0-1 Knapsack Problem

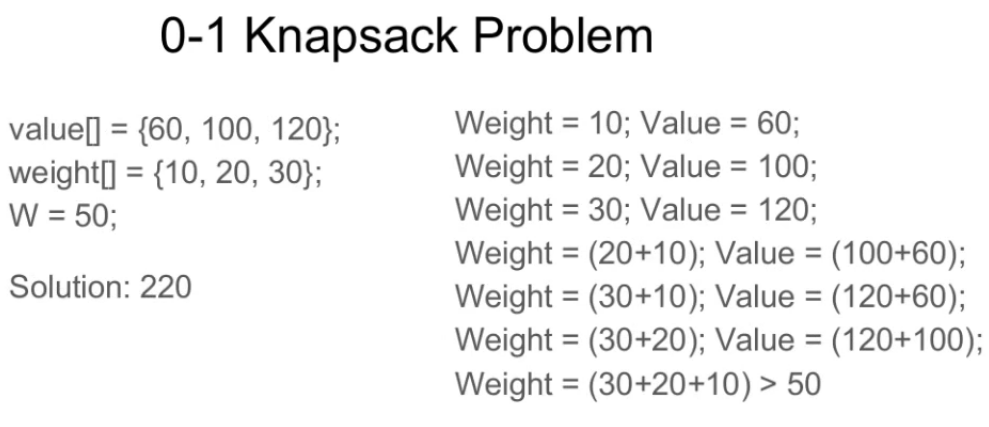
<https://www.youtube.com/watch?v=T4bY72lCQac&index=10&list=PLqM7alHXFySGbXhWx7sBJEwY2DnhDjmxm>

## Problem:

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack

## Recursive approach:

Or find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).



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.

**1) 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 c:

/\* A Naive recursive implementation of 0-1 Knapsack problem \*/

#include<stdio.h>

// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }

// Returns the maximum value that can be put in a knapsack of capacity W

int knapSack(int W, int wt[], int val[], int n)

{

   // Base Case

   if (n == 0 || W == 0)

       return 0;

   // If weight of the nth item is more than Knapsack capacity W, then

   // this item cannot be included in the optimal solution

   if (wt[n-1] > W)

       return knapSack(W, wt, val, n-1);

   // Return the maximum of two cases:

   // (1) nth item included

   // (2) not included

   else return max( val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),

                    knapSack(W, wt, val, n-1)

                  );

}

// Driver program to test above function

int main()

{

    int val[] = {60, 100, 120};

    int wt[] = {10, 20, 30};

    int  W = 50;

    int n = sizeof(val)/sizeof(val[0]);

    printf("%d", knapSack(W, wt, val, n));

    return 0;

}

In python:

#A naive recursive implementation of 0-1 Knapsack Problem

# Returns the maximum value that can be put in a knapsack of

# capacity W

def knapSack(W , wt , val , n):

    # Base Case

    if n == 0 or W == 0 :

        return 0

    # If weight of the nth item is more than Knapsack of capacity

    # W, then this item cannot be included in the optimal solution

    if (wt[n-1] > W):

        return knapSack(W , wt , val , n-1)

    # return the maximum of two cases:

    # (1) nth item included

    # (2) not included

    else:

        return max(val[n-1] + knapSack(W-wt[n-1] , wt , val , n-1),

                   knapSack(W , wt , val , n-1))

# end of function knapSack

# To test above function

val = [60, 100, 120]

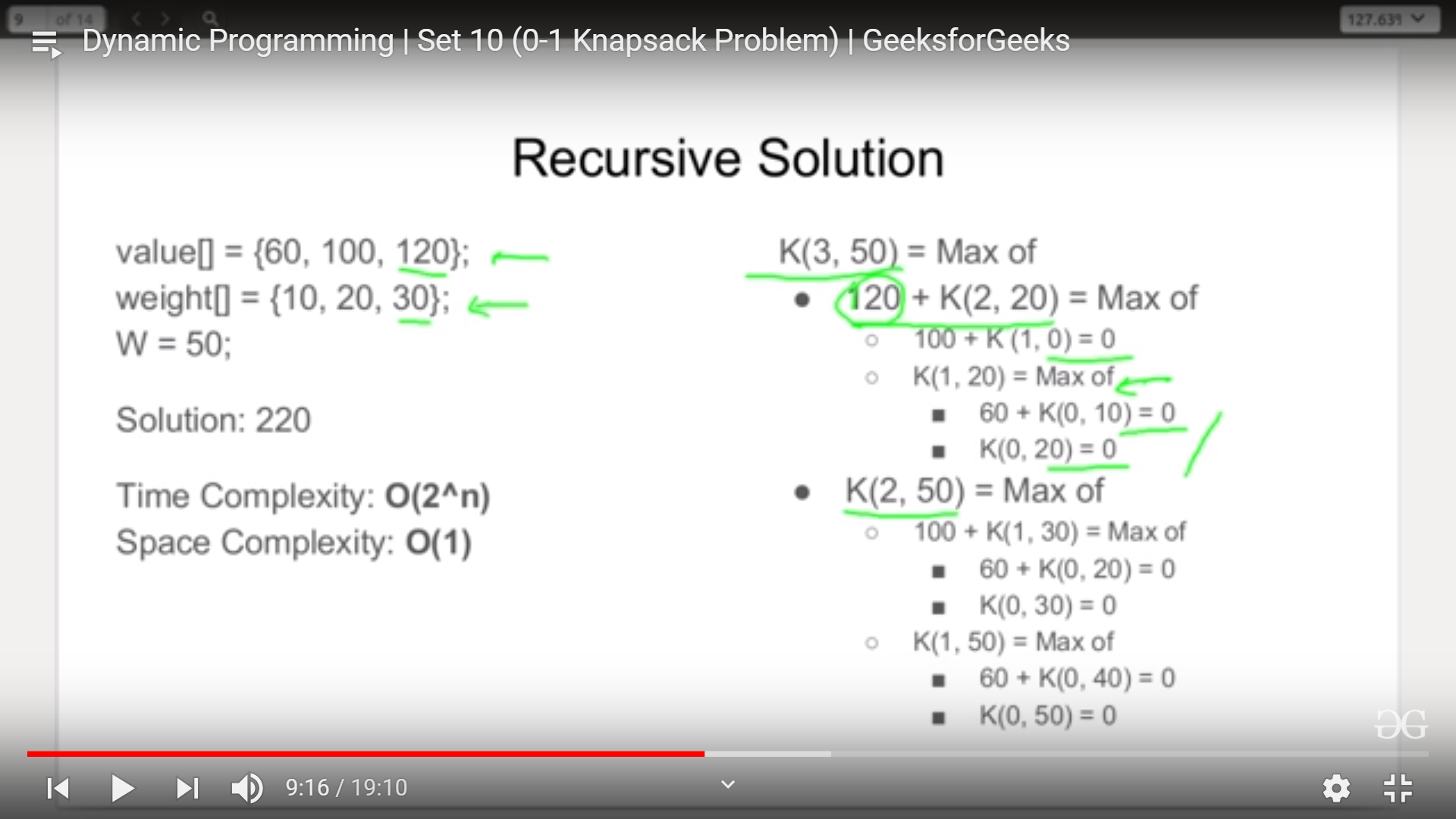
wt = [10, 20, 30]

W = 50

n = len(val)

print knapSack(W , wt , val , n)

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice. Time complexity of this naive recursive solution is exponential (2^n).



In last pic as you can see

K(3,50) : Means 3 items and 50 is total weight if you add 3rd item in your bucket it will be k(2,20)

Otherwise it will be k(2,50)

Same with k(2,20) also if you add 2nd item in your bucket it will be k(1,0) because 20 is weight of 2nd item and value added will be 100 and if you don’t add 2nd item it will be k(1,20)

Same with k(2,50) also if you add 2nd item it will be k(1,30) and again either 60 + k(0,20) or k(0,30).

And every time we will pick max from child calculation : like for k(2,20) : MAX ( (100+k(1,0) = 100) and (k(1,20) = MAX( 60+k(0,10) = 60 ) and (k(0,20) = 0 ) = 60 )) = 100

So k(2,20) = 100

**Same k(3,50 ) = MAX( k(2,50) and 120+k(2,20)) = ( MAX ( 160, 220) ) = 220**

## Dynamic programming approach:

Since suproblems are evaluated again, this problem has Overlapping Subprolems property. So the 0-1 Knapsack problem has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem.

Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), **recomputations of same subproblems can be avoided by constructing a temporary array K[][] in bottom up manner. Following is Dynamic Programming based implementation.**

// A Dynamic Programming based solution for 0-1 Knapsack problem

#include<stdio.h>

// A utility function that returns maximum of two integers

int max(int a, int b) { return (a > b)? a : b; }

// 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];

}

int main()

{

    int val[] = {60, 100, 120};

    int wt[] = {10, 20, 30};

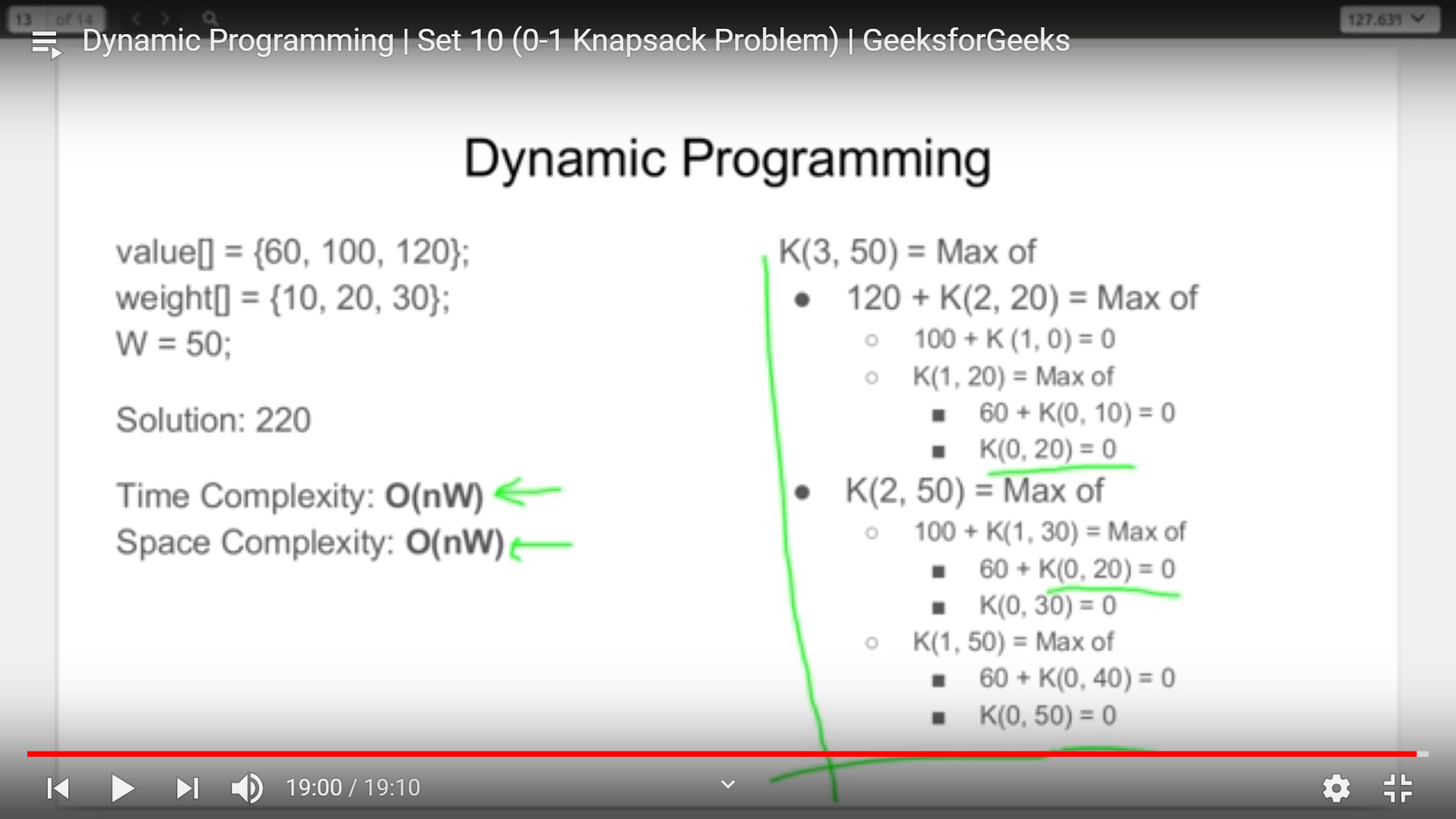
    int  W = 50;

    int n = sizeof(val)/sizeof(val[0]);

    printf("%d", knapSack(W, wt, val, n));

    return 0;

}



# Egg Dropping Puzzle:

<https://www.youtube.com/watch?v=amdKmQlATmQ&list=PLqM7alHXFySGbXhWx7sBJEwY2DnhDjmxm&index=11>

<https://www.geeksforgeeks.org/egg-dropping-puzzle-dp-11/>

## Problem:

The following is a description of the instance of this famous puzzle involving n=2 eggs and a building with k=36 floors.

Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

…..An egg that survives a fall can be used again.  
…..A broken egg must be discarded.  
…..The effect of a fall is the same for all eggs.  
…..If an egg breaks when dropped, then it would break if dropped from a higher floor.  
…..If an egg survives a fall then it would survive a shorter fall.  
…..It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor do not cause an egg to break.

## Recursive approach:

**1) Optimal Substructure:**  
When we drop an egg from a floor x, there can be two cases (1) The egg breaks (2) The egg doesn’t break.

**1) If the egg breaks after dropping from xth floor, then we only need to check for floors lower than x with remaining eggs; so the problem reduces to x-1 floors and n-1 eggs  
2) If the egg doesn’t break after dropping from the xth floor, then we only need to check for floors higher than x; so the problem reduces to k-x floors and n eggs.**

Since we need to minimize the number of trials in worst case, we take the maximum of two cases. We consider the max of above two cases for every floor and choose the floor which yields minimum number of trials.

k ==> Number of floors

n ==> Number of Eggs

eggDrop(n, k) ==> Minimum number of trials needed to find the critical

floor in worst case.

eggDrop(n, k) = 1 + min{max(eggDrop(n - 1, x - 1), eggDrop(n, k - x)):

x in {1, 2, ..., k}}

# include <stdio.h>

# include <limits.h>

// A utility function to get maximum of two integers

int max(int a, int b) { return (a > b)? a: b; }

/\* Function to get minimum number of trials needed in worst

  case with n eggs and k floors \*/

int eggDrop(int n, int k)

{

    // If there are no floors, then no trials needed. OR if there is

    // one floor, one trial needed.

    if (k == 1 || k == 0)

        return k;

    // We need k trials for one egg and k floors

    if (n == 1)

        return k;

    int min = INT\_MAX, x, res;

    // Consider all droppings from 1st floor to kth floor and

    // return the minimum of these values plus 1.

    for (x = 1; x <= k; x++)

    {

        res = max(eggDrop(n-1, x-1), eggDrop(n, k-x));

        if (res < min)

            min = res;

    }

    return min + 1;

}

/\* Driver program to test to pront printDups\*/

int main()

{

    int n = 2, k = 10;

    printf ("nMinimum number of trials in worst case with %d eggs and "

             "%d floors is %d \n", n, k, eggDrop(n, k));

    return 0;

}

Python:

import sys

# Function to get minimum number of trials

# needed in worst case with n eggs and k floors

def eggDrop(n, k):

    # If there are no floors, then no trials

    # needed. OR if there is one floor, one

    # trial needed.

    if (k == 1 or k == 0):

        return k

    # We need k trials for one egg

    # and k floors

    if (n == 1):

        return k

    min = sys.maxsize

    # Consider all droppings from 1st

    # floor to kth floor and return

    # the minimum of these values plus 1.

    for x in range(1, k + 1):

        res = max(eggDrop(n - 1, x - 1),

                  eggDrop(n, k - x))

        if (res < min):

            min = res

    return min + 1

# Driver Code

if \_\_name\_\_ == "\_\_main\_\_":

    n = 2

    k = 10

    print("Minimum number of trials in worst case with",

           n, "eggs and", k, "floors is", eggDrop(n, k))

# This code is contributed by ita\_c

For eggDrop(2,4) :

X = 1: res = max(eggDrop(1,0) , eddDrop(2,3)) = max ( 0 + MAX( ED(1,2) , ED(2,

## Dynamic programming approach:

Since same suproblems are called again, this problem has Overlapping Subprolems property. So Egg Dropping Puzzle has both properties (see [this](https://www.geeksforgeeks.org/overlapping-subproblems-property-in-dynamic-programming-dp-1/)and [this](https://www.geeksforgeeks.org/optimal-substructure-property-in-dynamic-programming-dp-2/)) of a dynamic programming problem. Like other typical [Dynamic Programming(DP) problems](https://www.geeksforgeeks.org/archives/tag/dynamic-programming), recomputations of same subproblems can be avoided by constructing a temporary array eggFloor[][] in bottom up manner.

# Longest Palindromic Substring

<https://www.geeksforgeeks.org/longest-palindrome-substring-set-1/>

## Problem:

Given a string, find the longest substring which is palindrome. For example, if the given string is “forgeeksskeegfor”, the output should be “geeksskeeg”.

**Method 1 (Brute Force)**

The simple approach is to check each substring whether the substring is a palindrome or not. We can run three loops, the outer two loops pick all substrings one by one by fixing the corner characters, the inner loop checks whether the picked substring is palindrome or not.

Time complexity: O ( n^3 )  
Auxiliary complexity: O ( 1 )

**Method 2 (Dynamic Programming )**

The time complexity can be reduced by storing results of subproblems. The idea is similar to [this](https://www.geeksforgeeks.org/archives/19155)post. We maintain a boolean table[n][n] that is filled in bottom up manner. The value of table[i][j] is true, if the substring is palindrome, otherwise false. To calculate table[i][j], we first check the value of table[i+1][j-1], if the value is true and str[i] is same as str[j], then we make table[i][j] true. Otherwise, the value of table[i][j] is made false.

// A dynamic programming solution for longest palindr.

// This code is adopted from following link

// <http://www.leetcode.com/2011/11/longest-palindromic-substring-part-i.html>

#include <stdio.h>

#include <string.h>

// A utility function to print a substring str[low..high]

void printSubStr( char\* str, int low, int high )

{

    for( int i = low; i <= high; ++i )

        printf("%c", str[i]);

}

// This function prints the longest palindrome substring

// of str[].

// It also returns the length of the longest palindrome

int longestPalSubstr( char \*str )

{

    int n = strlen( str ); // get length of input string

    // table[i][j] will be false if substring str[i..j]

    // is not palindrome.

    // Else table[i][j] will be true

    bool table[n][n];

    memset(table, 0, sizeof(table));

    // All substrings of length 1 are palindromes

    int maxLength = 1;

    for (int i = 0; i < n; ++i)

        table[i][i] = true;

    // check for sub-string of length 2.

    int start = 0;

    for (int i = 0; i < n-1; ++i)

    {

        if (str[i] == str[i+1])

        {

            table[i][i+1] = true;

            start = i;

            maxLength = 2;

        }

    }

    // Check for lengths greater than 2. k is length

    // of substring

    for (int k = 3; k <= n; ++k)

    {

        // Fix the starting index

        for (int i = 0; i < n-k+1 ; ++i)

        {

            // Get the ending index of substring from

            // starting index i and length k

            int j = i + k - 1;

            // checking for sub-string from ith index to

            // jth index iff str[i+1] to str[j-1] is a

            // palindrome

            if (table[i+1][j-1] && str[i] == str[j])

            {

                table[i][j] = true;

                if (k > maxLength)

                {

                    start = i;

                    maxLength = k;

                }

            }

        }

    }

    printf("Longest palindrome substring is: ");

    printSubStr( str, start, start + maxLength - 1 );

    return maxLength; // return length of LPS

}

// Driver program to test above functions

int main()

{

    char str[] = "forgeeksskeegfor";

    printf("\nLength is: %d\n", longestPalSubstr( str ) );

    return 0;

}