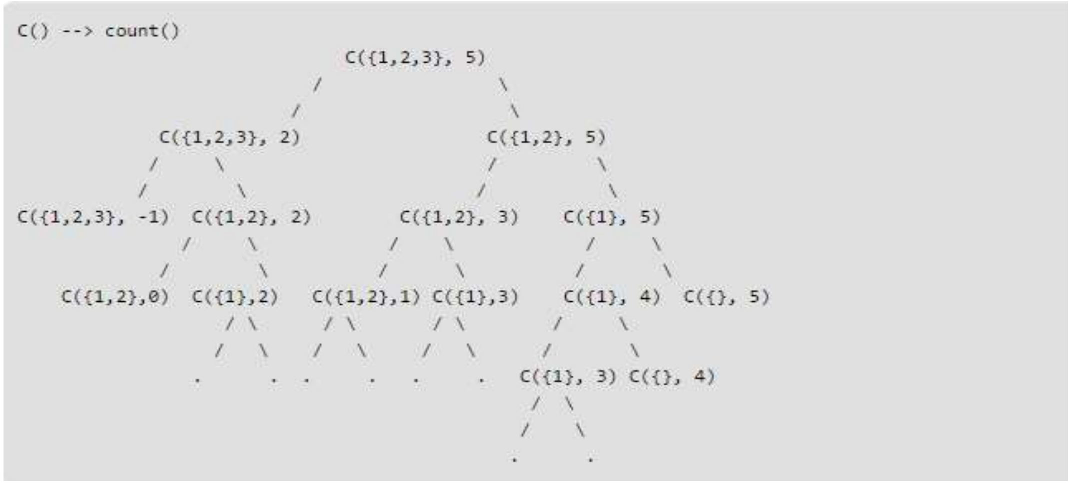
## Coin Change

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.

For 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 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} and {5,5}. So the output should be 5.

## Optimal Substructure

To count total number solutions, we can divide all set solutions in two sets.
1) Solutions that do not contain mth coin (or Sm).
2) Solutions that contain at least one Sm.
Let count(S[], m, n) be the function to count the number of solutions, then it can be written as sum of count(S[], m-1, n) and count(S[], m, n-Sm).



```
C() --> count()
                              C({1,2,3}, 5)
                            /                \
                          /                   \
             C({1,2,3}, 2)                 C({1,2}, 5)
            /      \                        /         \
           /         \                     /           \
C({1,2,3}, -1)  C({1,2}, 2)        C({1,2}, 3)    C({1}, 5)
               /     \             /    \             /     \
             /         \          /       \          /        \
    C({1,2},0)  C({1},2)   C({1,2},1) C({1},3)    C({1}, 4)  C({}, 5)
                   / \      / \        / \        /     \
                  /   \    /   \     /   \       /       \
                .      .  .     .   .     .   C({1}, 3) C({}, 4)
                                                / \
                                               /    \
                                              .      .
```

// Returns the count of ways we can sum S[0...m-1] coins to get sum n

```
int count( int S[], int m, int n )
{
   // If n is 0 then there is 1 solution (do not include any coin)
   if (n == 0)
      return 1;

   // If n is less than 0 then no solution exists
   if (n < 0)
      return 0;

   // If there are no coins and n is greater than 0, then no solution exist
   if (m <=0 && n >= 1)
      return 0;

   // count is sum of solutions (i) including S[m-1] (ii) excluding S[m-1]
   return count( S, m - 1, n ) + count( S, m, n-S[m-1] );
}
```

```
int count( int S[], int m, int n )
{
    int i, j, x, y;

    // We need n+1 rows as the table is constructed in bottom up manner using
    // the base case 0 value case (n = 0)
    int table[n+1][m];

    // Fill the entries for 0 value case (n = 0)
    for (i=0; i<m; i++)
        table[0][i] = 1;

    // Fill rest of the table entries in bottom up manner
    for (i = 1; i < n+1; i++)
    {
        for (j = 0; j < m; j++)
        {
            // Count of solutions including S[j]
            x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;

            // Count of solutions excluding S[j]
            y = (j >= 1)? table[i][j-1]: 0;

            // total count
            table[i][j] = x + y;
        }
    }
    return table[n][m-1];
}
```

## Edit Distance

Given two strings str1 and str2 and below operations that can performed on str1. Find minimum number of edits (operations) required to convert ‘str1’ into ‘str2’.

Insert

Remove

Replace

All of the above operations are of equal cost.

```
Input:   str1 = "geek", str2 = "gesek"
Output:  1
We can convert str1 into str2 by inserting a 's'.

Input:   str1 = "cat", str2 = "cut"
Output:  1
We can convert str1 into str2 by replacing 'a' with 'u'.

Input:   str1 = "sunday", str2 = "saturday"
Output:  3
Last three and first characters are same.  We basically
need to convert "un" to "atur".  This can be done using
below three operations.
Replace 'n' with 'r', insert t, insert a
```

## Subproblems

The idea is process all characters one by one staring from either from left or right sides of both strings.

Let we traverse from right corner, there are two possibilities for every pair of character being traversed.

m: Length of str1 (first string)

n: Length of str2 (second string)

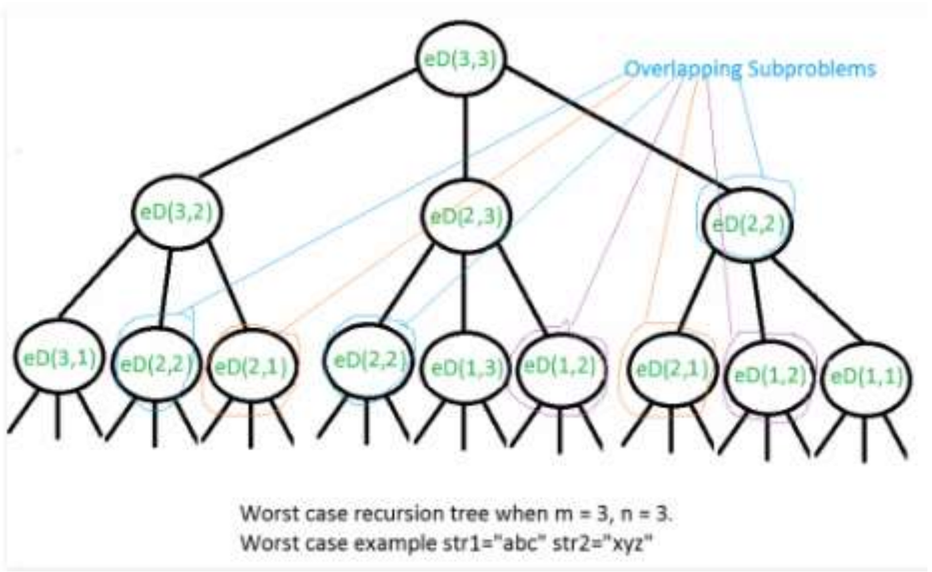
1. If last characters of two strings are same, nothing much to do. Ignore last characters and get count for remaining strings. So we recur for lengths m-1 and n-1.
2. Else (If last characters are not same), we consider all operations on ‘str1’, consider all three operations on last character of first string, recursively compute minimum cost for all three operations and take minimum of three values.

Insert: Recur for m and n-1

Remove: Recur for m-1 and n

```
int editDist(string str1 , string str2 , int m ,int n)
{
    // If first string is empty, the only option is to
    // insert all characters of second string into first
    if (m == 0) return n;

    // If second string is empty, the only option is to
    // remove all characters of first string
    if (n == 0) return m;

    // If last characters of two strings are same, nothing
    // much to do. Ignore last characters and get count for
    // remaining strings.
    if (str1[m-1] == str2[n-1])
        return editDist(str1, str2, m-1, n-1);

    // If last characters are not same, consider all three
    // operations on last character of first string, recursively
    // compute minimum cost for all three operations and take
    // minimum of three values.
    return 1 + min ( editDist(str1,  str2, m, n-1),    // Insert
                     editDist(str1,  str2, m-1, n),   // Remove
                     editDist(str1,  str2, m-1, n-1) // Replace
                   );
}
```

The time complexity of above solution is exponential. In worst case, we may end up doing O(3m) operations. The worst case happens when none of characters of two strings match. Below is a recursive call diagram for worst case.



## Recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subpriblems

```
int editDistDP(string str1, string str2, int m, int n)
{
    // Create a table to store results of subproblems
    int dp[m+1][n+1];

    // Fill d[][] in bottom up manner
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            // If first string is empty, only option is to
            // isnert all characters of second string
            if (i==0)
                dp[i][j] = j;  // Min. operations = j

            // If second string is empty, only option is to
            // remove all characters of second string
            else if (j==0)
                dp[i][j] = i; // Min. operations = i

            // If last characters are same, ignore last char
            // and recur for remaining string
            else if (str1[i-1] == str2[j-1])
                dp[i][j] = dp[i-1][j-1];

            // If last character are different, consider all
            // possibilities and find minimum
            else
                dp[i][j] = 1 + min(dp[i][j-1],  // Insert
                                   dp[i-1][j],  // Remove
                                   dp[i-1][j-1]); // Replace
        }
    }

    return dp[m][n];
}
```

# Floyd–Warshall

Algorithm for finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles)

```
/* Add all vertices one by one to the set of intermediate vertices. */

for (k = 0; k < V; k++)
  {
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
      // Pick all vertices as destination for the
      // above picked source
      for (j = 0; j < V; j++)
      {
        // If vertex k is on the shortest path from
        // i to j, then update the value of dist[i][j]
        if (dist[i][k] + dist[k][j] < dist[i][j])
          dist[i][j] = dist[i][k] + dist[k][j];
      }
    }
  }
```

## Tabulation vs Memoization

- 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
    - No overhead for recursion and less overhead for maintaining table
    - There are some problems for which the regular pattern of table accesses in the dynamic-programming algorithm can be exploited to reduce the time or space requirements even further
- 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

## Links

https://www.topcoder.com/community/data-science/data-science-tutorials/dynamic-programming-from-novice-to-advanced/

http://www.geeksforgeeks.org/fundamentals-of-algorithms/#DynamicProgramming

https://www.hackerrank.com/domains/algorithms/dynamic-programming/page:1

http://codeforces.com/problemset/tags/dp?order=BY_SOLVED_DESC

https://www.topcoder.com/tc?module=ProblemArchive&sr=&er=&sc=&sd=&class=&cat=Dynamic+Programming&div1l=&div2l=&mind1s=&mind2s=&maxd1s=&maxd2s=&wr=

## Review: Dynamic Programming

- Summary of the basic idea:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- Variations:
  - "Table" could be 3-dimensional, triangular, a tree, etc.

## Thank you for attention!

Find us at eliftech.com
Have a question? Contact us:
info@eliftech.com