

# Contents

<b>1</b>	<b>Set 1 (Overlapping Subproblems Property)</b>	<b>9</b>
	Source . . . . .	12
<b>2</b>	<b>Set 2 (Optimal Substructure Property)</b>	<b>13</b>
	Source . . . . .	14
<b>3</b>	<b>Set 3 (Longest Increasing Subsequence)</b>	<b>15</b>
	Source . . . . .	18
<b>4</b>	<b>Set 4 (Longest Common Subsequence)</b>	<b>20</b>
	Source . . . . .	24
<b>5</b>	<b>Set 5 (Edit Distance)</b>	<b>25</b>
	Source . . . . .	32
<b>6</b>	<b>Set 6 (Min Cost Path)</b>	<b>33</b>
	Source . . . . .	38
<b>7</b>	<b>Set 7 (Coin Change)</b>	<b>39</b>
	Source . . . . .	44
<b>8</b>	<b>Set 8 (Matrix Chain Multiplication)</b>	<b>45</b>
	Source . . . . .	51
<b>9</b>	<b>Set 9 (Binomial Coefficient)</b>	<b>52</b>
	Source . . . . .	56

<b>10 Set 10 ( 0-1 Knapsack Problem)</b>	<b>57</b>
Source . . . . .	61
<b>11 Set 11 (Egg Dropping Puzzle)</b>	<b>62</b>
Source . . . . .	68
<b>12 Set 12 (Longest Palindromic Subsequence)</b>	<b>69</b>
Source . . . . .	74
<b>13 Set 13 (Cutting a Rod)</b>	<b>75</b>
Source . . . . .	79
<b>14 Set 14 (Maximum Sum Increasing Subsequence)</b>	<b>80</b>
Source . . . . .	82
<b>15 Set 15 (Longest Bitonic Subsequence)</b>	<b>83</b>
Source . . . . .	85
<b>16 Set 16 (Floyd Warshall Algorithm)</b>	<b>86</b>
Source . . . . .	92
<b>17 Set 17 (Palindrome Partitioning)</b>	<b>93</b>
Source . . . . .	98
<b>18 Set 18 (Partition problem)</b>	<b>99</b>
Source . . . . .	104
<b>19 Set 19 (Word Wrap Problem)</b>	<b>105</b>
Source . . . . .	110
<b>20 Set 20 (Maximum Length Chain of Pairs)</b>	<b>111</b>
Source . . . . .	113
<b>21 Set 21 (Variations of LIS)</b>	<b>114</b>
Source . . . . .	115

<b>22 Set 22 (Box Stacking Problem)</b>	<b>116</b>
Source . . . . .	119
<b>23 Set 23 (Bellman–Ford Algorithm)</b>	<b>120</b>
Source . . . . .	130
<b>24 Set 24 (Optimal Binary Search Tree)</b>	<b>131</b>
Source . . . . .	136
<b>25 Set 25 (Subset Sum Problem)</b>	<b>138</b>
Source . . . . .	141
<b>26 Set 26 (Largest Independent Set Problem)</b>	<b>142</b>
Source . . . . .	147
<b>27 Set 27 (Maximum sum rectangle in a 2D matrix)</b>	<b>148</b>
Source . . . . .	151
<b>28 Set 28 (Minimum insertions to form a palindrome)</b>	<b>153</b>
Source . . . . .	158
<b>29 Set 29 (Longest Common Substring)</b>	<b>160</b>
Source . . . . .	162
<b>30 Set 30 (Dice Throw)</b>	<b>163</b>
Source . . . . .	166
<b>31 Set 31 (Optimal Strategy for a Game)</b>	<b>167</b>
Source . . . . .	170
<b>32 Set 32 (Word Break Problem)</b>	<b>171</b>
Source . . . . .	176
<b>33 Set 33 (Find if a string is interleaved of two other strings)</b>	<b>177</b>
Source . . . . .	180

<b>34 Set 34 (Assembly Line Scheduling)</b>	<b>181</b>
Source . . . . .	186
<b>35 Set 35 (Longest Arithmetic Progression)</b>	<b>187</b>
Source . . . . .	191
<b>36 Set 36 (Maximum Product Cutting)</b>	<b>192</b>
Source . . . . .	196
<b>37 Set 37 (Boolean Parenthesization Problem)</b>	<b>197</b>
Source . . . . .	201
<b>38 Bitmasking and Set 1 (Count ways to assign unique cap to every person)</b>	<b>202</b>
Source . . . . .	206
<b>39 Collect maximum points in a grid using two traversals</b>	<b>207</b>
Source . . . . .	211
<b>40 Compute sum of digits in all numbers from 1 to n</b>	<b>212</b>
Source . . . . .	216
<b>41 Count Possible Decodings of a given Digit Sequence</b>	<b>217</b>
Source . . . . .	220
<b>42 Count all possible paths from top left to bottom right of a mXn matrix</b>	<b>221</b>
Source . . . . .	223
<b>43 Count all possible walks from a source to a destination with exactly k edges</b>	<b>224</b>
Source . . . . .	228
<b>44 Count even length binary sequences with same sum of first and second half bits</b>	<b>229</b>
Source . . . . .	235

<b>45 Count number of binary strings without consecutive 1's</b>	<b>236</b>
Source . . . . .	238
<b>46 Count number of ways to cover a distance</b>	<b>239</b>
Source . . . . .	241
<b>47 Count number of ways to reach a given score in a game</b>	<b>242</b>
Source . . . . .	244
<b>48 Count of n digit numbers whose sum of digits equals to given sum</b>	<b>245</b>
Source . . . . .	249
<b>49 Count possible ways to construct buildings</b>	<b>250</b>
Source . . . . .	253
<b>50 Count total number of N digit numbers such that the difference between sum of even and odd digits is 1</b>	<b>254</b>
Source . . . . .	257
<b>51 Count ways to reach the n'th stair</b>	<b>258</b>
Source . . . . .	262
<b>52 Find length of the longest consecutive path from a given starting character</b>	<b>263</b>
Source . . . . .	267
<b>53 Find minimum number of coins that make a given value</b>	<b>268</b>
Source . . . . .	271
<b>54 Find minimum possible size of array with given rules for removing elements</b>	<b>273</b>
Source . . . . .	276
<b>55 Find number of solutions of a linear equation of n variables</b>	<b>277</b>
Source . . . . .	280

<b>56 Find the longest path in a matrix with given constraints</b>	<b>281</b>
Source . . . . .	283
<b>57 Find the minimum cost to reach destination using a train</b>	<b>285</b>
Source . . . . .	289
<b>58 How to print maximum number of A's using given four keys</b>	<b>290</b>
Source . . . . .	295
<b>59 Largest Sum Contiguous Subarray</b>	<b>296</b>
Source . . . . .	301
<b>60 Length of the longest substring without repeating characters</b>	<b>302</b>
Source . . . . .	305
<b>61 Longest Even Length Substring such that Sum of First and     Second Half is same</b>	<b>306</b>
Source . . . . .	312
<b>62 Longest Palindromic Substring   Set 1</b>	<b>313</b>
Source . . . . .	316
<b>63 Longest Repeating Subsequence</b>	<b>317</b>
Source . . . . .	319
<b>64 Maximum profit by buying and selling a share at most twice</b>	<b>320</b>
Source . . . . .	324
<b>65 Maximum size square sub-matrix with all 1s</b>	<b>325</b>
Source . . . . .	328
<b>66 Maximum weight transformation of a given string</b>	<b>329</b>
Source . . . . .	332
<b>67 Minimum Cost Polygon Triangulation</b>	<b>333</b>
Source . . . . .	338

<b>68 Minimum Initial Points to Reach Destination</b>	<b>339</b>
Source . . . . .	342
<b>69 Minimum number of jumps to reach end</b>	<b>343</b>
Source . . . . .	347
<b>70 Minimum number of squares whose sum equals to given number     n</b>	<b>348</b>
Source . . . . .	351
<b>71 Mobile Numeric Keypad Problem</b>	<b>352</b>
Source . . . . .	361
<b>72 Number of paths with exactly k coins</b>	<b>362</b>
Source . . . . .	365
<b>73 Program for Fibonacci numbers</b>	<b>366</b>
Source . . . . .	372
<b>74 Program for nth Catalan Number</b>	<b>373</b>
Source . . . . .	377
<b>75 Remove minimum elements from either side such that 2*min     becomes more than max</b>	<b>378</b>
Source . . . . .	384
<b>76 Shortest Common Supersequence</b>	<b>385</b>
Source . . . . .	390
<b>77 Shortest path with exactly k edges in a directed and weighted     graph</b>	<b>391</b>
Source . . . . .	395
<b>78 Tiling Problem</b>	<b>396</b>
Source . . . . .	398

<b>79 Total number of non-decreasing numbers with n digits</b>	<b>399</b>
Source . . . . .	403
<b>80 Travelling Salesman Problem   Set 1 (Naive and Dynamic Programming)</b>	<b>404</b>
Source . . . . .	406
<b>81 Ugly Numbers</b>	<b>407</b>
Source . . . . .	412
<b>82 Vertex Cover Problem   Set 2 (Dynamic Programming Solution for Tree)</b>	<b>413</b>
Source . . . . .	418
<b>83 Weighted Job Scheduling</b>	<b>419</b>
Source . . . . .	424



# Chapter 1

## Set 1 (Overlapping Subproblems Property)

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.

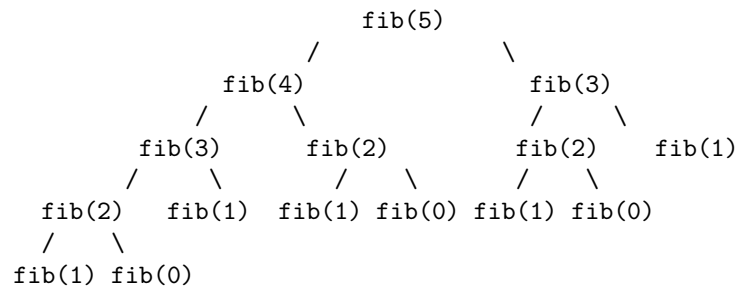
- 1) Overlapping Subproblems
- 2) Optimal Substructure

### 1) Overlapping Subproblems:

Like Divide and Conquer, Dynamic Programming combines solutions to subproblems. 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 be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take example of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
    if ( n <= 1 )
        return n;
    return fib(n-1) + fib(n-2);
}
```

Recursion tree for execution of *fib(5)*



We can see that the function *f(3)* is being called 2 times. If we would have stored the value of *f(3)*, then instead of computing it again, we would have reused the old stored value. 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):*

a) *Memoization (Top Down):* The memoized program for a problem is similar to the recursive version with a small modification that it looks into a lookup table before computing solutions. We initialize a lookup array with all initial values as NIL. Whenever we need solution to a subproblem, we first look into the lookup table. If the precomputed value is there then we return that value, otherwise we calculate the value and put the result in lookup table so that it can be reused later.

Following is the memoized version for nth Fibonacci Number.

```

/* Memoized version for nth Fibonacci number */
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Function to initialize NIL values in lookup table */
void _initialize()
{

```

```

    int i;
    for (i = 0; i < MAX; i++)
        lookup[i] = NIL;
}

/* 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);
    }

    return lookup[n];
}

int main ()
{
    int n = 40;
    _initialize();
    printf("Fibonacci number is %d ", fib(n));
    getchar();
    return 0;
}

```

b) *Tabulation (Bottom Up)*: The tabulated program for a given problem builds a table in bottom up fashion and returns the last entry from table.

```

/* tabulated version */
#include<stdio.h>
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];
}

```

```

}

int main ()
{
    int n = 9;
    printf("Fibonacci number is %d ", fib(n));
    getchar();
    return 0;
}

```

Both tabulated and Memoized store the solutions of subproblems. In Memoized version, table is filled on demand while in tabulated version, starting from the first entry, all entries are filled one by one. Unlike the tabulated version, all entries of the lookup table are not necessarily filled in memoized version. For example, memoized solution of LCS problem doesn't necessarily fill all entries.

To see the optimization achieved by memoized and tabulated versions over the basic recursive version, see the time taken by following runs for 40th Fibonacci number.

Simple recursive program

Memoized version

tabulated version

Also see method 2 of Ugly Number post for one more simple example where we have overlapping subproblems and we store the results of subproblems.

We will be covering Optimal Substructure Property and some more example problems in future posts on Dynamic Programming.

Try following questions as an exercise of this post.

- 1) Write a memoized version for LCS problem. Note that the tabular version is given in the CLRS book.
- 2) How would you choose between Memoization and Tabulation?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

<http://www.youtube.com/watch?v=V5hZoJ6uK-s>

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-1/>

## Chapter 2

# Set 2 (Optimal Substructure Property)

As we discussed in Set 1, following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming.

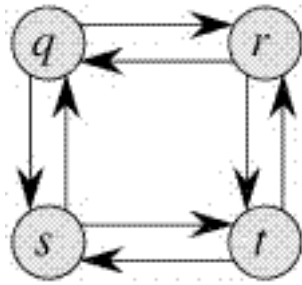
- 1) Overlapping Subproblems
- 2) Optimal Substructure

We have already discussed Overlapping Subproblem property in the Set 1. Let us discuss Optimal Substructure property here.

**2) Optimal Substructure:** 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. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph given in the CLRS book. There are two longest paths from  $q$  to  $t$ :  $q \rightarrow r \rightarrow t$  and  $q \rightarrow s \rightarrow t$ . Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path  $q \rightarrow r \rightarrow t$  is not a combination of longest path from  $q$  to  $r$  and longest path from  $r$  to  $t$ , because the longest path from  $q$  to  $r$  is  $q \rightarrow s \rightarrow t \rightarrow r$ .



We will be covering some example problems in future posts on Dynamic Programming.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**References:**

[http://en.wikipedia.org/wiki/Optimal\\_substructure](http://en.wikipedia.org/wiki/Optimal_substructure)

CLRS book

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-2-optimal-substructure-property/>

Category: Misc Tags: Dynamic Programming

## Chapter 3

### Set 3 (Longest Increasing Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively.

Let us discuss Longest Increasing Subsequence (LIS) problem as an example problem that can be solved using Dynamic Programming.

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

#### **Optimal Substructure:**

Let  $arr[0..n-1]$  be the input array and  $L(i)$  be the length of the LIS till index  $i$  such that  $arr[i]$  is part of LIS and  $arr[i]$  is the last element in LIS, then  $L(i)$  can be recursively written as.

$$L(i) = \{ 1 + \text{Max} ( L(j) ) \} \text{ where } j$$

*To get LIS of a given array, we need to return  $\max(L(i))$  where 0 Overlapping Subproblems:*

*Following is simple recursive implementation of the LIS problem. The implementation simply follows the recursive structure mentioned above. The value of lis ending with every element is returned using `max_ending_here`. The overall lis is returned using pointer to a variable `max`.*

```
/* A Naive recursive implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>
```

```
/* To make use of recursive calls, this function must return two things:
```

```

1) Length of LIS ending with element arr[n-1]. We use max_ending_here
   for this purpose
2) Overall maximum as the LIS may end with an element before arr[n-1]
   max_ref is used this purpose.
The value of LIS of full array of size n is stored in *max_ref which is our final result
*/
int _lis( int arr[], int n, int *max_ref)
{
    /* Base case */
    if(n == 1)
        return 1;

    int res, max_ending_here = 1; // length of LIS ending with arr[n-1]

    /* Recursively get all LIS ending with arr[0], arr[1] ... ar[n-2]. If
       arr[i-1] is smaller than arr[n-1], and max ending with arr[n-1] needs
       to be updated, then update it */
    for(int i = 1; i < n; i++)
    {
        res = _lis(arr, i, max_ref);
        if (arr[i-1] < arr[n-1] && res + 1 > max_ending_here)
            max_ending_here = res + 1;
    }

    // Compare max_ending_here with the overall max. And update the
    // overall max if needed
    if (*max_ref < max_ending_here)
        *max_ref = max_ending_here;

    // Return length of LIS ending with arr[n-1]
    return max_ending_here;
}

// The wrapper function for _lis()
int lis(int arr[], int n)
{
    // The max variable holds the result
    int max = 1;

    // The function _lis() stores its result in max
    _lis( arr, n, &max );

    // returns max
    return max;
}

```



```

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n",  lis( arr, n ));
    getchar();
    return 0;
}

```

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 tabulated implementation for the LIS problem.

```

/* Dynamic Programming implementation of LIS problem */
#include<stdio.h>
#include<stdlib.h>

/* lis() returns the length of the longest increasing subsequence in
   arr[] of size n */
int lis( int arr[], int n )
{
    int *lis, i, j, max = 0;
    lis = (int*) malloc ( sizeof( int ) * n );

    /* 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];

    /* Free memory to avoid memory leak */
    free( lis );

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = { 10, 22, 9, 33, 21, 50, 41, 60 };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LIS is %d\n", lis( arr, n ) );

    getchar();
    return 0;
}

```

Note that the time complexity of the above Dynamic Programmig (DP) solution is  $O(n^2)$  and there is a  $O(n\log n)$  solution for the LIS problem (see this). We have not discussed the  $n\log n$  solution here as the purpose of this post is to explain Dynamic Programmig with a simple example.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-3-longest-increasing-subsequence/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Set 2 (Optimal Substructure Property) A Boolean Array Puzzle →

Writing code in comment? Please use [code.geeksforgeeks.org](https://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 4

# Set 4 (Longest Common Subsequence)

We have discussed Overlapping Subproblems and Optimal Substructure properties in Set 1 and Set 2 respectively. We also discussed one example problem in Set 3. Let us discuss Longest Common Subsequence (LCS) problem as one more example problem that can be solved using Dynamic Programming.

*LCS Problem Statement:* 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.

The naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem.

### 1) 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  
 $L(X[0..m-1], Y[0..n-1]) = \text{MAX} ( L(X[0..m-2], Y[0..n-1]), L(X[0..m-1], Y[0..n-2]) )$

Examples:

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

$L(\text{“AGGTAB”}, \text{“GXTXAYB”}) = 1 + L(\text{“AGGTA”}, \text{“GXTXAY”})$

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

$L(\text{“ABCDGH”}, \text{“AEDFHR”}) = \text{MAX} ( L(\text{“ABCDG”}, \text{“AEDFHR”}), L(\text{“ABCDGH”}, \text{“AEDFH”}) )$

So the LCS problem has optimal substructure property as the main problem can be solved using solutions to subproblems.

## 2) Overlapping Subproblems:

Following is simple recursive implementation of the LCS problem. The implementation simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs( char *X, char *Y, int m, int n )
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m-1] == Y[n-1])
        return 1 + lcs(X, Y, m-1, n-1);
    else
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
```

```

int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    getchar();
    return 0;
}

```

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")

```

In the above partial recursion tree, lcs(“AXY”, “AYZ”) is being solved twice. If we draw the complete recursion tree, then 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 tabulated implementation for the LCS problem.

```

/* Dynamic Programming implementation of LCS problem */
#include<stdio.h>
#include<stdlib.h>

int max(int a, int b);

/* 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];
}

/* Utility function to get max of 2 integers */
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs( X, Y, m, n ) );

    getchar();
    return 0;
}

```

Time Complexity of the above implementation is  $O(mn)$  which is much better than the worst case time complexity of Naive Recursive implementation.

The above algorithm/code returns only length of LCS. Please see the following post for printing the LCS.

#### Printing Longest Common Subsequence

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

#### References:

<http://www.youtube.com/watch?v=V5hZoJ6uK-s>

[http://www.algorithmist.com/index.php/Longest\\_Common\\_Subsequence](http://www.algorithmist.com/index.php/Longest_Common_Subsequence)

<http://www.ics.uci.edu/~eppstein/161/960229.html>

[http://en.wikipedia.org/wiki/Longest\\_common\\_subsequence\\_problem](http://en.wikipedia.org/wiki/Longest_common_subsequence_problem)

#### Source

<http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/>



## Chapter 5

### Set 5 (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'.

1. Insert
2. Remove
3. Replace

All of the above operations are of equal cost.

#### Examples:

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

**What are the subproblems in this case?**

The idea is process all characters one by one starting 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.
  - (a) Insert: Recur for m and n-1
  - (b) Remove: Recur for m-1 and n
  - (c) Replace: Recur for m-1 and n-1

Below is C++ implementation of above Naive recursive solution.

C++

```
// A Naive recursive C++ program to find minimum number
// operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

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
                );
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDist( str1 , str2 , str1.length(), str2.length());

    return 0;
}

```

## Python

```

# A Naive recursive Python program to find minimum number
# operations to convert str1 to str2
def editDistance(str1, str2, m , 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 editDistance(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(editDistance(str1, str2, m, n-1),    # Insert
                editDistance(str1, str2, m-1, n),    # Remove
                editDistance(str1, str2, m-1, n-1)    # Replace
                )

# Driver program to test the above function
str1 = "sunday"
str2 = "saturday"
print editDistance(str1, str2, len(str1), len(str2))

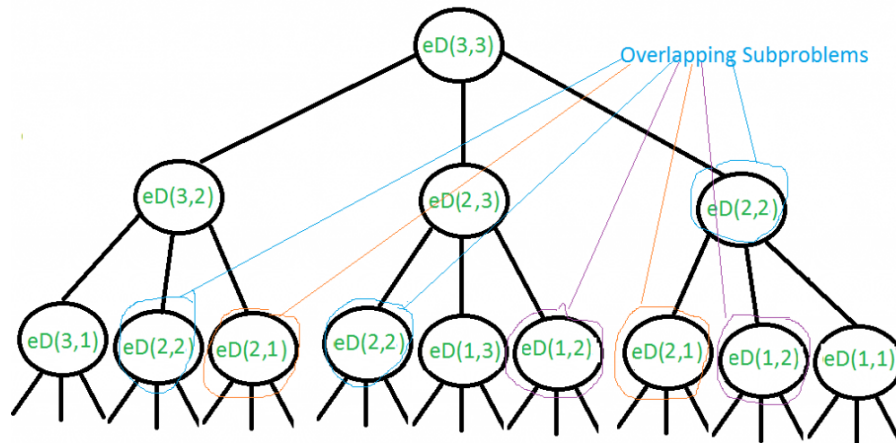
# This code is contributed by Bhavya Jain

```

Output:

3

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



Worst case recursion tree when  $m = 3$ ,  $n = 3$ .  
Worst case example  $str1 = "abc"$   $str2 = "xyz"$

We can see that many subproblems are solved again and again, for example  $eD(2,2)$  is called three times. Since same subproblems are called again, this problem has Overlapping Subproblems property. So Edit Distance problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array that stores results of subproblems.

C++

```
// A Dynamic Programming based C++ program to find minimum
// number operations to convert str1 to str2
#include<bits/stdc++.h>
using namespace std;

// Utility function to find minimum of three numbers
int min(int x, int y, int z)
{
    return min(min(x, y), z);
}

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
        // insert 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];
}

// Driver program
int main()
{
    // your code goes here
    string str1 = "sunday";
    string str2 = "saturday";

    cout << editDistDP(str1, str2, str1.length(), str2.length());

    return 0;
}

```

**Python**

```

# A Dynamic Programming based Python program for edit
# distance problem
def editDistDP(str1, str2, m, n):
    # Create a table to store results of subproblems
    dp = [[0 for x in range(n+1)] for x in range(m+1)]

    # Fill d[][] in bottom up manner
    for i in range(m+1):
        for j in range(n+1):

            # If first string is empty, only option is to
            # insert 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
            elif j == 0:
                dp[i][j] = i    # Min. operations = i

            # If last characters are same, ignore last char
            # and recur for remaining string
            elif 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]

# Driver program
str1 = "sunday"
str2 = "saturday"

print(editDistDP(str1, str2, len(str1), len(str2)))
# This code is contributed by Bhavya Jain

```

Output:

Time Complexity:  $O(m \times n)$

Auxiliary Space:  $O(m \times n)$

**Applications:** There are many practical applications of edit distance algorithm, refer Lucene API for sample. Another example, display all the words in a dictionary that are near proximity to a given word\incorrectly spelled word.

Thanks to Vivek Kumar for suggesting above updates.

Thanks to **Venki** for providing initial post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-5-edit-distance/>



## Chapter 6

### Set 6 (Min Cost Path)

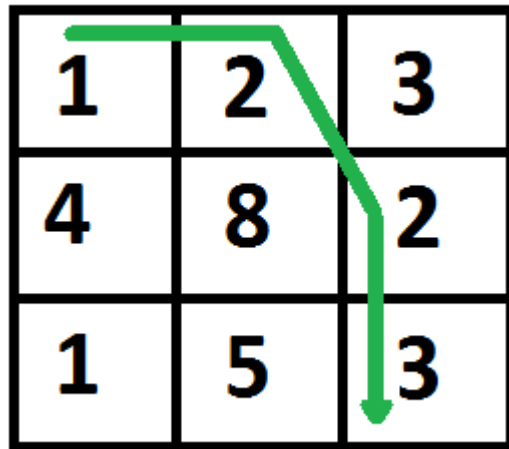
Given a cost matrix `cost[][]` and a position  $(m, n)$  in `cost[][]`, write a function that returns cost of minimum cost path to reach  $(m, n)$  from  $(0, 0)$ . Each cell of the matrix represents a cost to traverse through that cell. Total cost of a path to reach  $(m, n)$  is sum of all the costs on that path (including both source and destination). You can only traverse down, right and diagonally lower cells from a given cell, i.e., from a given cell  $(i, j)$ , cells  $(i+1, j)$ ,  $(i, j+1)$  and  $(i+1, j+1)$  can be traversed. You may assume that all costs are positive integers.

For example, in the following figure, what is the minimum cost path to  $(2, 2)$ ?

<b>1</b>	<b>2</b>	<b>3</b>
<b>4</b>	<b>8</b>	<b>2</b>
<b>1</b>	<b>5</b>	<b>3</b>

The path with minimum cost is highlighted in the following figure. The path is  $(0, 0) \rightarrow (0, 1) \rightarrow (1, 2) \rightarrow (2, 2)$ . The cost of the path is 8 ( $1 + 2 + 2 + 3$ ).

1	2	3
4	8	2
1	5	3



### 1) Optimal Substructure

The path to reach (m, n) must be through one of the 3 cells: (m-1, n-1) or (m-1, n) or (m, n-1). So minimum cost to reach (m, n) can be written as “minimum of the 3 cells plus cost[m][n]”.

$$\text{minCost}(m, n) = \min(\text{minCost}(m-1, n-1), \text{minCost}(m-1, n), \text{minCost}(m, n-1)) + \text{cost}[m][n]$$

### 2) Overlapping Subproblems

Following is simple recursive implementation of the MCP (Minimum Cost Path) problem. The implementation simply follows the recursive structure mentioned above.

```
/* A Naive recursive implementation of MCP(Minimum Cost Path) problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

/* Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C] */
int minCost(int cost[R][C], int m, int n)
{
    if (n < 0 || m < 0)
        return INT_MAX;
    else if (m == 0 && n == 0)
        return cost[m][n];
    else
        return cost[m][n] + min( minCost(cost, m-1, n-1),
                                minCost(cost, m-1, n),
                                minCost(cost, m, n-1) );
}
```

```

        minCost(cost, m-1, n),
        minCost(cost, m, n-1) );
    }

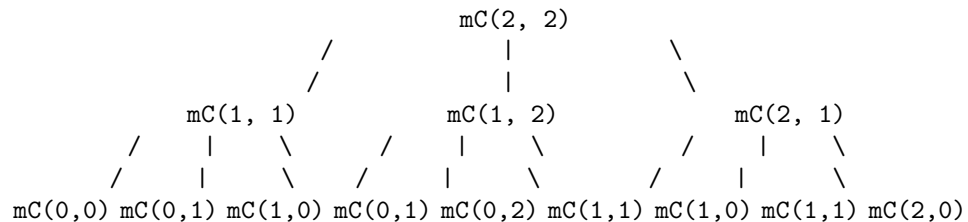
/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
    if (x < y)
        return (x < z)? x : z;
    else
        return (y < z)? y : z;
}

/* Driver program to test above functions */
int main()
{
    int cost[R][C] = { {1, 2, 3},
                        {4, 8, 2},
                        {1, 5, 3} };
    printf(" %d ", minCost(cost, 2, 2));
    return 0;
}

```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree, there are many nodes which appear more than once. Time complexity of this naive recursive solution is exponential and it is terribly slow.

mC refers to minCost()



So the MCP problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `tc[][]` in bottom up manner.

C++

```
/* Dynamic Programming implementation of MCP problem */
#include<stdio.h>
#include<limits.h>
#define R 3
#define C 3

int min(int x, int y, int z);

int minCost(int cost[R][C], int m, int n)
{
    int i, j;

    // Instead of following line, we can use int tc[m+1][n+1] or
    // dynamically allocate memory to save space. The following line is
    // used to keep the program simple and make it working on all compilers.
    int tc[R][C];

    tc[0][0] = cost[0][0];

    /* Initialize first column of total cost(tc) array */
    for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];

    /* Initialize first row of tc array */
    for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];

    /* Construct rest of the tc array */
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) + cost[i][j];

    return tc[m][n];
}

/* A utility function that returns minimum of 3 integers */
int min(int x, int y, int z)
{
    if (x < y)
        return (x < z)? x : z;
    else
        return (y < z)? y : z;
}
```

```

/* Driver program to test above functions */
int main()
{
    int cost[R][C] = { {1, 2, 3},
                        {4, 8, 2},
                        {1, 5, 3} };
    printf(" %d ", minCost(cost, 2, 2));
    return 0;
}

```

## Python

```

# Dynamic Programming Python implementation of Min Cost Path
# problem
R = 3
C = 3

def minCost(cost, m, n):

    # Instead of following line, we can use int tc[m+1][n+1] or
    # dynamically allocate memoery to save space. The following
    # line is used to keep te program simple and make it working
    # on all compilers.
    tc = [[0 for x in range(C)] for x in range(R)]

    tc[0][0] = cost[0][0]

    # Initialize first column of total cost(tc) array
    for i in range(1, m+1):
        tc[i][0] = tc[i-1][0] + cost[i][0]

    # Initialize first row of tc array
    for j in range(1, n+1):
        tc[0][j] = tc[0][j-1] + cost[0][j]

    # Construct rest of the tc array
    for i in range(1, m+1):
        for j in range(1, n+1):
            tc[i][j] = min(tc[i-1][j-1], tc[i-1][j], tc[i][j-1]) + cost[i][j]

    return tc[m][n]

```

```
# Driver program to test above functions
cost = [[1, 2, 3],
        [4, 8, 2],
        [1, 5, 3]]
print(minCost(cost, 2, 2))

# This code is contributed by Bhavya Jain
```

Output:

8

Time Complexity of the DP implementation is  $O(mn)$  which is much better than Naive Recursive implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-6-min-cost-path/>

## Chapter 7

### Set 7 (Coin Change)

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.

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.

#### 1) Optimal Substructure

To count total number solutions, we can divide all set solutions in two sets.

- 1) Solutions that do not contain  $m$ th coin (or  $S_m$ ).
- 2) Solutions that contain at least one  $S_m$ .

Let  $\text{count}(S[], m, n)$  be the function to count the number of solutions, then it can be written as sum of  $\text{count}(S[], m-1, n)$  and  $\text{count}(S[], m, n-S_m)$ .

Therefore, the problem has optimal substructure property as the problem can be solved using solutions to subproblems.

#### 2) Overlapping Subproblems

Following is a simple recursive implementation of the Coin Change problem. The implementation simply follows the recursive structure mentioned above.

```
#include<stdio.h>

// 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;

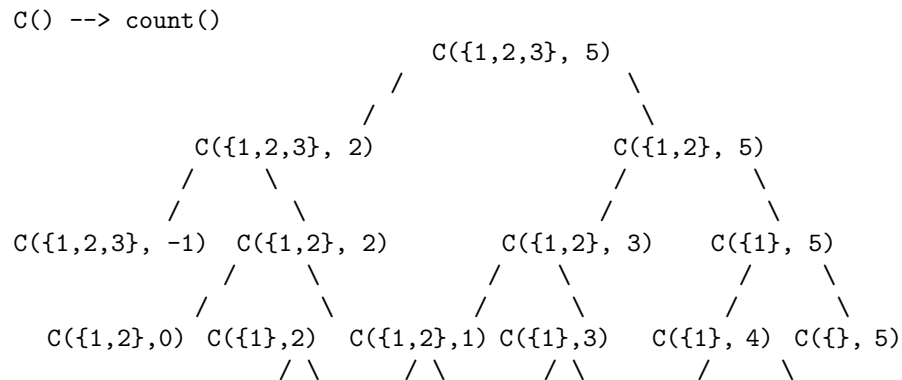
// If there are no coins and n is greater than 0, then no solution exist
if (m <= 0 && n >= 1)
    return 0;

// 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] );
}

// Driver program to test above function
int main()
{
    int i, j;
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    printf("%d ", count(arr, m, 4));
    getchar();
    return 0;
}

```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for  $S = \{1, 2, 3\}$  and  $n = 5$ . The function  $C(\{1\}, 3)$  is called two times. If we draw the complete tree, then we can see that there are many subproblems being called more than once.







```

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

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3};
    int m = sizeof(arr)/sizeof(arr[0]);
    int n = 4;
    printf(" %d ", count(arr, m, n));
    return 0;
}

```

## Python

```

# Dynamic Programming Python implementation of Coin Change problem
def count(S, m, n):
    # We need n+1 rows as the table is constructed in bottom up
    # manner using the base case 0 value case (n = 0)
    table = [[0 for x in range(m)] for x in range(n+1)]

    # Fill the enteries for 0 value case (n = 0)
    for i in range(m):
        table[0][i] = 1

    # Fill rest of the table enteries in bottom up manner
    for i in range(1, n+1):
        for j in range(m):
            # Count of solutions including S[j]
            x = table[i - S[j]][j] if i-S[j] >= 0 else 0

            # Count of solutions excluding S[j]
            y = table[i][j-1] if j >= 1 else 0

            # total count
            table[i][j] = x + y

    return table[n][m-1]

```

```
# Driver program to test above function
arr = [1, 2, 3]
m = len(arr)
n = 4
print(count(arr, m, n))

# This code is contributed by Bhavya Jain
```

Output:

4

Time Complexity:  $O(mn)$

Following is a simplified version of method 2. The auxiliary space required here is  $O(n)$  only.

```
int count( int S[], int m, int n )
{
    // table[i] will be storing the number of solutions for
    // value i. We need n+1 rows as the table is constructed
    // in bottom up manner using the base case (n = 0)
    int table[n+1];

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // Pick all coins one by one and update the table[] values
    // after the index greater than or equal to the value of the
    // picked coin
    for(int i=0; i<m; i++)
        for(int j=S[i]; j<=n; j++)
            table[j] += table[j-S[i]];

    return table[n];
}
```

Thanks to Rohan Laishram for suggesting this space optimized version.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:

[http://www.algorithmist.com/index.php/Coin\\_Change](http://www.algorithmist.com/index.php/Coin_Change)

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/>

## Chapter 8

# Set 8 (Matrix Chain Multiplication)

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

We have many options to multiply a chain of matrices because matrix multiplication is associative. In other words, no matter how we parenthesize the product, the result will be the same. For example, if we had four matrices A, B, C, and D, we would have:

$$(ABC)D = (AB)(CD) = A(BCD) = \dots$$

However, the order in which we parenthesize the product affects the number of simple arithmetic operations needed to compute the product, or the efficiency. For example, suppose A is a  $10 \times 30$  matrix, B is a  $30 \times 5$  matrix, and C is a  $5 \times 60$  matrix. Then,

$$\begin{aligned}(AB)C &= (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operations} \\ A(BC) &= (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operations}.\end{aligned}$$

Clearly the first parenthesization requires less number of operations.

Given an array  $p[]$  which represents the chain of matrices such that the  $i$ th matrix  $A_i$  is of dimension  $p[i-1] \times p[i]$ . We need to write a function `MatrixChainOrder()` that should return the minimum number of multiplications needed to multiply the chain.

Input:  $p[] = \{40, 20, 30, 10, 30\}$

Output: 26000

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

$(A(BC))D \rightarrow 20 \times 30 \times 10 + 40 \times 20 \times 10 + 40 \times 10 \times 30$

Input:  $p[] = \{10, 20, 30, 40, 30\}$

Output: 30000

There are 4 matrices of dimensions 10x20, 20x30, 30x40 and 40x30.

Let the input 4 matrices be A, B, C and D. The minimum number of multiplications are obtained by putting parenthesis in following way

$((AB)C)D \rightarrow 10 \times 20 \times 30 + 10 \times 30 \times 40 + 10 \times 40 \times 30$

Input:  $p[] = \{10, 20, 30\}$

Output: 6000

There are only two matrices of dimensions 10x20 and 20x30. So there is only one way to multiply the matrices, cost of which is  $10 \times 20 \times 30$

### 1) Optimal Substructure:

A simple solution is to place parenthesis at all possible places, calculate the cost for each placement and return the minimum value. In a chain of matrices of size  $n$ , we can place the first set of parenthesis in  $n-1$  ways. For example, if the given chain is of 4 matrices. let the chain be ABCD, then there are 3 way to place first set of parenthesis: A(BCD), (AB)CD and (ABC)D. So when we place a set of parenthesis, we divide the problem into subproblems of smaller size. Therefore, the problem has optimal substructure property and can be easily solved using recursion.

Minimum number of multiplication needed to multiply a chain of size  $n = \text{Minimum of all } n-1 \text{ placements (these placements create subproblems of smaller size)}$

### 2) Overlapping Subproblems

Following is a recursive implementation that simply follows the above optimal substructure property.

```

/* A naive recursive implementation that simply follows the above optimal
   substructure property */
#include<stdio.h>
#include<limits.h>

// Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
int MatrixChainOrder(int p[], int i, int j)
{
    if(i == j)
        return 0;
    int k;
    int min = INT_MAX;
    int count;

    // place parenthesis at different places between first and last matrix,
    // recursively calculate count of multiplications for each parenthesis
    // placement and return the minimum count
    for (k = i; k < j; k++)
    {
        count = MatrixChainOrder(p, i, k) +
                MatrixChainOrder(p, k+1, j) +
                p[i-1]*p[k]*p[j];

        if (count < min)
            min = count;
    }

    // Return minimum count
    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 2, 3, 4, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
           MatrixChainOrder(arr, 1, n-1));

    getchar();
    return 0;
}

```

The diagram illustrates a hierarchical tree structure. The root node is 1..4. It has four children: 1..1, 2..4, 1..2, and 1..3. Additionally, there is a node 4..4 shown as a child of 1..4. The tree further decomposes into grandchildren and great-grandchildren. Some nodes are highlighted in gray, indicating a specific path or set of nodes.

```

graph TD
    14[1..4] --- 11[1..1]
    14 --- 24[2..4]
    14 --- 12[1..2]
    14 --- 13[1..3]
    14 --- 44[4..4]
    24 --- 22[2..2]
    24 --- 34[3..4]
    24 --- 23[2..3]
    24 --- 44_2[4..4]
    12 --- 11_2[1..1]
    12 --- 22_2[2..2]
    13 --- 11_3[1..1]
    13 --- 23_2[2..3]
    13 --- 12_2[1..2]
    13 --- 33_2[3..3]
    34 --- 33_1[3..3]
    34 --- 44_1[4..4]
    23_2 --- 33_3[3..3]
    23_2 --- 44_3[4..4]
    22_2 --- 33_4[3..3]
    22_2 --- 44_4[4..4]
    12_2 --- 11_4[1..1]
    12_2 --- 22_4[2..2]
  
```

### Dynamic Programming Solution

C

48



```

the matrix  $A[i]A[i+1]\dots A[j] = A[i..j]$  where dimension of  $A[i]$  is
     $p[i-1] \times p[i]$  */

// cost is zero when multiplying one matrix.
for (i = 1; i < n; i++)
    m[i][i] = 0;

// L is chain length.
for (L=2; L<n; L++)
{
    for (i=1; i<=n-L+1; i++)
    {
        j = i+L-1;
        m[i][j] = INT_MAX;
        for (k=i; k<=j-1; k++)
        {
            // q = cost/scalar multiplications
            q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j];
            if (q < m[i][j])
                m[i][j] = q;
        }
    }
}

return m[1][n-1];
}

int main()
{
    int arr[] = {1, 2, 3, 4};
    int size = sizeof(arr)/sizeof(arr[0]);

    printf("Minimum number of multiplications is %d ",
           MatrixChainOrder(arr, size));

    getchar();
    return 0;
}

```

## Python

# Dynamic Programming Python implementation of Matrix Chain Multiplication

```

# See the Cormen book for details of the following algorithm
import sys

# Matrix Ai has dimension p[i-1] x p[i] for i = 1..n
def MatrixChainOrder(p, n):
    # For simplicity of the program, one extra row and one extra column are
    # allocated in m[] []. 0th row and 0th column of m[] [] are not used
    m = [[0 for x in range(n)] for x in range(n)]

    # m[i,j] = Minimum number of scalar multiplications needed to compute
    # the matrix A[i]A[i+1]...A[j] = A[i..j] where dimation of A[i] is
    # p[i-1] x p[i]

    # cost is zero when multiplying one matrix.
    for i in range(1, n):
        m[i][i] = 0

    # L is chain length.
    for L in range(2, n):
        for i in range(1, n-L+1):
            j = i+L-1
            m[i][j] = sys.maxint
            for k in range(i, j):

                # q = cost/scalar multiplications
                q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
                if q < m[i][j]:
                    m[i][j] = q

    return m[1][n-1]

# Driver program to test above function
arr = [1, 2, 3 ,4]
size = len(arr)

print("Minimum number of multiplications is " + str(MatrixChainOrder(arr, size)))
# This Code is contributed by Bhavya Jain

```

Output:

Minimum number of multiplications is 18

Time Complexity:  $O(n^3)$

Auxiliary Space:  $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**References:**

[http://en.wikipedia.org/wiki/Matrix\\_chain\\_multiplication](http://en.wikipedia.org/wiki/Matrix_chain_multiplication)

<http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm>

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/>

## Chapter 9

# Set 9 (Binomial Coefficient)

Following are common definition of Binomial Coefficients.

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

2) 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 an  $n$ -element set.

### The Problem

*Write a function that takes two parameters  $n$  and  $k$  and returns the value of Binomial Coefficient  $C(n, k)$ .* For example, your function should return 6 for  $n = 4$  and  $k = 2$ , and it should return 10 for  $n = 5$  and  $k = 2$ .

### 1) Optimal Substructure

The value of  $C(n, k)$  can recursively calculated using following standard formula for Binomial Coefficients.

$$\begin{aligned}C(n, k) &= C(n-1, k-1) + C(n-1, k) \\C(n, 0) &= C(n, n) = 1\end{aligned}$$

### 2) Overlapping Subproblems

Following is simple recursive implementation that simply follows the recursive structure mentioned above.

```
// A Naive Recursive Implementation
#include<stdio.h>
```

```

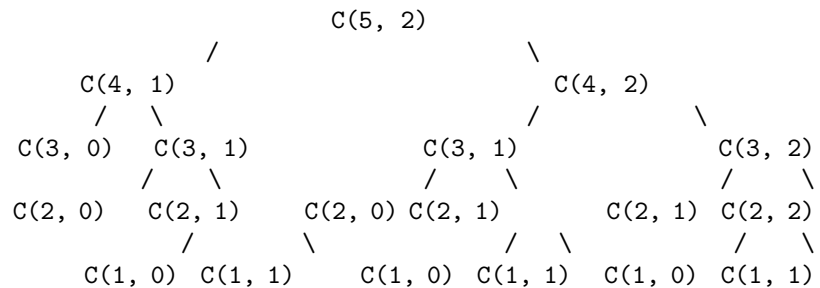
// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    // Base Cases
    if (k==0 || k==n)
        return 1;

    // Recur
    return binomialCoeff(n-1, k-1) + binomialCoeff(n-1, k);
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k));
    return 0;
}

```

It should be noted that the above function computes the same subproblems again and again. See the following recursion tree for  $n = 5$  and  $k = 2$ . The function  $C(3, 1)$  is called two times. For large values of  $n$ , there will be many common subproblems.



Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Binomial Coefficient problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array  $C[][]$  in bottom up manner. Following is Dynamic Programming based implementation.

C

```
// A Dynamic Programming based solution that uses table C[] [] to
// calculate the Binomial Coefficient
#include<stdio.h>

// Prototype of a utility function that returns minimum of two integers
int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
    int C[n+1][k+1];
    int i, j;

    // Caculate value of Binomial Coefficient in bottom up manner
    for (i = 0; i <= n; i++)
    {
        for (j = 0; j <= min(i, k); j++)
        {
            // Base Cases
            if (j == 0 || j == i)
                C[i][j] = 1;

            // Calculate value using previosly stored values
            else
                C[i][j] = C[i-1][j-1] + C[i-1][j];
        }
    }

    return C[n][k];
}

// A utility function to return minimum of two integers
int min(int a, int b)
{
    return (a<b)? a: b;
}

/* Drier program to test above function*/
int main()
{
    int n = 5, k = 2;
    printf ("Value of C(%d, %d) is %d ", n, k, binomialCoeff(n, k) );
    return 0;
}
```

```
}
```

## Python

```
# A Dynamic Programming based Python Program that uses table C[] []
# to calculate the Binomial Coefficient

# Returns value of Binomial Coefficient C(n, k)
def binomialCoef(n, k):
    C = [[0 for x in range(k+1)] for x in range(n+1)]

    # Calculate value of Binomial Coefficient in bottom up manner
    for i in range(n+1):
        for j in range(min(i, k)+1):
            # Base Cases
            if j == 0 or j == i:
                C[i][j] = 1

            # Calculate value using previously stored values
            else:
                C[i][j] = C[i-1][j-1] + C[i-1][j]

    return C[n][k]

# Driver program to test above function
n = 5
k = 2
print("Value of C[" + str(n) + "][" + str(k) + "] is "
      + str(binomialCoef(n,k)))

# This code is contributed by Bhavya Jain
```

Output:

```
Value of C[5][2] is 10
```

Time Complexity:  $O(n*k)$   
Auxiliary Space:  $O(n*k)$

Following is a space optimized version of the above code. The following code only uses  $O(k)$ . Thanks to AKfor suggesting this method.

```
// A space optimized Dynamic Programming Solution
int binomialCoeff(int n, int k)
{
    int* C = (int*)calloc(k+1, sizeof(int));
    int i, j, res;

    C[0] = 1;

    for(i = 1; i <= n; i++)
    {
        for(j = min(i, k); j > 0; j--)
            C[j] = C[j] + C[j-1];
    }

    res = C[k]; // Store the result before freeing memory

    free(C); // free dynamically allocated memory to avoid memory leak

    return res;
}
```

Time Complexity:  $O(n*k)$

Auxiliary Space:  $O(k)$

References:

<http://www.csl.mtu.edu/cs4321/www/Lectures/Lecture%2015%20-%20Dynamic%20Programming%20Binomial%20Coefficients.htm>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-9-binomial-coefficient/>



## Chapter 10

# Set 10 ( 0-1 Knapsack 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. In other words, given two integer arrays  $val[0..n-1]$  and  $wt[0..n-1]$  which represent values and weights associated with  $n$  items respectively. Also given an integer  $W$  which represents knapsack capacity, 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  $n$ th item).
- 2) Value of  $n$ th item plus maximum value obtained by  $n-1$  items and  $W$  minus weight of the  $n$ th item (including  $n$ th item).

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

### 2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```

/* 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)
                    );
}

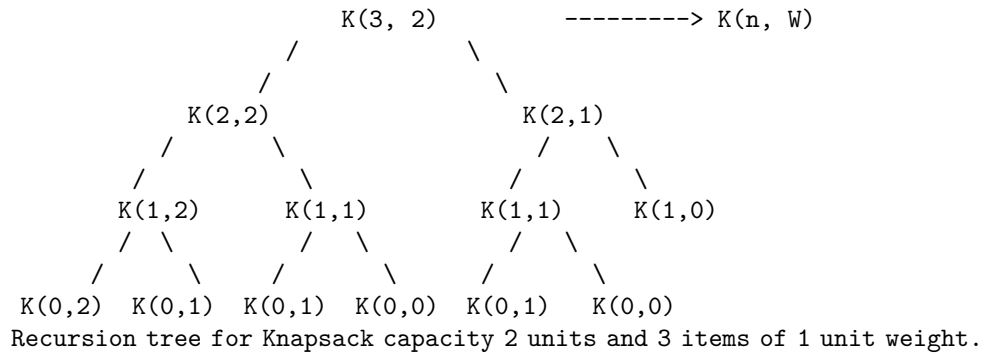
// Driver program to test above function
int main()
{
    int val[] = {60, 100, 120};
    int wt[] = {10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);
    printf("%d", knapSack(W, wt, val, n));
    return 0;
}

```

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 the following recursion tree,  $K()$  refers to  $\text{knapSack}()$ . 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}
```



Since subproblems are evaluated again, this problem has Overlapping Subproblems property. So the 0-1 Knapsack problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array  $K[][]$  in bottom up manner. Following is Dynamic Programming based implementation.

C++

```
// 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;
}

```

## Python

```

# A Dynamic Programming based Python Program for 0-1 Knapsack problem
# Returns the maximum value that can be put in a knapsack of capacity W
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]

    # Build table K[][] in bottom up manner
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif 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]

# Driver program to test above function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50

```

```
n = len(val)
print(knapSack(W, wt, val, n))

# This code is contributed by Bhavya Jain
```

Output:

220

Time Complexity:  $O(nW)$  where  $n$  is the number of items and  $W$  is the capacity of knapsack.

References:

<http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf>

<http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/>

## Chapter 11

# Set 11 (Egg Dropping Puzzle)

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.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the second floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg-droppings that is guaranteed to work in all cases?

The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that total number of trials are minimized.

Source: Wiki for Dynamic Programming

In this post, we will discuss solution to a general problem with  $n$  eggs and  $k$  floors. The solution is to try dropping an egg from every floor (from 1 to  $k$ ) and

recursively calculate the minimum number of droppings needed in worst case. The floor which gives the minimum value in worst case is going to be part of the solution.

In the following solutions, we return the minimum number of trails in worst case; these solutions can be easily modified to print floor numbers of every trials also.

### 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  $x$ th 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  $x$ th 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 trails 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}}
```

### 2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
# 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 trails 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;
}

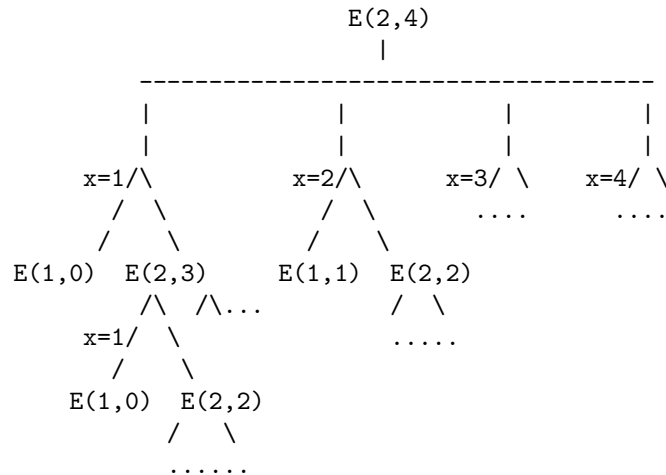
```

Output:

Minimum number of trials in worst case with 2 eggs and 10 floors is 4

It should be noted that the above function computes the same subproblems again and again. See the following partial recursion tree,  $E(2, 2)$  is being evaluated twice. There will many repeated subproblems when you draw the complete recursion tree even for small values of  $n$  and  $k$ .





Partial recursion tree for 2 eggs and 4 floors.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Egg Dropping Puzzle has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `eggFloor[][]` in bottom up manner.

### Dynamic Programming Solution

Following are C++ and Python implementations for Egg Dropping problem using Dynamic Programming.

#### C++

```

# A Dynamic Programming based C++ Program for the Egg Dropping Puzzle
# 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)
{

```

```

/* A 2D table where entry eggFloor[i][j] will represent minimum
   number of trials needed for i eggs and j floors. */
int eggFloor[n+1][k+1];
int res;
int i, j, x;

// We need one trial for one floor and 0 trials for 0 floors
for (i = 1; i <= n; i++)
{
    eggFloor[i][1] = 1;
    eggFloor[i][0] = 0;
}

// We always need j trials for one egg and j floors.
for (j = 1; j <= k; j++)
    eggFloor[1][j] = j;

// Fill rest of the entries in table using optimal substructure
// property
for (i = 2; i <= n; i++)
{
    for (j = 2; j <= k; j++)
    {
        eggFloor[i][j] = INT_MAX;
        for (x = 1; x <= j; x++)
        {
            res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x]);
            if (res < eggFloor[i][j])
                eggFloor[i][j] = res;
        }
    }
}

// eggFloor[n][k] holds the result
return eggFloor[n][k];
}

/* Driver program to test to print printDups*/
int main()
{
    int n = 2, k = 36;
    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

```
# A Dynamic Programming based Python Program for the Egg Dropping Puzzle
INT_MAX = 32767

# Function to get minimum number of trails needed in worst
# case with n eggs and k floors
def eggDrop(n, k):
    # A 2D table where entry eggFloor[i][j] will represent minimum
    # number of trials needed for i eggs and j floors.
    eggFloor = [[0 for x in range(k+1)] for x in range(n+1)]

    # We need one trial for one floor and 0 trials for 0 floors
    for i in range(1, n+1):
        eggFloor[i][1] = 1
        eggFloor[i][0] = 0

    # We always need j trials for one egg and j floors.
    for j in range(1, k+1):
        eggFloor[1][j] = j

    # Fill rest of the entries in table using optimal substructure
    # property
    for i in range(2, n+1):
        for j in range(2, k+1):
            eggFloor[i][j] = INT_MAX
            for x in range(1, j+1):
                res = 1 + max(eggFloor[i-1][x-1], eggFloor[i][j-x])
                if res < eggFloor[i][j]:
                    eggFloor[i][j] = res

    # eggFloor[n][k] holds the result
    return eggFloor[n][k]

# Driver program to test to print printDups
n = 2
k = 36
print("Minimum number of trials in worst case with" + str(n) + "eggs and " + str(k) + " floors is " + str(eggDrop(n, k)))

# This code is contributed by Bhavya Jain
```

**Output:**

Minimum number of trials in worst case with 2 eggs and 36 floors is 8

Time Complexity:  $O(nk^2)$

Auxiliary Space:  $O(nk)$

As an exercise, you may try modifying the above DP solution to print all intermediate floors (The floors used for minimum trail solution).

**References:**

<http://archive.itejournal.informs.org/Vol4No1/Sniedovich/index.php>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-11-egg-dropping-puzzle/>

## Chapter 12

# Set 12 (Longest Palindromic Subsequence)

Given a sequence, find the length of the longest palindromic subsequence in it. For example, if the given sequence is “BBABCBCAB”, then the output should be 7 as “BABCBAB” is the longest palindromic subsequence in it. “BBBBB” and “BBCBB” are also palindromic subsequences of the given sequence, but not the longest ones.

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

### 1) Optimal Substructure:

Let  $X[0..n-1]$  be the input sequence of length  $n$  and  $L(0, n-1)$  be the length of the longest palindromic subsequence of  $X[0..n-1]$ .

If last and first characters of  $X$  are same, then  $L(0, n-1) = L(1, n-2) + 2$ .

Else  $L(0, n-1) = \text{MAX} (L(1, n-1), L(0, n-2))$ .

Following is a general recursive solution with all cases handled.

```
// Every single character is a palindrom of length 1
L(i, i) = 1 for all indexes i in given sequence

// IF first and last characters are not same
If (X[i] != X[j]) L(i, j) = max{L(i + 1, j), L(i, j - 1)}

// If there are only 2 characters and both are same
```

```

Else if (j == i + 1) L(i, j) = 2

// If there are more than two characters, and first and last
// characters are same
Else L(i, j) = L(i + 1, j - 1) + 2

```

## 2) Overlapping Subproblems

Following is simple recursive implementation of the LPS problem. The implementation simply follows the recursive structure mentioned above.

```

#include<stdio.h>
#include<string.h>

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *seq, int i, int j)
{
    // Base Case 1: If there is only 1 character
    if (i == j)
        return 1;

    // Base Case 2: If there are only 2 characters and both are same
    if (seq[i] == seq[j] && i + 1 == j)
        return 2;

    // If the first and last characters match
    if (seq[i] == seq[j])
        return lps (seq, i+1, j-1) + 2;

    // If the first and last characters do not match
    return max( lps(seq, i, j-1), lps(seq, i+1, j) );
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKSFORGEEKS";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq, 0, n-1));
    getchar();
}

```

```

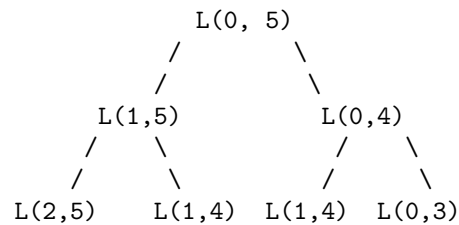
    return 0;
}

```

Output:

The length of the LPS is 5

Considering the above implementation, following is a partial recursion tree for a sequence of length 6 with all different characters.



In the above partial recursion tree,  $L(1, 4)$  is being solved twice. If we draw the complete recursion tree, then we can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So LPS problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array  $L[][]$  in bottom up manner.

### Dynamic Programming Solution

C++

```

# A Dynamic Programming based Python program for LPS problem
# Returns the length of the longest palindromic subsequence in seq
#include<stdio.h>
#include<string.h>

```

```

// A utility function to get max of two integers
int max (int x, int y) { return (x > y)? x : y; }

// Returns the length of the longest palindromic subsequence in seq
int lps(char *str)
{
    int n = strlen(str);
    int i, j, cl;
    int L[n][n]; // Create a table to store results of subproblems

    // Strings of length 1 are palindrome of length 1
    for (i = 0; i < n; i++)
        L[i][i] = 1;

    // Build the table. Note that the lower diagonal values of table are
    // useless and not filled in the process. The values are filled in a
    // manner similar to Matrix Chain Multiplication DP solution (See
    // http://www.geeksforgeeks.org/archives/15553). cl is length of
    // substring
    for (cl=2; cl<=n; cl++)
    {
        for (i=0; i<n-cl+1; i++)
        {
            j = i+cl-1;
            if (str[i] == str[j] && cl == 2)
                L[i][j] = 2;
            else if (str[i] == str[j])
                L[i][j] = L[i+1][j-1] + 2;
            else
                L[i][j] = max(L[i][j-1], L[i+1][j]);
        }
    }

    return L[0][n-1];
}

/* Driver program to test above functions */
int main()
{
    char seq[] = "GEEKS FOR GEEKS";
    int n = strlen(seq);
    printf ("The length of the LPS is %d", lps(seq));
    getchar();
    return 0;
}

```



## Python

```
# A Dynamic Programming based Python program for LPS problem
# Returns the length of the longest palindromic subsequence in seq
def lps(str):
    n = len(str)

    # Create a table to store results of subproblems
    L = [[0 for x in range(n)] for x in range(n)]

    # Strings of length 1 are palindrome of length 1
    for i in range(n):
        L[i][i] = 1

    # Build the table. Note that the lower diagonal values of table are
    # useless and not filled in the process. The values are filled in a
    # manner similar to Matrix Chain Multiplication DP solution (See
    # http://www.geeksforgeeks.org/dynamic-programming-set-8-matrix-chain-multiplication/
    # cl is length of substring
    for cl in range(2, n+1):
        for i in range(n-cl+1):
            j = i+cl-1
            if str[i] == str[j] and cl == 2:
                L[i][j] = 2
            elif str[i] == str[j]:
                L[i][j] = L[i+1][j-1] + 2
            else:
                L[i][j] = max(L[i][j-1], L[i+1][j]);

    return L[0][n-1]

# Driver program to test above functions
seq = "GEEKS FOR GEEKS"
n = len(seq)
print("The length of the LPS is " + str(lps(seq)))

# This code is contributed by Bhavya Jain
```

Output:

The length of the LPS is 7

Time Complexity of the above implementation is  $O(n^2)$  which is much better than the worst case time complexity of Naive Recursive implementation.

This problem is close to the Longest Common Subsequence (LCS) problem. In fact, we can use LCS as a subroutine to solve this problem. Following is the two step solution that uses LCS.

- 1) Reverse the given sequence and store the reverse in another array say `rev[0..n-1]`
- 2) LCS of the given sequence and `rev[]` will be the longest palindromic sequence. This solution is also a  $O(n^2)$  solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**References:**

<http://users.eecs.northwestern.edu/~dda902/336/hw6-sol.pdf>

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-12-longest-palindromic-subsequence/>

## Chapter 13

### Set 13 (Cutting a Rod)

Given a rod of length  $n$  inches and an array of prices that contains prices of all pieces of size smaller than  $n$ . Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example, if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length		1	2	3	4	5	6	7	8
-----									
price		1	5	8	9	10	17	17	20

And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length		1	2	3	4	5	6	7	8
-----									
price		3	5	8	9	10	17	17	20

The naive solution for this problem is to generate all configurations of different pieces and find the highest priced configuration. This solution is exponential in term of time complexity. Let us see how this problem possesses both important properties of a Dynamic Programming (DP) Problem and can efficiently solved using Dynamic Programming.

### 1) Optimal Substructure:

We can get the best price by making a cut at different positions and comparing the values obtained after a cut. We can recursively call the same function for a piece obtained after a cut.

Let  $\text{cutRod}(n)$  be the required (best possible price) value for a rod of length  $n$ .  $\text{cutRod}(n)$  can be written as following.

$$\text{cutRod}(n) = \max(\text{price}[i] + \text{cutRod}(n-i-1)) \text{ for all } i \text{ in } \{0, 1 \dots n-1\}$$

### 2) Overlapping Subproblems

Following is simple recursive implementation of the Rod Cutting problem. The implementation simply follows the recursive structure mentioned above.

```
// A Naive recursive solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
int max(int a, int b) { return (a > b)? a : b;}

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
    if (n <= 0)
        return 0;
    int max_val = INT_MIN;

    // Recursively cut the rod in different pieces and compare different
    // configurations
    for (int i = 0; i<n; i++)
        max_val = max(max_val, price[i] + cutRod(price, n-i-1));

    return max_val;
}

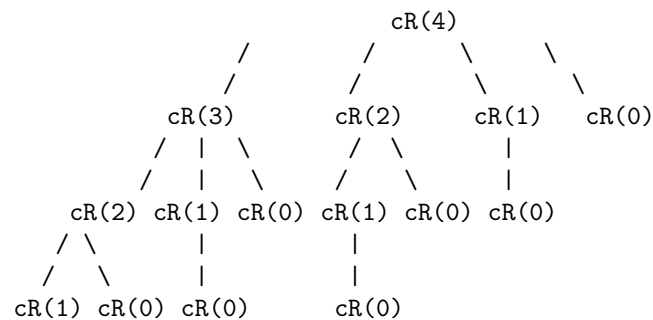
/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}
```

Output:

Maximum Obtainable Value is 22

Considering the above implementation, following is recursion tree for a Rod of length 4.

cR() ---> cutRod()



In the above partial recursion tree, cR(2) is being solved twice. We can see that there are many subproblems which are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So the Rod Cutting problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array val[] in bottom up manner.

C++

```
// A Dynamic Programming solution for Rod cutting problem
#include<stdio.h>
#include<limits.h>

// A utility function to get the maximum of two integers
```

```

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

/* Returns the best obtainable price for a rod of length n and
   price[] as prices of different pieces */
int cutRod(int price[], int n)
{
    int val[n+1];
    val[0] = 0;
    int i, j;

    // Build the table val[] in bottom up manner and return the last entry
    // from the table
    for (i = 1; i<=n; i++)
    {
        int max_val = INT_MIN;
        for (j = 0; j < i; j++)
            max_val = max(max_val, price[j] + val[i-j-1]);
        val[i] = max_val;
    }

    return val[n];
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 5, 8, 9, 10, 17, 17, 20};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum Obtainable Value is %d\n", cutRod(arr, size));
    getchar();
    return 0;
}

```

## Python

```

# A Dynamic Programming solution for Rod cutting problem
INT_MIN = -32767

# Returns the best obtainable price for a rod of length n and
# price[] as prices of different pieces
def cutRod(price, n):
    val = [0 for x in range(n+1)]

```

```

val[0] = 0

# Build the table val[] in bottom up manner and return
# the last entry from the table
for i in range(1, n+1):
    max_val = INT_MIN
    for j in range(i):
        max_val = max(max_val, price[j] + val[i-j-1])
    val[i] = max_val

return val[n]

# Driver program to test above functions
arr = [1, 5, 8, 9, 10, 17, 17, 20]
size = len(arr)
print("Maximum Obtainable Value is " + str(cutRod(arr, size)))

# This code is contributed by Bhavya Jain

```

Output:

```
Maximum Obtainable Value is 22
```

Time Complexity of the above implementation is  $O(n^2)$  which is much better than the worst case time complexity of Naive Recursive implementation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-13-cutting-a-rod/>

## Chapter 14

# Set 14 (Maximum Sum Increasing Subsequence)

Given an array of  $n$  positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 ( $1 + 2 + 3 + 100$ ), if the input array is {3, 4, 5, 10}, then output should be 22 ( $3 + 4 + 5 + 10$ ) and if the input array is {10, 5, 4, 3}, then output should be 10

### Solution

This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. We need a slight change in the Dynamic Programming solution of LIS problem. All we need to change is to use sum as a criteria instead of length of increasing subsequence.

Following are C/C++ and Python implementations for Dynamic Programming solution of the problem.

### C/C++

```
/* Dynamic Programming implementation of Maximum Sum Increasing
Subsequence (MSIS) problem */
#include<stdio.h>

/* maxSumIS() returns the maximum sum of increasing subsequence
   in arr[] of size n */
int maxSumIS( int arr[], int n )
{
    int i, j, max = 0;
```



```

int msis[n];

/* Initialize msis values for all indexes */
for ( i = 0; i < n; i++ )
    msis[i] = arr[i];

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

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

return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Sum of maximum sum increasing subsequence is %d\n",
        maxSumIS( arr, n ) );
    return 0;
}

```

## Python

```

# Dynamic Programming based Python implementation of Maximum Sum Increasing
# Subsequence (MSIS) problem

# maxSumIS() returns the maximum sum of increasing subsequence in arr[] of
# size n
def maxSumIS(arr, n):
    max = 0
    msis = [0 for x in range(n)]

    # Initialize msis values for all indexes

```

```

for i in range(n):
    msis[i] = arr[i]

# Compute maximum sum values in bottom up manner
for i in range(1, n):
    for j in range(i):
        if arr[i] > arr[j] and msis[i] < msis[j] + arr[i]:
            msis[i] = msis[j] + arr[i]

# Pick maximum of all msis values
for i in range(n):
    if max < msis[i]:
        max = msis[i]

return max

# Driver program to test above function
arr = [1, 101, 2, 3, 100, 4, 5]
n = len(arr)
print("Sum of maximum sum increasing subsequence is " +
      str(maxSumIS(arr, n)))

# This code is contributed by Bhavya Jain

```

Output:

```
Sum of maximum sum increasing subsequence is 106
```

Time Complexity:  $O(n^2)$

Source: Maximum Sum Increasing Subsequence Problem

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-14-maximum-sum-increasing-subsequence/>

## Chapter 15

# Set 15 (Longest Bitonic Subsequence)

Given an array `arr[0 ... n-1]` containing `n` positive integers, a subsequence of `arr[]` is called Bitonic if it is first increasing, then decreasing. Write a function that takes an array as argument and returns the length of the longest bitonic subsequence.

A sequence, sorted in increasing order is considered Bitonic with the decreasing part as empty. Similarly, decreasing order sequence is considered Bitonic with the increasing part as empty.

### Examples:

```
Input arr[] = {1, 11, 2, 10, 4, 5, 2, 1};
```

```
Output: 6 (A Longest Bitonic Subsequence of length 6 is 1, 2, 10, 4, 2, 1)
```

```
Input arr[] = {12, 11, 40, 5, 3, 1}
```

```
Output: 5 (A Longest Bitonic Subsequence of length 5 is 12, 11, 5, 3, 1)
```

```
Input arr[] = {80, 60, 30, 40, 20, 10}
```

```
Output: 5 (A Longest Bitonic Subsequence of length 5 is 80, 60, 30, 20, 10)
```

Source: Microsoft Interview Question

### Solution

This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. Let the input array be `arr[]` of length `n`. We need to construct two

arrays `lis[]` and `lds[]` using Dynamic Programming solution of LIS problem. `lis[i]` stores the length of the Longest Increasing subsequence ending with `arr[i]`. `lds[i]` stores the length of the longest Decreasing subsequence starting from `arr[i]`. Finally, we need to return the max value of `lis[i] + lds[i] - 1` where `i` is from 0 to `n-1`.

Following is C++ implementation of the above Dynamic Programming solution.

```
/* Dynamic Programming implementation of longest bitonic subsequence problem */
#include<stdio.h>
#include<stdlib.h>

/* lbs() returns the length of the Longest Bitonic Subsequence in
arr[] of size n. The function mainly creates two temporary arrays
lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

lis[i] ==> Longest Increasing subsequence ending with arr[i]
lds[i] ==> Longest decreasing subsequence starting with arr[i]
*/
int lbs( int arr[], int n )
{
    int i, j;

    /* Allocate memory for LIS[] and initialize LIS values as 1 for
    all indexes */
    int *lis = new int[n];
    for ( i = 0; i < n; i++ )
        lis[i] = 1;

    /* Compute LIS values from left to right */
    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;

    /* Allocate memory for lds and initialize LDS values for
    all indexes */
    int *lds = new int [n];
    for ( i = 0; i < n; i++ )
        lds[i] = 1;

    /* Compute LDS values from right to left */
    for ( i = n-2; i >= 0; i-- )
        for ( j = n-1; j > i; j-- )
            if ( arr[i] > arr[j] && lds[i] < lds[j] + 1)
```

```

        lds[i] = lds[j] + 1;

    /* Return the maximum value of lis[i] + lds[i] - 1*/
    int max = lis[0] + lds[0] - 1;
    for (i = 1; i < n; i++)
        if (lis[i] + lds[i] - 1 > max)
            max = lis[i] + lds[i] - 1;
    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LBS is %d\n", lbs( arr, n ) );

    getchar();
    return 0;
}

```

Output:

```

    Length of LBS is 7

```

Time Complexity:  $O(n^2)$

Auxiliary Space:  $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-15-longest-bitonic-subsequence/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Use of explicit keyword in C++ Set 16 (Floyd Warshall Algorithm) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 16

# Set 16 (Floyd Warshall Algorithm)

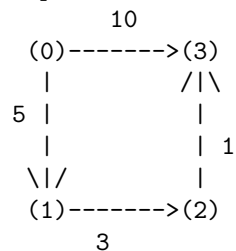
The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.

Example:

Input:

```
graph[] [] = { {0, 5, INF, 10},
                {INF, 0, 3, INF},
                {INF, INF, 0, 1},
                {INF, INF, INF, 0} }
```

which represents the following graph



Note that the value of `graph[i][j]` is 0 if `i` is equal to `j`.  
And `graph[i][j]` is INF (infinite) if there is no edge from vertex `i` to `j`.

Output:

Shortest distance matrix

```
0      5      8      9
```

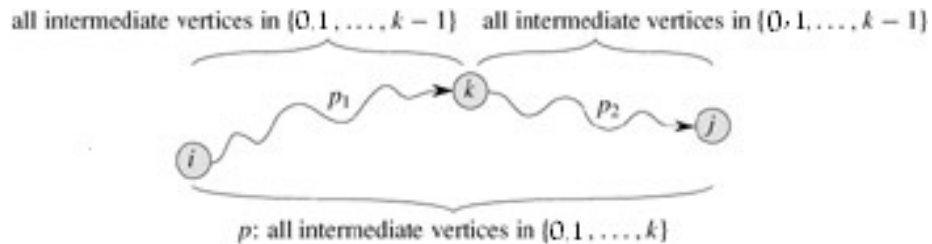
INF	0	3	4
INF	INF	0	1
INF	INF	INF	0

### Floyd Warshall Algorithm

We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number  $k$  as an intermediate vertex, we already have considered vertices  $\{0, 1, 2, \dots, k-1\}$  as intermediate vertices. For every pair  $(i, j)$  of source and destination vertices respectively, there are two possible cases.

- 1)  $k$  is not an intermediate vertex in shortest path from  $i$  to  $j$ . We keep the value of  $\text{dist}[i][j]$  as it is.
- 2)  $k$  is an intermediate vertex in shortest path from  $i$  to  $j$ . We update the value of  $\text{dist}[i][j]$  as  $\text{dist}[i][k] + \text{dist}[k][j]$ .

The following figure is taken from the Cormen book. It shows the above optimal substructure property in the all-pairs shortest path problem.



Following is implementations of the Floyd Warshall algorithm.

C/C++

```
// C Program for Floyd Warshall Algorithm
#include<stdio.h>

// Number of vertices in the graph
#define V 4

/* Define Infinite as a large enough value. This value will be used
   for vertices not connected to each other */
#define INF 99999
```

```

// A function to print the solution matrix
void printSolution(int dist[][V]);

// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
void floydWarshall (int graph[][V])
{
    /* dist[] [] will be the output matrix that will finally have the shortest
       distances between every pair of vertices */
    int dist[V][V], i, j, k;

    /* Initialize the solution matrix same as input graph matrix. Or
       we can say the initial values of shortest distances are based
       on shortest paths considering no intermediate vertex. */
    for (i = 0; i < V; i++)
        for (j = 0; j < V; j++)
            dist[i][j] = graph[i][j];

    /* Add all vertices one by one to the set of intermediate vertices.
       ---> Before start of a iteration, we have shortest distances between all
       pairs of vertices such that the shortest distances consider only the
       vertices in set {0, 1, 2, .. k-1} as intermediate vertices.
       ----> After the end of a iteration, vertex no. k is added to the set of
       intermediate vertices and the set becomes {0, 1, 2, .. k} */
    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];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}

/* A utility function to print solution */
void printSolution(int dist[][V])

```



```

{
    printf ("Following matrix shows the shortest distances"
           " between every pair of vertices \n");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++)
        {
            if (dist[i][j] == INF)
                printf ("%7s", "INF");
            else
                printf ("%7d", dist[i][j]);
        }
        printf ("\n");
    }
}

// driver program to test above function
int main()
{
    /* Let us create the following weighted graph
        10
        (0)----->(3)
        |           /\
        5 |         |
        |         | 1
        \|\        |
        (1)----->(2)
           3      */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
    };

    // Print the solution
    floydWarshell(graph);
    return 0;
}

```

## Java

```

// A Java program for Floyd Warshall All Pairs Shortest

```

```

// Path algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

class AllPairShortestPath
{
    final static int INF = 99999, V = 4;

    void floydWarshall(int graph[][])
    {
        int dist[][] = new int[V][V];
        int i, j, k;

        /* Initialize the solution matrix same as input graph matrix.
        Or we can say the initial values of shortest distances
        are based on shortest paths considering no intermediate
        vertex. */
        for (i = 0; i < V; i++)
            for (j = 0; j < V; j++)
                dist[i][j] = graph[i][j];

        /* Add all vertices one by one to the set of intermediate
        vertices.
        ---> Before start of a iteration, we have shortest
        distances between all pairs of vertices such that
        the shortest distances consider only the vertices in
        set {0, 1, 2, .. k-1} as intermediate vertices.
        ----> After the end of a iteration, vertex no. k is added
        to the set of intermediate vertices and the set
        becomes {0, 1, 2, .. k} */
        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];
                }
            }
        }
    }
}

```

```

    }
}

// Print the shortest distance matrix
printSolution(dist);
}

void printSolution(int dist[][])
{
    System.out.println("Following matrix shows the shortest "+
        "distances between every pair of vertices");
    for (int i=0; i<V; ++i)
    {
        for (int j=0; j<V; ++j)
        {
            if (dist[i][j]==INF)
                System.out.print("INF ");
            else
                System.out.print(dist[i][j]+" ");
        }
        System.out.println();
    }
}

// Driver program to test above function
public static void main (String[] args)
{
    /* Let us create the following weighted graph
    10
    (0)----->(3)
    |           /\
    5 |         |
    |         | 1
    \|\        |
    (1)----->(2)
    3           */
    int graph[][] = { {0, 5, INF, 10},
        {INF, 0, 3, INF},
        {INF, INF, 0, 1},
        {INF, INF, INF, 0}
    };

    AllPairShortestPath a = new AllPairShortestPath();

    // Print the solution
    a.floydWarshall(graph);
}

```

```

}

// Contributed by Aakash Hasija

```

Output:

Following matrix shows the shortest distances between every pair of vertices

0	5	8	9
INF	0	3	4
INF	INF	0	1
INF	INF	INF	0

Time Complexity:  $O(V^3)$

The above program only prints the shortest distances. We can modify the solution to print the shortest paths also by storing the predecessor information in a separate 2D matrix.

Also, the value of INF can be taken as INT\_MAX from limits.h to make sure that we handle maximum possible value. When we take INF as INT\_MAX, we need to change the if condition in the above program to avoid arithmetic overflow.

```

#include<limits.h>

#define INF INT_MAX
.....
if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j])
    dist[i][j] = dist[i][k] + dist[k][j];
.....

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-16-floyd-warshall-algorithm/>

## Chapter 17

# Set 17 (Palindrome Partitioning)

Given a string, a partitioning of the string is a *palindrome partitioning* if every substring of the partition is a palindrome. For example, “aba|b|bbabb|a|b|aba” is a palindrome partitioning of “ababbbabbababa”. Determine the fewest cuts needed for palindrome partitioning of a given string. For example, minimum 3 cuts are needed for “ababbbabbababa”. The three cuts are “a|babbbab|b|ababa”. If a string is palindrome, then minimum 0 cuts are needed. If a string of length  $n$  containing all different characters, then minimum  $n-1$  cuts are needed.

### Solution

This problem is a variation of Matrix Chain Multiplication problem. If the string is palindrome, then we simply return 0. Else, like the Matrix Chain Multiplication problem, we try making cuts at all possible places, recursively calculate the cost for each cut and return the minimum value.

Let the given string be `str` and `minPalPartion()` be the function that returns the fewest cuts needed for palindrome partitioning. following is the optimal substructure property.

```
// i is the starting index and j is the ending index. i must be passed as 0 and j as n-1
minPalPartion(str, i, j) = 0 if i == j. // When string is of length 1.
minPalPartion(str, i, j) = 0 if str[i..j] is palindrome.

// If none of the above conditions is true, then minPalPartion(str, i, j) can be
// calculated recursively using the following formula.
minPalPartion(str, i, j) = Min { minPalPartion(str, i, k) + 1 +
                                minPalPartion(str, k+1, j) }
                                where k varies from i to j-1
```

Following is Dynamic Programming solution. It stores the solutions to subproblems in two arrays  $P[][]$  and  $C[][]$ , and reuses the calculated values.

```
// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
    C[i][j] = Minimum number of cuts needed for palindrome partitioning
              of substring str[i..j]
    P[i][j] = true if substring str[i..j] is palindrome, else false
    Note that C[i][j] is 0 if P[i][j] is true */
    int C[n][n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
        C[i][i] = 0;
    }

    /* L is substring length. Build the solution in bottom up manner by
    considering all substrings of length starting from 2 to n.
    The loop structure is same as Matrix Chain Multiplication problem (
    See http://www.geeksforgeeks.org/archives/15553 )*/
    for (L=2; L<=n; L++)
    {
```

```

// For substring of length L, set different possible starting indexes
for (i=0; i<n-L+1; i++)
{
    j = i+L-1; // Set ending index

    // If L is 2, then we just need to compare two characters. Else
    // need to check two corner characters and value of P[i+1][j-1]
    if (L == 2)
        P[i][j] = (str[i] == str[j]);
    else
        P[i][j] = (str[i] == str[j]) && P[i+1][j-1];

    // IF str[i..j] is palindrome, then C[i][j] is 0
    if (P[i][j] == true)
        C[i][j] = 0;
    else
    {
        // Make a cut at every possible location starting from i to j,
        // and get the minimum cost cut.
        C[i][j] = INT_MAX;
        for (k=i; k<=j-1; k++)
            C[i][j] = min (C[i][j], C[i][k] + C[k+1][j]+1);
    }
}

// Return the min cut value for complete string. i.e., str[0..n-1]
return C[0][n-1];
}

// Driver program to test above function
int main()
{
    char str[] = "ababbbabbababa";
    printf("Min cuts needed for Palindrome Partitioning is %d",
        minPalPartion(str));
    return 0;
}

```

Output:

Min cuts needed for Palindrome Partitioning is 3

Time Complexity:  $O(n^3)$

#### An optimization to above approach

In above approach, we can calculating minimum cut while finding all palindromic substring. If we finding all palindromic substring 1<sup>st</sup> and then we calculate minimum cut, time complexity will reduce to  $O(n^2)$ .

Thanks for **Vivek** for suggesting this optimization.

```
// Dynamic Programming Solution for Palindrome Partitioning Problem
#include <stdio.h>
#include <string.h>
#include <limits.h>

// A utility function to get minimum of two integers
int min (int a, int b) { return (a < b)? a : b; }

// Returns the minimum number of cuts needed to partition a string
// such that every part is a palindrome
int minPalPartion(char *str)
{
    // Get the length of the string
    int n = strlen(str);

    /* Create two arrays to build the solution in bottom up manner
       C[i] = Minimum number of cuts needed for palindrome partitioning
              of substring str[0..i]
       P[i][j] = true if substring str[i..j] is palindrome, else false
       Note that C[i] is 0 if P[0][i] is true */
    int C[n];
    bool P[n][n];

    int i, j, k, L; // different looping variables

    // Every substring of length 1 is a palindrome
    for (i=0; i<n; i++)
    {
        P[i][i] = true;
    }

    /* L is substring length. Build the solution in bottom up manner by
       considering all substrings of length starting from 2 to n. */
    for (L=2; L<=n; L++)
    {
        // For substring of length L, set different possible starting indexes
        for (i=0; i<n-L+1; i++)
```



```

    {
        j = i+L-1; // Set ending index

        // If L is 2, then we just need to compare two characters. Else
        // need to check two corner characters and value of P[i+1][j-1]
        if (L == 2)
            P[i][j] = (str[i] == str[j]);
        else
            P[i][j] = (str[i] == str[j]) && P[i+1][j-1];
    }
}

for (i=0; i<n; i++)
{
    if (P[0][i] == true)
        C[i] = 0;
    else
    {
        C[i] = INT_MAX;
        for(j=0; j<i; j++)
        {
            if(P[j+1][i] == true && 1+C[j]<C[i])
                C[i]=1+C[j];
        }
    }
}

// Return the min cut value for complete string. i.e., str[0..n-1]
return C[n-1];
}

// Driver program to test above function
int main()
{
    char str[] = "ababbbabbababa";
    printf("Min cuts needed for Palindrome Partitioning is %d",
           minPalPartion(str));
    return 0;
}

```

Output:

Min cuts needed for Palindrome Partitioning is 3

Time Complexity:  $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-17-palindrome-partitioning/>

Category: Strings Tags: Dynamic Programming

Post navigation

← Operating Systems | Set 13 Operating Systems | Set 14 →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 18

# Set 18 (Partition problem)

Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Examples

```
arr[] = {1, 5, 11, 5}  
Output: true  
The array can be partitioned as {1, 5, 5} and {11}
```

```
arr[] = {1, 5, 3}  
Output: false  
The array cannot be partitioned into equal sum sets.
```

Following are the two main steps to solve this problem:

- 1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
- 2) If sum of array elements is even, calculate  $\text{sum}/2$  and find a subset of array with sum equal to  $\text{sum}/2$ .

The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

### Recursive Solution

Following is the recursive property of the second step mentioned above.

Let `isSubsetSum(arr, n, sum/2)` be the function that returns true if

there is a subset of arr[0..n-1] with sum equal to sum/2

The isSubsetSum problem can be divided into two subproblems

- a) isSubsetSum() without considering last element  
(reducing n to n-1)
- b) isSubsetSum considering the last element  
(reducing sum/2 by arr[n-1] and n to n-1)

If any of the above the above subproblems return true, then return true.

```
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||  
                             isSubsetSum (arr, n-1, sum/2 - arr[n-1])
```

```
// A recursive solution for partition problem
```

```
#include <stdio.h>
```

```
// A utility function that returns true if there is a subset of arr[]
```

```
// with sum equal to given sum
```

```
bool isSubsetSum (int arr[], int n, int sum)
```

```
{
```

```
    // Base Cases
```

```
    if (sum == 0)
```

```
        return true;
```

```
    if (n == 0 && sum != 0)
```

```
        return false;
```

```
    // If last element is greater than sum, then ignore it
```

```
    if (arr[n-1] > sum)
```

```
        return isSubsetSum (arr, n-1, sum);
```

```
    /* else, check if sum can be obtained by any of the following
```

```
        (a) including the last element
```

```
        (b) excluding the last element
```

```
    */
```

```
    return isSubsetSum (arr, n-1, sum) || isSubsetSum (arr, n-1, sum-arr[n-1]);
```

```
}
```

```
// Returns true if arr[] can be partitioned in two subsets of
```

```
// equal sum, otherwise false
```

```
bool findPartiion (int arr[], int n)
```

```
{
```

```
    // Calculate sum of the elements in array
```

```
    int sum = 0;
```

```
    for (int i = 0; i < n; i++)
```

```
        sum += arr[i];
```

```

        // If sum is odd, there cannot be two subsets with equal sum
        if (sum%2 != 0)
            return false;

        // Find if there is subset with sum equal to half of total sum
        return isSubsetSum (arr, n, sum/2);
    }

    // Driver program to test above function
    int main()
    {
        int arr[] = {3, 1, 5, 9, 12};
        int n = sizeof(arr)/sizeof(arr[0]);
        if (findPartiion(arr, n) == true)
            printf("Can be divided into two subsets of equal sum");
        else
            printf("Can not be divided into two subsets of equal sum");
        getchar();
        return 0;
    }

```

Output:

```

    Can be divided into two subsets of equal sum

```

Time Complexity:  $O(2^n)$  In worst case, this solution tries two possibilities (whether to include or exclude) for every element.

### Dynamic Programming Solution

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array `part[][]` of size  $(\text{sum}/2) * (n+1)$ . And we can construct the solution in bottom up manner such that every filled entry has following property

```

part[i][j] = true if a subset of {arr[0], arr[1], ..arr[j-1]} has sum
              equal to i, otherwise false

```

```

// A Dynamic Programming solution to partition problem
#include <stdio.h>

// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    int sum = 0;
    int i, j;

    // Caculcate sun of all elements
    for (i = 0; i < n; i++)
        sum += arr[i];

    if (sum%2 != 0)
        return false;

    bool part[sum/2+1][n+1];

    // initialize top row as true
    for (i = 0; i <= n; i++)
        part[0][i] = true;

    // initialize leftmost column, except part[0][0], as 0
    for (i = 1; i <= sum/2; i++)
        part[i][0] = false;

    // Fill the partition table in botton up manner
    for (i = 1; i <= sum/2; i++)
    {
        for (j = 1; j <= n; j++)
        {
            part[i][j] = part[i][j-1];
            if (i >= arr[j-1])
                part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
        }
    }

    /* // uncomment this part to print table
    for (i = 0; i <= sum/2; i++)
    {
        for (j = 0; j <= n; j++)
            printf ("%4d", part[i][j]);
        printf("\n");
    } */

```

```

        return part[sum/2][n];
    }

// Driver program to test above function
int main()
{
    int arr[] = {3, 1, 1, 2, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (findPartiion(arr, n) == true)
        printf("Can be divided into two subsets of equal sum");
    else
        printf("Can not be divided into two subsets of equal sum");
    getchar();
    return 0;
}

```

Output:

Can be divided into two subsets of equal sum

Following diagram shows the values in partition table. The diagram is taken from the wiki page of partition problem.

The entry  $part[i][j]$  indicates whether there is a subset of  $\{arr[0], arr[1], \dots, arr[j-1]\}$  that sums to  $i$

	{}	{3}	{3,1}	{3,1,1}	{3,1,1,2}	{3,1,1,2,2}	{3,1,1,2,2,1}
0	True	True	True	True	True	True	True
1	False	False	True	True	True	True	True
2	False	False	False	True	True	True	True
3	False	True	True	True	True	True	True
4	False	False	True	True	True	True	True
5	False	False	False	True	True	True	True

Dynamic Programming table for  
 $arr[] = \{3, 1, 1, 2, 2, 1\}$

Time Complexity:  $O(\text{sum} * n)$

Auxiliary Space:  $O(\text{sum} * n)$

Please note that this solution will not be feasible for arrays with big sum.

**References:**

[http://en.wikipedia.org/wiki/Partition\\_problem](http://en.wikipedia.org/wiki/Partition_problem)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-18-partition-problem/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Automata Theory | Set 5 Database Management Systems | Set 11 →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 19

# Set 19 (Word Wrap Problem)

Given a sequence of words, and a limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width.

The word processors like MS Word do task of placing line breaks. The idea is to have balanced lines. In other words, not have few lines with lots of extra spaces and some lines with small amount of extra spaces.

The extra spaces includes spaces put at the end of every line except the last one.

The problem is to minimize the following total cost.

Cost of a line = (Number of extra spaces in the line)<sup>3</sup>

Total Cost = Sum of costs for all lines

For example, consider the following string and line width  $M = 15$

"Geeks for Geeks presents word wrap problem"

Following is the optimized arrangement of words in 3 lines

Geeks for Geeks

presents word

wrap problem

The total extra spaces in line 1, line 2 and line 3 are 0, 2 and 3 respectively.

So optimal value of total cost is  $0 + 2*2 + 3*3 = 13$

Please note that the total cost function is not sum of extra spaces, but sum of cubes (or square is also used) of extra spaces. The idea behind this cost function is to balance the spaces among lines. For example, consider the following two arrangement of same set of words:

1) There are 3 lines. One line has 3 extra spaces and all other lines have 0 extra spaces. Total extra spaces =  $3 + 0 + 0 = 3$ . Total cost =  $3*3*3 + 0*0*0 + 0*0*0 = 27$ .

2) There are 3 lines. Each of the 3 lines has one extra space. Total extra spaces =  $1 + 1 + 1 = 3$ . Total cost =  $1*1*1 + 1*1*1 + 1*1*1 = 3$ .

Total extra spaces are 3 in both scenarios, but second arrangement should be preferred because extra spaces are balanced in all three lines. The cost function with cubic sum serves the purpose because the value of total cost in second scenario is less.

#### Method 1 (Greedy Solution)

The greedy solution is to place as many words as possible in the first line. Then do the same thing for the second line and so on until all words are placed. This solution gives optimal solution for many cases, but doesn't give optimal solution in all cases. For example, consider the following string "aaa bb cc dddd" and line width as 6. Greedy method will produce following output.

```
aaa bb
cc
dddd
```

Extra spaces in the above 3 lines are 0, 4 and 1 respectively. So total cost is  $0 + 64 + 1 = 65$ .

But the above solution is not the best solution. Following arrangement has more balanced spaces. Therefore less value of total cost function.

```
aaa
bb cc
dddd
```

Extra spaces in the above 3 lines are 3, 1 and 1 respectively. So total cost is  $27 + 1 + 1 = 29$ .

Despite being sub-optimal in some cases, the greedy approach is used by many word processors like MS Word and OpenOffice.org Writer.

### Method 2 (Dynamic Programming)

The following Dynamic approach strictly follows the algorithm given in solution of Cormen book. First we compute costs of all possible lines in a 2D table  $lc[i][j]$ . The value  $lc[i][j]$  indicates the cost to put words from  $i$  to  $j$  in a single line where  $i$  and  $j$  are indexes of words in the input sequences. If a sequence of words from  $i$  to  $j$  cannot fit in a single line, then  $lc[i][j]$  is considered infinite (to avoid it from being a part of the solution). Once we have the  $lc[i][j]$  table constructed, we can calculate total cost using following recursive formula. In the following formula,  $C[j]$  is the optimized total cost for arranging words from 1 to  $j$ .

$$c[j] = \begin{cases} 0 & \text{if } j = 0, \\ \min_{1 \leq i \leq j} (c[i - 1] + lc[i, j]) & \text{if } j > 0. \end{cases}$$

The above recursion has overlapping subproblem property. For example, the solution of subproblem  $c(2)$  is used by  $c(3)$ ,  $C(4)$  and so on. So Dynamic Programming is used to store the results of subproblems. The array  $c[]$  can be computed from left to right, since each value depends only on earlier values.

To print the output, we keep track of what words go on what lines, we can keep a parallel  $p$  array that points to where each  $c$  value came from. The last line starts at word  $p[n]$  and goes through word  $n$ . The previous line starts at word  $p[p[n]]$  and goes through word  $p[p[n]] - 1$ , etc. The function `printSolution()` uses  $p[]$  to print the solution.

In the below program, input is an array  $l[]$  that represents lengths of words in a sequence. The value  $l[i]$  indicates length of the  $i$ th word ( $i$  starts from 1) in the input sequence.

```
// A Dynamic programming solution for Word Wrap Problem
#include <limits.h>
#include <stdio.h>
#define INF INT_MAX

// A utility function to print the solution
int printSolution (int p[], int n);

// l[] represents lengths of different words in input sequence. For example,
// l[] = {3, 2, 2, 5} is for a sentence like "aaa bb cc dddd". n is size of
// l[] and M is line width (maximum no. of characters that can fit in a line)
void solveWordWrap (int l[], int n, int M)
{
    // For simplicity, 1 extra space is used in all below arrays
```

```

// extras[i][j] will have number of extra spaces if words from i
// to j are put in a single line
int extras[n+1][n+1];

// lc[i][j] will have cost of a line which has words from
// i to j
int lc[n+1][n+1];

// c[i] will have total cost of optimal arrangement of words
// from 1 to i
int c[n+1];

// p[] is used to print the solution.
int p[n+1];

int i, j;

// calculate extra spaces in a single line. The value extra[i][j]
// indicates extra spaces if words from word number i to j are
// placed in a single line
for (i = 1; i <= n; i++)
{
    extras[i][i] = M - l[i-1];
    for (j = i+1; j <= n; j++)
        extras[i][j] = extras[i][j-1] - l[j-1] - 1;
}

// Calculate line cost corresponding to the above calculated extra
// spaces. The value lc[i][j] indicates cost of putting words from
// word number i to j in a single line
for (i = 1; i <= n; i++)
{
    for (j = i; j <= n; j++)
    {
        if (extras[i][j] < 0)
            lc[i][j] = INF;
        else if (j == n && extras[i][j] >= 0)
            lc[i][j] = 0;
        else
            lc[i][j] = extras[i][j]*extras[i][j];
    }
}

// Calculate minimum cost and find minimum cost arrangement.
// The value c[j] indicates optimized cost to arrange words

```

```

        // from word number 1 to j.
        c[0] = 0;
        for (j = 1; j <= n; j++)
        {
            c[j] = INF;
            for (i = 1; i <= j; i++)
            {
                if (c[i-1] != INF && lc[i][j] != INF && (c[i-1] + lc[i][j] < c[j]))
                {
                    c[j] = c[i-1] + lc[i][j];
                    p[j] = i;
                }
            }
        }

        printSolution(p, n);
    }

int printSolution (int p[], int n)
{
    int k;
    if (p[n] == 1)
        k = 1;
    else
        k = printSolution (p, p[n]-1) + 1;
    printf ("Line number %d: From word no. %d to %d \n", k, p[n], n);
    return k;
}

// Driver program to test above functions
int main()
{
    int l[] = {3, 2, 2, 5};
    int n = sizeof(l)/sizeof(l[0]);
    int M = 6;
    solveWordWrap (l, n, M);
    return 0;
}

```

Output:

```

Line number 1: From word no. 1 to 1
Line number 2: From word no. 2 to 3

```

Line number 3: From word no. 4 to 4

Time Complexity:  $O(n^2)$

Auxiliary Space:  $O(n^2)$  The auxiliary space used in the above program can be optimized to  $O(n)$  (See the reference 2 for details)

**References:**

[http://en.wikipedia.org/wiki/Word\\_wrap](http://en.wikipedia.org/wiki/Word_wrap)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-18-word-wrap/>

## Chapter 20

# Set 20 (Maximum Length Chain of Pairs)

You are given n pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b Amazon Interview | Set 2

For example, if the given pairs are {{5, 24}, {39, 60}, {15, 28}, {27, 40}, {50, 90} }, then the longest chain that can be formed is of length 3, and the chain is {{5, 24}, {27, 40}, {50, 90}}

This problem is a variation of standard Longest Increasing Subsequence problem. Following is a simple two step process.

- 1) Sort given pairs in increasing order of first (or smaller) element.
- 2) Now run a modified LIS process where we compare the second element of already finalized LIS with the first element of new LIS being constructed.

The following code is a slight modification of method 2 of this post.

```
#include<stdio.h>
#include<stdlib.h>

// Structure for a pair
struct pair
{
    int a;
    int b;
};

// This function assumes that arr[] is sorted in increasing order
```

```

// according the first (or smaller) values in pairs.
int maxChainLength( struct pair arr[], int n)
{
    int i, j, max = 0;
    int *mcl = (int*) malloc ( sizeof( int ) * n );

    /* Initialize MCL (max chain length) values for all indexes */
    for ( i = 0; i < n; i++ )
        mcl[i] = 1;

    /* Compute optimized chain length values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i].a > arr[j].b && mcl[i] < mcl[j] + 1)
                mcl[i] = mcl[j] + 1;

    // mcl[i] now stores the maximum chain length ending with pair i

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

    /* Free memory to avoid memory leak */
    free( mcl );

    return max;
}

/* Driver program to test above function */
int main()
{
    struct pair arr[] = { {5, 24}, {15, 25},
                          {27, 40}, {50, 60} };
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of maximum size chain is %d\n",
           maxChainLength( arr, n ));
    return 0;
}

```

Output:

Length of maximum size chain is 3



Time Complexity:  $O(n^2)$  where  $n$  is the number of pairs.

The given problem is also a variation of Activity Selection problem and can be solved in  $(n \log n)$  time. To solve it as an activity selection problem, consider the first element of a pair as start time in activity selection problem, and the second element of pair as end time. Thanks to Palash for suggesting this approach.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

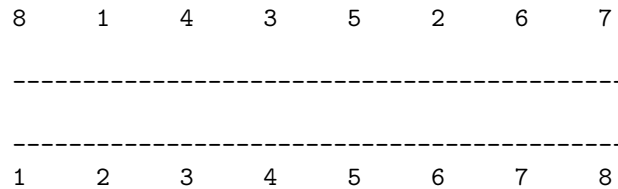
<http://www.geeksforgeeks.org/dynamic-programming-set-20-maximum-length-chain-of-pairs/>

## Chapter 21

### Set 21 (Variations of LIS)

We have discussed Dynamic Programming solution for Longest Increasing Subsequence problem in thispost and a  $O(n\text{Log}n)$  solution in thispost. Following are commonly asked variations of the standardLIS problem.

**1. Building Bridges:** Consider a 2-D map with a horizontal river passing through its center. There are  $n$  cities on the southern bank with x-coordinates  $a(1) \dots a(n)$  and  $n$  cities on the northern bank with x-coordinates  $b(1) \dots b(n)$ . You want to connect as many north-south pairs of cities as possible with bridges such that no two bridges cross. When connecting cities, you can only connect city  $i$  on the northern bank to city  $i$  on the southern bank.



Source:Dynamic Programming Practice Problems. The link also has well explained solution for the problem.

**2. Maximum Sum Increasing Subsequence:** Given an array of  $n$  positive integers. Write a program to find the maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is  $\{1, 101, 2, 3, 100, 4, 5\}$ , then output should be  $\{1, 2, 3, 100\}$ . The solution to this problem has been published here.

**3. The Longest Chain** You are given pairs of numbers. In a pair, the first number is smaller with respect to the second number. Suppose you have two sets (a, b) and (c, d), the second set can follow the first set if b < c.

**4. Box Stacking** You are given a set of n types of rectangular 3-D boxes, where the i<sup>th</sup> box has height h(i), width w(i) and depth d(i) (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.  
Source: Dynamic Programming Practice Problems. The link also has well explained solution for the problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

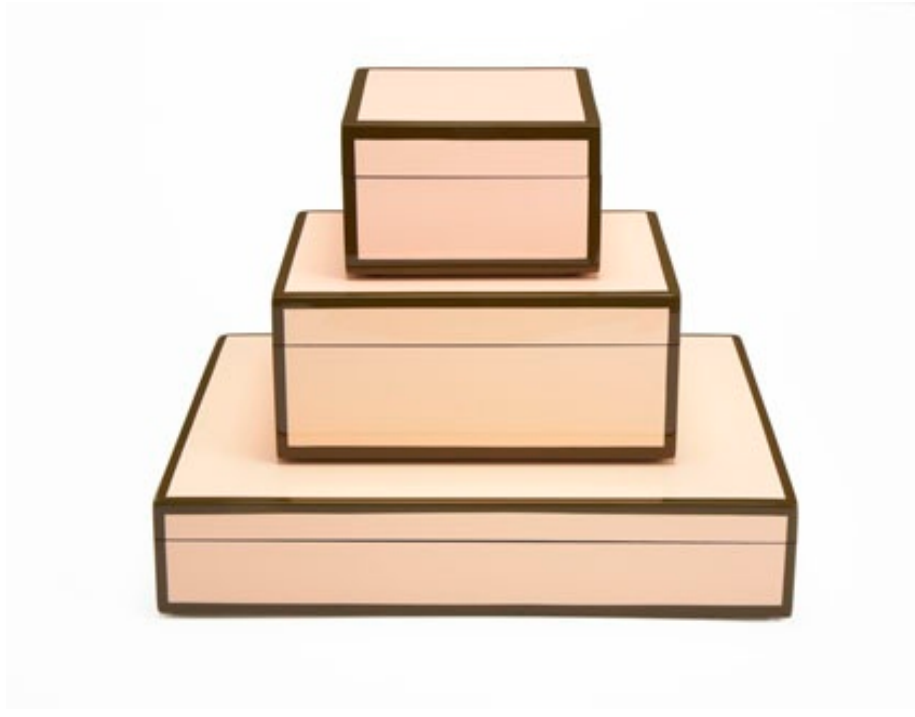
<http://www.geeksforgeeks.org/dynamic-programming-set-14-variations-of-lis/>

## Chapter 22

# Set 22 (Box Stacking Problem)

You are given a set of  $n$  types of rectangular 3-D boxes, where the  $i^{\text{th}}$  box has height  $h(i)$ , width  $w(i)$  and depth  $d(i)$  (all real numbers). You want to create a stack of boxes which is as tall as possible, but you can only stack a box on top of another box if the dimensions of the 2-D base of the lower box are each strictly larger than those of the 2-D base of the higher box. Of course, you can rotate a box so that any side functions as its base. It is also allowable to use multiple instances of the same type of box.

Source: <http://people.csail.mit.edu/bdean/6.046/dp/>. The link also has video for explanation of solution.



The Box Stacking problem is a variation of LIS problem. We need to build a maximum height stack.

Following are the key points to note in the problem statement:

- 1) A box can be placed on top of another box only if both width and depth of the upper placed box are smaller than width and depth of the lower box respectively.
- 2) We can rotate boxes. For example, if there is a box with dimensions  $\{1 \times 2 \times 3\}$  where 1 is height,  $2 \times 3$  is base, then there can be three possibilities,  $\{1 \times 2 \times 3\}$ ,  $\{2 \times 1 \times 3\}$  and  $\{3 \times 1 \times 2\}$ .
- 3) We can use multiple instances of boxes. What it means is, we can have two different rotations of a box as part of our maximum height stack.

Following is the **solution** based on DP solution of LIS problem.

- 1) Generate all 3 rotations of all boxes. The size of rotation array becomes 3 times the size of original array. For simplicity, we consider depth as always smaller than or equal to width.
- 2) Sort the above generated  $3n$  boxes in decreasing order of base area.
- 3) After sorting the boxes, the problem is same as LIS with following optimal substructure property.

$MSH(i)$  = Maximum possible Stack Height with box  $i$  at top of stack

$MSH(i) = \{ \text{Max} ( MSH(j) ) + \text{height}(i) \}$  where  $j$  width( $i$ ) and depth( $j$ ) >

depth(i).

If there is no such j then  $MSH(i) = height(i)$

4) To get overall maximum height, we return  $\max(MSH(i))$  where 0 /\* Dynamic Programming implementation of Box Stacking problem \*/ #include<stdio.h> #include<stdlib.h> /\* Representation of a box \*/ struct Box { // h -> height, w -> width, d -> depth int h, w, d; // for simplicity of solution, always keep w <= d }; // A utility function to get minimum of two integers int min (int x, int y) { return (x < y)? x : y; } // A utility function to get maximum of two integers int max (int x, int y) { return (x > y)? x : y; } /\* Following function is needed for library function qsort(). We use qsort() to sort boxes in decreasing order of base area. Refer following link for help of qsort() and compare() <http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/> \*/ int compare (const void \*a, const void \*b) { return ( (\*(Box \*)b).d \* (\*(Box \*)b).w ) - ( (\*(Box \*)a).d \* (\*(Box \*)a).w ); } /\* Returns the height of the tallest stack that can be formed with give type of boxes \*/ int maxStackHeight( Box arr[], int n ) { /\* Create an array of all rotations of given boxes For example, for a box {1, 2, 3}, we consider three instances{{1, 2, 3}, {2, 1, 3}, {3, 1, 2}} \*/ Box rot[3\*n]; int index = 0; for (int i = 0; i < n; i++) { // Copy the original box rot[index] = arr[i]; index++; // First rotation of box rot[index].h = arr[i].w; rot[index].d = max(arr[i].h, arr[i].d); rot[index].w = min(arr[i].h, arr[i].d); index++; // Second rotation of box rot[index].h = arr[i].d; rot[index].d = max(arr[i].h, arr[i].w); rot[index].w = min(arr[i].h, arr[i].w); index++; } // Now the number of boxes is 3n n = 3\*n; /\* Sort the array 'rot[]' in decreasing order, using library function for quick sort \*/ qsort (rot, n, sizeof(rot[0]), compare); // Uncomment following two lines to print all rotations // for (int i = 0; i < n; i++) // printf(“%d x %d x %d\n”, rot[i].h, rot[i].w, rot[i].d); /\* Initialize msh values for all indexes msh[i] -> Maximum possible Stack Height with box i on top \*/ int msh[n]; for (int i = 0; i < n; i++) msh[i] = rot[i].h; /\* Compute optimized msh values in bottom up manner \*/ for (int i = 1; i < n; i++) for (int j = 0; j < i; j++) if ( rot[i].w < rot[j].w && rot[i].d < rot[j].d && msh[i] < msh[j] + rot[i].h ) { msh[i] = msh[j] + rot[i].h; } /\* Pick maximum of all msh values \*/ int max = -1; for ( int i = 0; i < n; i++) if ( max < msh[i] ) max = msh[i]; return max; } /\* Driver program to test above function \*/ int main() { Box arr[] = { {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32} }; int n = sizeof(arr)/sizeof(arr[0]); printf(“The maximum possible height of stack is %d\n”, maxStackHeight (arr, n) ); return 0; }

Output:

The maximum possible height of stack is 60

In the above program, given input boxes are {4, 6, 7}, {1, 2, 3}, {4, 5, 6}, {10, 12, 32}. Following are all rotations of the boxes in decreasing order of base area.

10 x 12 x 32  
12 x 10 x 32  
32 x 10 x 12  
4 x 6 x 7  
4 x 5 x 6  
6 x 4 x 7  
5 x 4 x 6  
7 x 4 x 6  
6 x 4 x 5  
1 x 2 x 3  
2 x 1 x 3  
3 x 1 x 2

The height 60 is obtained by boxes { **3**, 1, 2}, {**1**, 2, 3}, {**6**, 4, 5}, {**4**, 5, 6}, {**4**, 6, 7}, {**32**, 10, 12}, {**10**, 12, 32}}

Time Complexity:  $O(n^2)$

Auxiliary Space:  $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-21-box-stacking-problem/>

## Chapter 23

# Set 23 (Bellman–Ford Algorithm)

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges. We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is  $O(V \log V)$  (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is  $O(VE)$ , which is more than Dijkstra.*

### Algorithm

Following are the detailed steps.

*Input:* Graph and a source vertex *src*

*Output:* Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

1) This step initializes distances from source to all vertices as infinite and distance to source itself as 0. Create an array `dist[]` of size  $|V|$  with all values as infinite except `dist[src]` where *src* is source vertex.

2) This step calculates shortest distances. Do following  $|V|-1$  times where  $|V|$  is the number of vertices in given graph.

.....a) Do following for each edge *u-v*

.....If `dist[v] > dist[u] + weight of edge uv`, then update `dist[v]`

.....`dist[v] = dist[u] + weight of edge uv`

3) This step reports if there is a negative weight cycle in graph. Do following for each edge *u-v*



.....If  $\text{dist}[v] > \text{dist}[u] + \text{weight of edge } uv$ , then “Graph contains negative weight cycle”

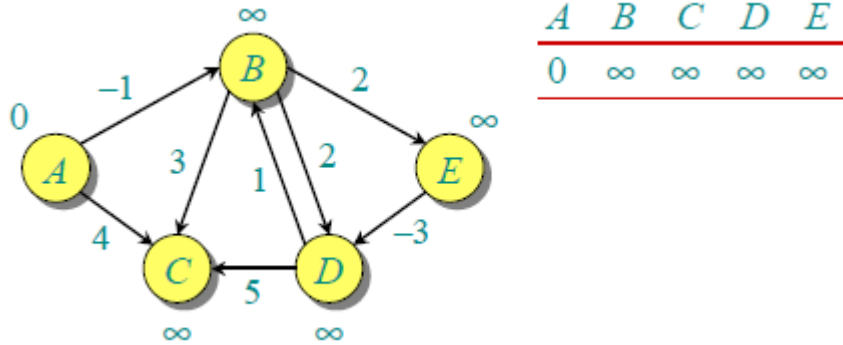
The idea of step 3 is, step 2 guarantees shortest distances if graph doesn't contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

**How does this work?** Like other Dynamic Programming Problems, the algorithm calculate shortest paths in bottom-up manner. It first calculates the shortest distances for the shortest paths which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on. After the  $i$ th iteration of outer loop, the shortest paths with at most  $i$  edges are calculated. There can be maximum  $|V| - 1$  edges in any simple path, that is why the outer loop runs  $|V| - 1$  times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most  $i$  edges, then an iteration over all edges guarantees to give shortest path with at-most  $(i+1)$  edges (Proof is simple, you can refer this or MIT Video Lecture)

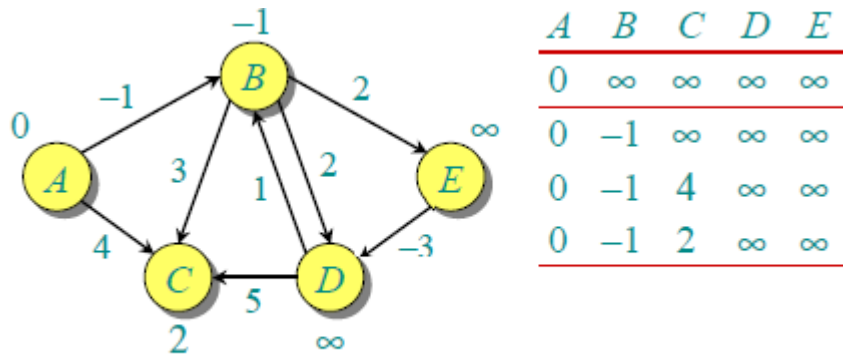
### Example

Let us understand the algorithm with following example graph. The images are taken from thissource.

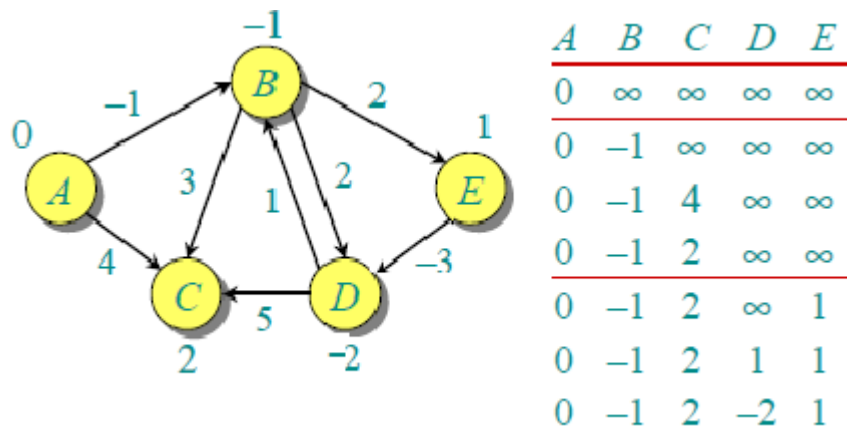
Let the given source vertex be 0. Initialize all distances as infinite, except the distance to source itself. Total number of vertices in the graph is 5, so *all edges must be processed 4 times*.



Let all edges are processed in following order: (B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D). We get following distances when all edges are processed first time. The first row in shows initial distances. The second row shows distances when edges (B,E), (D,B), (B,D) and (A,B) are processed. The third row shows distances when (A,C) is processed. The fourth row shows when (D,C), (B,C) and (E,D) are processed.



The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get following distances when all edges are processed second time (The last row shows final values).



The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

#### Implementation:

C++

```
// A C / C++ program for Bellman-Ford's single source
// shortest path algorithm.

#include <stdio.h>
```

```

#include <stdlib.h>
#include <string.h>
#include <limits.h>

// a structure to represent a weighted edge in graph
struct Edge
{
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph
{
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph =
        (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge =
        (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A utility function used to print the solution
void printArr(int dist[], int n)
{
    printf("Vertex    Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

// The main function that finds shortest distances from src to
// all other vertices using Bellman-Ford algorithm. The function
// also detects negative weight cycle

```

```

void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V-1; i++)
    {
        for (int j = 0; j < E; j++)
        {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles. The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle. If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++)
    {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
            printf("Graph contains negative weight cycle");
    }

    printArr(dist, V);

    return;
}

// Driver program to test above functions

```

```

int main()
{
    /* Let us create the graph given in above example */
    int V = 5; // Number of vertices in graph
    int E = 8; // Number of edges in graph
    struct Graph* graph = createGraph(V, E);

    // add edge 0-1 (or A-B in above figure)
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = -1;

    // add edge 0-2 (or A-C in above figure)
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 4;

    // add edge 1-2 (or B-C in above figure)
    graph->edge[2].src = 1;
    graph->edge[2].dest = 2;
    graph->edge[2].weight = 3;

    // add edge 1-3 (or B-D in above figure)
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 2;

    // add edge 1-4 (or A-E in above figure)
    graph->edge[4].src = 1;
    graph->edge[4].dest = 4;
    graph->edge[4].weight = 2;

    // add edge 3-2 (or D-C in above figure)
    graph->edge[5].src = 3;
    graph->edge[5].dest = 2;
    graph->edge[5].weight = 5;

    // add edge 3-1 (or D-B in above figure)
    graph->edge[6].src = 3;
    graph->edge[6].dest = 1;
    graph->edge[6].weight = 1;

    // add edge 4-3 (or E-D in above figure)
    graph->edge[7].src = 4;
    graph->edge[7].dest = 3;
    graph->edge[7].weight = -3;
}

```

```

        BellmanFord(graph, 0);

    return 0;
}

```

## Java

```

// A Java program for Bellman-Ford's single source shortest path
// algorithm.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a connected, directed and weighted graph
class Graph
{
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;
        Edge() {
            src = dest = weight = 0;
        }
    };

    int V, E;
    Edge edge[];

    // Creates a graph with V vertices and E edges
    Graph(int v, int e)
    {
        V = v;
        E = e;
        edge = new Edge[e];
        for (int i=0; i<e; ++i)
            edge[i] = new Edge();
    }

    // The main function that finds shortest distances from src
    // to all other vertices using Bellman-Ford algorithm. The
    // function also detects negative weight cycle
    void BellmanFord(Graph graph,int src)

```

```

{
    int V = graph.V, E = graph.E;
    int dist[] = new int[V];

    // Step 1: Initialize distances from src to all other
    // vertices as INFINITE
    for (int i=0; i<V; ++i)
        dist[i] = Integer.MAX_VALUE;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple
    // shortest path from src to any other vertex can
    // have at-most |V| - 1 edges
    for (int i=1; i<V; ++i)
    {
        for (int j=0; j<E; ++j)
        {
            int u = graph.edge[j].src;
            int v = graph.edge[j].dest;
            int weight = graph.edge[j].weight;
            if (dist[u]!=Integer.MAX_VALUE &&
                dist[u]+weight<dist[v])
                dist[v]=dist[u]+weight;
        }
    }

    // Step 3: check for negative-weight cycles. The above
    // step guarantees shortest distances if graph doesn't
    // contain negative weight cycle. If we get a shorter
    // path, then there is a cycle.
    for (int j=0; j<E; ++j)
    {
        int u = graph.edge[j].src;
        int v = graph.edge[j].dest;
        int weight = graph.edge[j].weight;
        if (dist[u]!=Integer.MAX_VALUE &&
            dist[u]+weight<dist[v])
            System.out.println("Graph contains negative weight cycle");
    }
    printArr(dist, V);
}

// A utility function used to print the solution
void printArr(int dist[], int V)
{
    System.out.println("Vertex    Distance from Source");

```

```

        for (int i=0; i<V; ++i)
            System.out.println(i+"\t\t"+dist[i]);
    }

    // Driver method to test above function
    public static void main(String[] args)
    {
        int V = 5; // Number of vertices in graph
        int E = 8; // Number of edges in graph

        Graph graph = new Graph(V, E);

        // add edge 0-1 (or A-B in above figure)
        graph.edge[0].src = 0;
        graph.edge[0].dest = 1;
        graph.edge[0].weight = -1;

        // add edge 0-2 (or A-C in above figure)
        graph.edge[1].src = 0;
        graph.edge[1].dest = 2;
        graph.edge[1].weight = 4;

        // add edge 1-2 (or B-C in above figure)
        graph.edge[2].src = 1;
        graph.edge[2].dest = 2;
        graph.edge[2].weight = 3;

        // add edge 1-3 (or B-D in above figure)
        graph.edge[3].src = 1;
        graph.edge[3].dest = 3;
        graph.edge[3].weight = 2;

        // add edge 1-4 (or A-E in above figure)
        graph.edge[4].src = 1;
        graph.edge[4].dest = 4;
        graph.edge[4].weight = 2;

        // add edge 3-2 (or D-C in above figure)
        graph.edge[5].src = 3;
        graph.edge[5].dest = 2;
        graph.edge[5].weight = 5;

        // add edge 3-1 (or D-B in above figure)
        graph.edge[6].src = 3;
        graph.edge[6].dest = 1;
        graph.edge[6].weight = 1;
    }

```



```

        // add edge 4-3 (or E-D in above figure)
        graph.edge[7].src = 4;
        graph.edge[7].dest = 3;
        graph.edge[7].weight = -3;

        graph.BellmanFord(graph, 0);
    }
}
// Contributed by Aakash Hasija

```

Output:

Vertex	Distance from Source
0	0
1	-1
2	2
3	-2
4	1

### Notes

1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.

2) Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijkstra's where we need to find minimum value of all vertices, in Bellman-Ford, edges are considered one by one.

### Exercise

1) The standard Bellman-Ford algorithm reports shortest path only if there is no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.

2) Can we use Dijkstra's algorithm for shortest paths for graphs with negative weights – one idea can be, calculate the minimum weight value, add a positive value (equal to absolute value of minimum weight value) to all weights and run the Dijkstra's algorithm for the modified graph. Will this algorithm work?

### References:

<http://www.youtube.com/watch?v=Ttezuzs39nk>

[http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford\\_algorithm](http://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm)

<http://www.cs.arizona.edu/classes/cs445/spring07/ShortestPath2.prn.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### **Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/>

## Chapter 24

# Set 24 (Optimal Binary Search Tree)

Given a sorted array  $keys[0.. n-1]$  of search keys and an array  $freq[0.. n-1]$  of frequency counts, where  $freq[i]$  is the number of searches to  $keys[i]$ . Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.

Let us first define the cost of a BST. The cost of a BST node is level of that node multiplied by its frequency. Level of root is 1.

### Example 1

Input:  $keys[] = \{10, 12\}$ ,  $freq[] = \{34, 50\}$

There can be following two possible BSTs



Frequency of searches of 10 and 12 are 34 and 50 respectively.

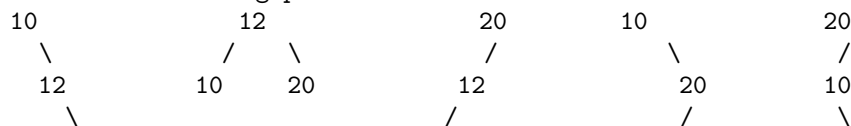
The cost of tree I is  $34*1 + 50*2 = 134$

The cost of tree II is  $50*1 + 34*2 = 118$

### Example 2

Input:  $keys[] = \{10, 12, 20\}$ ,  $freq[] = \{34, 8, 50\}$

There can be following possible BSTs



20	10	12	12
I	II	III	IV
V			

Among all possible BSTs, cost of the fifth BST is minimum.  
Cost of the fifth BST is  $1*50 + 2*34 + 3*8 = 142$

### 1) Optimal Substructure:

The optimal cost for  $\text{freq}[i..j]$  can be recursively calculated using following formula.

$$\text{optCost}(i, j) = \sum_{k=i}^j \text{freq}[k] + \min_{r=i}^j [\text{optCost}(i, r-1) + \text{optCost}(r+1, j)]$$

We need to calculate  $\text{optCost}(0, n-1)$  to find the result.

The idea of above formula is simple, we one by one try all nodes as root (r varies from i to j in second term). When we make  $r$ th node as root, we recursively calculate optimal cost from i to r-1 and r+1 to j.

We add sum of frequencies from i to j (see first term in the above formula), this is added because every search will go through root and one comparison will be done for every search.

### 2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of optimal binary search tree problem
#include <stdio.h>
#include <limits.h>

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

// A recursive function to calculate cost of optimal binary search tree
int optCost(int freq[], int i, int j)
{
    // Base cases
    if (j < i)          // If there are no elements in this subarray
        return 0;
    if (j == i)        // If there is one element in this subarray
        return freq[i];

    // Get sum of freq[i], freq[i+1], ... freq[j]
    int fsum = sum(freq, i, j);
```

```

        // Initialize minimum value
        int min = INT_MAX;

        // One by one consider all elements as root and recursively find cost
        // of the BST, compare the cost with min and update min if needed
        for (int r = i; r <= j; ++r)
        {
            int cost = optCost(freq, i, r-1) + optCost(freq, r+1, j);
            if (cost < min)
                min = cost;
        }

        // Return minimum value
        return min + fsum;
    }

// The main function that calculates minimum cost of a Binary Search Tree.
// It mainly uses optCost() to find the optimal cost.
int optimalSearchTree(int keys[], int freq[], int n)
{
    // Here array keys[] is assumed to be sorted in increasing order.
    // If keys[] is not sorted, then add code to sort keys, and rearrange
    // freq[] accordingly.
    return optCost(freq, 0, n-1);
}

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
        s += freq[k];
    return s;
}

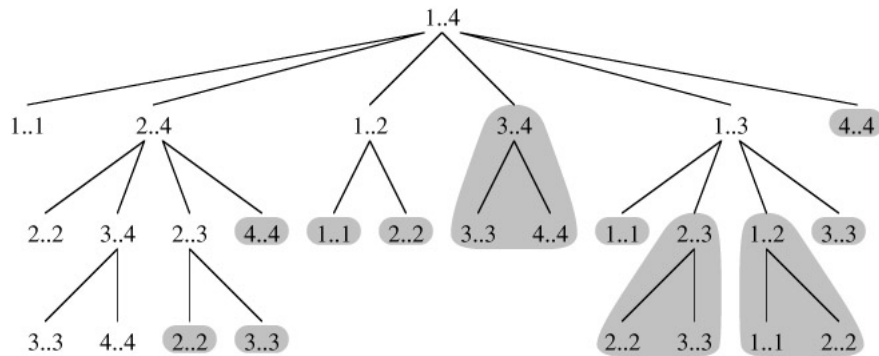
// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}

```

Output:

Cost of Optimal BST is 142

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. We can see many subproblems being repeated in the following recursion tree for `freq[1..4]`.



Since same subproblems are called again, this problem has Overlapping Subproblems property. So optimal BST problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `cost[][]` in bottom up manner.

### Dynamic Programming Solution

Following is C/C++ implementation for optimal BST problem using Dynamic Programming. We use an auxiliary array `cost[n][n]` to store the solutions of subproblems. `cost[0][n-1]` will hold the final result. The challenge in implementation is, all diagonal values must be filled first, then the values which lie on the line just above the diagonal. In other words, we must first fill all `cost[i][i]` values, then all `cost[i][i+1]` values, then all `cost[i][i+2]` values. So how to fill the 2D array in such manner? The idea used in the implementation is same as Matrix Chain Multiplication problem, we use a variable 'L' for chain length and increment 'L', one by one. We calculate column number 'j' using the values of 'i' and 'L'.

```
// Dynamic Programming code for Optimal Binary Search Tree Problem
#include <stdio.h>
#include <limits.h>
```

```

// A utility function to get sum of array elements freq[i] to freq[j]
int sum(int freq[], int i, int j);

/* A Dynamic Programming based function that calculates minimum cost of
a Binary Search Tree. */
int optimalSearchTree(int keys[], int freq[], int n)
{
    /* Create an auxiliary 2D matrix to store results of subproblems */
    int cost[n][n];

    /* cost[i][j] = Optimal cost of binary search tree that can be
    formed from keys[i] to keys[j].
    cost[0][n-1] will store the resultant cost */

    // For a single key, cost is equal to frequency of the key
    for (int i = 0; i < n; i++)
        cost[i][i] = freq[i];

    // Now we need to consider chains of length 2, 3, ... .
    // L is chain length.
    for (int L=2; L<=n; L++)
    {
        // i is row number in cost[][]
        for (int i=0; i<=n-L+1; i++)
        {
            // Get column number j from row number i and chain length L
            int j = i+L-1;
            cost[i][j] = INT_MAX;

            // Try making all keys in interval keys[i..j] as root
            for (int r=i; r<=j; r++)
            {
                // c = cost when keys[r] becomes root of this subtree
                int c = ((r > i)? cost[i][r-1]:0) +
                    ((r < j)? cost[r+1][j]:0) +
                    sum(freq, i, j);
                if (c < cost[i][j])
                    cost[i][j] = c;
            }
        }
    }
    return cost[0][n-1];
}

// A utility function to get sum of array elements freq[i] to freq[j]

```

```

int sum(int freq[], int i, int j)
{
    int s = 0;
    for (int k = i; k <=j; k++)
        s += freq[k];
    return s;
}

// Driver program to test above functions
int main()
{
    int keys[] = {10, 12, 20};
    int freq[] = {34, 8, 50};
    int n = sizeof(keys)/sizeof(keys[0]);
    printf("Cost of Optimal BST is %d ", optimalSearchTree(keys, freq, n));
    return 0;
}

```

Output:

```
Cost of Optimal BST is 142
```

### Notes

- 1) The time complexity of the above solution is  $O(n^4)$ . The time complexity can be easily reduced to  $O(n^3)$  by pre-calculating sum of frequencies instead of calling sum() again and again.
- 2) In the above solutions, we have computed optimal cost only. The solutions can be easily modified to store the structure of BSTs also. We can create another auxiliary array of size n to store the structure of tree. All we need to do is, store the chosen 'r' in the innermost loop.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

### Source

<http://www.geeksforgeeks.org/dynamic-programming-set-24-optimal-binary-search-tree/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Strand Life Sciences Interview | Set 1 Facebook Interview | Set 1 →



Writing code in comment? Please use [code.geeksforgeeks.org](https://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 25

# Set 25 (Subset Sum Problem)

Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

Examples: `set[] = {3, 34, 4, 12, 5, 2}, sum = 9`  
Output: `True` //There is a subset (4, 5) with sum 9.

Let `isSubSetSum(int set[], int n, int sum)` be the function to find whether there is a subset of `set[]` with sum equal to *sum*. *n* is the number of elements in `set[]`.

The `isSubsetSum` problem can be divided into two subproblems

...a) Include the last element, recur for  $n = n-1$ ,  $sum = sum - set[n-1]$

...b) Exclude the last element, recur for  $n = n-1$ .

If any of the above the above subproblems return true, then return true.

Following is the recursive formula for `isSubsetSum()` problem.

```
isSubsetSum(set, n, sum) = isSubsetSum(set, n-1, sum) ||  
                           isSubsetSum(arr, n-1, sum-set[n-1])
```

Base Cases:

```
isSubsetSum(set, n, sum) = false, if sum > 0 and n == 0
```

```
isSubsetSum(set, n, sum) = true, if sum == 0
```

Following is naive recursive implementation that simply follows the recursive structure mentioned above.

```
// A recursive solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sum equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0 && sum != 0)
        return false;

    // If last element is greater than sum, then ignore it
    if (set[n-1] > sum)
        return isSubsetSum(set, n-1, sum);

    /* else, check if sum can be obtained by any of the following
    (a) including the last element
    (b) excluding the last element */
    return isSubsetSum(set, n-1, sum) || isSubsetSum(set, n-1, sum-set[n-1]);
}

// Driver program to test above function
int main()
{
    int set[] = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    int n = sizeof(set)/sizeof(set[0]);
    if (isSubsetSum(set, n, sum) == true)
        printf("Found a subset with given sum");
    else
        printf("No subset with given sum");
    return 0;
}
```

Output:

Found a subset with given sum

The above solution may try all subsets of given set in worst case. Therefore time complexity of the above solution is exponential. The problem is in-fact NP-Complete (There is no known polynomial time solution for this problem).

**We can solve the problem in Pseudo-polynomial time using Dynamic programming.** We create a boolean 2D table subset[][] and fill it in bottom up manner. The value of subset[i][j] will be true if there is a subset of set[0..j-1] with sum equal to i., otherwise false. Finally, we return subset[sum][n]

```
// A Dynamic Programming solution for subset sum problem
#include <stdio.h>

// Returns true if there is a subset of set[] with sun equal to given sum
bool isSubsetSum(int set[], int n, int sum)
{
    // The value of subset[i][j] will be true if there is a subset of set[0..j-1]
    // with sum equal to i
    bool subset[sum+1][n+1];

    // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[0][i] = true;

    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
        subset[i][0] = false;

    // Fill the subset table in botton up manner
    for (int i = 1; i <= sum; i++)
    {
        for (int j = 1; j <= n; j++)
        {
            subset[i][j] = subset[i][j-1];
            if (i >= set[j-1])
                subset[i][j] = subset[i][j] || subset[i - set[j-1]][j-1];
        }
    }

    /* // uncomment this code to print table
    for (int i = 0; i <= sum; i++)
    {
        for (int j = 0; j <= n; j++)
            printf ("%4d", subset[i][j]);
        printf("\n");
    } */
```

```

        return subset[sum][n];
    }

    // Driver program to test above function
    int main()
    {
        int set[] = {3, 34, 4, 12, 5, 2};
        int sum = 9;
        int n = sizeof(set)/sizeof(set[0]);
        if (isSubsetSum(set, n, sum) == true)
            printf("Found a subset with given sum");
        else
            printf("No subset with given sum");
        return 0;
    }

```

Output:

```

    Found a subset with given sum

```

Time complexity of the above solution is  $O(\text{sum} * n)$ .

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-subset-sum-problem/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Nvidia Interview | Set 1 Measure one litre using two vessels and infinite water supply →

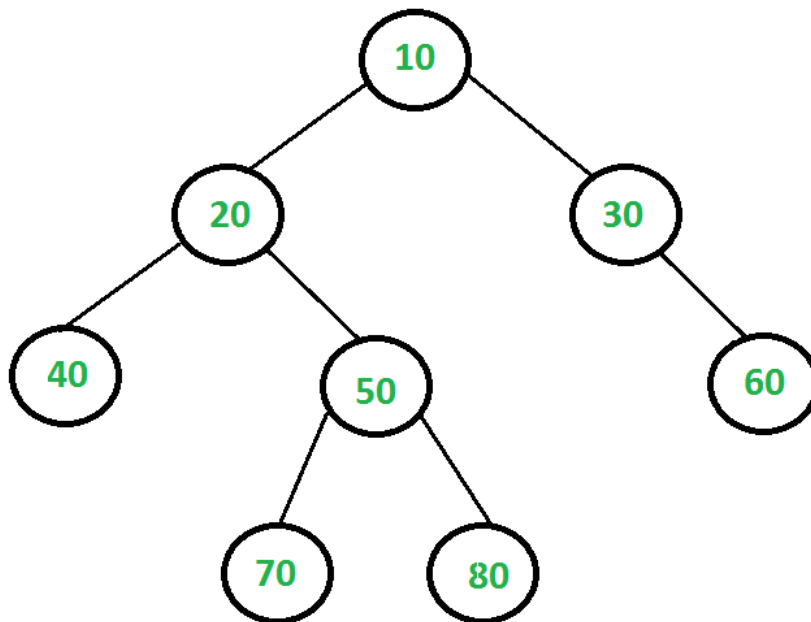
Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 26

### Set 26 (Largest Independent Set Problem)

Given a Binary Tree, find size of the **Largest Independent Set(LIS)** in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

For example, consider the following binary tree. The largest independent set(LIS) is  $\{10, 40, 60, 70, 80\}$  and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

### 1) Optimal Substructure:

Let  $LISS(X)$  indicates size of largest independent set of a tree with root X.

$$LISS(X) = \text{MAX} \{ (1 + \text{sum of LISS for all grandchildren of X}), \\ (\text{sum of LISS for all children of X}) \}$$

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of  $LISS(X)$  is 1 plus LISS of all grandchildren. If X is not a member, then the value is sum of LISS of all children.

### 2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of Largest Independent Set problem
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the largest independent set in a given
// binary tree
int LISS(struct node *root)
{
```

```

    if (root == NULL)
        return 0;

    // Caculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
    if (root->left)
        size_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        size_incl += LISS(root->right->left) + LISS(root->right->right);

    // Return the maximum of two sizes
    return max(size_incl, size_excl);
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root
        = newNode(20);
    root->left
        = newNode(8);
    root->left->left
        = newNode(4);
    root->left->right
        = newNode(12);
    root->left->right->left
        = newNode(10);
    root->left->right->right
        = newNode(14);
    root->right
        = newNode(22);
    root->right->right
        = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}

```



Output:

Size of the Largest Independent Set is 5

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So LISS problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'liss' is added to tree nodes. The initial value of 'liss' is set as 0 for all nodes. The recursive function LISS() calculates 'liss' for a node only if it is not already set.

```
/* Dynamic programming based program for Largest Independent Set problem */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    int liss;
    struct node *left, *right;
};

// A memoization function returns size of the largest independent set in
// a given binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    if (root->liss)
        return root->liss;
```

```

    if (root->left == NULL && root->right == NULL)
        return (root->liss = 1);

    // Calculate size excluding the current node
    int liss_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int liss_incl = 1;
    if (root->left)
        liss_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        liss_incl += LISS(root->right->left) + LISS(root->right->right);

    // Maximum of two sizes is LISS, store it for future uses.
    root->liss = max(liss_incl, liss_excl);

    return root->liss;
}

// A utility function to create a node
struct node* newNode(int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->liss = 0;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left              = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left    = newNode(10);
    root->left->right->right   = newNode(14);
    root->right              = newNode(22);
    root->right->right        = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}

```

```
}
```

Output

```
Size of the Largest Independent Set is 5
```

Time Complexity:  $O(n)$  where  $n$  is the number of nodes in given Binary tree.

Following extensions to above solution can be tried as an exercise.

- 1) Extend the above solution for  $n$ -ary tree.
- 2) The above solution modifies the given tree structure by adding an additional field 'liss' to tree nodes. Extend the solution so that it doesn't modify the tree structure.
- 3) The above solution only returns size of LIS, it doesn't print elements of LIS. Extend the solution to print all nodes that are part of LIS.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/largest-independent-set-problem/>

## Chapter 27

# Set 27 (Maximum sum rectangle in a 2D matrix)

Given a 2D array, find the maximum sum subarray in it. For example, in the following 2D array, the maximum sum subarray is highlighted with blue rectangle and sum of this subarray is 29.

This problem is mainly an extension of Largest Sum Contiguous Subarray for 1D array.

The **naive solution** for this problem is to check every possible rectangle in given 2D array. This solution requires 4 nested loops and time complexity of this solution would be  $O(n^4)$ .

**Kadane's algorithm** for 1D array can be used to reduce the time complexity to  $O(n^3)$ . The idea is to fix the left and right columns one by one and find the maximum sum contiguous rows for every left and right column pair. We basically find top and bottom row numbers (which have maximum sum) for every fixed left and right column pair. To find the top and bottom row numbers, calculate sum of elements in every row from left to right and store these sums in an array say temp[]. So temp[i] indicates sum of elements from left to right in row i. If we apply Kadane's 1D algorithm on temp[], and get the maximum sum subarray of temp, this maximum sum would be the maximum possible sum with left and right as boundary columns. To get the overall maximum sum, we compare this sum with the maximum sum so far.

```
// Program to find maximum sum subarray in a given 2D array
#include <stdio.h>
#include <string.h>
#include <limits.h>
```

```

#define ROW 4
#define COL 5

// Implementation of Kadane's algorithm for 1D array. The function returns the
// maximum sum and stores starting and ending indexes of the maximum sum subarray
// at addresses pointed by start and finish pointers respectively.
int kadane(int* arr, int* start, int* finish, int n)
{
    // initialize sum, maxSum and
    int sum = 0, maxSum = INT_MIN, i;

    // Just some initial value to check for all negative values case
    *finish = -1;

    // local variable
    int local_start = 0;

    for (i = 0; i < n; ++i)
    {
        sum += arr[i];
        if (sum < 0)
        {
            sum = 0;
            local_start = i+1;
        }
        else if (sum > maxSum)
        {
            maxSum = sum;
            *start = local_start;
            *finish = i;
        }
    }

    // There is at-least one non-negative number
    if (*finish != -1)
        return maxSum;

    // Special Case: When all numbers in arr[] are negative
    maxSum = arr[0];
    *start = *finish = 0;

    // Find the maximum element in array
    for (i = 1; i < n; i++)
    {
        if (arr[i] > maxSum)
        {

```

```

        maxSum = arr[i];
        *start = *finish = i;
    }
}
return maxSum;
}

// The main function that finds maximum sum rectangle in M[] []
void findMaxSum(int M[] [COL])
{
    // Variables to store the final output
    int maxSum = INT_MIN, finalLeft, finalRight, finalTop, finalBottom;

    int left, right, i;
    int temp[ROW], sum, start, finish;

    // Set the left column
    for (left = 0; left < COL; ++left)
    {
        // Initialize all elements of temp as 0
        memset(temp, 0, sizeof(temp));

        // Set the right column for the left column set by outer loop
        for (right = left; right < COL; ++right)
        {
            // Calculate sum between current left and right for every row 'i'
            for (i = 0; i < ROW; ++i)
                temp[i] += M[i][right];

            // Find the maximum sum subarray in temp[]. The kadane() function
            // also sets values of start and finish. So 'sum' is sum of
            // rectangle between (start, left) and (finish, right) which is the
            // maximum sum with boundary columns strictly as left and right.
            sum = kadane(temp, &start, &finish, ROW);

            // Compare sum with maximum sum so far. If sum is more, then update
            // maxSum and other output values
            if (sum > maxSum)
            {
                maxSum = sum;
                finalLeft = left;
                finalRight = right;
                finalTop = start;
                finalBottom = finish;
            }
        }
    }
}

```

```

    }

    // Print final values
    printf("(Top, Left) (%d, %d)\n", finalTop, finalLeft);
    printf("(Bottom, Right) (%d, %d)\n", finalBottom, finalRight);
    printf("Max sum is: %d\n", maxSum);
}

// Driver program to test above functions
int main()
{
    int M[ROW][COL] = {{1, 2, -1, -4, -20},
                        {-8, -3, 4, 2, 1},
                        {3, 8, 10, 1, 3},
                        {-4, -1, 1, 7, -6}
                       };

    findMaxSum(M);

    return 0;
}

```

Output:

```

(Top, Left) (1, 1)
(Bottom, Right) (3, 3)
Max sum is: 29

```

Time Complexity:  $O(n^3)$

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-27-max-sum-rectangle-in-a-2d-matrix/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← D E Shaw Interview | Set 1 [TopTalent.in] In Conversation With Nithin On What It Takes To Get Into Goldman Sachs →

Writing code in comment? Please use [code.geeksforgeeks.org](https://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 28

# Set 28 (Minimum insertions to form a palindrome)

Given a string, find the minimum number of characters to be inserted to convert it to palindrome.

Before we go further, let us understand with few examples:

ab: Number of insertions required is 1. **bab**

aa: Number of insertions required is 0. **aa**

abcd: Number of insertions required is 3. **dcbabcd**

abcda: Number of insertions required is 2. **adc bcda** which is same as number of insertions in the substring **bcd**(Why?).

abcde: Number of insertions required is 4. **edcbabcde**

Let the input string be *str[l.....h]*. The problem can be broken down into three parts:

1. Find the minimum number of insertions in the substring *str[l+1,.....h]*.
2. Find the minimum number of insertions in the substring *str[l.....h-1]*.
3. Find the minimum number of insertions in the substring *str[l+1.....h-1]*.

### Recursive Solution

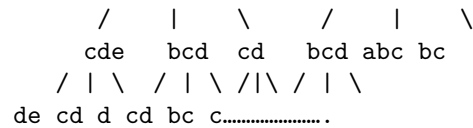
The minimum number of insertions in the string *str[l.....h]* can be given as:

*minInsertions(str[l+1.....h-1])* if *str[l]* is equal to *str[h]*

*min(minInsertions(str[l.....h-1]), minInsertions(str[l+1.....h])) + 1* otherwise

```
// A Naive recursive program to find minimum number insertions
// needed to make a string palindrome
#include <stdio.h>
#include <limits.h>
#include <string.h>
```





The substrings in bold show that the recursion to be terminated and the recursion tree cannot originate from there. Substring in the same color indicates overlapping subproblems.

*How to reuse solutions of subproblems?*

We can create a table to store results of subproblems so that they can be used directly if same subproblem is encountered again.

The below table represents the stored values for the string abcde.

	a	b	c	d	e
-----					
0	1	2	3	4	
0	0	1	2	3	
0	0	0	1	2	
0	0	0	0	1	
0	0	0	0	0	

*How to fill the table?*

The table should be filled in diagonal fashion. For the string abcde, 0....4, the following should be order in which the table is filled:

Gap = 1:  
(0, 1) (1, 2) (2, 3) (3, 4)

Gap = 2:  
(0, 2) (1, 3) (2, 4)

Gap = 3:  
(0, 3) (1, 4)

Gap = 4:  
(0, 4)

// A Dynamic Programming based program to find minimum number

```

// insertions needed to make a string palindrome
#include <stdio.h>
#include <string.h>

// A utility function to find minimum of two integers
int min(int a, int b)
{   return a < b ? a : b;   }

// A DP function to find minimum number of insertions
int findMinInsertionsDP(char str[], int n)
{
    // Create a table of size n*n. table[i][j] will store
    // minimum number of insertions needed to convert str[i..j]
    // to a palindrome.
    int table[n][n], l, h, gap;

    // Initialize all table entries as 0
    memset(table, 0, sizeof(table));

    // Fill the table
    for (gap = 1; gap < n; ++gap)
        for (l = 0, h = gap; h < n; ++l, ++h)
            table[l][h] = (str[l] == str[h])? table[l+1][h-1] :
                (min(table[l][h-1], table[l+1][h]) + 1);

    // Return minimum number of insertions for str[0..n-1]
    return table[0][n-1];
}

// Driver program to test above function.
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsDP(str, strlen(str)));
    return 0;
}

```

Output:

3

Time complexity:  $O(N^2)$   
 Auxiliary Space:  $O(N^2)$

### Another Dynamic Programming Solution (Variation of Longest Common Subsequence Problem)

The problem of finding minimum insertions can also be solved using Longest Common Subsequence (LCS) Problem. If we find out LCS of string and its reverse, we know how many maximum characters can form a palindrome. We need insert remaining characters. Following are the steps.

- 1) Find the length of LCS of input string and its reverse. Let the length be 'l'.
- 2) The minimum number insertions needed is length of input string minus 'l'.

```
// An LCS based program to find minimum number insertions needed to
// make a string palindrome
#include<stdio.h>
#include <string.h>

/* Utility function to get max of 2 integers */
int max(int a, int b)
{   return (a > b)? a : b; }

/* Returns length of LCS for X[0..m-1], Y[0..n-1].
   See http://goo.gl/bHQVP for details of this function */
int lcs( char *X, char *Y, int m, int n )
{
    int L[n+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];
}
```

```

// LCS based function to find minimum number of insertions
int findMinInsertionsLCS(char str[], int n)
{
    // Create another string to store reverse of 'str'
    char rev[n+1];
    strcpy(rev, str);
    strrev(rev);

    // The output is length of string minus length of lcs of
    // str and its reverse
    return (n - lcs(str, rev, n, n));
}

// Driver program to test above functions
int main()
{
    char str[] = "geeks";
    printf("%d", findMinInsertionsLCS(str, strlen(str)));
    return 0;
}

```

Output:

3

Time complexity of this method is also  $O(n^2)$  and this method also requires  $O(n^2)$  extra space.

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-28-minimum-insertions-to-form-a-palindrome/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Longest prefix matching – A Trie based solution in Java Set 29 (Longest Common Substring) →

Writing code in comment? Please use [code.geeksforgeeks.org](https://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 29

# Set 29 (Longest Common Substring)

Given two strings 'X' and 'Y', find the length of the longest common substring. For example, if the given strings are "GeeksforGeeks" and "GeeksQuiz", the output should be 5 as longest common substring is "Geeks"

Let m and n be the lengths of first and second strings respectively.

A **simple solution** is to one by one consider all substrings of first string and for every substring check if it is a substring in second string. Keep track of the maximum length substring. There will be  $O(m^2)$  substrings and we can find whether a string is substring on another string in  $O(n)$  time (See this). So overall time complexity of this method would be  $O(n * m^2)$

**Dynamic Programming** can be used to find the longest common substring in  $O(m*n)$  time. The idea is to find length of the longest common suffix for all substrings of both strings and store these lengths in a table.

The longest common suffix has following optimal substructure property

$$\begin{aligned} \text{LCSuff}(X, Y, m, n) &= \text{LCSuff}(X, Y, m-1, n-1) + 1 \text{ if } X[m-1] = Y[n-1] \\ &0 \text{ Otherwise (if } X[m-1] \neq Y[n-1]) \end{aligned}$$

The maximum length Longest Common Suffix is the longest common substring.

$$\text{LCSuff}(X, Y, m, n) = \text{Max}(\text{LCSuff}(X, Y, i, j)) \text{ where } 1$$

Following is C++ implementation of the above solution.

```
/* Dynamic Programming solution to find length of the longest common substring */
#include<iostream>
#include<string.h>
```



```

using namespace std;

// A utility function to find maximum of two integers
int max(int a, int b)
{   return (a > b)? a : b; }

/* Returns length of longest common substring of X[0..m-1] and Y[0..n-1] */
int LCSuffStr(char *X, char *Y, int m, int n)
{
    // Create a table to store lengths of longest common suffixes of
    // substrings. Note that LCSuff[i][j] contains length of longest
    // common suffix of X[0..i-1] and Y[0..j-1]. The first row and
    // first column entries have no logical meaning, they are used only
    // for simplicity of program
    int LCSuff[m+1][n+1];
    int result = 0; // To store length of the longest common substring

    /* Following steps build LCSuff[m+1][n+1] in bottom up fashion. */
    for (int i=0; i<=m; i++)
    {
        for (int j=0; j<=n; j++)
        {
            if (i == 0 || j == 0)
                LCSuff[i][j] = 0;

            else if (X[i-1] == Y[j-1])
            {
                LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
                result = max(result, LCSuff[i][j]);
            }
            else LCSuff[i][j] = 0;
        }
    }
    return result;
}

/* Driver program to test above function */
int main()
{
    char X[] = "OldSite:GeeksforGeeks.org";
    char Y[] = "NewSite:GeeksQuiz.com";

    int m = strlen(X);
    int n = strlen(Y);

    cout << "Length of Longest Common Substring is " << LCSuffStr(X, Y, m, n);
}

```

```
        return 0;
    }
```

Output:

```
Length of Longest Common Substring is 10
```

Time Complexity:  $O(m*n)$

Auxiliary Space:  $O(m*n)$

**References:**     [http://en.wikipedia.org/wiki/Longest\\_common\\_substring\\_problem](http://en.wikipedia.org/wiki/Longest_common_substring_problem)

The longest substring can also be solved in  $O(n+m)$  time using Suffix Tree. We will be covering Suffix Tree based solution in a separate post.

**Exercise:** The above solution prints only length of the longest common substring. Extend the solution to print the substring also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/longest-common-substring/>

Category: Strings Tags: Dynamic Programming

## Chapter 30

### Set 30 (Dice Throw)

Given  $n$  dice each with  $m$  faces, numbered from 1 to  $m$ , find the number of ways to get sum  $X$ .  $X$  is the summation of values on each face when all the dice are thrown.

The **Naive approach** is to find all the possible combinations of values from  $n$  dice and keep on counting the results that sum to  $X$ .

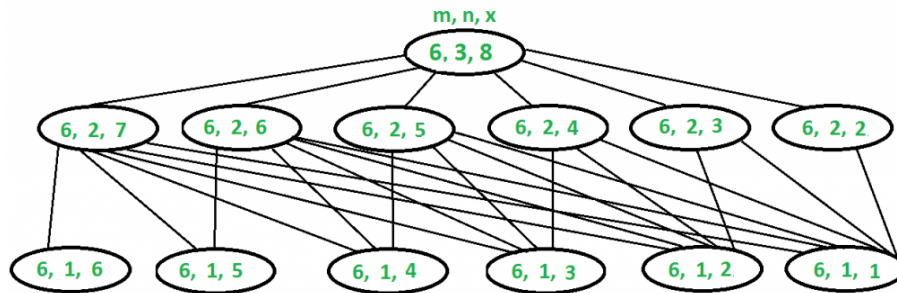
This problem can be efficiently solved using **Dynamic Programming (DP)**.

```
Let the function to find X from n dice is: Sum(m, n, X)
The function can be represented as:
Sum(m, n, X) = Finding Sum (X - 1) from (n - 1) dice plus 1 from nth dice
               + Finding Sum (X - 2) from (n - 1) dice plus 2 from nth dice
               + Finding Sum (X - 3) from (n - 1) dice plus 3 from nth dice
               .....
               .....
               .....
               + Finding Sum (X - m) from (n - 1) dice plus m from nth dice
```

```
So we can recursively write Sum(m, n, x) as following
Sum(m, n, X) = Sum(m, n - 1, X - 1) +
               Sum(m, n - 1, X - 2) +
               ..... +
               Sum(m, n - 1, X - m)
```

#### Why DP approach?

The above problem exhibits overlapping subproblems. See the below diagram. Also, see this recursive implementation. Let there be 3 dice, each with 6 faces and we need to find the number of ways to get sum 8:



$$\text{Sum}(6, 3, 8) = \text{Sum}(6, 2, 7) + \text{Sum}(6, 2, 6) + \text{Sum}(6, 2, 5) + \text{Sum}(6, 2, 4) + \text{Sum}(6, 2, 3) + \text{Sum}(6, 2, 2)$$

To evaluate  $\text{Sum}(6, 3, 8)$ , we need to evaluate  $\text{Sum}(6, 2, 7)$  which can recursively written as following:

$$\text{Sum}(6, 2, 7) = \text{Sum}(6, 1, 6) + \text{Sum}(6, 1, 5) + \text{Sum}(6, 1, 4) + \text{Sum}(6, 1, 3) + \text{Sum}(6, 1, 2) + \text{Sum}(6, 1, 1)$$

We also need to evaluate  $\text{Sum}(6, 2, 6)$  which can recursively written as following:

$$\text{Sum}(6, 2, 6) = \text{Sum}(6, 1, 5) + \text{Sum}(6, 1, 4) + \text{Sum}(6, 1, 3) + \text{Sum}(6, 1, 2) + \text{Sum}(6, 1, 1)$$

.....  
 .....

$$\text{Sum}(6, 2, 2) = \text{Sum}(6, 1, 1)$$

Please take a closer look at the above recursion. The sub-problems in RED are solved first time and sub-problems in BLUE are solved again (exhibit overlapping sub-problems). Hence, storing the results of the solved sub-problems saves time.

Following is C++ implementation of Dynamic Programming approach.

```
// C++ program to find number of ways to get sum 'x' with 'n'
// dice where every dice has 'm' faces
#include <iostream>
#include <string.h>
using namespace std;

// The main function that returns number of ways to get sum 'x'
// with 'n' dice and 'm' with m faces.
int findWays(int m, int n, int x)
```

```

{
    // Create a table to store results of subproblems. One extra
    // row and column are used for simplicity (Number of dice
    // is directly used as row index and sum is directly used
    // as column index). The entries in 0th row and 0th column
    // are never used.
    int table[n + 1][x + 1];
    memset(table, 0, sizeof(table)); // Initialize all entries as 0

    // Table entries for only one dice
    for (int j = 1; j <= m && j <= x; j++)
        table[1][j] = 1;

    // Fill rest of the entries in table using recursive relation
    // i: number of dice, j: sum
    for (int i = 2; i <= n; i++)
        for (int j = 1; j <= x; j++)
            for (int k = 1; k <= m && k < j; k++)
                table[i][j] += table[i-1][j-k];

    /* Uncomment these lines to see content of table
    for (int i = 0; i <= n; i++)
    {
        for (int j = 0; j <= x; j++)
            cout << table[i][j] << " ";
        cout << endl;
    } */
    return table[n][x];
}

// Driver program to test above functions
int main()
{
    cout << findWays(4, 2, 1) << endl;
    cout << findWays(2, 2, 3) << endl;
    cout << findWays(6, 3, 8) << endl;
    cout << findWays(4, 2, 5) << endl;
    cout << findWays(4, 3, 5) << endl;

    return 0;
}

```

Output:

0  
2  
21  
4  
6

**Time Complexity:**  $O(m * n * x)$  where  $m$  is number of faces,  $n$  is number of dice and  $x$  is given sum.

We can add following two conditions at the beginning of `findWays()` to improve performance of program for extreme cases ( $x$  is too high or  $x$  is too low)

```
// When x is so high that sum can not go beyond x even when we
// get maximum value in every dice throw.
if (m*n <= x)
    return (m*n == x);

// When x is too low
if (n >= x)
    return (n == x);
```

With above conditions added, time complexity becomes  $O(1)$  when  $x \geq m*n$  or when  $x$  Exercise:

Extend the above algorithm to find the probability to get  $\text{Sum} > X$ .

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/dice-throw-problem/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Biconnected graph [TopTalent.in] Interview With Nandini from VNIT Who Bagged an Off Campus Job in Microsoft →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 31

# Set 31 (Optimal Strategy for a Game)

Problem statement: Consider a row of  $n$  coins of values  $v_1 \dots v_n$ , where  $n$  is even. We play a game against an opponent by alternating turns. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. Determine the maximum possible amount of money we can definitely win if we move first.

Note: The opponent is as clever as the user.

Let us understand the problem with few examples:

1. 5, 3, 7, 10 : The user collects maximum value as  $15(10 + 5)$
2. 8, 15, 3, 7 : The user collects maximum value as  $22(7 + 15)$

Does choosing the best at each move give an optimal solution?

No. In the second example, this is how the game can finish:

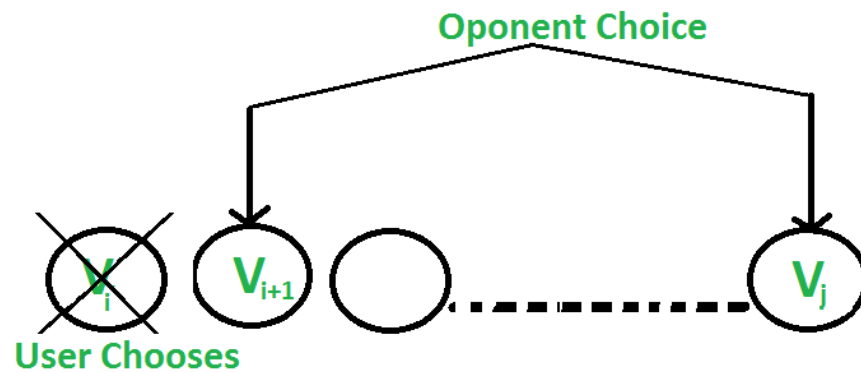
1.  
.....User chooses 8.  
.....Opponent chooses 15.  
.....User chooses 7.  
.....Opponent chooses 3.  
Total value collected by user is  $15(8 + 7)$
2.  
.....User chooses 7.  
.....Opponent chooses 8.  
.....User chooses 15.  
.....Opponent chooses 3.  
Total value collected by user is  $22(7 + 15)$

So if the user follows the second game state, maximum value can be collected although the first move is not the best.

There are two choices:

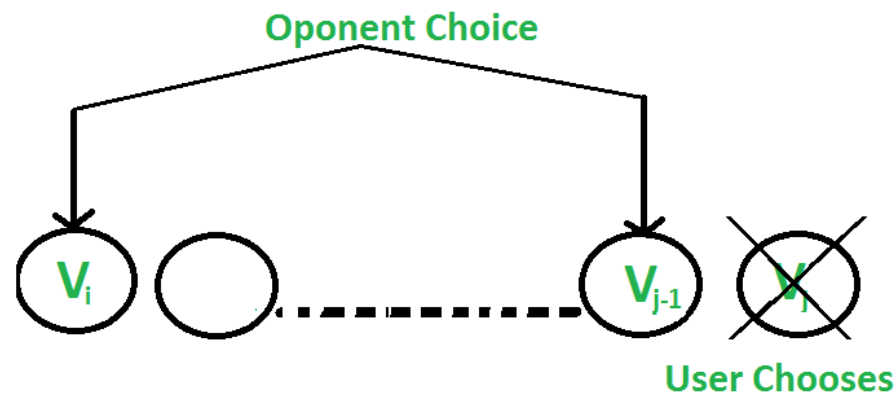
1. The user chooses the  $i$ th coin with value  $V_i$ : The opponent either chooses  $(i+1)$ th coin or  $j$ th coin. The opponent intends to choose the coin which leaves the user with minimum value.

i.e. The user can collect the value  $V_i + \min(F(i+2, j), F(i+1, j-1))$



2. The user chooses the  $j$ th coin with value  $V_j$ : The opponent either chooses  $i$ th coin or  $(j-1)$ th coin. The opponent intends to choose the coin which leaves the user with minimum value.

i.e. The user can collect the value  $V_j + \min(F(i+1, j-1), F(i, j-2))$



Following is recursive solution that is based on above two choices. We take the maximum of two choices.



$F(i, j)$  represents the maximum value the user can collect from  $i$ 'th coin to  $j$ 'th coin.

$$F(i, j) = \text{Max}(V_i + \min(F(i+2, j), F(i+1, j-1)), V_j + \min(F(i+1, j-1), F(i, j-2)))$$

Base Cases

$$F(i, j) = V_i \quad \text{If } j == i$$

$$F(i, j) = \max(V_i, V_j) \quad \text{If } j == i+1$$

### Why Dynamic Programming?

The above relation exhibits overlapping sub-problems. In the above relation,  $F(i+1, j-1)$  is calculated twice.

```
// C program to find out maximum value from a given sequence of coins
#include <stdio.h>
#include <limits.h>

// Utility functions to get maximum and minimum of two integers
int max(int a, int b) { return a > b ? a : b; }
int min(int a, int b) { return a < b ? a : b; }

// Returns optimal value possible that a player can collect from
// an array of coins of size n. Note that n must be even
int optimalStrategyOfGame(int* arr, int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, x, y, z;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap)
    {
        for (i = 0, j = gap; j < n; ++i, ++j)
        {
            // Here x is value of F(i+2, j), y is F(i+1, j-1) and
            // z is F(i, j-2) in above recursive formula
            x = ((i+2) <= j) ? table[i+2][j] : 0;
            y = ((i+1) <= (j-1)) ? table[i+1][j-1] : 0;
            z = (i <= (j-2)) ? table[i][j-2] : 0;

            table[i][j] = max(arr[i] + min(x, y), arr[j] + min(y, z));
        }
    }
}
```

```

    }

    return table[0][n-1];
}

// Driver program to test above function
int main()
{
    int arr1[] = {8, 15, 3, 7};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    printf("%d\n", optimalStrategyOfGame(arr1, n));

    int arr2[] = {2, 2, 2, 2};
    n = sizeof(arr2)/sizeof(arr2[0]);
    printf("%d\n", optimalStrategyOfGame(arr2, n));

    int arr3[] = {20, 30, 2, 2, 2, 10};
    n = sizeof(arr3)/sizeof(arr3[0]);
    printf("%d\n", optimalStrategyOfGame(arr3, n));

    return 0;
}

```

Output:

```

22
4
42

```

### Exercise

Your thoughts on the strategy when the user wishes to only win instead of winning with the maximum value. Like above problem, number of coins is even. Can Greedy approach work quite well and give an optimal solution? Will your answer change if number of coins is odd?

This article is compiled by Aashish Barnwal. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/dynamic-programming-set-31-optimal-strategy-for-a-game/>

## Chapter 32

# Set 32 (Word Break Problem)

Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words. See following examples for more details.

This is a famous Google interview question, also being asked by many other companies now a days.

```
Consider the following dictionary
{ i, like, sam, sung, samsung, mobile, ice,
  cream, icecream, man, go, mango}
```

```
Input:  ilike
```

```
Output: Yes
```

```
The string can be segmented as "i like".
```

```
Input:  ilikesamsung
```

```
Output: Yes
```

```
The string can be segmented as "i like samsung" or "i like sam sung".
```

### **Recursive implementation:**

The idea is simple, we consider each prefix and search it in dictionary. If the prefix is present in dictionary, we recur for rest of the string (or suffix). If the recursive call for suffix returns true, we return true, otherwise we try next prefix. If we have tried all prefixes and none of them resulted in a solution, we return false.

We strongly recommend to see **substr** function which is used extensively in following implementations.

```
// A recursive program to test whether a given string can be segmented into
// space separated words in dictionary
#include <iostream>
using namespace std;

/* A utility function to check whether a word is present in dictionary or not.
An array of strings is used for dictionary. Using array of strings for
dictionary is definitely not a good idea. We have used for simplicity of
the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile", "samsung", "sam", "sung", "man", "mango",
                           "icecream", "and", "go", "i", "like", "ice", "cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}

// returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();

    // Base case
    if (size == 0) return true;

    // Try all prefixes of lengths from 1 to size
    for (int i=1; i<=size; i++)
    {
        // The parameter for dictionaryContains is str.substr(0, i)
        // str.substr(0, i) which is prefix (of input string) of
        // length 'i'. We first check whether current prefix is in
        // dictionary. Then we recursively check for remaining string
        // str.substr(i, size-i) which is suffix of length size-i
        if (dictionaryContains( str.substr(0, i) ) &&
            wordBreak( str.substr(i, size-i) ))
            return true;
    }
}
```

```

        // If we have tried all prefixes and none of them worked
        return false;
    }

    // Driver program to test above functions
    int main()
    {
        wordBreak("ilikesamsung"? cout <<"Yes\n": cout << "No\n";
        wordBreak("iiiiiiii"? cout <<"Yes\n": cout << "No\n";
        wordBreak(")? cout <<"Yes\n": cout << "No\n";
        wordBreak("ilikelikeymangoiii"? cout <<"Yes\n": cout << "No\n";
        wordBreak("samsungandmango"? cout <<"Yes\n": cout << "No\n";
        wordBreak("samsungandmangok"? cout <<"Yes\n": cout << "No\n";
        return 0;
    }

```

Output:

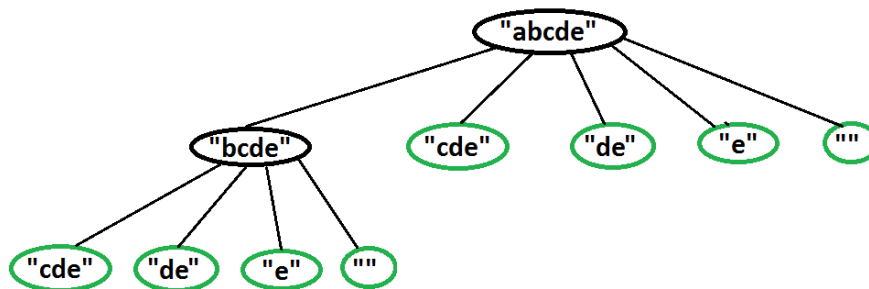
```

    Yes
    Yes
    Yes
    Yes
    Yes
    No

```

### Dynamic Programming

Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string “abcde” in worst case.



Partial recursion tree for input string “abcde”. The subproblems encircled with green color are overlapping subproblems

```

// A Dynamic Programming based program to test whether a given string can
// be segmented into space separated words in dictionary
#include <iostream>
#include <string.h>
using namespace std;

/* A utility function to check whether a word is present in dictionary or not.
An array of strings is used for dictionary. Using array of strings for
dictionary is definitely not a good idea. We have used for simplicity of
the program*/
int dictionaryContains(string word)
{
    string dictionary[] = {"mobile", "samsung", "sam", "sung", "man", "mango",
                           "icecream", "and", "go", "i", "like", "ice", "cream"};
    int size = sizeof(dictionary)/sizeof(dictionary[0]);
    for (int i = 0; i < size; i++)
        if (dictionary[i].compare(word) == 0)
            return true;
    return false;
}

// Returns true if string can be segmented into space separated
// words, otherwise returns false
bool wordBreak(string str)
{
    int size = str.size();
    if (size == 0)    return true;

    // Create the DP table to store results of subproblems. The value wb[i]
    // will be true if str[0..i-1] can be segmented into dictionary words,
    // otherwise false.
    bool wb[size+1];
    memset(wb, 0, sizeof(wb)); // Initialize all values as false.

    for (int i=1; i<=size; i++)
    {
        // if wb[i] is false, then check if current prefix can make it true.
        // Current prefix is "str.substr(0, i)"
        if (wb[i] == false && dictionaryContains( str.substr(0, i) ))
            wb[i] = true;

        // wb[i] is true, then check for all substrings starting from
        // (i+1)th character and store their results.
        if (wb[i] == true)
        {

```

```

        // If we reached the last prefix
        if (i == size)
            return true;

        for (int j = i+1; j <= size; j++)
        {
            // Update wb[j] if it is false and can be updated
            // Note the parameter passed to dictionaryContains() is
            // substring starting from index 'i' and length 'j-i'
            if (wb[j] == false && dictionaryContains( str.substr(i, j-i) ))
                wb[j] = true;

            // If we reached the last character
            if (j == size && wb[j] == true)
                return true;
        }
    }
}

/* Uncomment these lines to print DP table "wb[]"
for (int i = 1; i <= size; i++)
    cout << " " << wb[i]; */

// If we have tried all prefixes and none of them worked
return false;
}

// Driver program to test above functions
int main()
{
    wordBreak("ilikesamsung")? cout <<"Yes\n": cout << "No\n";
    wordBreak("iiiiiiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("")? cout <<"Yes\n": cout << "No\n";
    wordBreak("ilikelikeimangoiii")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmango")? cout <<"Yes\n": cout << "No\n";
    wordBreak("samsungandmangok")? cout <<"Yes\n": cout << "No\n";
    return 0;
}

```

Output:

Yes

Yes  
Yes  
Yes  
Yes  
No

**Exercise:**

The above solutions only find out whether a given string can be segmented or not. Extend the above Dynamic Programming solution to print all possible partitions of input string. See extended recursive solution for reference.

Examples:

```
Input: ilikeicecreamandmango
Output:
i like ice cream and man go
i like ice cream and mango
i like icecream and man go
i like icecream and mango
```

```
Input: ilikesamsungmobile
Output:
i like sam sung mobile
i like samsung mobile
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-32-word-break-problem/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Amazon Interview | Set 33 Custom Tree Problem →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 33

# Set 33 (Find if a string is interleaved of two other strings)

Given three strings A, B and C. Write a function that checks whether C is an interleaving of A and B. C is said to be interleaving A and B, if it contains all characters of A and B and order of all characters in individual strings is preserved.

We have discussed a simple solution of this problem here. The simple solution doesn't work if strings A and B have some common characters. For example A = "XXY", string B = "XXZ" and string C = "XXZXXXY". To handle all cases, two possibilities need to be considered.

- a) If first character of C matches with first character of A, we move one character ahead in A and C and recursively check.
- b) If first character of C matches with first character of B, we move one character ahead in B and C and recursively check.

If any of the above two cases is true, we return true, else false. Following is simple recursive implementation of this approach (Thanks to Frederic for suggesting this)

```
// A simple recursive function to check whether C is an interleaving of A and B
bool isInterleaved(char *A, char *B, char *C)
{
    // Base Case: If all strings are empty
    if (!(*A || *B || *C))
```

```

        return true;

    // If C is empty and any of the two strings is not empty
    if (*C == '\0')
        return false;

    // If any of the above mentioned two possibilities is true,
    // then return true, otherwise false
    return ( (*C == *A) && isInterleaved(A+1, B, C+1))
        || ((*C == *B) && isInterleaved(A, B+1, C+1));
}

```

### Dynamic Programming

The worst case time complexity of recursive solution is  $O(2^n)$ . The above recursive solution certainly has many overlapping subproblems. For example, if we consider  $A = \text{"XXX"}$ ,  $B = \text{"XXX"}$  and  $C = \text{"XXXXXX"}$  and draw recursion tree, there will be many overlapping subproblems.

Therefore, like other typical Dynamic Programming problems, we can solve it by creating a table and store results of subproblems in bottom up manner. Thanks to Abhinav Ramana for suggesting this method and implementation.

```

// A Dynamic Programming based program to check whether a string C is
// an interleaving of two other strings A and B.
#include <iostream>
#include <string.h>
using namespace std;

// The main function that returns true if C is
// an interleaving of A and B, otherwise false.
bool isInterleaved(char* A, char* B, char* C)
{
    // Find lengths of the two strings
    int M = strlen(A), N = strlen(B);

    // Let us create a 2D table to store solutions of
    // subproblems. C[i][j] will be true if C[0..i+j-1]
    // is an interleaving of A[0..i-1] and B[0..j-1].
    bool IL[M+1][N+1];

    memset(IL, 0, sizeof(IL)); // Initialize all values as false.

    // C can be an interleaving of A and B only if sum

```

```

// of lengths of A & B is equal to length of C.
if ((M+N) != strlen(C))
    return false;

// Process all characters of A and B
for (int i=0; i<=M; ++i)
{
    for (int j=0; j<=N; ++j)
    {
        // two empty strings have an empty string
        // as interleaving
        if (i==0 && j==0)
            IL[i][j] = true;

        // A is empty
        else if (i==0 && B[j-1]==C[j-1])
            IL[i][j] = IL[i][j-1];

        // B is empty
        else if (j==0 && A[i-1]==C[i-1])
            IL[i][j] = IL[i-1][j];

        // Current character of C matches with current character of A,
        // but doesn't match with current character of B
        else if (A[i-1]==C[i+j-1] && B[j-1]!=C[i+j-1])
            IL[i][j] = IL[i-1][j];

        // Current character of C matches with current character of B,
        // but doesn't match with current character of A
        else if (A[i-1]!=C[i+j-1] && B[j-1]==C[i+j-1])
            IL[i][j] = IL[i][j-1];

        // Current character of C matches with that of both A and B
        else if (A[i-1]==C[i+j-1] && B[j-1]==C[i+j-1])
            IL[i][j]=(IL[i-1][j] || IL[i][j-1]) ;
    }
}

return IL[M][N];
}

// A function to run test cases
void test(char *A, char *B, char *C)
{
    if (isInterleaved(A, B, C))
        cout << C << " is interleaved of " << A << " and " << B << endl;
}

```

```

        else
            cout << C << " is not interleaved of " << A << " and " << B << endl;
    }

// Driver program to test above functions
int main()
{
    test("XXY", "XXZ", "XXZXXXY");
    test("XY" , "WZ" , "WZXY");
    test ("XY", "X", "XXY");
    test ("YX", "X", "XXY");
    test ("XXY", "XXZ", "XXXXZY");
    return 0;
}

```

Output:

```

    XXZXXXY is not interleaved of XXY and XXZ
    WZXY is interleaved of XY and WZ
    XXY is interleaved of XY and X
    XXY is not interleaved of YX and X
    XXXXZY is interleaved of XXY and XXZ

```

See this for more test cases.

Time Complexity:  $O(MN)$

Auxiliary Space:  $O(MN)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

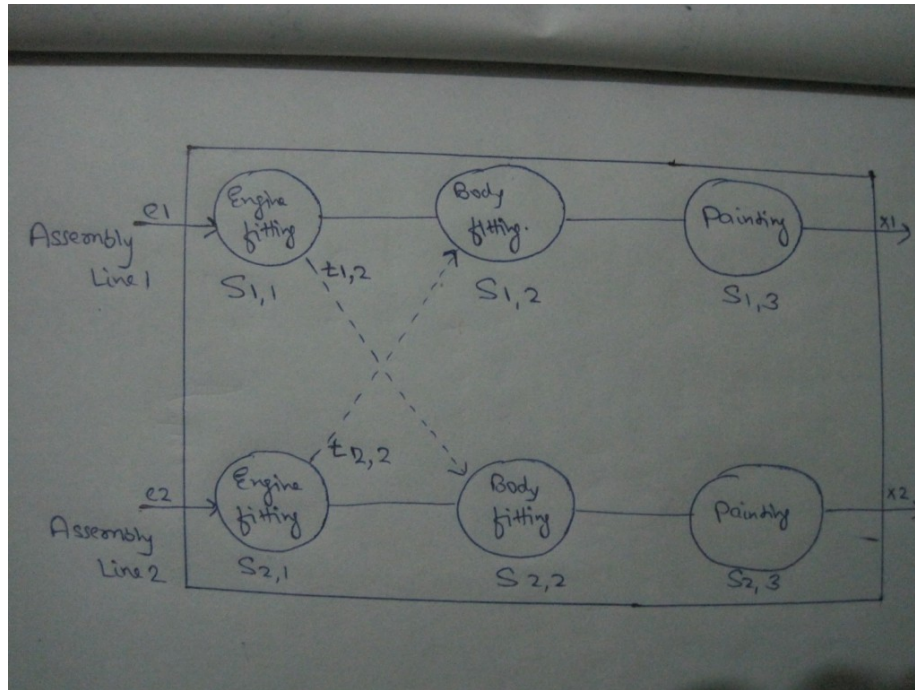
<http://www.geeksforgeeks.org/check-whether-a-given-string-is-an-interleaving-of-two-other-given-strings-set-1/>

## Chapter 34

### Set 34 (Assembly Line Scheduling)

A car factory has two assembly lines, each with  $n$  stations. A station is denoted by  $S_{i,j}$  where  $i$  is either 1 or 2 and indicates the assembly line the station is on, and  $j$  indicates the number of the station. The time taken per station is denoted by  $a_{i,j}$ . Each station is dedicated to some sort of work like engine fitting, body fitting, painting and so on. So, a car chassis must pass through each of the  $n$  stations in order before exiting the factory. The parallel stations of the two assembly lines perform the same task. After it passes through station  $S_{i,j}$ , it will continue to station  $S_{i,j+1}$  unless it decides to transfer to the other line. Continuing on the same line incurs no extra cost, but transferring from line  $i$  at station  $j - 1$  to station  $j$  on the other line takes time  $t_{i,j}$ . Each assembly line takes an entry time  $e_i$  and exit time  $x_i$  which may be different for the two lines. Give an algorithm for computing the minimum time it will take to build a car chassis.

The below figure presents the problem in a clear picture:



The following information can be extracted from the problem statement to make it simpler:

- Two assembly lines, 1 and 2, each with stations from 1 to  $n$ .
- A car chassis must pass through all stations from 1 to  $n$  in order (in any of the two assembly lines). i.e. it cannot jump from station  $i$  to station  $j$  if they are not at one move distance.
- The car chassis can move one station forward in the same line, or one station diagonally in the other line. It incurs an extra cost  $t_{i,j}$  to move to station  $j$  from line  $i$ . No cost is incurred for movement in same line.
- The time taken in station  $j$  on line  $i$  is  $a_{i,j}$ .
- $S_{i,j}$  represents a station  $j$  on line  $i$ .

#### Breaking the problem into smaller sub-problems:

We can easily find the  $i$ th factorial if  $(i-1)$ th factorial is known. Can we apply the similar funda here?

If the minimum time taken by the chassis to leave station  $S_{i,j-1}$  is known, the minimum time taken to leave station  $S_{i,j}$  can be calculated quickly by combining  $a_{i,j}$  and  $t_{i,j}$ .

**T1(j)** indicates the minimum time taken by the car chassis to leave station  $j$  on assembly line 1.

**T2(j)** indicates the minimum time taken by the car chassis to leave station  $j$  on assembly line 2.

**Base cases:**

The entry time  $e_i$  comes into picture only when the car chassis enters the car factory.

Time taken to leave first station in line 1 is given by:

$$T1(1) = \text{Entry time in Line 1} + \text{Time spent in station } S_{1,1}$$

$$T1(1) = e_1 + a_{1,1}$$

Similarly, time taken to leave first station in line 2 is given by:

$$T2(1) = e_2 + a_{2,1}$$

**Recursive Relations:**

If we look at the problem statement, it quickly boils down to the below observations:

The car chassis at station  $S_{1,j}$  can come either from station  $S_{1,j-1}$  or station  $S_{2,j-1}$ .

Case #1: Its previous station is  $S_{1,j-1}$

The minimum time to leave station  $S_{1,j}$  is given by:

$$T1(j) = \text{Minimum time taken to leave station } S_{1,j-1} + \text{Time spent in station } S_{1,j}$$

$$T1(j) = T1(j-1) + a_{1,j}$$

Case #2: Its previous station is  $S_{2,j-1}$

The minimum time to leave station  $S_{1,j}$  is given by:

$$T1(j) = \text{Minimum time taken to leave station } S_{2,j-1} + \text{Extra cost incurred to change the assembly line} + \text{Time spent in station } S_{1,j}$$

$$T1(j) = T2(j-1) + t_{2,j} + a_{1,j}$$

The minimum time  $T1(j)$  is given by the minimum of the two obtained in cases #1 and #2.

$$T1(j) = \min((T1(j-1) + a_{1,j}), (T2(j-1) + t_{2,j} + a_{1,j}))$$

Similarly the minimum time to reach station  $S_{2,j}$  is given by:

$$T2(j) = \min((T2(j-1) + a_{2,j}), (T1(j-1) + t_{1,j} + a_{2,j}))$$

The total minimum time taken by the car chassis to come out of the factory is given by:

$$Tmin = \min(\text{Time taken to leave station } S_{1,n} + \text{Time taken to exit the car factory})$$

$$Tmin = \min(T1(n) + x_1, T2(n) + x_2)$$

**Why dynamic programming?**

The above recursion exhibits overlapping sub-problems. There are two ways to reach station  $S_{1,j}$ :

1. From station  $S_{1,j-1}$
2. From station  $S_{2,j-1}$

So, to find the minimum time to leave station  $S_{1,j}$  the minimum time to leave the previous two stations must be calculated(as explained in above recursion). Similarly, there are two ways to reach station  $S_{2,j}$ :

1. From station  $S_{2, j-1}$
2. From station  $S_{1, j-1}$

Please note that the minimum times to leave stations  $S_{1, j-1}$  and  $S_{2, j-1}$  have already been calculated.

So, we need two tables to store the partial results calculated for each station in an assembly line. The table will be filled in bottom-up fashion.

**Note:**

In this post, the word “leave” has been used in place of “reach” to avoid the confusion. Since the car chassis must spend a fixed time in each station, the word leave suits better.

**Implementation:**

```
// A C program to find minimum possible time by the car chassis to complete
#include <stdio.h>
#define NUM_LINE 2
#define NUM_STATION 4

// Utility function to find minimum of two numbers
int min(int a, int b) { return a < b ? a : b; }

int carAssembly(int a[][NUM_STATION], int t[][NUM_STATION], int *e, int *x)
{
    int T1[NUM_STATION], T2[NUM_STATION], i;

    T1[0] = e[0] + a[0][0]; // time taken to leave first station in line 1
    T2[0] = e[1] + a[1][0]; // time taken to leave first station in line 2

    // Fill tables T1[] and T2[] using the above given recursive relations
    for (i = 1; i < NUM_STATION; ++i)
    {
        T1[i] = min(T1[i-1] + a[0][i], T2[i-1] + t[1][i] + a[0][i]);
        T2[i] = min(T2[i-1] + a[1][i], T1[i-1] + t[0][i] + a[1][i]);
    }

    // Consider exit times and return minimum
    return min(T1[NUM_STATION-1] + x[0], T2[NUM_STATION-1] + x[1]);
}

int main()
{
    int a[][NUM_STATION] = {{4, 5, 3, 2},
                             {2, 10, 1, 4}};
```



```

int t[][NUM_STATION] = {{0, 7, 4, 5},
                        {0, 9, 2, 8}};
int e[] = {10, 12}, x[] = {18, 7};

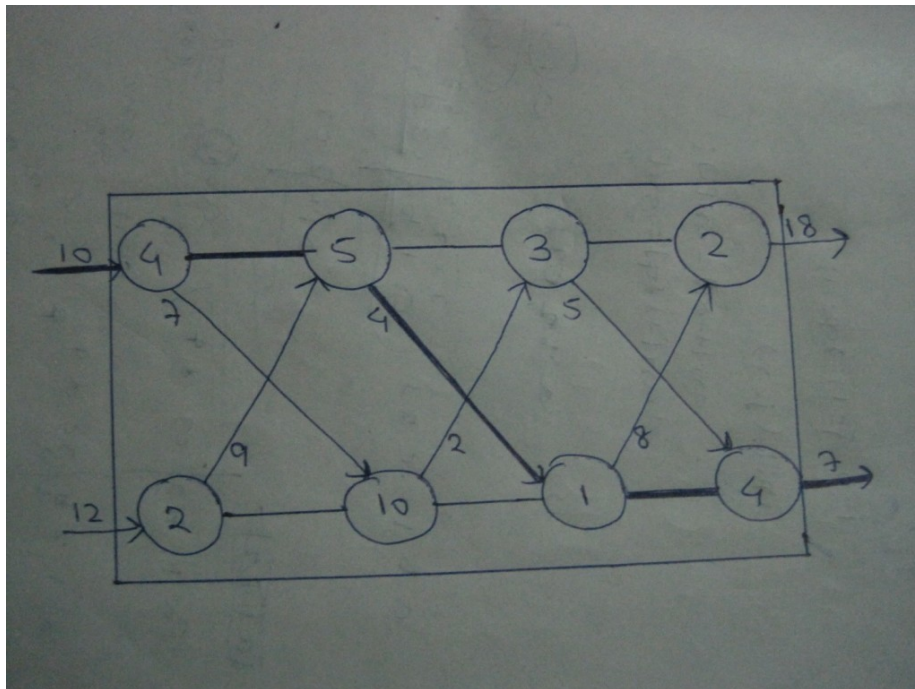
printf("%d", carAssembly(a, t, e, x));

return 0;
}

```

Output:

35



The bold line shows the path covered by the car chassis for given input values.

**Exercise:**

Extend the above algorithm to print the path covered by the car chassis in the factory.

**References:**

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

This article is compiled by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/dynamic-programming-set-34-assembly-line-scheduling/>

## Chapter 35

# Set 35 (Longest Arithmetic Progression)

Given a set of numbers, find the **Length of the Longest Arithmetic Progression (LLAP)** in it.

Examples:

```
set[] = {1, 7, 10, 15, 27, 29}
output = 3
The longest arithmetic progression is {1, 15, 29}

set[] = {5, 10, 15, 20, 25, 30}
output = 6
The whole set is in AP
```

For simplicity, we have assumed that the given set is sorted. We can always add a pre-processing step to first sort the set and then apply the below algorithms.

A **simple solution** is to one by one consider every pair as first two elements of AP and check for the remaining elements in sorted set. To consider all pairs as first two elements, we need to run a  $O(n^2)$  nested loop. Inside the nested loops, we need a third loop which linearly looks for the more elements in **Arithmetic Progression (AP)**. This process takes  $O(n^3)$  time.

We can solve this problem in  $O(n^2)$  time **using Dynamic Programming**. To get idea of the DP solution, let us first discuss solution of following simpler problem.

***Given a sorted set, find if there exist three elements in Arithmetic Progression or not***

Please note that, the answer is true if there are 3 or more elements in AP, otherwise false.

To find the three elements, we first fix an element as middle element and search for other two (one smaller and one greater). We start from the second element and fix every element as middle element. For an element  $\text{set}[j]$  to be middle of AP, there must exist elements ' $\text{set}[i]$ ' and ' $\text{set}[k]$ ' such that  $\text{set}[i] + \text{set}[k] = 2 * \text{set}[j]$  where  $0 \leq i < j < k \leq n-1$ . How to efficiently find  $i$  and  $k$  for a given  $j$ ? We can find  $i$  and  $k$  in linear time using following simple algorithm.

1) Initialize  $i$  as  $j-1$  and  $k$  as  $j+1$   
2) Do following while  $i \geq 0$  and  $j < n$  a) If  $\text{set}[i] + \text{set}[k]$  is equal to  $2 * \text{set}[j]$ , then we are done.  
.....b) If  $\text{set}[i] + \text{set}[k] > 2 * \text{set}[j]$ , then decrement  $i$  (do  $i--$ ).  
.....c) Else if  $\text{set}[i] + \text{set}[k] < 2 * \text{set}[j]$ , then increment  $k$  (do  $k++$ ).  
// The function returns true if there exist three elements in AP // Assumption:  $\text{set}[0..n-1]$  is sorted. // The code strictly implements the algorithm provided in the reference. bool arithmeticThree(int  $\text{set}[]$ , int  $n$ ) { // One by fix every element as middle element for (int  $j=1$ ;  $j < n-1$ ;  $j++$ ) { // Initialize  $i$  and  $k$  for the current  $j$  int  $i = j-1$ ,  $k = j+1$ ; // Find if there exist  $i$  and  $k$  that form AP // with  $j$  as middle element while ( $i \geq 0$  &&  $k < n-1$ ) { if ( $\text{set}[i] + \text{set}[k] == 2 * \text{set}[j]$ ) return true; if ( $\text{set}[i] + \text{set}[k] < 2 * \text{set}[j]$ )  $k++$ ; if ( $\text{set}[i] + \text{set}[k] > 2 * \text{set}[j]$ )  $i--$ ; } } return false; }

See this for a complete running program.

***How to extend the above solution for the original problem?***

The above function returns a boolean value. The required output of original problem is Length of the Longest Arithmetic Progression (LLAP) which is an integer value. If the given set has two or more elements, then the value of LLAP is at least 2 (Why?).

The idea is to create a 2D table  $L[n][n]$ . An entry  $L[i][j]$  in this table stores LLAP with  $\text{set}[i]$  and  $\text{set}[j]$  as first two elements of AP and  $j > i$ . The last column of the table is always 2 (Why – see the meaning of  $L[i][j]$ ). Rest of the table is filled from bottom right to top left. To fill rest of the table,  $j$  (second element in AP) is first fixed.  $i$  and  $k$  are searched for a fixed  $j$ . If  $i$  and  $k$  are found such that  $i, j, k$  form an AP, then the value of  $L[i][j]$  is set as  $L[j][k] + 1$ . Note that the value of  $L[j][k]$  must have been filled before as the loop traverses from right to left columns.

Following is C++ implementation of the Dynamic Programming algorithm.

```
// C++ program to find Length of the Longest AP (llap) in a given sorted set.
// The code strictly implements the algorithm provided in the reference.
#include <iostream>
using namespace std;
```

```

// Returns length of the longest AP subset in a given set
int lengthOfLongestAP(int set[], int n)
{
    if (n <= 2) return n;

    // Create a table and initialize all values as 2. The value of
    // L[i][j] stores LLAP with set[i] and set[j] as first two
    // elements of AP. Only valid entries are the entries where j>i
    int L[n][n];
    int llap = 2; // Initialize the result

    // Fill entries in last column as 2. There will always be
    // two elements in AP with last number of set as second
    // element in AP
    for (int i = 0; i < n; i++)
        L[i][n-1] = 2;

    // Consider every element as second element of AP
    for (int j=n-2; j>=1; j--)
    {
        // Search for i and k for j
        int i = j-1, k = j+1;
        while (i >= 0 && k <= n-1)
        {
            if (set[i] + set[k] < 2*set[j])
                k++;

            // Before changing i, set L[i][j] as 2
            else if (set[i] + set[k] > 2*set[j])
            { L[i][j] = 2, i--; }

            else
            {
                // Found i and k for j, LLAP with i and j as first two
                // elements is equal to LLAP with j and k as first two
                // elements plus 1. L[j][k] must have been filled
                // before as we run the loop from right side
                L[i][j] = L[j][k] + 1;

                // Update overall LLAP, if needed
                llap = max(llap, L[i][j]);

                // Change i and k to fill more L[i][j] values for
                // current j
                i--; k++;
            }
        }
    }
}

```

```

    }

    // If the loop was stopped due to k becoming more than
    // n-1, set the remaining entities in column j as 2
    while (i >= 0)
    {
        L[i][j] = 2;
        i--;
    }
}
return llap;
}

/* Drier program to test above function*/
int main()
{
    int set1[] = {1, 7, 10, 13, 14, 19};
    int n1 = sizeof(set1)/sizeof(set1[0]);
    cout <<  lengthOfLongestAP(set1, n1) << endl;

    int set2[] = {1, 7, 10, 15, 27, 29};
    int n2 = sizeof(set2)/sizeof(set2[0]);
    cout <<  lengthOfLongestAP(set2, n2) << endl;

    int set3[] = {2, 4, 6, 8, 10};
    int n3 = sizeof(set3)/sizeof(set3[0]);
    cout <<  lengthOfLongestAP(set3, n3) << endl;

    return 0;
}

```

Output:

```

4
3
5

```

**Time Complexity:**  $O(n^2)$

**Auxiliary Space:**  $O(n^2)$

**References:**

<http://www.cs.uiuc.edu/~jeffe/pubs/pdf/arith.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/length-of-the-longest-arithmetic-progression-in-a-sorted-array/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Set 34 (Assembly Line Scheduling) How to check if two given line segments intersect? →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 36

# Set 36 (Maximum Product Cutting)

Given a rope of length  $n$  meters, cut the rope in different parts of integer lengths in a way that maximizes product of lengths of all parts. You must make at least one cut. Assume that the length of rope is more than 2 meters.

Examples:

Input:  $n = 2$

Output: 1 (Maximum obtainable product is  $1*1$ )

Input:  $n = 3$

Output: 2 (Maximum obtainable product is  $1*2$ )

Input:  $n = 4$

Output: 4 (Maximum obtainable product is  $2*2$ )

Input:  $n = 5$

Output: 6 (Maximum obtainable product is  $2*3$ )

Input:  $n = 10$

Output: 36 (Maximum obtainable product is  $3*3*4$ )

### 1) Optimal Substructure:

This problem is similar to Rod Cutting Problem. We can get the maximum product by making a cut at different positions and comparing the values obtained



after a cut. We can recursively call the same function for a piece obtained after a cut.

Let  $\text{maxProd}(n)$  be the maximum product for a rope of length  $n$ .  $\text{maxProd}(n)$  can be written as following.

$\text{maxProd}(n) = \max(i*(n-i), \text{maxProdRec}(n-i)*i)$  for all  $i$  in  $\{1, 2, 3 \dots n\}$

## 2) Overlapping Subproblems

Following is simple recursive C++ implementation of the problem. The implementation simply follows the recursive structure mentioned above.

```
// A Naive Recursive method to find maxium product
#include <iostream>
using namespace std;

// Utility function to get the maximum of two and three integers
int max(int a, int b) { return (a > b)? a : b;}
int max(int a, int b, int c) { return max(a, max(b, c));}

// The main function that returns maximum product obtainable
// from a rope of length n
int maxProd(int n)
{
    // Base cases
    if (n == 0 || n == 1) return 0;

    // Make a cut at different places and take the maximum of all
    int max_val = 0;
    for (int i = 1; i < n; i++)
        max_val = max(max_val, i*(n-i), maxProd(n-i)*i);

    // Return the maximum of all values
    return max_val;
}

/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}
```

Output:

Considering the above implementation, following is recursion tree for a Rope of length 5.

```

graph TD
    mP5[mP(5)] --> mP4[mP(4)]
    mP5 --> mP3_1[mP(3)]
    mP5 --> mP2_1[mP(2)]
    mP5 --> mP1_1[mP(1)]
    mP4 --> mP3_2[mP(3)]
    mP4 --> mP2_2[mP(2)]
    mP4 --> mP1_2[mP(1)]
    mP3_1 --> mP2_3[mP(2)]
    mP3_1 --> mP1_3[mP(1)]
    mP2_1 --> mP1_4[mP(1)]
    mP3_2 --> mP2_4[mP(2)]
    mP3_2 --> mP1_5[mP(1)]
    mP2_2 --> mP1_6[mP(1)]
    mP2_3 --> mP1_7[mP(1)]
    mP2_4 --> mP1_8[mP(1)]
    mP1_2 --> mP1_9[mP(1)]
    mP1_3 --> mP1_10[mP(1)]
    mP1_4 --> mP1_11[mP(1)]
    mP1_5 --> mP1_12[mP(1)]
    mP1_6 --> mP1_13[mP(1)]
    mP1_7 --> mP1_14[mP(1)]
    mP1_8 --> mP1_15[mP(1)]
    mP1_9 --> mP1_16[mP(1)]
    mP1_10 --> mP1_17[mP(1)]
    mP1_11 --> mP1_18[mP(1)]
    mP1_12 --> mP1_19[mP(1)]
    mP1_13 --> mP1_20[mP(1)]
    mP1_14 --> mP1_21[mP(1)]
    mP1_15 --> mP1_22[mP(1)]
    mP1_16 --> mP1_23[mP(1)]
    mP1_17 --> mP1_24[mP(1)]
    mP1_18 --> mP1_25[mP(1)]
    mP1_19 --> mP1_26[mP(1)]
    mP1_20 --> mP1_27[mP(1)]
    mP1_21 --> mP1_28[mP(1)]
    mP1_22 --> mP1_29[mP(1)]
    mP1_23 --> mP1_30[mP(1)]
    mP1_24 --> mP1_31[mP(1)]
    mP1_25 --> mP1_32[mP(1)]
    mP1_26 --> mP1_33[mP(1)]
    mP1_27 --> mP1_34[mP(1)]
    mP1_28 --> mP1_35[mP(1)]
    mP1_29 --> mP1_36[mP(1)]
    mP1_30 --> mP1_37[mP(1)]
    mP1_31 --> mP1_38[mP(1)]
    mP1_32 --> mP1_39[mP(1)]
    mP1_33 --> mP1_40[mP(1)]
    mP1_34 --> mP1_41[mP(1)]
    mP1_35 --> mP1_42[mP(1)]
    mP1_36 --> mP1_43[mP(1)]
    mP1_37 --> mP1_44[mP(1)]
    mP1_38 --> mP1_45[mP(1)]
    mP1_39 --> mP1_46[mP(1)]
    mP1_40 --> mP1_47[mP(1)]
    mP1_41 --> mP1_48[mP(1)]
    mP1_42 --> mP1_49[mP(1)]
    mP1_43 --> mP1_50[mP(1)]
    mP1_44 --> mP1_51[mP(1)]
    mP1_45 --> mP1_52[mP(1)]
    mP1_46 --> mP1_53[mP(1)]
    mP1_47 --> mP1_54[mP(1)]
    mP1_48 --> mP1_55[mP(1)]
    mP1_49 --> mP1_56[mP(1)]
    mP1_50 --> mP1_57[mP(1)]
    mP1_51 --> mP1_58[mP(1)]
    mP1_52 --> mP1_59[mP(1)]
    mP1_53 --> mP1_60[mP(1)]
    mP1_54 --> mP1_61[mP(1)]
    mP1_55 --> mP1_62[mP(1)]
    mP1_56 --> mP1_63[mP(1)]
    mP1_57 --> mP1_64[mP(1)]
    mP1_58 --> mP1_65[mP(1)]
    mP1_59 --> mP1_66[mP(1)]
    mP1_60 --> mP1_67[mP(1)]
    mP1_61 --> mP1_68[mP(1)]
    mP1_62 --> mP1_69[mP(1)]
    mP1_63 --> mP1_70[mP(1)]
    mP1_64 --> mP1_71[mP(1)]
    mP1_65 --> mP1_72[mP(1)]
    mP1_66 --> mP1_73[mP(1)]
    mP1_67 --> mP1_74[mP(1)]
    mP1_68 --> mP1_75[mP(1)]
    mP1_69 --> mP1_76[mP(1)]
    mP1_70 --> mP1_77[mP(1)]
    mP1_71 --> mP1_78[mP(1)]
    mP1_72 --> mP1_79[mP(1)]
    mP1_73 --> mP1_80[mP(1)]
    mP1_74 --> mP1_81[mP(1)]
    mP1_75 --> mP1_82[mP(1)]
    mP1_76 --> mP1_83[mP(1)]
    mP1_77 --> mP1_84[mP(1)]
    mP1_78 --> mP1_85[mP(1)]
    mP1_79 --> mP1_86[mP(1)]
    mP1_80 --> mP1_87[mP(1)]
    mP1_81 --> mP1_88[mP(1)]
    mP1_82 --> mP1_89[mP(1)]
    mP1_83 --> mP1_90[mP(1)]
    mP1_84 --> mP1_91[mP(1)]
    mP1_85 --> mP1_92[mP(1)]
    mP1_86 --> mP1_93[mP(1)]
    mP1_87 --> mP1_94[mP(1)]
    mP1_88 --> mP1_95[mP(1)]
    mP1_89 --> mP1_96[mP(1)]
    mP1_90 --> mP1_97[mP(1)]
    mP1_91 --> mP1_98[mP(1)]
    mP1_92 --> mP1_99[mP(1)]
    mP1_93 --> mP1_100[mP(1)]
    mP1_94 --> mP1_101[mP(1)]
    mP1_95 --> mP1_102[mP(1)]
    mP1_96 --> mP1_103[mP(1)]
    mP1_97 --> mP1_104[mP(1)]
    mP1_98 --> mP1_105[mP(1)]
    mP1_99 --> mP1_106[mP(1)]
    mP1_100 --> mP1_107[mP(1)]
    mP1_101 --> mP1_108[mP(1)]
    mP1_102 --> mP1_109[mP(1)]
    mP1_103 --> mP1_110[mP(1)]
    mP1_104 --> mP1_111[mP(1)]
    mP1_105 --> mP1_112[mP(1)]
    mP1_106 --> mP1_113[mP(1)]
    mP1_107 --> mP1_114[mP(1)]
    mP1_108 --> mP1_115[mP(1)]
    mP1_109 --> mP1_116[mP(1)]
    mP1_110 --> mP1_117[mP(1)]
    mP1_111 --> mP1_118[mP(1)]
    mP1_112 --> mP1_119[mP(1)]
    mP1_113 --> mP1_120[mP(1)]
    mP1_114 --> mP1_121[mP(1)]
    mP1_115 --> mP1_122[mP(1)]
    mP1_116 --> mP1_123[mP(1)]
    mP1_117 --> mP1_124[mP(1)]
    mP1_118 --> mP1_125[mP(1)]
    mP1_119 --> mP1_126[mP(1)]
    mP1_120 --> mP1_127[mP(1)]
    mP1_121 --> mP1_128[mP(1)]
    mP1_122 --> mP1_129[mP(1)]
    mP1_123 --> mP1_130[mP(1)]
    mP1_124 --> mP1_131[mP(1)]
    mP1_125 --> mP1_132[mP(1)]
    mP1_126 --> mP1_133[mP(1)]
    mP1_127 --> mP1_134[mP(1)]
    mP1_128 --> mP1_135[mP(1)]
    mP1_129 --> mP1_136[mP(1)]
    mP1_130 --> mP1_137[mP(1)]
    mP1_131 --> mP1_138[mP(1)]
    mP1_132 --> mP1_139[mP(1)]
    mP1_133 --> mP1_140[mP(1)]
    mP1_134 --> mP1_141[mP(1)]
    mP1_135 --> mP1_142[mP(1)]
    mP1_136 --> mP1_143[mP(1)]
    mP1_137 --> mP1_144[mP(1)]
    mP1_138 --> mP1_145[mP(1)]
    mP1_139 --> mP1_146[mP(1)]
    mP1_140 --> mP1_147[mP(1)]
    mP1_141 --> mP1_148[mP(1)]
    mP1_142 --> mP1_149[mP(1)]
    mP1_143 --> mP1_150[mP(1)]
    mP1_144 --> mP1_151[mP(1)]
    mP1_145 --> mP1_152[mP(1)]
    mP1_146 --> mP1_153[mP(1)]
    mP1_147 --> mP1_154[mP(1)]
    mP1_148 --> mP1_155[mP(1)]
    mP1_149 --> mP1_156[mP(1)]
    mP1_150 --> mP1_157[mP(1)]
    mP1_151 --> mP1_158[mP(1)]
    mP1_152 --> mP1_159[mP(1)]
    mP1_153 --> mP1_160[mP(1)]
    mP1_154 --> mP1_161[mP(1)]
    mP1_15
```

```
// A Dynamic Programming solution for Max Product Problem
int maxProd(int n)
{
    int val[n+1];
    val[0] = val[1] = 0;

    // Build the table val[] in bottom up manner and return
    // the last entry from the table
    for (int i = 1; i <= n; i++)
    {
        int max_val = 0;
        for (int j = 1; j <= i/2; j++)
            max_val = max(max_val, (i-j)*j, j*val[i-j]);
        val[i] = max_val;
    }
}
```

```

        return val[n];
    }

```

Time Complexity of the Dynamic Programming solution is  $O(n^2)$  and it requires  $O(n)$  extra space.

### A Tricky Solution:

If we see some examples of this problems, we can easily observe following pattern. The maximum product can be obtained by repeatedly cutting parts of size 3 while size is greater than 4, keeping the last part as size of 2 or 3 or 4. For example,  $n = 10$ , the maximum product is obtained by 3, 3, 4. For  $n = 11$ , the maximum product is obtained by 3, 3, 3, 2. Following is C++ implementation of this approach.

```

#include <iostream>
using namespace std;

/* The main function that returns the max possible product */
int maxProd(int n)
{
    // n equals to 2 or 3 must be handled explicitly
    if (n == 2 || n == 3) return (n-1);

    // Keep removing parts of size 3 while n is greater than 4
    int res = 1;
    while (n > 4)
    {
        n -= 3;
        res *= 3; // Keep multiplying 3 to res
    }
    return (n * res); // The last part multiplied by previous parts
}

/* Driver program to test above functions */
int main()
{
    cout << "Maximum Product is " << maxProd(10);
    return 0;
}

```

Output:

Maximum Product is 36

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/dynamic-programming-set-36-cut-a-rope-to-maximize-product/>

## Chapter 37

# Set 37 (Boolean Parenthesization Problem)

Given a boolean expression with following symbols.

Symbols

```
'T' ---> true
'F' ---> false
```

And following operators filled between symbols

Operators

```
& ---> boolean AND
| ---> boolean OR
^ ---> boolean XOR
```

Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

Let the input be in form of two arrays one contains the symbols (T and F) in order and other contains operators (&, | and ^)

**Examples:**

Input: symbol[] = {T, F, T}  
operator[] = {^, &}  
Output: 2  
The given expression is "T ^ F & T", it evaluates true  
in two ways "( (T ^ F) & T)" and "(T ^ (F & T))"

Input: symbol[] = {T, F, F}  
operator[] = {^, |}  
Output: 2  
The given expression is "T ^ F | F", it evaluates true  
in two ways "( (T ^ F) | F )" and "( T ^ (F | F) )".

Input: symbol[] = {T, T, F, T}  
operator[] = {|, &, ^}  
Output: 4  
The given expression is "T | T & F ^ T", it evaluates true  
in 4 ways ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T)  
and (T|((T&F)^T)).

### Solution:

Let **T(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to true.

$$T(i, j) = \sum_{k=i}^{j-1} \begin{cases} T(i, k) * T(k+1, j) & \text{If operator[k] is '&'} \\ Total(i, k) * Total(k+1, j) - F(i, k) * F(k+1, j) & \text{If operator[k] is '|'} \\ T(i, k) * F(k+1, j) + F(i, k) * T(k+1, j) & \text{If operator[k] is '^'} \end{cases}$$

$$Total(i, j) = T(i, j) + F(i, j)$$

Let **F(i, j)** represents the number of ways to parenthesize the symbols between i and j (both inclusive) such that the subexpression between i and j evaluates to false.

$$F(i, j) = \sum_{k=i}^{j-1} \begin{cases} Total(i, k) * Total(k+1, j) - T(i, k) * T(k+1, j) & \text{If operator[k] is '&'} \\ F(i, k) * F(k+1, j) & \text{If operator[k] is '|'} \\ T(i, k) * T(k+1, j) + F(i, k) * F(k+1, j) & \text{If operator[k] is '^'} \end{cases}$$

$$Total(i, j) = T(i, j) + F(i, j)$$

Base Cases:

$$\begin{aligned} T(i, i) &= 1 \text{ if symbol[i] = 'T'} \\ T(i, i) &= 0 \text{ if symbol[i] = 'F'} \\ F(i, i) &= 1 \text{ if symbol[i] = 'F'} \end{aligned}$$

$F(i, i) = 0$  if `symbol[i] = 'T'`

If we draw recursion tree of above recursive solution, we can observe that it many overlapping subproblems. Like other dynamic programming problems, it can be solved by filling a table in bottom up manner. Following is C++ implementation of dynamic programming solution.

```
#include<iostream>
#include<cstring>
using namespace std;

// Returns count of all possible parenthesizations that lead to
// result true for a boolean expression with symbols like true
// and false and operators like &, | and ^ filled between symbols
int countParenth(char symb[], char oper[], int n)
{
    int F[n][n], T[n][n];

    // Fill diagonal entries first
    // All diagonal entries in T[i][i] are 1 if symbol[i]
    // is T (true). Similarly, all F[i][i] entries are 1 if
    // symbol[i] is F (False)
    for (int i = 0; i < n; i++)
    {
        F[i][i] = (symb[i] == 'F')? 1: 0;
        T[i][i] = (symb[i] == 'T')? 1: 0;
    }

    // Now fill T[i][i+1], T[i][i+2], T[i][i+3]... in order
    // And F[i][i+1], F[i][i+2], F[i][i+3]... in order
    for (int gap=1; gap<n; ++gap)
    {
        for (int i=0, j=gap; j<n; ++i, ++j)
        {
            T[i][j] = F[i][j] = 0;
            for (int g=0; g<gap; g++)
            {
                // Find place of parenthesization using current value
                // of gap
                int k = i + g;

                // Store Total[i][k] and Total[k+1][j]
                int tik = T[i][k] + F[i][k];
```

```

        int tkj = T[k+1][j] + F[k+1][j];

        // Follow the recursive formulas according to the current
        // operator
        if (oper[k] == '&')
        {
            T[i][j] += T[i][k]*T[k+1][j];
            F[i][j] += (tik*tkj - T[i][k]*T[k+1][j]);
        }
        if (oper[k] == '|')
        {
            F[i][j] += F[i][k]*F[k+1][j];
            T[i][j] += (tik*tkj - F[i][k]*F[k+1][j]);
        }
        if (oper[k] == '^')
        {
            T[i][j] += F[i][k]*T[k+1][j] + T[i][k]*F[k+1][j];
            F[i][j] += T[i][k]*T[k+1][j] + F[i][k]*F[k+1][j];
        }
    }
}

return T[0][n-1];
}

// Driver program to test above function
int main()
{
    char symbols[] = "TTFT";
    char operators[] = "&|^";
    int n = strlen(symbols);

    // There are 4 ways
    // ((T|T)&(F^T)), (T|(T&(F^T))), (((T|T)&F)^T) and (T|((T&F)^T))
    cout << countParenth(symbols, operators, n);
    return 0;
}

```

Output:

4



Time Complexity:  $O(n^3)$

Auxiliary Space:  $O(n^2)$

**References:**

[http://people.cs.clemson.edu/~bcdcan/dp\\_practice/dp\\_9.swf](http://people.cs.clemson.edu/~bcdcan/dp_practice/dp_9.swf)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<http://www.geeksforgeeks.org/dynamic-programming-set-37-boolean-parenthesization-problem/>

Category: Misc

Post navigation

← Pilani Soft Labs (redBus) Interview Print substring of a given string without using any string function and loop in C →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 38

# Bitmasking and Set 1 (Count ways to assign unique cap to every person)

Consider the below problems statement.

There 100 different types of caps each having a unique id from 1 to 100. Also, there 'n' persons each having a collection of variable number of caps. One day all of these persons decide to go in a party wearing a cap but to look unique they decided that none them will wear the same type of cap. So, count the total number of arrangements or ways such that none of them is wearing same type of cap.

Constraints: 1 First line contains value of n, next n lines contain collections of all the n persons. Input: 3 5 100 1 // Collection of first person. 2 // Collection of second person. 5 100 // Collection of second person. Output: 4 Explanation: All valid possible ways are (5, 2, 100), (100, 2, 5), (1, 2, 5) and (1, 2, 100)

Since, number of ways could be large, so output modulo 1000000007

**We strongly recommend you to minimize your browser and try this yourself first.**

A **Simple Solution** is to try all possible combinations. Start by picking first element from first set, marking it as visited and recur for remaining sets. It is basically a Backtracking based solution.

A **better solution is to use Bitmasking and DP**. Let us first introduce Bitmasking.

**What is Bitmasking?**

Suppose we have a collection of elements which are numbered from 1 to N. If

we want to represent a subset of this set than it can be encoded by a sequence of  $N$  bits (we usually call this sequence a “mask”). In our chosen subset the  $i$ -th element belongs to it if and only if the  $i$ -th bit of the mask is set i.e., it equals to 1. For example, the mask 10000101 means that the subset of the set  $[1...8]$  consists of elements 1, 3 and 8. We know that for a set of  $N$  elements there are total  $2^N$  subsets thus  $2^N$  masks are possible, one representing each subset. Each mask is in fact an integer number written in binary notation.

Our main methodology is to assign a value to each mask (and, therefore, to each subset) and thus calculate the values for new masks using values of the already computed masks. Usually our main target is to calculate value/solution for the complete set i.e., for mask 11111111. Normally, to find the value for a subset  $X$  we remove an element in every possible way and use values for obtained subsets  $X'_1, X'_2... ,X'_k$  to compute the value/solution for  $X$ . This means that the values for  $X'_i$  must have been computed already, so we need to establish an ordering in which masks will be considered. It's easy to see that the natural ordering will do: go over masks in increasing order of corresponding numbers. Also, We sometimes, start with the empty subset  $X$  and we add elements in every possible way and use the values of obtained subsets  $X'_1, X'_2... ,X'_k$  to compute the value/solution for  $X$ .

We mostly use the following notations/operations on masks:

bit( $i$ ,mask) – the  $i$ -th bit of mask

count(mask) – the number of non-zero bits in mask

first(mask) – the number of the lowest non-zero bit in mask

set( $i$ , mask) – set the  $i$ th bit in mask

check( $i$ , mask) – check the  $i$ th bit in mask

### How is this problem solved using Bitmasking + DP?

The idea is to use the fact that there are upto 10 persons. So we can use a integer variable as a bitmask to store which person is wearing cap and which is not.

Let  $i$  be the current cap number (caps from 1 to  $i-1$  are already processed). Let integer variable mask indicates the the persons wearing and not wearing caps. If  $i$ 'th bit is set in mask, then  $i$ 'th person is wearing a cap, else not.

```
// consider the case when ith cap is not included
// in the arrangement
countWays(mask, i) = countWays(mask, i+1) +

// when ith cap is included in the arrangement
// so, assign this cap to all possible persons
// one by one and recur for remaining persons.
&Sum; countWays(mask | (1
```

If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. Since we want to access all persons that can wear a given cap, we use an array of vectors, `capList`. Below is C++ implementation of above idea.

```
// C++ program to find number of ways to wear hats
#include<bits/stdc++.h>
#define MOD 1000000007
using namespace std;

// capList[i]'th vector contains the list of persons having a cap with id i
// id is between 1 to 100 so we declared an array of 101 vectors as indexing
// starts from 0.
vector<int> capList[101];

// dp[2^10][101] .. in dp[i][j], i denotes the mask i.e., it tells that
// how many and which persons are wearing cap. j denotes the first j caps
// used. So, dp[i][j] tells the number ways we assign j caps to mask i
// such that none of them wears the same cap
int dp[1025][101];

// This is used for base case, it has all the N bits set
// so, it tells whether all N persons are wearing a cap.
int allmask;

// Mask is the set of persons, i is the number of
// caps processed starting from first cap.
long long int countWaysUtil(int mask, int i)
{
    // If all persons are wearing a cap so we
    // are done and this is one way so return 1
    if (mask == allmask) return 1;

    // If not everyone is wearing a cap and also there are no more
    // caps left to process, so there is no way, thus return 0;
    if (i > 100) return 0;

    // If we already have solved this subproblem, return the answer.
    if (dp[mask][i] != -1) return dp[mask][i];

    // Ways, when we don't include this cap in our arrangement
    // or solution set.
    long long int ways = countWaysUtil(mask, i+1);

    // size is the total number of persons having cap with id i.
    int size = capList[i].size();
```

```

// So, assign one by one ith cap to all the possible persons
// and recur for remaining caps.
for (int j = 0; j < size; j++)
{
    // if person capList[i][j] is already wearing a cap so continue as
    // we cannot assign him this cap
    if (mask & (1 << capList[i][j])) continue;

    // Else assign him this cap and recur for remaining caps with
    // new updated mask vector
    else ways += countWaysUtil(mask | (1 << capList[i][j]), i+1);
    ways %= MOD;
}

// Save the result and return it.
return dp[mask][i] = ways;
}

// Reads n lines from standard input for current test case
void countWays(int n)
{
    //----- READ INPUT -----
    string temp, str;
    int x;
    getline(cin, str); // to get rid of newline character
    for (int i=0; i<n; i++)
    {
        getline(cin, str);
        stringstream ss(str);

        // while there are words in the streamobject ss
        while (ss >> temp)
        {
            stringstream s;
            s << temp;
            s >> x;

            // add the ith person in the list of cap if with id x
            capList[x].push_back(i);
        }
    }
    //-----

    // All mask is used to check of all persons
    // are included or not, set all n bits as 1
    allmask = (1 << n) - 1;

```

```

        // Initialize all entries in dp as -1
        memset(dp, -1, sizeof dp);

        // Call recursive function count ways
        cout << countWaysUtil(0, 1) << endl;
    }

    // Driver Program
    int main()
    {
        int n;    // number of persons in every test case
        cin >> n;
        countWays(n);
        return 0;
    }

```

Input:

```

3
5 100 1
2
5 100

```

Output:

```

4

```

This article is contributed by Gaurav Ahirwar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/bitmasking-and-dynamic-programming-set-1-count-ways-to-assign-unique-cap>

## Chapter 39

# Collect maximum points in a grid using two traversals

Given a matrix where every cell represents points. How to collect maximum points using two traversals under following conditions?

Let the dimensions of given grid be  $R \times C$ .

- 1) The first traversal starts from top left corner, i.e.,  $(0, 0)$  and should reach left bottom corner, i.e.,  $(R-1, 0)$ . The second traversal starts from top right corner, i.e.,  $(0, C-1)$  and should reach bottom right corner, i.e.,  $(R-1, C-1)$ .
- 2) From a point  $(i, j)$ , we can move to  $(i+1, j+1)$  or  $(i+1, j-1)$  or  $(i+1, j)$ .
- 3) A traversal gets all points of a particular cell through which it passes. If one traversal has already collected points of a cell, then the other traversal gets no points if goes through that cell again.

Input :

```
int arr[R][C] = {{3, 6, 8, 2},
                  {5, 2, 4, 3},
                  {1, 1, 20, 10},
                  {1, 1, 20, 10},
                  {1, 1, 20, 10},
                  };
```

Output: 73

Explanation :

First traversal collects total points of value  $3 + 2 + 20 + 1 + 1 = 27$

Second traversal collects total points of value  $2 + 4 + 10 + 20 + 10 = 46$ .  
 Total Points collected =  $27 + 46 = 73$ .

Source: <http://qa.geeksforgeeks.org/1485/running-through-the-grid-to-get-maximum-nutritional-value>

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to do both traversals concurrently. We start first from (0, 0) and second traversal from (0, C-1) simultaneously. The important thing to note is, at any particular step both traversals will be in same row as in all possible three moves, row number is increased. Let (x1, y1) and (x2, y2) denote current positions of first and second traversals respectively. Thus at any time x1 will be equal to x2 as both of them move forward but variation is possible along y. Since variation in y could occur in 3 ways no change (y), go left (y - 1), go right (y + 1). So in total 9 combinations among y1, y2 are possible. The 9 cases as mentioned below after base cases.

```
Both traversals always move forward along x
Base Cases:
// If destinations reached
if (x == R-1 && y1 == 0 && y2 == C-1)
    maxPoints(arr, x, y1, y2) = arr[x][y1] + arr[x][y2];

// If any of the two locations is invalid (going out of grid)
if input is not valid
    maxPoints(arr, x, y1, y2) = -INF (minus infinite)

// If both traversals are at same cell, then we count the value of cell
// only once.
If y1 and y2 are same
    result = arr[x][y1]
Else
    result = arr[x][y1] + arr[x][y2]

result += max { // Max of 9 cases
    maxPoints(arr, x+1, y1+1, y2),
    maxPoints(arr, x+1, y1+1, y2+1),
    maxPoints(arr, x+1, y1+1, y2-1),
    maxPoints(arr, x+1, y1-1, y2),
    maxPoints(arr, x+1, y1-1, y2+1),
    maxPoints(arr, x+1, y1-1, y2-1),
    maxPoints(arr, x+1, y1, y2),
    maxPoints(arr, x+1, y1, y2+1),
```



```

        maxPoints(arr, x+1, y1, y2-1)
    }

```

The above recursive solution has many subproblems that are solved again and again. Therefore, we can use Dynamic Programming to solve the above problem more efficiently. Below is memoization (Memoization is alternative to table based iterative solution in Dynamic Programming) based implementation. In below implementation, we use a memoization table 'mem' to keep track of already solved problems.

```

// A Memoization based program to find maximum collection
// using two traversals of a grid
#include<bits/stdc++.h>
using namespace std;
#define R 5
#define C 4

// checks whether a given input is valid or not
bool isValid(int x, int y1, int y2)
{
    return (x >= 0 && x < R && y1 >=0 &&
            y1 < C && y2 >=0 && y2 < C);
}

// Driver function to collect max value
int getMaxUtil(int arr[R][C], int mem[R][C][C], int x, int y1, int y2)
{
    /*----- BASE CASES -----*/
    // if P1 or P2 is at an invalid cell
    if (!isValid(x, y1, y2)) return INT_MIN;

    // if both traversals reach their destinations
    if (x == R-1 && y1 == 0 && y2 == C-1)
        return arr[x][y1] + arr[x][y2];

    // If both traversals are at last row but not at their destination
    if (x == R-1) return INT_MIN;

    // If subproblem is already solved
    if (mem[x][y1][y2] != -1) return mem[x][y1][y2];

    // Initialize answer for this subproblem

```

```

int ans = INT_MIN;

// this variable is used to store gain of current cell(s)
int temp = (y1 == y2)? arr[x][y1]: arr[x][y1] + arr[x][y2];

/* Recur for all possible cases, then store and return the
   one with max value */
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2-1));
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2+1));
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1, y2));

ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2));
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2-1));
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1-1, y2+1));

ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2));
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2-1));
ans = max(ans, temp + getMaxUtil(arr, mem, x+1, y1+1, y2+1));

return (mem[x][y1][y2] = ans);
}

// This is mainly a wrapper over recursive function getMaxUtil().
// This function creates a table for memoization and calls
// getMaxUtil()
int geMaxCollection(int arr[R][C])
{
    // Create a memoization table and initialize all entries as -1
    int mem[R][C][C];
    memset(mem, -1, sizeof(mem));

    // Calculation maximum value using memoization based function
    // getMaxUtil()
    return getMaxUtil(arr, mem, 0, 0, C-1);
}

// Driver program to test above functions
int main()
{
    int arr[R][C] = {{3, 6, 8, 2},
                     {5, 2, 4, 3},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10},
                     {1, 1, 20, 10},
                     };
    cout << "Maximum collection is " << geMaxCollection(arr);
}

```

```
        return 0;  
    }
```

Output:

```
Maximum collection is 73
```

Thanks to Gaurav Ahirwar for suggesting above problem and solution here.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/collect-maximum-points-in-a-grid-using-two-traversals/>

Category: Arrays Tags: Dynamic Programming, Matrix

Post navigation

← Amazon Interview Experience | Set 213 (Off-Campus for SDE1) Amazon Interview Experience | 214 (On-Campus) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 40

# Compute sum of digits in all numbers from 1 to n

Given a number x, find sum of digits in all numbers from 1 to n.  
Examples:

Input: n = 5

Output: Sum of digits in numbers from 1 to 5 = 15

Input: n = 12

Output: Sum of digits in numbers from 1 to 12 = 51

Input: n = 328

Output: Sum of digits in numbers from 1 to 328 = 3241

### Naive Solution:

A naive solution is to go through every number x from 1 to n, and compute sum in x by traversing all digits of x. Below is C++ implementation of this idea.

```
// A Simple C++ program to compute sum of digits in numbers from 1 to n
#include<iostream>
using namespace std;

int sumOfDigits(int );
```

```

// Returns sum of all digits in numbers from 1 to n
int sumOfDigitsFrom1ToN(int n)
{
    int result = 0; // initialize result

    // One by one compute sum of digits in every number from
    // 1 to n
    for (int x=1; x<=n; x++)
        result += sumOfDigits(x);

    return result;
}

// A utility function to compute sum of digits in a
// given number x
int sumOfDigits(int x)
{
    int sum = 0;
    while (x != 0)
    {
        sum += x %10;
        x   = x /10;
    }
    return sum;
}

// Driver Program
int main()
{
    int n = 328;
    cout << "Sum of digits in numbers from 1 to " << n << " is "
         << sumOfDigitsFrom1ToN(n);
    return 0;
}

```

Output

```
Sum of digits in numbers from 1 to 328 is 3241
```

#### **Efficient Solution:**

Above is a naive solution. We can do it more efficiently by finding a pattern.

Let us take few examples.

$$\begin{aligned}
\text{sum}(9) &= 1 + 2 + 3 + 4 + \dots + 9 \\
&= 9*10/2 \\
&= 45
\end{aligned}$$

$$\begin{aligned}
\text{sum}(99) &= 45 + (10 + 45) + (20 + 45) + \dots + (90 + 45) \\
&= 45*10 + (10 + 20 + 30 + \dots + 90) \\
&= 45*10 + 10(1 + 2 + \dots + 9) \\
&= 45*10 + 45*10 \\
&= \text{sum}(9)*10 + 45*10
\end{aligned}$$

$$\text{sum}(999) = \text{sum}(99)*10 + 45*100$$

In general, we can compute  $\text{sum}(10^d - 1)$  using below formula

$$\text{sum}(10^d - 1) = \text{sum}(10^{d-1} - 1) * 10 + 45*(10^{d-1})$$

In below implementation, the above formula is implemented using dynamic programming as there are overlapping subproblems.

The above formula is one core step of the idea. Below is complete algorithm

**Algorithm: sum(n)**

- 1) Find number of digits minus one in n. Let this value be 'd'.  
For 328, d is 2.
- 2) Compute sum of digits in numbers from 1 to  $10^d - 1$ .  
Let this sum be w. For 328, we compute sum of digits from 1 to 99 using above formula.
- 3) Find Most significant digit (msd) in n. For 328, msd is 3.
- 4) Overall sum is sum of following terms
  - a) Sum of digits in 1 to " $\text{msd} * 10^d - 1$ ". For 328, sum of digits in numbers from 1 to 299.  
For 328, we compute  $3*\text{sum}(99) + (1 + 2)*100$ . Note that sum of  $\text{sum}(299)$  is  $\text{sum}(99) + \text{sum of digits from 100 to 199} + \text{sum of digits from 200 to 299}$ .  
Sum of 100 to 199 is  $\text{sum}(99) + 1*100$  and sum of 299 is  $\text{sum}(99) + 2*100$ .  
In general, this sum can be computed as  $w*\text{msd} + (\text{msd}*(\text{msd}-1)/2)*10^d$

b) Sum of digits in  $\text{msd} * 10^d$  to  $n$ . For 328, sum of digits in 300 to 328.  
 For 328, this sum is computed as  $3*29 + \text{recursive call "sum(28)"}$   
 In general, this sum can be computed as  $\text{msd} * (n \% (\text{msd}*10^d) + 1) + \text{sum}(n \% (10^d))$

Below is C++ implementation of above algorithm.

```
// C++ program to compute sum of digits in numbers from 1 to n
#include<bits/stdc++.h>
using namespace std;

// Function to computer sum of digits in numbers from 1 to n
// Comments use example of 328 to explain the code
int sumOfDigitsFrom1ToN(int n)
{
    // base case: if  $n < 10$  return sum of
    // first  $n$  natural numbers
    if ( $n < 10$ )
        return  $n*(n+1)/2$ ;

    //  $d$  = number of digits minus one in  $n$ . For 328,  $d$  is 2
    int  $d = \log_{10}(n)$ ;

    // computing sum of digits from 1 to  $10^d - 1$ ,
    //  $d=1$   $a[0]=0$ ;
    //  $d=2$   $a[1]=\text{sum of digit from 1 to 9} = 45$ 
    //  $d=3$   $a[2]=\text{sum of digit from 1 to 99} = a[1]*10 + 45*10^1 = 900$ 
    //  $d=4$   $a[3]=\text{sum of digit from 1 to 999} = a[2]*10 + 45*10^2 = 13500$ 
    int *a = new int[ $d+1$ ];
    a[0] = 0, a[1] = 45;
    for (int  $i=2$ ;  $i \leq d$ ;  $i++$ )
        a[i] = a[i-1]*10 + 45*ceil(pow(10, $i-1$ ));

    // computing  $10^d$ 
    int  $p = \text{ceil}(\text{pow}(10, d))$ ;

    // Most significant digit (msd) of  $n$ ,
    // For 328, msd is 3 which can be obtained using  $328/100$ 
    int  $\text{msd} = n/p$ ;

    // EXPLANATION FOR FIRST and SECOND TERMS IN BELOW LINE OF CODE
```

```

        // First two terms compute sum of digits from 1 to 299
        // (sum of digits in range 1-99 stored in a[d]) +
        // (sum of digits in range 100-199, can be calculated as 1*100 + a[d]
        // (sum of digits in range 200-299, can be calculated as 2*100 + a[d]
        // The above sum can be written as 3*a[d] + (1+2)*100

        // EXPLANATION FOR THIRD AND FOURTH TERMS IN BELOW LINE OF CODE
        // The last two terms compute sum of digits in number from 300 to 328
        // The third term adds 3*29 to sum as digit 3 occurs in all numbers
        //           from 300 to 328
        // The fourth term recursively calls for 28
        return msd*a[d] + (msd*(msd-1)/2)*p +
            msd*(1+n%p) + sumOfDigitsFrom1ToN(n%p);
    }

    // Driver Program
    int main()
    {
        int n = 328;
        cout << "Sum of digits in numbers from 1 to " << n << " is "
            << sumOfDigitsFrom1ToN(n);
        return 0;
    }

```

Output

```
Sum of digits in numbers from 1 to 328 is 3241
```

The efficient algorithm has one more advantage that we need to compute the array 'a[]' only once even when we are given multiple inputs.

This article is computed by **Shubham Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-sum-of-digits-in-numbers-from-1-to-n/>

Category: Misc Tags: Dynamic Programming, MathematicalAlgo



## Chapter 41

# Count Possible Decodings of a given Digit Sequence

Let 1 represent 'A', 2 represents 'B', etc. Given a digit sequence, count the number of possible decodings of the given digit sequence.

Examples:

```
Input: digits[] = "121"
Output: 3
// The possible decodings are "ABA", "AU", "LA"

Input: digits[] = "1234"
Output: 3
// The possible decodings are "ABCD", "LCD", "AWD"
```

An empty digit sequence is considered to have one decoding. It may be assumed that the input contains valid digits from 0 to 9 and there are no leading 0's, no extra trailing 0's and no two or more consecutive 0's.

**We strongly recommend to minimize the browser and try this yourself first.**

This problem is recursive and can be broken in sub-problems. We start from end of the given digit sequence. We initialize the total count of decodings as 0. We recur for two subproblems.

1) If the last digit is non-zero, recur for remaining (n-1) digits and add the result to total count.

2) If the last two digits form a valid character (or smaller than 27), recur for remaining (n-2) digits and add the result to total count.

Following is C++ implementation of the above approach.

```
// A naive recursive C++ implementation to count number of decodings
// that can be formed from a given digit sequence
#include <iostream>
#include <cstring>
using namespace std;

// Given a digit sequence of length n, returns count of possible
// decodings by replacing 1 with A, 2 with B, ... 26 with Z
int countDecoding(char *digits, int n)
{
    // base cases
    if (n == 0 || n == 1)
        return 1;

    int count = 0; // Initialize count

    // If the last digit is not 0, then last digit must add to
    // the number of words
    if (digits[n-1] > '0')
        count = countDecoding(digits, n-1);

    // If the last two digits form a number smaller than or equal to 26,
    // then consider last two digits and recur
    if (digits[n-2] < '2' || (digits[n-2] == '2' && digits[n-1] < '7'))
        count += countDecoding(digits, n-2);

    return count;
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
    int n = strlen(digits);
    cout << "Count is " << countDecoding(digits, n);
    return 0;
}
```

Output:

```
Count is 3
```

The time complexity of above the code is exponential. If we take a closer look at the above program, we can observe that the recursive solution is similar to Fibonacci Numbers. Therefore, we can optimize the above solution to work in  $O(n)$  time using Dynamic Programming. Following is C++ implementation for the same.

```
// A Dynamic Programming based C++ implementation to count decodings
#include <iostream>
#include <cstring>
using namespace std;

// A Dynamic Programming based function to count decodings
int countDecodingDP(char *digits, int n)
{
    int count[n+1]; // A table to store results of subproblems
    count[0] = 1;
    count[1] = 1;

    for (int i = 2; i <= n; i++)
    {
        count[i] = 0;

        // If the last digit is not 0, then last digit must add to
        // the number of words
        if (digits[i-1] > '0')
            count[i] = count[i-1];

        // If second last digit is smaller than 2 and last digit is
        // smaller than 7, then last two digits form a valid character
        if (digits[i-2] < '2' || (digits[i-2] == '2' && digits[i-1] < '7'))
            count[i] += count[i-2];
    }
    return count[n];
}

// Driver program to test above function
int main()
{
    char digits[] = "1234";
```

```
    int n = strlen(digits);  
    cout << "Count is " << countDecodingDP(digits, n);  
    return 0;  
}
```

Output:

```
Count is 3
```

Time Complexity of the above solution is  $O(n)$  and it requires  $O(n)$  auxiliary space. We can reduce auxiliary space to  $O(1)$  by using space optimized version discussed in the Fibonacci Number Post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-possible-decodings-given-digit-sequence/>

Category: Strings Tags: Dynamic Programming, Fibonacci numbers, MathematicalAlgo

## Chapter 42

# Count all possible paths from top left to bottom right of a mXn matrix

The problem is to count all the possible paths from top left to bottom right of a mXn matrix with the constraints that *from each cell you can either move only to right or down*

We have discussed a solution to print all possible paths, counting all paths is easier. Let NumberOfPaths(m, n) be the count of paths to reach row number m and column number n in the matrix, NumberOfPaths(m, n) can be recursively written as following.

```
#include <iostream>
using namespace std;

// Returns count of possible paths to reach cell at row number m and column
// number n from the topmost leftmost cell (cell at 1, 1)
int numberOfPaths(int m, int n)
{
    // If either given row number is first or given column number is first
    if (m == 1 || n == 1)
        return 1;

    // If diagonal movements are allowed then the last addition
    // is required.
    return  numberOfPaths(m-1, n) + numberOfPaths(m, n-1);
    // + numberOfPaths(m-1,n-1);
}
```

```

}

int main()
{
    cout << numberOfPaths(3, 3);
    return 0;
}

```

Output:

6

The time complexity of above recursive solution is exponential. There are many overlapping subproblems. We can draw a recursion tree for `numberOfPaths(3, 3)` and see many overlapping subproblems. The recursion tree would be similar to Recursion tree for Longest Common Subsequence problem.

So this problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `count[][]` in bottom up manner using the above recursive formula.

```

#include <iostream>
using namespace std;

// Returns count of possible paths to reach cell at row number m and column
// number n from the topmost leftmost cell (cell at 1, 1)
int numberOfPaths(int m, int n)
{
    // Create a 2D table to store results of subproblems
    int count[m][n];

    // Count of paths to reach any cell in first column is 1
    for (int i = 0; i < m; i++)
        count[i][0] = 1;

    // Count of paths to reach any cell in first column is 1
    for (int j = 0; j < n; j++)
        count[0][j] = 1;

    // Calculate count of paths for other cells in bottom-up manner using
    // the recursive solution

```

```

    for (int i = 1; i < m; i++)
    {
        for (int j = 1; j < n; j++)

            // By uncommenting the last part the code calculatest he total
            // possible paths if the diagonal Movements are allowed
            count[i][j] = count[i-1][j] + count[i][j-1]; //+ count[i-1][j-1];

    }
    return count[m-1][n-1];
}

// Driver program to test above functions
int main()
{
    cout << numberOfPaths(3, 3);
    return 0;
}

```

Output:

6

Time complexity of the above dynamic programming solution is  $O(mn)$ .

**Note the count can also be calculated using the formula  $(m-1 + n-1)!/(m-1)!(n-1)!$ . See this for more details.**

This article is contributed by **Hariprasad NG**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-possible-paths-top-left-bottom-right-nxm-matrix/>

Category: Arrays Tags: Dynamic Programming

## Chapter 43

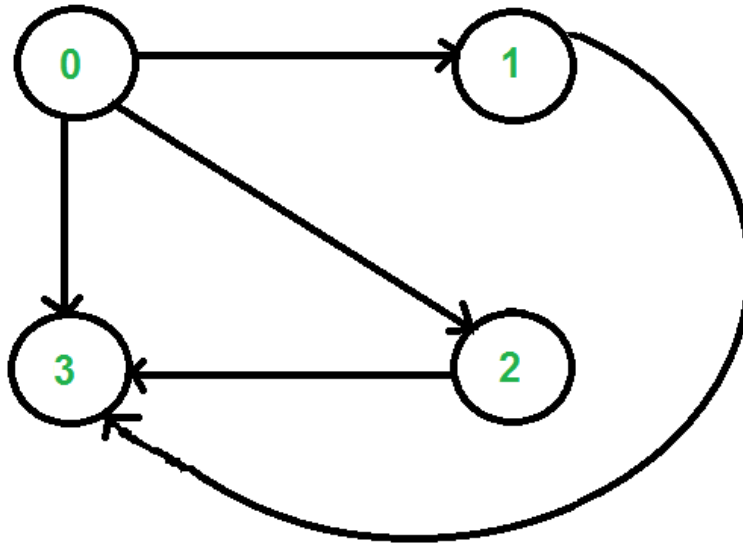
# Count all possible walks from a source to a destination with exactly k edges

Given a directed graph and two vertices 'u' and 'v' in it, count all possible walks from 'u' to 'v' with exactly k edges on the walk.

The graph is given as adjacency matrix representation where value of `graph[i][j]` as 1 indicates that there is an edge from vertex i to vertex j and a value 0 indicates no edge from i to j.

For example consider the following graph. Let source 'u' be vertex 0, destination 'v' be 3 and k be 2. The output should be 2 as there are two walk from 0 to 3 with exactly 2 edges. The walks are {0, 2, 3} and {0, 1, 3}





We strongly recommend to minimize the browser and try this yourself first.

A **simple solution** is to start from  $u$ , go to all adjacent vertices and recur for adjacent vertices with  $k$  as  $k-1$ , source as adjacent vertex and destination as  $v$ . Following is C++ implementation of this simple solution.

```

// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph
#define V 4

// A naive recursive function to count walks from u to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    // Base cases
    if (k == 0 && u == v) return 1;
    if (k == 1 && graph[u][v]) return 1;
    if (k <= 0) return 0;

    // Initialize result
    int count = 0;

```

```

        // Go to all adjacents of u and recur
        for (int i = 0; i < V; i++)
            if (graph[u][i]) // Check if i is adjacent of u
                count += countwalks(graph, i, v, k-1);

        return count;
    }

    // driver program to test above function
    int main()
    {
        /* Let us create the graph shown in above diagram*/
        int graph[V][V] = { {0, 1, 1, 1},
                             {0, 0, 0, 1},
                             {0, 0, 0, 1},
                             {0, 0, 0, 0}
                           };
        int u = 0, v = 3, k = 2;
        cout << countwalks(graph, u, v, k);
        return 0;
    }

```

Output:

2

The worst case time complexity of the above function is  $O(V^k)$  where  $V$  is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly  $n$  children. We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other Dynamic Programming problems, we fill the 3D table in bottom up manner.

```

// C++ program to count walks from u to v with exactly k edges
#include <iostream>
using namespace std;

// Number of vertices in the graph

```

```

#define V 4

// A Dynamic programming based function to count walks from u
// to v with k edges
int countwalks(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value count[i][j][e] will
    // store count of possible walks from i to j with exactly k edges
    int count[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++) // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                count[i][j][e] = 0;

                // from base cases
                if (e == 0 && i == j)
                    count[i][j][e] = 1;
                if (e == 1 && graph[i][j])
                    count[i][j][e] = 1;

                // go to adjacent only when number of edges is more than 1
                if (e > 1)
                {
                    for (int a = 0; a < V; a++) // adjacent of source i
                        if (graph[i][a])
                            count[i][j][e] += count[a][j][e-1];
                }
            }
        }
    }
    return count[u][v][k];
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 1, 1, 1},
                        {0, 0, 0, 1},
                        {0, 0, 0, 1},

```

```

                                {0, 0, 0, 0}
                                };
    int u = 0, v = 3, k = 2;
    cout << countwalks(graph, u, v, k);
    return 0;
}

```

Output:

2

Time complexity of the above DP based solution is  $O(V^3K)$  which is much better than the naive solution.

We can also use **Divide and Conquer** to solve the above problem in  $O(V^3 \log k)$  time. The count of walks of length  $k$  from  $u$  to  $v$  is the  $[u][v]$ 'th entry in  $(\text{graph}[V][V])^k$ . We can calculate power of by doing  $O(\log k)$  multiplication by using the divide and conquer technique to calculate power. A multiplication between two matrices of size  $V \times V$  takes  $O(V^3)$  time. Therefore overall time complexity of this method is  $O(V^3 \log k)$ .

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/count-possible-paths-source-destination-exactly-k-edges/>

Category: Graph Tags: Dynamic Programming

Post navigation

← Amazon Interview | Set 100 (On-Campus) A Problem in Many Binary Search Implementations →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 44

# Count even length binary sequences with same sum of first and second half bits

Given a number  $n$ , find count of all binary sequences of length  $2n$  such that sum of first  $n$  bits is same as sum of last  $n$  bits.

Examples:

```
Input:  n = 1
Output: 2
There are 2 sequences of length  $2*n$ , the
sequences are 00 and 11
```

```
Input:  n = 2
Output: 6
There are 6 sequences of length  $2*n$ , the
sequences are 0101, 0110, 1010, 1001, 0000
and 1111
```

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to fix first and last bits and then recur for  $n-1$ , i.e., remaining  $2(n-1)$  bits. There are following possibilities when we fix first and last bits.

1) First and last bits are same, remaining  $n-1$  bits on both sides should also

have the **same** sum.

2) First bit is 1 and last bit is 0, sum of remaining n-1 bits on left side should be 1 **less** than the sum n-1 bits on right side.

2) First bit is 0 and last bit is 1, sum of remaining n-1 bits on left side should be 1 **more** than the sum n-1 bits on right side.

Based on above facts, we get below recurrence formula.

**diff** is the expected difference between sum of first half digits and last half digits. Initially diff is 0.

```
                // When first and last bits are same
                // there are two cases, 00 and 11
count(n, diff) = 2*count(n-1, diff) +

                // When first bit is 1 and last bit is 0
count(n-1, diff-1) +

                // When first bit is 0 and last bit is 1
count(n-1, diff+1)
```

What should be base cases?

// When n == 1 (2 bit sequences)

1) If n == 1 and diff == 0, return 2

2) If n == 1 and |diff| == 1, return 1

// We can't cover difference of more than n with 2n bits

3) If |diff| > n, return 0

Below is C++ implementation based of above **Naive Recursive Solution**.

```
// A Naive Recursive C++ program to count even
// length binary sequences such that the sum of
// first and second half bits is same
#include<bits/stdc++.h>
using namespace std;

// diff is difference between sums first n bits
// and last n bits respectively
int countSeq(int n, int diff)
{
    // We can't cover difference of more
```

```

        // than n with 2n bits
        if (abs(diff) > n)
            return 0;

        // n == 1, i.e., 2 bit long sequences
        if (n == 1 && diff == 0)
            return 2;
        if (n == 1 && abs(diff) == 1)
            return 1;

        int res = // First bit is 0 & last bit is 1
                  countSeq(n-1, diff+1) +

                  // First and last bits are same
                  2*countSeq(n-1, diff) +

                  // First bit is 1 & last bit is 0
                  countSeq(n-1, diff-1);

        return res;
    }

    // Driver program
    int main()
    {
        int n = 2;
        cout << "Count of sequences is "
              << countSeq(2, 0);
        return 0;
    }

```

Output

```
Count of sequences is 6
```

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from  $n = 4$  and  $\text{diff} = 0$ , we can reach  $(3, 0)$  through multiple paths. Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see this and this) of a **Dynamic Programming** problem. Below is a memoization based solution that uses a lookup table to compute the result.

```

// A memoization based C++ program to count even
// length binary sequences such that the sum of
// first and second half bits is same
#include<bits/stdc++.h>
using namespace std;
#define MAX 1000

// A lookup table to store the results of subproblems
int lookup[MAX][MAX];

// dif is difference between sums of first n bits
// and last n bits i.e., dif = (Sum of first n bits) -
//                               (Sum of last n bits)
int countSeqUtil(int n, int dif)
{
    // We can't cover difference of more
    // than n with 2n bits
    if (abs(dif) > n)
        return 0;

    // n == 1, i.e., 2 bit long sequences
    if (n == 1 && dif == 0)
        return 2;
    if (n == 1 && abs(dif) == 1)
        return 1;

    // Check if this subproblem is already solved
    // n is added to dif to make sure index becomes
    // positive
    if (lookup[n][n+dif] != -1)
        return lookup[n][n+dif];

    int res = // First bit is 0 & last bit is 1
               countSeqUtil(n-1, dif+1) +

               // First and last bits are same
               2*countSeqUtil(n-1, dif) +

               // First bit is 1 & last bit is 0
               countSeqUtil(n-1, dif-1);

    // Store result in lookup table and return the result
    return lookup[n][n+dif] = res;
}

```



```

// A Wrapper over countSeqUtil(). It mainly initializes lookup
// table, then calls countSeqUtil()
int countSeq(int n)
{
    // Initialize all entries of lookup table as not filled
    memset(lookup, -1, sizeof(lookup));

    // call countSeqUtil()
    return countSeqUtil(n, 0);
}

// Driver program
int main()
{
    int n = 2;
    cout << "Count of sequences is "
         << countSeq(2);
    return 0;
}

```

Output

```
Count of sequences is 6
```

Worst case time complexity of this solution is  $O(n^2)$  as diff can be maximum  $n$ .

Below is  **$O(n)$  solution** for the same.

```

Number of n-bit strings with 0 ones =  $nC_0$ 
Number of n-bit strings with 1 ones =  $nC_1$ 
...
Number of n-bit strings with k ones =  $nC_k$ 
...
Number of n-bit strings with n ones =  $nC_n$ 

```

So, we can get required result using below

```

No. of  $2*n$  bit strings such that first  $n$  bits have 0 ones &
last  $n$  bits have 0 ones =  $nC_0 * nC_0$ 

```

No. of  $2^n$  bit strings such that first  $n$  bits have 1 ones & last  $n$  bits have 1 ones =  $nC_1 * nC_1$

....

and so on.

$$\begin{aligned} \text{Result} &= nC_0 * nC_0 + nC_1 * nC_1 + \dots + nC_n * nC_n \\ &= \sum_{k=0}^n (nC_k)^2 \end{aligned}$$

Below is C++ implementation based on above idea.

```
// A O(n) C++ program to count even length binary sequences
// such that the sum of first and second half bits is same
#include<iostream>
using namespace std;

// Returns the count of even length sequences
int countSeq(int n)
{
    int nCr=1, res = 1;

    // Calculate SUM ((nCr)^2)
    for (int r = 1; r<=n ; r++)
    {
        // Compute nCr using nC(r-1)
        // nCr/nC(r-1) = (n+1-r)/r;
        nCr = (nCr * (n+1-r))/r;

        res += nCr*nCr;
    }

    return res;
}

// Driver program
int main()
{
    int n = 3;
    cout << "Count of sequences is "
         << countSeq(n);
    return 0;
}
```

Output

Count of sequences is 6

Thanks to d\_geeks, Saurabh Jain and Mysterious Mind for suggesting above  $O(n)$  solution.

This article is contributed by Pawan. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-even-length-binary-sequences-with-same-sum-of-first-and-second-half-b>

Category: Misc Tags: Dynamic Programming

Post navigation

← 10 mistakes people tend to do in an Interview Adobe Interview Experience | Set 30 (Off-Campus For Member Technical Staff) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 45

# Count number of binary strings without consecutive 1's

Given a positive integer  $N$ , count all possible distinct binary strings of length  $N$  such that there are no consecutive 1's.

Examples:

```
Input:  N = 2
Output: 3
// The 3 strings are 00, 01, 10

Input: N = 3
Output: 5
// The 5 strings are 000, 001, 010, 100, 101
```

This problem can be solved using Dynamic Programming. Let  $a[i]$  be the number of binary strings of length  $i$  which do not contain any two consecutive 1's and which end in 0. Similarly, let  $b[i]$  be the number of such strings which end in 1. We can append either 0 or 1 to a string ending in 0, but we can only append 0 to a string ending in 1. This yields the recurrence relation:

$$\begin{aligned} a[i] &= a[i - 1] + b[i - 1] \\ b[i] &= a[i - 1] \end{aligned}$$

The base cases of above recurrence are  $a[1] = b[1] = 1$ . The total number of strings of length  $i$  is just  $a[i] + b[i]$ .

Following is C++ implementation of above solution. In the following implementation, indexes start from 0. So  $a[i]$  represents the number of binary strings for input length  $i+1$ . Similarly,  $b[i]$  represents binary strings for input length  $i+1$ .

```
// C++ program to count all distinct binary strings
// without two consecutive 1's
#include <iostream>
using namespace std;

int countStrings(int n)
{
    int a[n], b[n];
    a[0] = b[0] = 1;
    for (int i = 1; i < n; i++)
    {
        a[i] = a[i-1] + b[i-1];
        b[i] = a[i-1];
    }
    return a[n-1] + b[n-1];
}

// Driver program to test above functions
int main()
{
    cout << countStrings(3) << endl;
    return 0;
}
```

Output:

5

**Source:**

[courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf](https://courses.csail.mit.edu/6.006/oldquizzes/solutions/q2-f2009-sol.pdf)

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-number-binary-strings-without-consecutive-1s/>

Category: Arrays Tags: Dynamic Programming

## Chapter 46

# Count number of ways to cover a distance

Given a distance 'dist, count total number of ways to cover the distance with 1, 2 and 3 steps.

Examples:

```
Input:  n = 3
Output: 4
Below are the four ways
 1 step + 1 step + 1 step
 1 step + 2 step
 2 step + 1 step
 3 step
```

```
Input:  n = 4
Output: 7
```

**We strongly recommend you to minimize your browser and try this yourself first.**

```
// A naive recursive C++ program to count number of ways to cover
// a distance with 1, 2 and 3 steps
#include<iostream>
using namespace std;
```

```

// Returns count of ways to cover 'dist'
int printCountRec(int dist)
{
    // Base cases
    if (dist<0)    return 0;
    if (dist==0)  return 1;

    // Recur for all previous 3 and add the results
    return printCountRec(dist-1) +
           printCountRec(dist-2) +
           printCountRec(dist-3);
}

// driver program
int main()
{
    int dist = 4;
    cout << printCountRec(dist);
    return 0;
}

```

Output:

7

The time complexity of above solution is exponential, a close upper bound is  $O(3^n)$ . If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from  $n = 6$ , we can reach 4 by subtracting one 2 times and by subtracting 2 one times. So the subproblem for 4 is called twice.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `count[]` in bottom up manner.

Below is Dynamic Programming based C++ implementation.

```

// A Dynamic Programming based C++ program to count number of ways
// to cover a distance with 1, 2 and 3 steps

```



```

#include<iostream>
using namespace std;

int printCountDP(int dist)
{
    int count[dist+1];

    // Initialize base values. There is one way to cover 0 and 1
    // distances and two ways to cover 2 distance
    count[0] = 1, count[1] = 1, count[2] = 2;

    // Fill the count array in bottom up manner
    for (int i=3; i<=dist; i++)
        count[i] = count[i-1] + count[i-2] + count[i-3];

    return count[dist];
}

// driver program
int main()
{
    int dist = 4;
    cout << printCountDP(dist);
    return 0;
}

```

Output: 4

This article is contributed by Vignesh Venkatesan. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-number-of-ways-to-cover-a-distance/>

Category: Misc Tags: Dynamic Programming

## Chapter 47

# Count number of ways to reach a given score in a game

Consider a game where a player can score 3 or 5 or 10 points in a move. Given a total score  $n$ , find number of ways to reach the given score.

Examples:

```
Input: n = 20
Output: 4
There are following 4 ways to reach 20
(10, 10)
(5, 5, 10)
(5, 5, 5, 5)
(3, 3, 3, 3, 3, 5)
```

```
Input: n = 13
Output: 2
There are following 2 ways to reach 13
(3, 5, 5)
(3, 10)
```

We strongly recommend you to minimize the browser and try this yourself first.

This problem is a variation of coin change problem and can be solved in  $O(n)$  time and  $O(n)$  auxiliary space.

The idea is to create a table of size  $n+1$  to store counts of all scores from 0 to  $n$ . For every possible move (3, 5 and 10), increment values in table.

```
// A C program to count number of possible ways to a given score
// can be reached in a game where a move can earn 3 or 5 or 10
#include <stdio.h>

// Returns number of ways to reach score n
int count(int n)
{
    // table[i] will store count of solutions for
    // value i.
    int table[n+1], i;

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // One by one consider given 3 moves and update the table[]
    // values after the index greater than or equal to the
    // value of the picked move
    for (i=3; i<=n; i++)
        table[i] += table[i-3];
    for (i=5; i<=n; i++)
        table[i] += table[i-5];
    for (i=10; i<=n; i++)
        table[i] += table[i-10];

    return table[n];
}

// Driver program
int main(void)
{
    int n = 20;
    printf("Count for %d is %d\n", n, count(n));

    n = 13;
    printf("Count for %d is %d", n, count(n));
}
```

```
        return 0;
    }
```

Output:

```
    Count for 20 is 4
    Count for 13 is 2
```

**Exercise:** How to count score when (10, 5, 5), (5, 5, 10) and (5, 10, 5) are considered as different sequences of moves. Similarly, (5, 3, 3), (3, 5, 3) and (3, 3, 5) are considered different.

This article is contributed by **Rajeev Arora**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-number-ways-reach-given-score-game/>

Category: Misc Tags: Dynamic Programming

## Chapter 48

# Count of n digit numbers whose sum of digits equals to given sum

Given two integers 'n' and 'sum', find count of all n digit numbers with sum of digits as 'sum'. Leading 0's are not counted as digits.

1

Example:

Input: n = 2, sum = 2  
Output: 2  
Explanation: Numbers are 11 and 20

Input: n = 2, sum = 5  
Output: 5  
Explanation: Numbers are 14, 23, 32, 41 and 50

Input: n = 3, sum = 6  
Output: 21

**We strongly recommend you to minimize your browser and try this yourself first**

The idea is simple, we subtract all values from 0 to 9 from given sum and recur for sum minus that digit. Below is recursive formula.

```

countRec(n, sum) = finalCount(n-1, sum-x)
                    where 1 == 0
One important observation is, leading 0's must be
handled explicitly as they are not counted as digits.
So our final count can be written as below.

finalCount(n, sum) = finalCount(n-1, sum-x)
                    where 0 == 0

```

Below is a simple recursive solution based on above recursive formula.

```

// A recursive program to count numbers with sum
// of digits as given 'sum'
#include<bits/stdc++.h>
using namespace std;

// Recursive function to count 'n' digit numbers
// with sum of digits as 'sum'. This function
// considers leading 0's also as digits, that is
// why not directly called
unsigned long long int countRec(int n, int sum)
{
    // Base case
    if (n == 0)
        return sum == 0;

    // Initialize answer
    unsigned long long int ans = 0;

    // Traverse through every digit and count
    // numbers beginning with it using recursion
    for (int i=0; i<=9; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);

    return ans;
}

// This is mainly a wrapper over countRec. It
// explicitly handles leading digit and calls
// countRec() for remaining digits.
unsigned long long int finalCount(int n, int sum)

```

```

{
    // Initialize final answer
    unsigned long long int ans = 0;

    // Traverse through every digit from 1 to
    // 9 and count numbers beginning with it
    for (int i = 1; i <= 9; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);

    return ans;
}

// Driver program
int main()
{
    int n = 2, sum = 5;
    cout << finalCount(n, sum);
    return 0;
}

```

Output:

5

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, if we start with  $n = 3$  and  $\text{sum} = 10$ , we can reach  $n = 1$ ,  $\text{sum} = 8$ , by considering digit sequences 1,1 or 2, 0.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see this and this) of a dynamic programming problem.

Below is Memoization based C++ implementation.

```

// A memoization based recursive program to count
// numbers with sum of n as given 'sum'
#include<bits/stdc++.h>
using namespace std;

// A lookup table used for memoization

```

```

unsigned long long int lookup[101][50001];

// Memoization based implementation of recursive
// function
unsigned long long int countRec(int n, int sum)
{
    // Base case
    if (n == 0)
        return sum == 0;

    // If this subproblem is already evaluated,
    // return the evaluated value
    if (lookup[n][sum] != -1)
        return lookup[n][sum];

    // Initialize answer
    unsigned long long int ans = 0;

    // Traverse through every digit and
    // recursively count numbers beginning
    // with it
    for (int i=0; i<10; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);

    return lookup[n][sum] = ans;
}

// This is mainly a wrapper over countRec. It
// explicitly handles leading digit and calls
// countRec() for remaining n.
unsigned long long int finalCount(int n, int sum)
{
    // Initialize all entries of lookup table
    memset(lookup, -1, sizeof lookup);

    // Initialize final answer
    unsigned long long int ans = 0;

    // Traverse through every digit from 1 to
    // 9 and count numbers beginning with it
    for (int i = 1; i <= 9; i++)
        if (sum-i >= 0)
            ans += countRec(n-1, sum-i);

    return ans;
}

```



```
// Driver program
int main()
{
    int n = 3, sum = 5;
    cout << finalCount(n, sum);
    return 0;
}
```

Output:

5

Thanks to Gaurav Ahirwar for suggesting above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/count-of-n-digit-numbers-whose-sum-of-digits-equals-to-given-sum/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Minimum Initial Points to Reach Destination Microsoft IDC Interview Experience | Set 68 (For SDE) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 49

# Count possible ways to construct buildings

Given an input number of sections and each section has 2 plots on either sides of the road. Find all possible ways to construct buildings in the plots such that there is a space between any 2 buildings.

Example:

N = 1

Output = 4

Place a building on one side.

Place a building on other side

Do not place any building.

Place a building on both sides.

N = 3

Output = 25

3 sections, which means possible ways for one side are BSS, BSB, SSS, SBS, SSB where B represents a building and S represents an empty space

Total possible ways are 25, because a way to place on one side can correspond to any of 5 ways on other side.

N = 4

Output = 64

**We strongly recommend to minimize your browser and try this yourself first**

We can simplify the problem to first calculate for one side only. If we know the result for one side, we can always do square of the result and get result for two sides.

A new building can be placed on a section if section just before it has space. A space can be placed anywhere (it doesn't matter whether the previous section has a building or not).

```
Let countB(i) be count of possible ways with i sections
    ending with a building.
    countS(i) be count of possible ways with i sections
    ending with a space.

// A space can be added after a building or after a space.
countS(N) = countB(N-1) + countS(N-1)

// A building can only be added after a space.
countB[N] = countS(N-1)

// Result for one side is sum of the above two counts.
result1(N) = countS(N) + countB(N)

// Result for two sides is square of result1(N)
result2(N) = result1(N) * result1(N)
```

Below is C++ implementation of above idea.

```
// C++ program to count all possible way to construct buildings
#include<iostream>
using namespace std;

// Returns count of possible ways for N sections
int countWays(int N)
{
    // Base case
    if (N == 1)
        return 4; // 2 for one side and 4 for two sides

    // countB is count of ways with a building at the end
    // countS is count of ways with a space at the end
```

```

// prev_countB and prev_countS are previous values of
// countB and countS respectively.

// Initialize countB and countS for one side
int countB=1, countS=1, prev_countB, prev_countS;

// Use the above recursive formula for calculating
// countB and countS using previous values
for (int i=2; i<=N; i++)
{
    prev_countB = countB;
    prev_countS = countS;

    countS = prev_countB + prev_countS;
    countB = prev_countS;
}

// Result for one side is sum of ways ending with building
// and ending with space
int result = countS + countB;

// Result for 2 sides is square of result for one side
return (result*result);
}

// Driver program
int main()
{
    int N = 3;
    cout << "Count of ways for " << N
         << " sections is " << countWays(N);
    return 0;
}

```

Output:

25

Time complexity:  $O(N)$

Auxiliary Space:  $O(1)$

Algorithmic Paradigm: Dynamic Programming

### Optimized Solution:

Note that the above solution can be further optimized. If we take closer look at the results, for different values, we can notice that the results for two sides are squares of Fibonacci Numbers.

N = 1, result = 4 [result for one side = 2]  
N = 2, result = 9 [result for one side = 3]  
N = 3, result = 25 [result for one side = 5]  
N = 4, result = 64 [result for one side = 8]  
N = 5, result = 169 [result for one side = 13]

.....  
.....

In general, we can say

```
result(N) = fib(N+2)2
```

```
fib(N) is a function that returns N'th  
Fibonacci Number.
```

Therefore, we can use  $O(\log N)$  implementation of Fibonacci Numbers to find number of ways in  $O(\log N)$  time.

This article is contributed by **GOPINATH**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/count-possible-ways-to-construct-buildings/>

## Chapter 50

# Count total number of N digit numbers such that the difference between sum of even and odd digits is 1

Given a number n, we need to count total number of n digit numbers such that the sum of even digits is 1 more than the sum of odd digits. Here even and odd means positions of digits are like array indexes, for example, the leftmost (or leading) digit is considered as even digit, next to leftmost is considered as odd and so on.

Example

Input: n = 2

Output: Required Count of 2 digit numbers is 9

Explanation : 10, 21, 32, 43, 54, 65, 76, 87, 98.

Input: n = 3

Output: Required Count of 3 digit numbers is 54

Explanation: 100, 111, 122, ....., 980

We strongly recommend you to minimize your browser and try this yourself first.

This problem is mainly an extension of Count of n digit numbers whose sum of digits equals to given sum. Here the solution of subproblems depend on four variables: digits, esum (current even sum), osum (current odd sum), isEven(A flag to indicate whether current digit is even or odd).

Below is Memoization based solution for the same.

```
// A memoization based recursive program to count numbers
// with difference between odd and even digit sums as 1
#include<bits/stdc++.h>

using namespace std;

// A lookup table used for memoization.
unsigned long long int lookup[50][1000][1000][2];

// Memoization based recursive function to count numbers
// with even and odd digit sum difference as 1. This function
// considers leading zero as a digit
unsigned long long int countRec(int digits, int esum,
                                int osum, bool isOdd, int n)
{
    // Base Case
    if (digits == n)
        return (esum - osum == 1);

    // If current subproblem is already computed
    if (lookup[digits][esum][osum][isOdd] != -1)
        return lookup[digits][esum][osum][isOdd];

    // Initialize result
    unsigned long long int ans = 0;

    // If current digit is odd, then add it to odd sum and recur
    if (isOdd)
        for (int i = 0; i <= 9; i++)
            ans += countRec(digits+1, esum, osum+i, false, n);
    else // Add to even sum and recur
        for (int i = 0; i <= 9; i++)
            ans += countRec(digits+1, esum+i, osum, true, n);

    // Store current result in lookup table and return the same
    return lookup[digits][esum][osum][isOdd] = ans;
}
```

```

// This is mainly a wrapper over countRec. It
// explicitly handles leading digit and calls
// countRec() for remaining digits.
unsigned long long int finalCount(int n)
{
    // Initialize number digits considered so far
    int digits = 0;

    // Initialize all entries of lookup table
    memset(lookup, -1, sizeof lookup);

    // Initialize final answer
    unsigned long long int ans = 0;

    // Initialize even and odd sums
    int esum = 0, osum = 0;

    // Explicitly handle first digit and call recursive function
    // countRec for remaining digits. Note that the first digit
    // is considered as even digit.
    for (int i = 1; i <= 9; i++)
        ans += countRec(digits+1, esum + i, osum, true, n);

    return ans;
}

// Driver program
int main()
{
    int n = 3;
    cout << "Count of " << n << " digit numbers is " << finalCount(n);
    return 0;
}

```

Output:

```
Count of 3 digit numbers is 54
```

Thanks to Gaurav Ahirwar for providing above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above



## Source

<http://www.geeksforgeeks.org/count-total-number-of-n-digit-numbers-such-that-the-difference-between-the-s>

Category: Misc Tags: Dynamic Programming

Post navigation

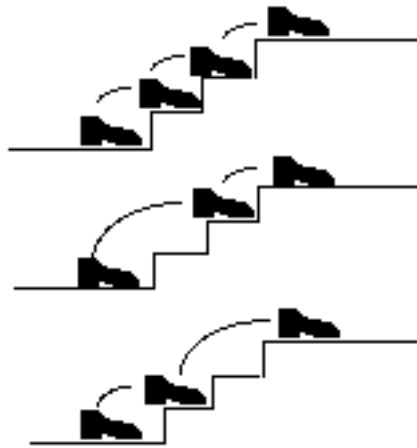
← Intuit Interview | Set 9 (On-Campus) USA's Love for Indian CEOs →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 51

# Count ways to reach the n'th stair

There are  $n$  stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top.



Consider the example shown in diagram. The value of  $n$  is 3. There are 3 ways to reach the top. The diagram is taken from Easier Fibonacci puzzles

### More Examples:

Input:  $n = 1$   
Output: 1  
There is only one way to climb 1 stair

Input:  $n = 2$   
Output: 2  
There are two ways: (1, 1) and (2)

Input:  $n = 4$   
Output: 5  
(1, 1, 1, 1), (1, 1, 2), (2, 1, 1), (1, 2, 1), (2, 2)

We can easily find recursive nature in above problem. The person can reach  $n$ 'th stair from either  $(n-1)$ 'th stair or from  $(n-2)$ 'th stair. Let the total number of ways to reach  $n$ 'th stair be 'ways( $n$ )'. The value of 'ways( $n$ )' can be written as following.

$$\text{ways}(n) = \text{ways}(n-1) + \text{ways}(n-2)$$

The above expression is actually the expression for Fibonacci numbers, but there is one thing to notice, the value of ways( $n$ ) is equal to fibonacci( $n+1$ ).

ways(1) = fib(2) = 1  
ways(2) = fib(3) = 2  
ways(3) = fib(4) = 3

So we can use function for fibonacci numbers to find the value of ways( $n$ ). Following is C++ implementation of the above idea.

```
// A C program to count number of ways to reach n't stair when
// a person can climb 1, 2, ..m stairs at a time.
#include<stdio.h>

// A simple recursive program to find n'th fibonacci number
```

```

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

// Returns number of ways to reach s'th stair
int countWays(int s)
{
    return fib(s + 1);
}

// Driver program to test above functions
int main ()
{
    int s = 4;
    printf("Number of ways = %d", countWays(s));
    getchar();
    return 0;
}

```

Output:

```

    Number of ways = 5

```

The time complexity of the above implementation is exponential (golden ratio raised to power  $n$ ). It can be optimized to work in  $O(\text{Log}n)$  time using the previously discussed Fibonacci function optimizations.

### Generalization of the above problem

How to count number of ways if the person can climb up to  $m$  stairs for a given value  $m$ ? For example if  $m$  is 4, the person can climb 1 stair or 2 stairs or 3 stairs or 4 stairs at a time.

We can write the recurrence as following.

$$\text{ways}(n, m) = \text{ways}(n-1, m) + \text{ways}(n-2, m) + \dots + \text{ways}(n-m, m)$$

Following is C++ implementation of above recurrence.

```

// A C program to count number of ways to reach n't stair when
// a person can climb either 1 or 2 stairs at a time
#include<stdio.h>

// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    if (n <= 1)
        return n;
    int res = 0;
    for (int i = 1; i<=m && i<=n; i++)
        res += countWaysUtil(n-i, m);
    return res;
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}

```

Output:

Number of ways = 5

The time complexity of above solution is exponential. It can be optimized to  $O(mn)$  by using dynamic programming. Following is dynamic programming based solution. We build a table `res[]` in bottom up manner.

```

// A C program to count number of ways to reach n't stair when
// a person can climb 1, 2, ...m stairs at a time
#include<stdio.h>

```

```

// A recursive function used by countWays
int countWaysUtil(int n, int m)
{
    int res[n];
    res[0] = 1; res[1] = 1;
    for (int i=2; i<n; i++)
    {
        res[i] = 0;
        for (int j=1; j<=m && j<=i; j++)
            res[i] += res[i-j];
    }
    return res[n-1];
}

// Returns number of ways to reach s'th stair
int countWays(int s, int m)
{
    return countWaysUtil(s+1, m);
}

// Driver program to test above functions
int main ()
{
    int s = 4, m = 2;
    printf("Nuber of ways = %d", countWays(s, m));
    return 0;
}

```

Output:

Number of ways = 5

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

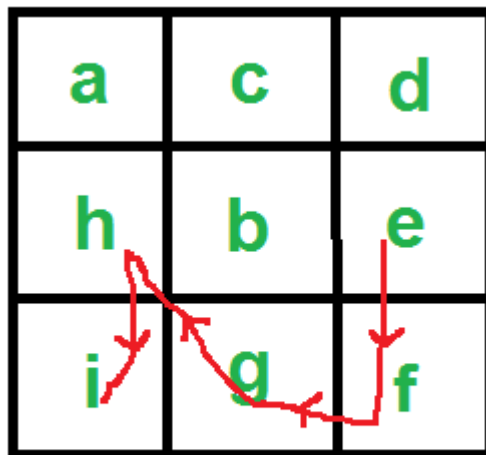
## Source

<http://www.geeksforgeeks.org/count-ways-reach-nth-stair/>

## Chapter 52

### Find length of the longest consecutive path from a given starting character

Given a matrix of characters. Find length of the longest path from a given character, such that all characters in the path are consecutive to each other, i.e., every character in path is next to previous in alphabetical order. It is allowed to move in all 8 directions from a cell.



**Starting Point 'e'**

Example

```

Input: mat[][] = { {a, c, d},
                   {h, b, e},
                   {i, g, f}}
        Starting Point = 'e'

```

Output: 5  
 If starting point is 'e', then longest path with consecutive characters is "e f g h i".

```

Input: mat[R][C] = { {b, e, f},
                     {h, d, a},
                     {i, c, a}};
        Starting Point = 'b'

```

Output: 1  
 'c' is not present in all adjacent cells of 'b'

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is to first search given starting character in the given matrix. Do Depth First Search (DFS) from all occurrences to find all consecutive paths. While doing DFS, we may encounter many subproblems again and again. So we use dynamic programming to store results of subproblems.

Below is C++ implementation of above idea.

```

// C++ program to find the longest consecutive path
#include<bits/stdc++.h>
#define R 3
#define C 3
using namespace std;

// tool matrices to recur for adjacent cells.
int x[] = {0, 1, 1, -1, 1, 0, -1, -1};
int y[] = {1, 0, 1, 1, -1, -1, 0, -1};

// dp[i][j] Stores length of longest consecutive path
// starting at arr[i][j].
int dp[R][C];

// check whether mat[i][j] is a valid cell or not.

```



```

bool isValid(int i, int j)
{
    if (i < 0 || j < 0 || i >= R || j >= C)
        return false;
    return true;
}

// Check whether current character is adjacent to previous
// character (character processed in parent call) or not.
bool isadjacent(char prev, char curr)
{
    return ((curr - prev) == 1);
}

// i, j are the indices of the current cell and prev is the
// character processed in the parent call.. also mat[i][j]
// is our current character.
int getLenUtil(char mat[R][C], int i, int j, char prev)
{
    // If this cell is not valid or current character is not
    // adjacent to previous one (e.g. d is not adjacent to b )
    // or if this cell is already included in the path than return 0.
    if (!isValid(i, j) || !isadjacent(prev, mat[i][j]))
        return 0;

    // If this subproblem is already solved , return the answer
    if (dp[i][j] != -1)
        return dp[i][j];

    int ans = 0; // Initialize answer

    // recur for paths with differnt adjacent cells and store
    // the length of longest path.
    for (int k=0; k<8; k++)
        ans = max(ans, 1 + getLenUtil(mat, i + x[k],
                                      j + y[k], mat[i][j]));

    // save the answer and return
    return dp[i][j] = ans;
}

// Returns length of the longest path with all characters consecutive
// to each other. This function first initializes dp array that
// is used to store results of subproblems, then it calls
// recursive DFS based function getLenUtil() to find max length path
int getLen(char mat[R][C], char s)

```

```

{
    memset(dp, -1, sizeof dp);
    int ans = 0;

    for (int i=0; i<R; i++)
    {
        for (int j=0; j<C; j++)
        {
            // check for each possible starting point
            if (mat[i][j] == s) {

                // recur for all eight adjacent cells
                for (int k=0; k<8; k++)
                    ans = max(ans, 1 + getLenUtil(mat,
                                                    i + x[k], j + y[k], s));
            }
        }
    }
    return ans;
}

// Driver program
int main() {

    char mat[R][C] = { {'a','c','d'},
                        { 'h','b','a'},
                        { 'i','g','f'} };

    cout << getLen(mat, 'a') << endl;
    cout << getLen(mat, 'e') << endl;
    cout << getLen(mat, 'b') << endl;
    cout << getLen(mat, 'f') << endl;
    return 0;
}

```

Output:

```

4
0
3
4

```

Thanks to Gaurav Ahirwar for above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/find-length-of-the-longest-consecutive-path-in-a-character-matrix/>

Category: Arrays Tags: Dynamic Programming, Matrix

Post navigation

← Fiberlink (maas360) Interview Experience | Set 4 (Off-Campus) Goldman Sachs Interview Experience | Set 10 (On-Campus) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 53

# Find minimum number of coins that make a given value

Given a value  $V$ , if we want to make change for  $V$  cents, and we have infinite supply of each of  $C = \{ C_1, C_2, \dots, C_m \}$  valued coins, what is the minimum number of coins to make the change?

Examples:

```
Input: coins[] = {25, 10, 5}, V = 30
Output: Minimum 2 coins required
We can use one coin of 25 cents and one of 5 cents
```

```
Input: coins[] = {9, 6, 5, 1}, V = 11
Output: Minimum 2 coins required
We can use one coin of 6 cents and 1 coin of 5 cents
```

**We strongly recommend you to minimize your browser and try this yourself first.**

This problem is a variation of the problem discussed Coin Change Problem. Here instead of finding total number of possible solutions, we need to find the solution with minimum number of coins.

The minimum number of coins for a value  $V$  can be computed using below recursive formula.

```

If V == 0, then 0 coins required.
If V > 0
    minCoin(coins[0..m-1], V) = min {1 + minCoins(V-coin[i])}
                                where i varies from 0 to m-1
                                and coin[i]

Below is recursive solution based on above recursive formula.

// A Naive recursive C++ program to find minimum of coins
// to make a given change V
#include<bits/stdc++.h>
using namespace std;

// m is size of coins array (number of different coins)
int minCoins(int coins[], int m, int V)
{
    // base case
    if (V == 0) return 0;

    // Initialize result
    int res = INT_MAX;

    // Try every coin that has smaller value than V
    for (int i=0; i<m; i++)
    {
        if (coins[i] <= V)
        {
            int sub_res = minCoins(coins, m, V-coins[i]);

            // Check for INT_MAX to avoid overflow and see if
            // result can minimized
            if (sub_res != INT_MAX && sub_res + 1 < res)
                res = sub_res + 1;
        }
    }
    return res;
}

// Driver program to test above function
int main()
{
    int coins[] = {9, 6, 5, 1};
    int m = sizeof(coins)/sizeof(coins[0]);
    int V = 11;
    cout << "Minimum coins required is "
         << minCoins(coins, m, V);
}

```

```

    return 0;
}

```

Output:

```

Minimum coins required is 2

```

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from  $V = 11$ , we can reach 6 by subtracting one 5 times and by subtracting 5 one times. So the subproblem for 6 is called twice.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So the min coins problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array `table[][]` in bottom up manner. Below is Dynamic Programming based solution.

```

// A Dynamic Programming based C++ program to find minimum of coins
// to make a given change V
#include<bits/stdc++.h>
using namespace std;

// m is size of coins array (number of different coins)
int minCoins(int coins[], int m, int V)
{
    // table[i] will be storing the minimum number of coins
    // required for i value. So table[V] will have result
    int table[V+1];

    // Base case (If given value V is 0)
    table[0] = 0;

    // Initialize all table values as Infinite
    for (int i=1; i<=V; i++)
        table[i] = INT_MAX;

    // Compute minimum coins required for all
    // values from 1 to V
    for (int i=1; i<=V; i++)

```

```

    {
        // Go through all coins smaller than i
        for (int j=0; j<m; j++)
            if (coins[j] <= i)
            {
                int sub_res = table[i-coins[j]];
                if (sub_res != INT_MAX && sub_res + 1 < table[i])
                    table[i] = sub_res + 1;
            }
    }
    return table[V];
}

// Driver program to test above function
int main()
{
    int coins[] = {9, 6, 5, 1};
    int m = sizeof(coins)/sizeof(coins[0]);
    int V = 11;
    cout << "Minimum coins required is "
          << minCoins(coins, m, V);
    return 0;
}

```

Output:

```
Minimum coins required is 2
```

Time complexity of the above solution is  $O(mV)$ .

Thanks to Goku for suggesting above solution in a comment here and thanks to Vignesh Mohan for suggesting this problem and initial solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/find-minimum-number-of-coins-that-make-a-change/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Drishti-Soft Solutions Interview | Set 2 (On-Campus Written) Adobe Interview Experience | Set 23 (1 Year Experienced) →

Writing code in comment? Please use [code.geeksforgeeks.org](https://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 54

# Find minimum possible size of array with given rules for removing elements

Given an array of numbers and a constant  $k$ , minimize size of array with following rules for removing elements.

- Exactly three elements can be removed at one go.
- The removed three elements must be adjacent in array, i.e.,  $\text{arr}[i]$ ,  $\text{arr}[i+1]$ ,  $\text{arr}[i+2]$ . And the second element must be  $k$  greater than first and third element must be  $k$  greater than second, i.e.,  $\text{arr}[i+1] - \text{arr}[i] = k$  and  $\text{arr}[i+2] - \text{arr}[i+1] = k$ .

Example:

```
Input: arr[] = {2, 3, 4, 5, 6, 4}, k = 1
Output: 0
We can actually remove all elements.
First remove 4, 5, 6 => We get {2, 3, 4}
Now remove 2, 3, 4   => We get empty array {}
```

```
Input: arr[] = {2, 3, 4, 7, 6, 4}, k = 1
Output: 3
We can only remove 2 3 4
```

Source: <https://code.google.com/codejam/contest/4214486/dashboard#s=p2>

**We strongly recommend you to minimize your browser and try this yourself first.**

For every element  $arr[i]$  there are two possibilities

- 1) Either the element is not removed.
- 2) OR element is removed (if it follows rules of removal). When an element is removed, there are again two possibilities.
  - ....a) It may be removed directly, i.e., initial  $arr[i+1]$  is  $arr[i]+k$  and  $arr[i+2]$  is  $arr[i] + 2*k$ .
  - ....b) There exist  $x$  and  $y$  such that  $arr[x] - arr[i] = k$ ,  $arr[y] - arr[x] = k$ , and subarrays " $arr[i+1...x-1]$ " & " $arr[x+1...y-1]$ " can be completely removed.

Below is recursive algorithm based on above idea.

```
// Returns size of minimum possible size of arr[low..high]
// after removing elements according to given rules
findMinSize(arr[], low, high, k)

// If there are less than 3 elements in arr[low..high]
1) If high-low+1 2) result = 1 + findMinSize(arr, low+1, high)

// Case when 'arr[low]' is part of some triplet and removed
// Try all possible triplets that have arr[low]
3) For all i from low+1 to high
    For all j from i+1 to high
        Update result if all of the following conditions are met
        a)  $arr[i] - arr[low] = k$ 
        b)  $arr[j] - arr[i] = k$ 
        c)  $findMinSize(arr, low+1, i-1, k)$  returns 0
        d)  $findMinSize(arr, i+1, j-1, k)$  also returns 0
        e) Result calculated for this triplet (low, i, j)
           is smaller than existing result.

4) Return result
```

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array  $dp[][]$  to store results of the subproblems. Below is Dynamic Programming based solution

Below is C++ implementation of above idea. The implementation is memoization based, i.e., it is recursive and uses a lookup table `dp[][]` to check if a subproblem is already solved or not.

```
// C++ program to find size of minimum possible array after
// removing elements according to given rules
#include <bits/stdc++.h>
using namespace std;
#define MAX 1000

// dp[i][j] denotes the minimum number of elements left in
// the subarray arr[i..j].
int dp[MAX][MAX];

int minSizeRec(int arr[], int low, int high, int k)
{
    // If already evaluated
    if (dp[low][high] != -1)
        return dp[low][high];

    // If size of array is less than 3
    if ( (high-low + 1) < 3)
        return high-low +1;

    // Initialize result as the case when first element is
    // separated (not removed using given rules)
    int res = 1 + minSizeRec(arr, low+1, high, k);

    // Now consider all cases when first element forms a triplet
    // and removed. Check for all possible triplets (low, i, j)
    for (int i = low+1; i<=high-1; i++)
    {
        for (int j = i+1; j <= high; j++ )
        {
            // Check if this triplet follows the given rules of
            // removal. And elements between 'low' and 'i' , and
            // between 'i' and 'j' can be recursively removed.
            if (arr[i] == (arr[low] + k) &&
                arr[j] == (arr[low] + 2*k) &&
                minSizeRec(arr, low+1, i-1, k) == 0 &&
                minSizeRec(arr, i+1, j-1, k) == 0)
            {
                res = min(res, minSizeRec(arr, j+1, high, k));
            }
        }
    }
}
```

```

    }

    // Insert value in table and return result
    return (dp[low][high] = res);
}

// This function mainlu initializes dp table and calls
// recursive function minSizeRec
int minSize(int arr[], int n, int k)
{
    memset(dp, -1, sizeof(dp));
    return minSizeRec(arr, 0, n-1, k);
}

// Driver prrogram to test above function
int main()
{
    int arr[] = {2, 3, 4, 5, 6, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    int k = 1;
    cout << minSize(arr, n, k) << endl;
    return 0;
}

```

Output:

0

This article is contributed by Ekta Goel. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/find-minimum-possible-size-of-array-with-given-rules-for-removal/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Microsoft Interview experience | Set 73 (On-Campus for IT SDE Intern)  
Microsoft Interview Experience | Set 74 (For Software Engineer in IT Team) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 55

# Find number of solutions of a linear equation of n variables

Given a linear equation of n variables, find number of non-negative integer solutions of it. For example, let the given equation be " $x + 2y = 5$ ", solutions of this equation are " $x = 1, y = 2$ ", " $x = 5, y = 0$ " and " $x = 0, y = 2.5$ ". It may be assumed that all coefficients in given equation are positive integers.

Example:

```
Input:  coeff[] = {1, 2}, rhs = 5
Output: 3
The equation "x + 2y = 5" has 3 solutions.
(x=3,y=1), (x=1,y=2), (x=5,y=0)
```

```
Input:  coeff[] = {2, 2, 3}, rhs = 4
Output: 3
The equation "2x + 2y + 3z = 4" has 3 solutions.
(x=0,y=2,z=0), (x=2,y=0,z=0), (x=1,y=1,z=0)
```

**We strongly recommend you to minimize your browser and try this yourself first.**

We can solve this problem recursively. The idea is to subtract first coefficient from rhs and then recur for remaining value of rhs.

```

If rhs = 0
    countSol(coeff, 0, rhs, n-1) = 1
Else
    countSol(coeff, 0, rhs, n-1) = &Sum;countSol(coeff, i, rhs-coeff[i], m-1)
                                where coeff[i]
Below is recursive C++ implementation of above solution.

// A naive recursive C++ program to find number of non-negative
// solutions for a given linear equation
#include<bits/stdc++.h>
using namespace std;

// Recursive function that returns count of solutions for given
// rhs value and coefficients coeff[start..end]
int countSol(int coeff[], int start, int end, int rhs)
{
    // Base case
    if (rhs == 0)
        return 1;

    int result = 0; // Initialize count of solutions

    // One by subtract all smaller or equal coefficients and recur
    for (int i=start; i<=end; i++)
        if (coeff[i] <= rhs)
            result += countSol(coeff, i, end, rhs-coeff[i]);

    return result;
}

// Driver program
int main()
{
    int coeff[] = {2, 2, 5};
    int rhs = 4;
    int n = sizeof(coeff)/sizeof(coeff[0]);
    cout << countSol(coeff, 0, n-1, rhs);
    return 0;
}

```

Output:

3

The time complexity of above solution is exponential. We can solve this problem in Pseudo Polynomial Time (time complexity is dependent on numeric value of input) using Dynamic Programming. The idea is similar to Dynamic Programming solution Subset Sum problem. Below is Dynamic Programming based C++ implementation.

```
// A Dynamic programming based C++ program to find number of
// non-negative solutions for a given linear equation
#include<bits/stdc++.h>
using namespace std;

// Returns counr of solutions for given rhs and coefficients
// coeff[0..n-1]
int countSol(int coeff[], int n, int rhs)
{
    // Create and initialize a table to store results of
    // subproblems
    int dp[rhs+1];
    memset(dp, 0, sizeof(dp));
    dp[0] = 1;

    // Fill table in bottom up manner
    for (int i=0; i<n; i++)
        for (int j=coeff[i]; j<=rhs; j++)
            dp[j] += dp[j-coeff[i]];

    return dp[rhs];
}

// Driver program
int main()
{
    int coeff[] = {2, 2, 5};
    int rhs = 4;
    int n = sizeof(coeff)/sizeof(coeff[0]);
    cout << countSol(coeff, n, rhs);
    return 0;
}
```

Output:

Time Complexity of above solution is  $O(n * \text{rhs})$

This article is contributed by **Ashish Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/find-number-of-solutions-of-a-linear-equation-of-n-variables/>

Category: Misc Tags: Dynamic Programming, MathematicalAlgo

Post navigation

← Factset Interview Experience | Set 6 (On-Campus) Dolat Capital Interview experience | Set 1 (On-Campus) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 56

# Find the longest path in a matrix with given constraints

Given a  $n \times n$  matrix where numbers all numbers are distinct and are distributed from range 1 to  $n^2$ , find the maximum length path (starting from any cell) such that all cells along the path are increasing order with a difference of 1.

We can move in 4 directions from a given cell  $(i, j)$ , i.e., we can move to  $(i+1, j)$  or  $(i, j+1)$  or  $(i-1, j)$  or  $(i, j-1)$  with the condition that the adjacent

Example:

```
Input:  mat[] [] = {{1, 2, 9}
                   {5, 3, 8}
                   {4, 6, 7}}
```

```
Output: 4
```

```
The longest path is 6-7-8-9.
```

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is simple, we calculate longest path beginning with every cell. Once we have computed longest for all cells, we return maximum of all longest paths. One important observation in this approach is many overlapping subproblems. Therefore this problem can be optimally solved using Dynamic Programming.

Below is Dynamic Programming based C implementation that uses a lookup table `dp[][]` to check if a problem is already solved or not.

```

#include<bits/stdc++.h>
#define n 3
using namespace std;

// Returns length of the longest path beginning with mat[i][j].
// This function mainly uses lookup table dp[n][n]
int findLongestFromACell(int i, int j, int mat[n][n], int dp[n][n])
{
    // Base case
    if (i<0 || i>=n || j<0 || j>=n)
        return 0;

    // If this subproblem is already solved
    if (dp[i][j] != -1)
        return dp[i][j];

    // Since all numbers are unique and in range from 1 to n*n,
    // there is atmost one possible direction from any cell
    if ((mat[i][j] +1) == mat[i][j+1])
        return dp[i][j] = 1 + findLongestFromACell(i,j+1,mat,dp);

    if (mat[i][j] +1 == mat[i][j-1])
        return dp[i][j] = 1 + findLongestFromACell(i,j-1,mat,dp);

    if (mat[i][j] +1 == mat[i-1][j])
        return dp[i][j] = 1 + findLongestFromACell(i-1,j,mat,dp);

    if (mat[i][j] +1 == mat[i+1][j])
        return dp[i][j] = 1 + findLongestFromACell(i+1,j,mat,dp);

    // If none of the adjacent fours is one greater
    return dp[i][j] = 1;
}

// Returns length of the longest path beginning with any cell
int finLongestOverAll(int mat[n][n])
{
    int result = 1; // Initialize result

    // Create a lookup table and fill all entries in it as -1
    int dp[n][n];
    memset(dp, -1, sizeof dp);

    // Compute longest path beginning from all cells
    for (int i=0; i<n; i++)

```

```

    {
        for (int j=0; j<n; j++)
        {
            if (dp[i][j] == -1)
                findLongestFromACell(i, j, mat, dp);

            // Update result if needed
            result = max(result, dp[i][j]);
        }
    }

    return result;
}

// Driver program
int main()
{
    int mat[n][n] = {{1, 2, 9},
                     {5, 3, 8},
                     {4, 6, 7}};

    cout << "Length of the longest path is "
          << finLongestOverAll(mat);
    return 0;
}

```

Output:

```
Length of the longest path is 4
```

Time complexity of the above solution is  $O(n^2)$ . It may seem more at first look. If we take a closer look, we can notice that all values of  $dp[i][j]$  are computed only once.

This article is contributed by Ekta Goel. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/find-the-longest-path-in-a-matrix-with-given-constraints/>

Category: Arrays Tags: Dynamic Programming, Matrix

Post navigation

← Sort a stack using recursion Paytm Interview Experience | Set 5 (Recruitment Drive) →

Writing code in comment? Please use [code.geeksforgeeks.org](https://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 57

# Find the minimum cost to reach destination using a train

There are N stations on route of a train. The train goes from station 0 to N-1. The ticket cost for all pair of stations (i, j) is given where j is greater than i. Find the minimum cost to reach the destination.

Consider the following example:

```
Input:
cost[N][N] = { {0, 15, 80, 90},
                {INF, 0, 40, 50},
                {INF, INF, 0, 70},
                {INF, INF, INF, 0}
              };
```

There are 4 stations and cost[i][j] indicates cost to reach j from i. The entries where j

We strongly recommend to minimize your browser and try this yourself first.  
The minimum cost to reach N-1 from 0 can be recursively written as following:

```
minCost(0, N-1) = MIN { cost[0][n-1],
                        cost[0][1] + minCost(1, N-1),
                        minCost(0, 2) + minCost(2, N-1),
                        .....,
                        minCost(0, N-2) + cost[N-2][n-1] }
```

The following is C++ implementation of above recursive formula.

```
// A naive recursive solution to find min cost path from station 0
// to station N-1
#include<iostream>
#include<climits>
using namespace std;

// infinite value
#define INF INT_MAX

// Number of stations
#define N 4

// A recursive function to find the shortest path from
// source 's' to destination 'd'.
int minCostRec(int cost[][N], int s, int d)
{
    // If source is same as destination
    // or destination is next to source
    if (s == d || s+1 == d)
        return cost[s][d];

    // Initialize min cost as direct ticket from
    // source 's' to destination 'd'.
    int min = cost[s][d];

    // Try every intermediate vertex to find minimum
    for (int i = s+1; i<d; i++)
    {
        int c = minCostRec(cost, s, i) +
                minCostRec(cost, i, d);
        if (c < min)
            min = c;
    }
    return min;
}

// This function returns the smallest possible cost to
// reach station N-1 from station 0. This function mainly
// uses minCostRec().
int minCost(int cost[][N])
{
    return minCostRec(cost, 0, N-1);
}
```

```

// Driver program to test above function
int main()
{
    int cost[N][N] = { {0, 15, 80, 90},
                        {INF, 0, 40, 50},
                        {INF, INF, 0, 70},
                        {INF, INF, INF, 0}
                      };

    cout << "The Minimum cost to reach station "
          << N << " is " << minCost(cost);
    return 0;
}

```

Output:

```
The Minimum cost to reach station 4 is 65
```

Time complexity of the above implementation is exponential as it tries every possible path from 0 to N-1. The above solution solves same subproblems multiple times (it can be seen by drawing recursion tree for `minCostPathRec(0, 5)`). Since this problem has both properties of dynamic programming problems ((see this and this)). Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

One dynamic programming solution is to create a 2D table and fill the table using above given recursive formula. The extra space required in this solution would be  $O(N^2)$  and time complexity would be  $O(N^3)$ .

We can solve this problem using  $O(N)$  extra space and  $O(N^2)$  time. The idea is based on the fact that given input matrix is a Directed Acyclic Graph (DAG). The shortest path in DAG can be calculated using the approach discussed in below post.

#### Shortest Path in Directed Acyclic Graph

We need to do less work here compared to above mentioned post as we know topological sorting of the graph. The topological sorting of vertices here is 0, 1, ..., N-1. Following is the idea once topological sorting is known.

The idea in below code is to first calculate min cost for station 1, then for station 2, and so on. These costs are stored in an array `dist[0...N-1]`.

- 1) The min cost for station 0 is 0, i.e., `dist[0] = 0`
- 2) The min cost for station 1 is `cost[0][1]`, i.e., `dist[1] = cost[0][1]`

- 3) The min cost for station 2 is minimum of following two.
- a)  $\text{dist}[0] + \text{cost}[0][2]$
  - b)  $\text{dist}[1] + \text{cost}[1][2]$
- 3) The min cost for station 3 is minimum of following three.
- a)  $\text{dist}[0] + \text{cost}[0][3]$
  - b)  $\text{dist}[1] + \text{cost}[1][3]$
  - c)  $\text{dist}[2] + \text{cost}[2][3]$

Similarly,  $\text{dist}[4]$ ,  $\text{dist}[5]$ , ...  $\text{dist}[N-1]$  are calculated.

Below is C++ implementation of above idea.

```
// A Dynamic Programming based solution to find min cost
// to reach station N-1 from station 0.
#include<iostream>
#include<climits>
using namespace std;

#define INF INT_MAX
#define N 4

// This function returns the smallest possible cost to
// reach station N-1 from station 0.
int minCost(int cost[][N])
{
    // dist[i] stores minimum cost to reach station i
    // from station 0.
    int dist[N];
    for (int i=0; i<N; i++)
        dist[i] = INF;
    dist[0] = 0;

    // Go through every station and check if using it
    // as an intermediate station gives better path
    for (int i=0; i<N; i++)
        for (int j=i+1; j<N; j++)
            if (dist[j] > dist[i] + cost[i][j])
                dist[j] = dist[i] + cost[i][j];

    return dist[N-1];
}

// Driver program to test above function
int main()
{
```



```

int cost[N][N] = { {0, 15, 80, 90},
                   {INF, 0, 40, 50},
                   {INF, INF, 0, 70},
                   {INF, INF, INF, 0}
                 };
cout << "The Minimum cost to reach station "
      << N << " is " << minCost(cost);
return 0;
}

```

Output:

```
The Minimum cost to reach station 4 is 65
```

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/find-the-minimum-cost-to-reach-a-destination-where-every-station-is-connected/>

Category: Graph Tags: Dynamic Programming

Post navigation

← Fiberlink (maas360) Interview | Set 2 (Written Test Question) VMWare Interview Experience | Set 2 (On-Campus) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 58

# How to print maximum number of A's using given four keys

This is a famous interview question asked in Google, Paytm and many other company interviews.

Below is the problem statement.

Imagine you have a special keyboard with the following keys:

Key 1: Prints 'A' on screen

Key 2: (Ctrl-A): Select screen

Key 3: (Ctrl-C): Copy selection to buffer

Key 4: (Ctrl-V): Print buffer on screen appending it  
after what has already been printed.

If you can only press the keyboard for N times (with the above four keys), write a program to produce maximum numbers of A's. That is to say, the input parameter is N (No. of keys that you can press), the output is M (No. of As that you can produce).

Examples:

Input: N = 3

Output: 3

We can at most get 3 A's on screen by pressing following key sequence.

A, A, A

Input: N = 7

Output: 9

We can at most get 9 A's on screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V

Input: N = 11

Output: 27

We can at most get 27 A's on screen by pressing following key sequence.

A, A, A, Ctrl A, Ctrl C, Ctrl V, Ctrl V, Ctrl A, Ctrl C, Ctrl V, Ctrl V

**We strongly recommend to minimize your browser and try this yourself first.**

Below are few important points to note.

a) For N 6).

The task is to find out the break=point after which we get the above suffix of keystrokes. Definition of a breakpoint is that instance after which we need to only press Ctrl-A, Ctrl-C once and the only Ctrl-V's afterwards to generate the optimal length. If we loop from N-3 to 1 and choose each of these values for the break-point, and compute that optimal string they would produce. Once the loop ends, we will have the maximum of the optimal lengths for various breakpoints, thereby giving us the optimal length for N keystrokes.

Below is C implementation based on above idea.

```
/* A recursive C program to print maximum number of A's using
   following four keys */
#include<stdio.h>

// A recursive function that returns the optimal length string
// for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;
```

```

// Initialize result
int max = 0;

// TRY ALL POSSIBLE BREAK-POINTS
// For any keystroke N, we need to loop from N-3 keystrokes
// back to 1 keystroke to find a breakpoint 'b' after which we
// will have Ctrl-A, Ctrl-C and then only Ctrl-V all the way.
int b;
for (b=N-3; b>=1; b--)
{
    // If the breakpoint is s at b'th keystroke then
    // the optimal string would have length
    // (n-b-1)*screen[b-1];
    int curr = (N-b-1)*findoptimal(b);
    if (curr > max)
        max = curr;
}
return max;
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}

```

Output:

```

Maximum Number of A's with 1 keystrokes is 1
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12
Maximum Number of A's with 9 keystrokes is 16

```

```

Maximum Number of A's with 10 keystrokes is 20
Maximum Number of A's with 11 keystrokes is 27
Maximum Number of A's with 12 keystrokes is 36
Maximum Number of A's with 13 keystrokes is 48
Maximum Number of A's with 14 keystrokes is 64
Maximum Number of A's with 15 keystrokes is 81
Maximum Number of A's with 16 keystrokes is 108
Maximum Number of A's with 17 keystrokes is 144
Maximum Number of A's with 18 keystrokes is 192
Maximum Number of A's with 19 keystrokes is 256
Maximum Number of A's with 20 keystrokes is 324

```

The above function computes the same subproblems again and again. Recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Below is Dynamic Programming based C implementation where an auxiliary array `screen[N]` is used to store result of subproblems.

```

/* A Dynamic Programming based C program to find maximum number of A's
   that can be printed using four keys */
#include<stdio.h>

// this function returns the optimal length string for N keystrokes
int findoptimal(int N)
{
    // The optimal string length is N when N is smaller than 7
    if (N <= 6)
        return N;

    // An array to store result of subproblems
    int screen[N];

    int b; // To pick a breakpoint

    // Initializing the optimal lengths array for upto 6 input
    // strokes.
    int n;
    for (n=1; n<=6; n++)
        screen[n-1] = n;

    // Solve all subproblems in bottom manner
    for (n=7; n<=N; n++)
    {

```

```

        // Initialize length of optimal string for n keystrokes
        screen[n-1] = 0;

        // For any keystroke n, we need to loop from n-3 keystrokes
        // back to 1 keystroke to find a breakpoint 'b' after which we
        // will have ctrl-a, ctrl-c and then only ctrl-v all the way.
        for (b=n-3; b>=1; b--)
        {
            // if the breakpoint is at b'th keystroke then
            // the optimal string would have length
            // (n-b-1)*screen[b-1];
            int curr = (n-b-1)*screen[b-1];
            if (curr > screen[n-1])
                screen[n-1] = curr;
        }
    }

    return screen[N-1];
}

// Driver program
int main()
{
    int N;

    // for the rest of the array we will rely on the previous
    // entries to compute new ones
    for (N=1; N<=20; N++)
        printf("Maximum Number of A's with %d keystrokes is %d\n",
            N, findoptimal(N));
}

```

Output:

```

Maximum Number of A's with 1 keystrokes is 1
Maximum Number of A's with 2 keystrokes is 2
Maximum Number of A's with 3 keystrokes is 3
Maximum Number of A's with 4 keystrokes is 4
Maximum Number of A's with 5 keystrokes is 5
Maximum Number of A's with 6 keystrokes is 6
Maximum Number of A's with 7 keystrokes is 9
Maximum Number of A's with 8 keystrokes is 12

```

Maximum Number of A's with 9 keystrokes is 16  
Maximum Number of A's with 10 keystrokes is 20  
Maximum Number of A's with 11 keystrokes is 27  
Maximum Number of A's with 12 keystrokes is 36  
Maximum Number of A's with 13 keystrokes is 48  
Maximum Number of A's with 14 keystrokes is 64  
Maximum Number of A's with 15 keystrokes is 81  
Maximum Number of A's with 16 keystrokes is 108  
Maximum Number of A's with 17 keystrokes is 144  
Maximum Number of A's with 18 keystrokes is 192  
Maximum Number of A's with 19 keystrokes is 256  
Maximum Number of A's with 20 keystrokes is 324

Thanks to **Gaurav Saxena** for providing the above approach to solve this problem.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/how-to-print-maximum-number-of-a-using-given-four-keys/>

Category: Misc Tags: Dynamic Programming

Post navigation

← CouponDunia Interview Experience | Set 2 (Fresher) Amazon Interview Experience | Set 179 (For SDE-1) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 59

# Largest Sum Contiguous Subarray

Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

**Kadane's Algorithm:**

```
Initialize:
    max_so_far = 0
    max_ending_here = 0

Loop for each element of the array
    (a) max_ending_here = max_ending_here + a[i]
    (b) if(max_ending_here
```

Explanation:

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array.

Lets take the example:  
{-2, -3, 4, -1, -2, 1, 5, -3}

max\_so\_far = max\_ending\_here = 0

```
for i=0, a[0] = -2
max_ending_here = max_ending_here + (-2)
Set max_ending_here = 0 because max_ending_here
```

Program:



C++

```
// C++ program to print largest contiguous array sum
#include<iostream>
using namespace std;

int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;

    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)
            max_ending_here = 0;
        if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

/*Driver program to test maxSubArraySum*/
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is \n" << max_sum;
    return 0;
}
```

Python

```
# Python program to find maximum contiguous subarray

# Function to find the maximum contiguous subarray
def maxSubArraySum(a,size):

    max_so_far = 0
```

```

max_ending_here = 0

for i in range(0, size):
    max_ending_here = max_ending_here + a[i]
    if max_ending_here < 0:
        max_ending_here = 0

    if (max_so_far < max_ending_here):
        max_so_far = max_ending_here

return max_so_far

# Driver function to check the above function
a = [-2, -3, 4, -1, -2, 1, 5, -3]
print"Maximum contiguous sum is", maxSubArraySum(a,len(a))

#This code is contributed by _Devesh Agrawal_

```

Output:

```
Maximum contiguous sum is 7
```

#### Notes:

Algorithm doesn't work for all negative numbers. It simply returns 0 if all numbers are negative. For handling this we can add an extra phase before actual implementation. The phase will look if all numbers are negative, if they are it will return maximum of them (or smallest in terms of absolute value). There may be other ways to handle it though.

Above program can be optimized further, if we compare max\_so\_far with max\_ending\_here only if max\_ending\_here is greater than 0.

#### C++

```

int maxSubArraySum(int a[], int size)
{
    int max_so_far = 0, max_ending_here = 0;
    for (int i = 0; i < size; i++)
    {
        max_ending_here = max_ending_here + a[i];
        if (max_ending_here < 0)

```

```

        max_ending_here = 0;

        /* Do not compare for all elements. Compare only
        when max_ending_here > 0 */
        else if (max_so_far < max_ending_here)
            max_so_far = max_ending_here;
    }
    return max_so_far;
}

```

## Python

```

def maxSubArraySum(a,size):

    max_so_far = 0
    max_ending_here = 0

    for i in range(0, size):
        max_ending_here = max_ending_here + a[i]
        if max_ending_here < 0:
            max_ending_here = 0

        # Do not compare for all elements. Compare only
        # when max_ending_here > 0
        elif (max_so_far < max_ending_here):
            max_so_far = max_ending_here

    return max_so_far

```

**Time Complexity:**  $O(n)$

**Algorithmic Paradigm:** Dynamic Programming

Following is another simple implementation suggested by **Mohit Kumar**. The implementation handles the case when all numbers in array are negative.

## C++

```

#include<iostream>
using namespace std;

```

```

int maxSubArraySum(int a[], int size)
{
    int max_so_far = a[0];
    int curr_max = a[0];

    for (int i = 1; i < size; i++)
    {
        curr_max = max(a[i], curr_max+a[i]);
        max_so_far = max(max_so_far, curr_max);
    }
    return max_so_far;
}

/* Driver program to test maxSubArraySum */
int main()
{
    int a[] = {-2, -3, 4, -1, -2, 1, 5, -3};
    int n = sizeof(a)/sizeof(a[0]);
    int max_sum = maxSubArraySum(a, n);
    cout << "Maximum contiguous sum is " << max_sum;
    return 0;
}

```

## Python

```

# Python program to find maximum contiguous subarray

def maxSubArraySum(a,size):

    max_so_far =a[0]
    curr_max = a[0]

    for i in range(1,size):
        curr_max = max(a[i], curr_max + a[i])
        max_so_far = max(max_so_far,curr_max)

    return max_so_far

# Driver function to check the above function
a = [-2, -3, 4, -1, -2, 1, 5, -3]
print"Maximum contiguous sum is" , maxSubArraySum(a,len(a))

```

```
#This code is contributed by _Devesh Agrawal_
```

Output:

```
Maximum contiguous sum is 7
```

Now try below question

Given an array of integers (possibly some of the elements negative), write a C program to find out the \*maximum product\* possible by adding 'n' consecutive integers in the array, n

**References:**

[http://en.wikipedia.org/wiki/Kadane%27s\\_Algorithm](http://en.wikipedia.org/wiki/Kadane%27s_Algorithm)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Category: Arrays Tags: Dynamic Programming

Post navigation

← Find the Number Occurring Odd Number of Times Implement Your Own sizeof →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Source

<http://www.geeksforgeeks.org/largest-sum-contiguous-subarray/>

## Chapter 60

# Length of the longest substring without repeating characters

Given a string, find the length of the longest substring without repeating characters. For example, the longest substrings without repeating characters for “ABDEFGABEF” are “BDEFGA” and “DEFGAB”, with length 6. For “BBBB” the longest substring is “B”, with length 1. For “GEEKSFORGEEKS”, there are two longest substrings shown in the below diagrams, with length 7.



The desired time complexity is  $O(n)$  where  $n$  is the length of the string.

### Method 1 (Simple)

We can consider all substrings one by one and check for each substring whether it contains all unique characters or not. There will be  $n*(n+1)/2$  substrings. Whether a substring contains all unique characters or not can be checked in linear time by scanning it from left to right and keeping a map of visited characters. Time complexity of this solution would be  $O(n^3)$ .

### Method 2 (Linear Time)

Let us talk about the linear time solution now. This solution uses extra space to store the last indexes of already visited characters. The idea is to scan the string from left to right, keep track of the maximum length Non-Repeating Character Substring (NRCS) seen so far. Let the maximum length be `max_len`. When we traverse the string, we also keep track of length of the current NRCS using `cur_len` variable. For every new character, we look for it in already processed part of the string (A temp array called `visited[]` is used for this purpose). If it is not present, then we increase the `cur_len` by 1. If present, then there are two cases:

- a) The previous instance of character is not part of current NRCS (The NRCS which is under process). In this case, we need to simply increase `cur_len` by 1.
- b) If the previous instance is part of the current NRCS, then our current NRCS changes. It becomes the substring starting from the next character of previous instance to currently scanned character. We also need to compare `cur_len` and `max_len`, before changing current NRCS (or changing `cur_len`).

### Implementation

```
#include<stdlib.h>
#include<stdio.h>
#define NO_OF_CHARS 256

int min(int a, int b);

int longestUniqueSubsttr(char *str)
{
    int n = strlen(str);
    int cur_len = 1; // To store the length of current substring
    int max_len = 1; // To store the result
    int prev_index; // To store the previous index
    int i;
    int *visited = (int *)malloc(sizeof(int)*NO_OF_CHARS);

    /* Initialize the visited array as -1, -1 is used to indicate that
       character has not been visited yet. */
    for (i = 0; i < NO_OF_CHARS; i++)
        visited[i] = -1;
```

```

/* Mark first character as visited by storing the index of first
   character in visited array. */
visited[str[0]] = 0;

/* Start from the second character. First character is already processed
   (cur_len and max_len are initialized as 1, and visited[str[0]] is set */
for (i = 1; i < n; i++)
{
    prev_index = visited[str[i]];

    /* If the current character is not present in the already processed
       substring or it is not part of the current NRCS, then do cur_len++ */
    if (prev_index == -1 || i - cur_len > prev_index)
        cur_len++;

    /* If the current character is present in currently considered NRCS,
       then update NRCS to start from the next character of previous instance. */
    else
    {
        /* Also, when we are changing the NRCS, we should also check whether
           length of the previous NRCS was greater than max_len or not.*/
        if (cur_len > max_len)
            max_len = cur_len;

        cur_len = i - prev_index;
    }

    visited[str[i]] = i; // update the index of current character
}

// Compare the length of last NRCS with max_len and update max_len if needed
if (cur_len > max_len)
    max_len = cur_len;

free(visited); // free memory allocated for visited

return max_len;
}

/* A utility function to get the minimum of two integers */
int min(int a, int b)
{
    return (a>b)?b:a;
}

```



```

/* Driver program to test above function */
int main()
{
    char str[] = "ABDEFGABEF";
    printf("The input string is %s \n", str);
    int len = longestUniqueSubsttr(str);
    printf("The length of the longest non-repeating character substring is %d", len);

    getchar();
    return 0;
}

```

## Output

```

The input string is ABDEFGABEF
The length of the longest non-repeating character substring is 6

```

**Time Complexity:**  $O(n + d)$  where  $n$  is length of the input string and  $d$  is number of characters in input string alphabet. For example, if string consists of lowercase English characters then value of  $d$  is 26.

**Auxiliary Space:**  $O(d)$

**Algorithmic Paradigm:** Dynamic Programming

As an exercise, try the modified version of the above problem where you need to print the maximum length NRCS also (the above program only prints length of it).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/length-of-the-longest-substring-without-repeating-characters/>

## Chapter 61

# Longest Even Length Substring such that Sum of First and Second Half is same

Given a string 'str' of digits, find length of the longest substring of 'str', such that the length of the substring is 2k digits and sum of left k digits is equal to the sum of right k digits.

Examples:

Input: str = "123123"

Output: 6

The complete string is of even length and sum of first and second half digits is same

Input: str = "1538023"

Output: 4

The longest substring with same first and second half sum is "5380"

### Simple Solution [ $O(n^3)$ ]

A Simple Solution is to check every substring of even length. The following is C based implementation of simple approach.

```

// A simple C based program to find length of longest even length
// substring with same sum of digits in left and right
#include<stdio.h>
#include<string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen =0; // Initialize result

    // Choose starting point of every substring
    for (int i=0; i<n; i++)
    {
        // Choose ending point of even length substring
        for (int j =i+1; j<n; j += 2)
        {
            int length = j-i+1;//Find length of current substr

            // Calculate left & right sums for current substr
            int leftsum = 0, rightsum =0;
            for (int k =0; k<length/2; k++)
            {
                leftsum += (str[i+k]-'0');
                rightsum += (str[i+k+length/2]-'0');
            }

            // Update result if needed
            if (leftsum == rightsum && maxlen < length)
                maxlen = length;
        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "1538023";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}

```

Output:

Length of the substring is 4

### Dynamic Programming [ $O(n^2)$ and $O(n^2)$ extra space]

The above solution can be optimized to work in  $O(n^2)$  using **Dynamic Programming**. The idea is to build a 2D table that stores sums of substrings. The following is C based implementation of Dynamic Programming approach.

```
// A C based program that uses Dynamic Programming to find length of the
// longest even substring with same sum of digits in left and right half
#include <stdio.h>
#include <string.h>

int findLength(char *str)
{
    int n = strlen(str);
    int maxlen = 0; // Initialize result

    // A 2D table where sum[i][j] stores sum of digits
    // from str[i] to str[j]. Only filled entries are
    // the entries where j >= i
    int sum[n][n];

    // Fill the diagonal values for sunstrings of length 1
    for (int i = 0; i < n; i++)
        sum[i][i] = str[i] - '0';

    // Fill entries for substrings of length 2 to n
    for (int len = 2; len <= n; len++)
    {
        // Pick i and j for current substring
        for (int i = 0; i < n - len + 1; i++)
        {
            int j = i + len - 1;
            int k = len / 2;

            // Calculate value of sum[i][j]
            sum[i][j] = sum[i][j - k] + sum[j - k + 1][j];

            // Update result if 'len' is even, left and right
            // sums are same and len is more than maxlen
            if (len % 2 == 0 && sum[i][j - k] == sum[j - k + 1][j]
                && len > maxlen)
                maxlen = len;
        }
    }
}
```

```

        }
    }
    return maxlen;
}

// Driver program to test above function
int main(void)
{
    char str[] = "153803";
    printf("Length of the substring is %d", findLength(str));
    return 0;
}

```

Output:

Length of the substring is 4

Time complexity of the above solution is  $O(n^2)$ , but it requires  $O(n^2)$  extra space.

#### [A $O(n^2)$ and $O(n)$ extra space solution]

The idea is to use a single dimensional array to store cumulative sum.

```

// A  $O(n^2)$  time and  $O(n)$  extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int sum[n+1]; // To store cumulative sum from first digit to nth digit
    sum[0] = 0;

    /* Store cumulative sum of digits from first to last digit */
    for (int i = 1; i <= n; i++)
        sum[i] = (sum[i-1] + str[i-1] - '0'); /* convert chars to int */

    int ans = 0; // initialize result

    /* consider all even length substrings one by one */
    for (int len = 2; len <= n; len += 2)
    {
        for (int i = 0; i <= n-len; i++)

```

```

        {
            int j = i + len - 1;

            /* Sum of first and second half is same than update ans */
            if (sum[i+len/2] - sum[i] == sum[i+len] - sum[i+len/2])
                ans = max(ans, len);
        }
    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
    cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}

```

Output:

```

    Length of the substring is 6

```

Thanks to Gaurav Ahirwar for suggesting this method.

#### [A $O(n^2)$ time and $O(1)$ extra space solution]

The idea is to consider all possible mid points (of even length substrings) and keep expanding on both sides to get and update optimal length as the sum of two sides become equal.

Below is C++ implementation of the above idea.

```

// A  $O(n^2)$  time and  $O(1)$  extra space solution
#include<bits/stdc++.h>
using namespace std;

int findLength(string str, int n)
{
    int ans = 0; // Initialize result

    // Consider all possible midpoints one by one
    for (int i = 0; i <= n-2; i++)

```

```

{
    /* For current midpoint 'i', keep expanding substring on
       both sides, if sum of both sides becomes equal update
       ans */
    int l = i, r = i + 1;

    /* initialize left and right sum */
    int lsum = 0, rsum = 0;

    /* move on both sides till indexes go out of bounds */
    while (r < n && l >= 0)
    {
        lsum += str[l] - '0';
        rsum += str[r] - '0';
        if (lsum == rsum)
            ans = max(ans, r-l+1);
        l--;
        r++;
    }
    return ans;
}

// Driver program to test above function
int main()
{
    string str = "123123";
    cout << "Length of the substring is " << findLength(str, str.length());
    return 0;
}

```

Output:

```
Length of the substring is 6
```

Thanks to Gaurav Ahirwar for suggesting this method.

This article is contributed by **Ashish Bansal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/longest-even-length-substring-sum-first-second-half/>

Category: Strings Tags: Dynamic Programming



## Chapter 62

# Longest Palindromic Substring | Set 1

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 thispost. 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;
}

```

Output:

```

    Longest palindrome substring is: geeksskeeg
    Length is: 10

```

Time complexity:  $O(n^2)$

Auxiliary Space:  $O(n^2)$

We will soon be adding more optimized methods as separate posts.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

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

Category: Strings Tags: Dynamic Programming

## Chapter 63

# Longest Repeating Subsequence

Given a string, find length of the longest repeating subsequence such that the two subsequence don't have same string character at same position, i.e., any i'th character in the two subsequences shouldn't have the same index in the original string.

Examples:

```
Input: str = "abc"
Output: 0
There is no repeating subsequence
```

```
Input: str = "aab"
Output: 1
The two subsequence are 'a'(first) and 'a'(second).
Note that 'b' cannot be considered as part of subsequence
as it would be at same index in bot.
```

```
Input: str = "aabb"
Output: 2
```

```
Input: str = "axxy"
Output: 2
```

We strongly recommend you to minimize your browser and try this yourself first.

This problem is just the modification of Longest Common Subsequence problem. The idea is to find the LCS(str, str) where str is the input string with the restriction that when both the characters are same, they shouldn't be on the same index in the two strings.

Below is C++ implementation of the idea.

```
// C++ program to find the longest repeating
// subsequence
#include <iostream>
#include <string>
using namespace std;

int findLongestRepeatingSubSeq(string str)
{
    int n = str.length();

    // Create and initialize DP table
    int dp[n+1][n+1];
    for (int i=0; i<=n; i++)
        for (int j=0; j<=n; j++)
            dp[i][j] = 0;

    // Fill dp table (similar to LCS loops)
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            // If characters match and indexes are not same
            if (str[i-1] == str[j-1] && i!=j)
                dp[i][j] = 1 + dp[i-1][j-1];
            // If characters do not match
            else
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
        }
    }
    return dp[n][n];
}

// Driver Program
int main()
{
    string str = "aabb";
    cout << "The length of the largest subsequence that"
          " repeats itself is : "
```

```
        << findLongestRepeatingSubSeq(str);  
    return 0;  
}
```

Output:

The length of the largest subsequence that repeats itself is : 2

This article is contributed by Ekta Goel. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/longest-repeating-subsequence/>

Category: Strings Tags: Dynamic Programming

## Chapter 64

# Maximum profit by buying and selling a share at most twice

In a daily share trading, a buyer buys shares in the morning and sells it on same day. If the trader is allowed to make at most 2 transactions in a day, where as second transaction can only start after first one is complete (Sell->buy->sell->buy). Given stock prices throughout day, find out maximum profit that a share trader could have made.

Examples:

```
Input:  price[] = {10, 22, 5, 75, 65, 80}
Output: 87
Trader earns 87 as sum of 12 and 75
Buy at price 10, sell at 22, buy at 5 and sell at 80
```

```
Input:  price[] = {2, 30, 15, 10, 8, 25, 80}
Output: 100
Trader earns 100 as sum of 28 and 72
Buy at price 2, sell at 30, buy at 8 and sell at 80
```

```
Input:  price[] = {100, 30, 15, 10, 8, 25, 80};
Output: 72
Buy at price 8 and sell at 80.
```

```
Input:  price[] = {90, 80, 70, 60, 50}
Output: 0
```



Not possible to earn.

**We strongly recommend to minimize your browser and try this yourself first.**

A **Simple Solution** is to consider every index 'i' and do following

```
Max profit with at most two transactions =  
    MAX {max profit with one transaction and subarray price[0..i] +  
          max profit with one transaction and subarray price[i+1..n-1] }  
    i varies from 0 to n-1.
```

Maximum possible using one transaction can be calculated using following  $O(n)$  algorithm

Maximum difference between two elements such that larger element appears after the smaller number

Time complexity of above simple solution is  $O(n^2)$ .

We can do this  $O(n)$  using following **Efficient Solution**. The idea is to store maximum possible profit of every subarray and solve the problem in following two phases.

- 1) Create a table profit[0..n-1] and initialize all values in it 0.
- 2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]
- 3) Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0..i].
- 4) Return profit[n-1]

To do step 1, we need to keep track of maximum price from right to left side and to do step 2, we need to keep track of minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in second step, we use same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After an iteration i, the array profit[0..i] contains maximum profit with 2 transactions and profit[i+1..n-1] contains profit with two transactions.

Below are implementations of above idea.

C++

```

// C++ program to find maximum possible profit with at most
// two transactions
#include<iostream>
using namespace std;

// Returns maximum profit with two transactions on a given
// list of stock prices, price[0..n-1]
int maxProfit(int price[], int n)
{
    // Create profit array and initialize it as 0
    int *profit = new int[n];
    for (int i=0; i<n; i++)
        profit[i] = 0;

    /* Get the maximum profit with only one transaction
       allowed. After this loop, profit[i] contains maximum
       profit from price[i..n-1] using at most one trans. */
    int max_price = price[n-1];
    for (int i=n-2; i>=0; i--)
    {
        // max_price has maximum of price[i..n-1]
        if (price[i] > max_price)
            max_price = price[i];

        // we can get profit[i] by taking maximum of:
        // a) previous maximum, i.e., profit[i+1]
        // b) profit by buying at price[i] and selling at
        //     max_price
        profit[i] = max(profit[i+1], max_price-price[i]);
    }

    /* Get the maximum profit with two transactions allowed
       After this loop, profit[n-1] contains the result */
    int min_price = price[0];
    for (int i=1; i<n; i++)
    {
        // min_price is minimum price in price[0..i]
        if (price[i] < min_price)
            min_price = price[i];

        // Maximum profit is maximum of:
        // a) previous maximum, i.e., profit[i-1]
        // b) (Buy, Sell) at (min_price, price[i]) and add
        //     profit of other trans. stored in profit[i]
        profit[i] = max(profit[i-1], profit[i] +

```

```

        (price[i]-min_price) );
    }
    int result = profit[n-1];

    delete [] profit; // To avoid memory leak

    return result;
}

// Drive program
int main()
{
    int price[] = {2, 30, 15, 10, 8, 25, 80};
    int n = sizeof(price)/sizeof(price[0]);
    cout << "Maximum Profit = " << maxProfit(price, n);
    return 0;
}

```

## Python

```

# Returns maximum profit with two transactions on a given
# list of stock prices price[0..n-1]
def maxProfit(price,n):

    # Create profit array and initialize it as 0
    profit = [0]*n

    # Get the maximum profit with only one transaction
    # allowed. After this loop, profit[i] contains maximum
    # profit from price[i..n-1] using at most one trans.
    max_price=price[n-1]

    for i in range( n-2, 0 ,-1):

        if price[i]> max_price:
            max_price = price[i]

        # we can get profit[i] by taking maximum of:
        # a) previous maximum, i.e., profit[i+1]
        # b) profit by buying at price[i] and selling at
        #    max_price
        profit[i] = max(profit[i+1], max_price - price[i])

```

```

# Get the maximum profit with two transactions allowed
# After this loop, profit[n-1] contains the result
min_price=price[0]

for i in range(1,n):

    if price[i] < min_price:
        min_price = price[i]

    # Maximum profit is maximum of:
    # a) previous maximum, i.e., profit[i-1]
    # b) (Buy, Sell) at (min_price, A[i]) and add
    #    profit of other trans. stored in profit[i]
    profit[i] = max(profit[i-1], profit[i]+(price[i]-min_price))

result = profit[n-1]

return result

# Driver function
price = [2, 30, 15, 8, 25, 80]
print "Maximum profit is", maxProfit(price, len(price))

# This code is contributed by __Devesh Agrawal__

```

Output:

```
Maximum Profit = 100
```

Time complexity of the above solution is  $O(n)$ .

Algorithmic Paradigm: Dynamic Programming

This article is contributed by **Amit Jaiswal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-twice/>

## Chapter 65

# Maximum size square sub-matrix with all 1s

Given a binary matrix, find out the maximum size square sub-matrix with all 1s.

For example, consider the below binary matrix.

0	1	1	0	1
1	1	0	1	0
0	1	1	1	0
1	1	1	1	0
1	1	1	1	1
0	0	0	0	0

The maximum square sub-matrix with all set bits is

1	1	1
1	1	1
1	1	1

Algorithm:

Let the given binary matrix be  $M[R][C]$ . The idea of the algorithm is to construct an auxiliary size matrix  $S[][]$  in which each entry  $S[i][j]$  represents size of the square sub-matrix with all 1s including  $M[i][j]$  where  $M[i][j]$  is the rightmost and bottommost entry in sub-matrix.

- 1) Construct a sum matrix  $S[R][C]$  for the given  $M[R][C]$ .
  - a) Copy first row and first columns as it is from  $M[][]$  to  $S[][]$
  - b) For other entries, use following expressions to construct  $S[][]$ 

```

          If  $M[i][j]$  is 1 then
               $S[i][j] = \min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1$ 
          Else /*If  $M[i][j]$  is 0*/
               $S[i][j] = 0$ 

```
- 2) Find the maximum entry in  $S[R][C]$
- 3) Using the value and coordinates of maximum entry in  $S[i]$ , print sub-matrix of  $M[][]$

For the given  $M[R][C]$  in above example, constructed  $S[R][C]$  would be:

```

0  1  1  0  1
1  1  0  1  0
0  1  1  1  0
1  1  2  2  0
1  2  2  3  1
0  0  0  0  0

```

The value of maximum entry in above matrix is 3 and coordinates of the entry are (4, 3). Using the maximum value and its coordinates, we can find out the required sub-matrix.

```

#include<stdio.h>
#define bool int
#define R 6
#define C 5

void printMaxSubSquare(bool M[R][C])
{
    int i,j;
    int S[R][C];
    int max_of_s, max_i, max_j;

    /* Set first column of S[][] */
    for(i = 0; i < R; i++)
        S[i][0] = M[i][0];

    /* Set first row of S[][] */

```

```

for(j = 0; j < C; j++)
    S[0][j] = M[0][j];

/* Construct other entries of S[] */
for(i = 1; i < R; i++)
{
    for(j = 1; j < C; j++)
    {
        if(M[i][j] == 1)
            S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1;
        else
            S[i][j] = 0;
    }
}

/* Find the maximum entry, and indexes of maximum entry
   in S[] */
max_of_s = S[0][0]; max_i = 0; max_j = 0;
for(i = 0; i < R; i++)
{
    for(j = 0; j < C; j++)
    {
        if(max_of_s < S[i][j])
        {
            max_of_s = S[i][j];
            max_i = i;
            max_j = j;
        }
    }
}

printf("\n Maximum size sub-matrix is: \n");
for(i = max_i; i > max_i - max_of_s; i--)
{
    for(j = max_j; j > max_j - max_of_s; j--)
    {
        printf("%d ", M[i][j]);
    }
    printf("\n");
}

/* UTILITY FUNCTIONS */
/* Function to get minimum of three values */
int min(int a, int b, int c)
{

```

```

int m = a;
if (m > b)
    m = b;
if (m > c)
    m = c;
return m;
}

/* Driver function to test above functions */
int main()
{
    bool M[R][C] = {{0, 1, 1, 0, 1},
                     {1, 1, 0, 1, 0},
                     {0, 1, 1, 1, 0},
                     {1, 1, 1, 1, 0},
                     {1, 1, 1, 1, 1},
                     {0, 0, 0, 0, 0}};

    printMaxSubSquare(M);
    getchar();
}

```

Time Complexity:  $O(m*n)$  where  $m$  is number of rows and  $n$  is number of columns in the given matrix.

Auxiliary Space:  $O(m*n)$  where  $m$  is number of rows and  $n$  is number of columns in the given matrix.

Algorithmic Paradigm: Dynamic Programming

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

## Source

<http://www.geeksforgeeks.org/maximum-size-sub-matrix-with-all-1s-in-a-binary-matrix/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Print list items containing all characters of a given word Inorder Tree Traversal without Recursion →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 66

# Maximum weight transformation of a given string

Given a string consisting of only A's and B's. We can transform the given string to another string by toggling any character. Thus many transformations of the given string are possible. The task is to find Weight of the maximum weight transformation.

Weight of a sting is calculated using below formula.

$$\text{Weight of string} = \text{Weight of total pairs} + \text{weight of single characters} - \text{Total number of toggles.}$$

Two consecutive characters are considered as pair only if they are different.

Weight of a single pair (both character are different) = 4

Weight of a single character = 1

Examples:

Input: str = "AA"

Output: 3

Transformations of given string are "AA", "AB", "BA" and "BB".  
 Maximum weight transformation is "AB" or "BA". And weight  
 is "One Pair - One Toggle" = 4-1 = 3.

Input: str = "ABB"

Output: 5

Transformations are "ABB", "ABA", "AAB", "AAA", "BBB",  
 "BBA", "BAB" and "BAA"

Maximum weight is of original string 4+1 (One Pair + 1  
 character)

**We strongly recommend you to minimize your browser and try this  
 yourself first.**

We can recursively find maximum weight using below formula.

```

If (n == 1)
    maxWeight(str[0..n-1]) = 1

Else If str[0] != str[1]
    // Max of two cases: First character considered separately
    //                      First pair considered separately
    maxWeight(str[0..n-1]) = Max (1 + maxWeight(str[1..n-1]),
                                4 + getMaxRec(str[2..n-1]))

Else
    // Max of two cases: First character considered separately
    //                      First pair considered separately
    // Since first two characters are same and a toggle is
    // required to form a pair, 3 is added for pair instead
    // of 4
    maxWeight(str[0..n-1]) = Max (1 + maxWeight(str[1..n-1]),
                                3 + getMaxRec(str[2..n-1]))
    
```

If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems.

Below is a memoization based solution. A lookup table is used to see if a problem is already computed.

```

// C++ program to find maximum weight transformation
// of a given string
#include<bits/stdc++.h>
using namespace std;

// Returns wieght of the maximum weight
// transformation
int getMaxRec(string &str, int i, int n, int lookup[])
{
    // Base case
    if (i >= n) return 0;

    //If this subproblem is already solved
    if (lookup[i] != -1) return lookup[i];

    // Don't make pair, so weight gained is 1
    int ans = 1 + getMaxRec(str, i+1, n, lookup);

    // If we can make pair
    if (i+1 < n)
    {
        // If elements are dissimilar, weight gained is 4
        if (str[i] != str[i+1])
            ans = max(4 + getMaxRec(str, i+2, n, lookup),
                    ans);

        // if elements are similar so for making a pair
        // we toggle any of them. Since toggle cost is
        // 1 so overall weight gain becomes 3
        else ans = max(3 + getMaxRec(str, i+2, n, lookup),
                    ans);
    }

    // save and return maximum of above cases
    return lookup[i] = ans;
}

// Initializes lookup table and calls getMaxRec()
int getMaxWeight(string str)
{
    int n = str.length();

    // Create and initialize lookup table
    int lookup[n];
    memset(lookup, -1, sizeof lookup);
}

```

```

        // Call recursive function
        return getMaxRec(str, 0, str.length(), lookup);
    }

    // Driver program
    int main()
    {
        string str = "AAAAABB";
        cout << "Maximum weight of a transformation of "
             << str << " is " << getMaxWeight(str);
        return 0;
    }

```

Output:

```
Maximum weight of a transformation of AAAAABB is 11
```

Thanks to Gaurav Ahirwar for providing above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/maximum-weight-transformation-of-a-given-string/>

Category: Strings Tags: Dynamic Programming

Post navigation

← SAP Labs Interview Experience | Set 11 (On-Campus) Bitmasking and Set 1 (Count ways to assign unique cap to every person) →

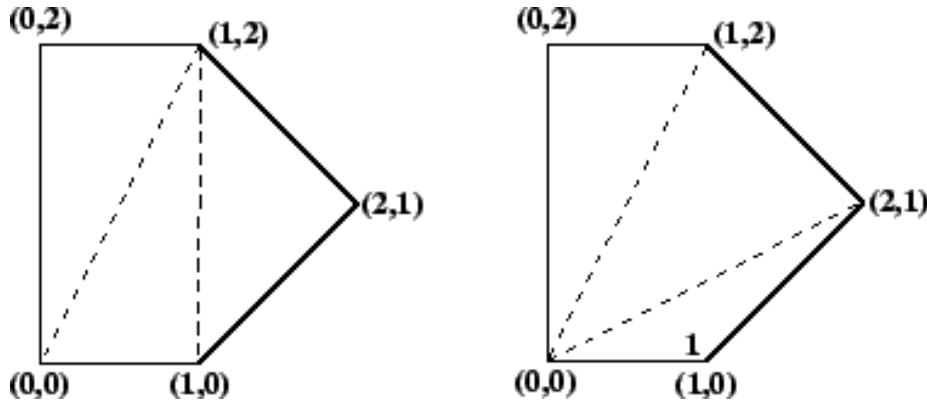
Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 67

# Minimum Cost Polygon Triangulation

A triangulation of a convex polygon is formed by drawing diagonals between non-adjacent vertices (corners) such that the diagonals never intersect. The problem is to find the cost of triangulation with the minimum cost. The cost of a triangulation is sum of the weights of its component triangles. Weight of each triangle is its perimeter (sum of lengths of all sides)

See following example taken from [this source](#).



*Two triangulations of the same convex pentagon. The triangulation on the left has a cost of  $8 + 2\sqrt{2} + 2\sqrt{5}$  (approximately 15.30), the one on the right has a cost of  $4 + 2\sqrt{2} + 4\sqrt{5}$  (approximately 15.77).*

This problem has recursive substructure. The idea is to divide the polygon into three parts: a single triangle, the sub-polygon to the left, and the sub-polygon to the right. We try all possible divisions like this and find the one that minimizes the cost of the triangle plus the cost of the triangulation of the two sub-polygons.

```

Let Minimum Cost of triangulation of vertices from i to j be minCost(i, j)
If j
Following is C++ implementation of above naive recursive formula.

// Recursive implementation for minimum cost convex polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// A recursive function to find minimum cost of polygon triangulation
// The polygon is represented by points[i..j].
double mTC(Point points[], int i, int j)
{
    // There must be at least three points between i and j
    // (including i and j)
    if (j < i+2)
        return 0;

```

```

// Initialize result as infinite
double res = MAX;

// Find minimum triangulation by considering all
for (int k=i+1; k<j; k++)
    res = min(res, (mTC(points, i, k) + mTC(points, k, j) +
                    cost(points, i, k, j)));
return res;
}

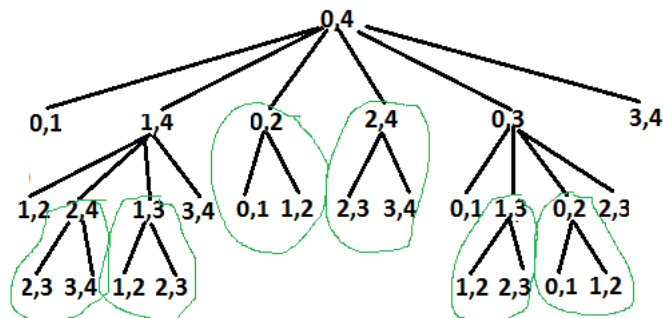
// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTC(points, 0, n-1);
    return 0;
}

```

Output:

15.3006

The above problem is similar to Matrix Chain Multiplication. The following is recursion tree for mTC(points[], 0, 4).



Recursion Tree for recursive implementation. Overlapping subproblems are encircled.

It can be easily seen in the above recursion tree that the problem has many overlapping subproblems. Since the problem has both properties: Optimal Substructure and Overlapping Subproblems, it can be efficiently solved using dynamic programming.

Following is C++ implementation of dynamic programming solution.

```
// A Dynamic Programming based program to find minimum cost of convex
// polygon triangulation
#include <iostream>
#include <cmath>
#define MAX 1000000.0
using namespace std;

// Structure of a point in 2D plane
struct Point
{
    int x, y;
};

// Utility function to find minimum of two double values
double min(double x, double y)
{
    return (x <= y)? x : y;
}

// A utility function to find distance between two points in a plane
double dist(Point p1, Point p2)
{
    return sqrt((p1.x - p2.x)*(p1.x - p2.x) +
                (p1.y - p2.y)*(p1.y - p2.y));
}

// A utility function to find cost of a triangle. The cost is considered
// as perimeter (sum of lengths of all edges) of the triangle
double cost(Point points[], int i, int j, int k)
{
    Point p1 = points[i], p2 = points[j], p3 = points[k];
    return dist(p1, p2) + dist(p2, p3) + dist(p3, p1);
}

// A Dynamic programming based function to find minimum cost for convex
// polygon triangulation.
double mTCDP(Point points[], int n)
{

```



```

// There must be at least 3 points to form a triangle
if (n < 3)
    return 0;

// table to store results of subproblems. table[i][j] stores cost of
// triangulation of points from i to j. The entry table[0][n-1] stores
// the final result.
double table[n][n];

// Fill table using above recursive formula. Note that the table
// is filled in diagonal fashion i.e., from diagonal elements to
// table[0][n-1] which is the result.
for (int gap = 0; gap < n; gap++)
{
    for (int i = 0, j = gap; j < n; i++, j++)
    {
        if (j < i+2)
            table[i][j] = 0.0;
        else
        {
            table[i][j] = MAX;
            for (int k = i+1; k < j; k++)
            {
                double val = table[i][k] + table[k][j] + cost(points,i,j,k);
                if (table[i][j] > val)
                    table[i][j] = val;
            }
        }
    }
}
return table[0][n-1];
}

// Driver program to test above functions
int main()
{
    Point points[] = {{0, 0}, {1, 0}, {2, 1}, {1, 2}, {0, 2}};
    int n = sizeof(points)/sizeof(points[0]);
    cout << mTCDP(points, n);
    return 0;
}

```

Output:

15.3006

Time complexity of the above dynamic programming solution is  $O(n^3)$ .

Please note that the above implementations assume that the points of convex polygon are given in order (either clockwise or anticlockwise)

**Exercise:**

Extend the above solution to print triangulation also. For the above example, the optimal triangulation is 0 3 4, 0 1 3, and 1 2 3.

**Sources:**

<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture12.html>

<http://www.cs.utoronto.ca/~heap/Courses/270F02/A4/chains/node2.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<http://www.geeksforgeeks.org/minimum-cost-polygon-triangulation/>

Category: Misc Tags: Dynamic Programming, geometric algorithms

Post navigation

← D E Shaw Interview | Set 6 (Off-Campus) Find n'th number in a number system with only 3 and 4 →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 68

# Minimum Initial Points to Reach Destination

Given a grid with each cell consisting of positive, negative or no points i.e, zero points. We can move across a cell only if we have positive points ( $> 0$ ). Whenever we pass through a cell, points in that cell are added to our overall points. We need to find minimum initial points to reach cell  $(m-1, n-1)$  from  $(0, 0)$ .

Constraints :

- From a cell  $(i, j)$  we can move to  $(i+1, j)$  or  $(i, j+1)$ .
- We cannot move from  $(i, j)$  if your overall points at  $(i, j)$  is
- We have to reach at  $(n-1, m-1)$  with minimum positive points i.e.,  $> 0$ .

```
Input: points[m][n] = { {-2, -3,  3},
                        {-5, -10, 1},
                        {10,  30, -5}
                      };
```

Output: 7

Explanation:

7 is the minimum value to reach destination with positive throughout the path. Below is the path.

$(0,0) \rightarrow (0,1) \rightarrow (0,2) \rightarrow (1, 2) \rightarrow (2, 2)$

We start from  $(0, 0)$  with 7, we reach  $(0, 1)$  with 5,  $(0, 2)$  with 2,  $(1, 2)$  with 5,  $(2, 2)$  with and finally we have 1 point (we needed

greater than 0 points at the end).

**We strongly recommend you to minimize your browser and try this yourself first.**

At the first look, this problem looks similar Max/Min Cost Path, but maximum overall points gained will not guarantee the minimum initial points. Also, it is compulsory in the current problem that the points never drops to zero or below. For instance, Suppose following two paths exists from source to destination cell.

We can solve this problem through bottom-up table filling dynamic programming technique.

- To begin with, we should maintain a 2D array `dp` of the same size as the grid, where `dp[i][j]` represents the minimum points that guarantees the continuation of the journey to destination before entering the cell  $(i, j)$ . It's but obvious that `dp[0][0]` is our final solution. Hence, for this problem, we need to fill the table from the bottom right corner to left top.
- Now, let us decide minimum points needed to leave cell  $(i, j)$  (remember we are moving from bottom to up). There are only two paths to choose:  $(i+1, j)$  and  $(i, j+1)$ . Of course we will choose the cell that the player can finish the rest of his journey with a smaller initial points. Therefore we have: **`min_Points_on_exit = min(dp[i+1][j], dp[i][j+1])`**

Now we know how to compute `min_Points_on_exit`, but we need to fill the table `dp[][]` to get the solution in `dp[0][0]`.

**How to compute `dp[i][j]`?**

The value of `dp[i][j]` can be written as below.

$$dp[i][j] = \max(\text{min\_Points\_on\_exit} - \text{points}[i][j], 1)$$

Let us see how above expression covers all cases.

- If `points[i][j] == 0`, then nothing is gained in this cell; the player can leave the cell with the same points as he enters the room with, i.e. `dp[i][j] = min_Points_on_exit`.
- If `dp[i][j]` If `dp[i][j] > 0`, then the player could enter  $(i, j)$  with points as little as `min_Points_on_exit - points[i][j]`. since he could gain “`points[i][j]`” points in this cell. However, the value of `min_Points_on_exit - points[i][j]` might drop to 0 or below in this situation. When this happens, we must clip the value to 1 in order to make sure `dp[i][j]` stays positive:  
**`dp[i][j] = max(min_Points_on_exit - points[i][j], 1)`**.

Finally return `dp[0][0]` which is our answer.

Below is C++ implementation of above algorithm.

```

// C++ program to find minimum initial points to reach destination
#include<bits/stdc++.h>
#define R 3
#define C 3
using namespace std;

int minInitialPoints(int points[][C])
{
    // dp[i][j] represents the minimum initial points player
    // should have so that when starts with cell(i, j) successfully
    // reaches the destination cell(m-1, n-1)
    int dp[R][C];
    int m = R, n = C;

    // Base case
    dp[m-1][n-1] = points[m-1][n-1] > 0? 1:
        abs(points[m-1][n-1]) + 1;

    // Fill last row and last column as base to fill
    // entire table
    for (int i = m-2; i >= 0; i--)
        dp[i][n-1] = max(dp[i+1][n-1] - points[i][n-1], 1);
    for (int j = n-2; j >= 0; j--)
        dp[m-1][j] = max(dp[m-1][j+1] - points[m-1][j], 1);

    // fill the table in bottom-up fashion
    for (int i=m-2; i>=0; i--)
    {
        for (int j=n-2; j>=0; j--)
        {
            int min_points_on_exit = min(dp[i+1][j], dp[i][j+1]);
            dp[i][j] = max(min_points_on_exit - points[i][j], 1);
        }
    }

    return dp[0][0];
}

// Driver Program
int main()
{
    int points[R][C] = { {-2,-3,3},
                          {-5,-10,1},
                          {10,30,-5}
    };
}

```

```
        };\n        cout << "Minimum Initial Points Required: "\n              << minInitialPoints(points);\n        return 0;\n    }\n}
```

Output:

```
Minimum Initial Points Required: 7
```

This article is contributed by Gaurav Ahirwar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/minimum-positive-points-to-reach-destination/>

Category: Arrays Tags: Dynamic Programming, Matrix

Post navigation

← Amazon Interview Experience | Set 227 (On-Campus for Internship and Full-Time) Count of n digit numbers whose sum of digits equals to given sum →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 69

# Minimum number of jumps to reach end

Given an array of integers where each element represents the max number of steps that can be made forward from that element. Write a function to return the minimum number of jumps to reach the end of the array (starting from the first element). If an element is 0, then cannot move through that element.

Example:

```
Input: arr[] = {1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9}
Output: 3 (1-> 3 -> 8 ->9)
```

First element is 1, so can only go to 3. Second element is 3, so can make at most 3 steps eg to 5 or 8 or 9.

### Method 1 (Naive Recursive Approach)

A naive approach is to start from the first element and recursively call for all the elements reachable from first element. The minimum number of jumps to reach end from first can be calculated using minimum number of jumps needed to reach end from the elements reachable from first.

$minJumps(start, end) = Min ( minJumps(k, end) )$  for all  $k$  reachable from  $start$

```
#include <stdio.h>
#include <limits.h>
```

```

// Returns minimum number of jumps to reach arr[h] from arr[l]
int minJumps(int arr[], int l, int h)
{
    // Base case: when source and destination are same
    if (h == l)
        return 0;

    // When nothing is reachable from the given source
    if (arr[l] == 0)
        return INT_MAX;

    // Traverse through all the points reachable from arr[l]. Recursively
    // get the minimum number of jumps needed to reach arr[h] from these
    // reachable points.
    int min = INT_MAX;
    for (int i = l+1; i <= h && i <= l + arr[l]; i++)
    {
        int jumps = minJumps(arr, i, h);
        if(jumps != INT_MAX && jumps + 1 < min)
            min = jumps + 1;
    }

    return min;
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 3, 2, 3, 6, 8, 9, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr, 0, n-1));
    return 0;
}

```

If we trace the execution of this method, we can see that there will be overlapping subproblems. For example, `minJumps(3, 9)` will be called two times as `arr[3]` is reachable from `arr[1]` and `arr[2]`. So this problem has both properties (optimal substructure and overlapping subproblems) of Dynamic Programming.

### Method 2 (Dynamic Programming)

In this method, we build a `jumps[]` array from left to right such that `jumps[i]` indicates the minimum number of jumps needed to reach `arr[i]` from `arr[0]`. Finally, we return `jumps[n-1]`.



```

#include <stdio.h>
#include <limits.h>

int min(int x, int y) { return (x < y)? x: y; }

// Returns minimum number of jumps to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    int *jumps = new int[n]; // jumps[n-1] will hold the result
    int i, j;

    if (n == 0 || arr[0] == 0)
        return INT_MAX;

    jumps[0] = 0;

    // Find the minimum number of jumps to reach arr[i]
    // from arr[0], and assign this value to jumps[i]
    for (i = 1; i < n; i++)
    {
        jumps[i] = INT_MAX;
        for (j = 0; j < i; j++)
        {
            if (i <= j + arr[j] && jumps[j] != INT_MAX)
            {
                jumps[i] = min(jumps[i], jumps[j] + 1);
                break;
            }
        }
    }
    return jumps[n-1];
}

// Driver program to test above function
int main()
{
    int arr[] = {1, 3, 6, 1, 0, 9};
    int size = sizeof(arr)/sizeof(int);
    printf("Minimum number of jumps to reach end is %d ", minJumps(arr,size));
    return 0;
}

```

Output:

Minimum number of jumps to reach end is 3

Thanks to parasfor suggesting this method.

Time Complexity:  $O(n^2)$

### Method 3 (Dynamic Programming)

In this method, we build jumps[] array from right to left such that jumps[i] indicates the minimum number of jumps needed to reach arr[n-1] from arr[i]. Finally, we return arr[0].

```
int minJumps(int arr[], int n)
{
    int *jumps = new int[n]; // jumps[0] will hold the result
    int min;

    // Minimum number of jumps needed to reach last element
    // from last elements itself is always 0
    jumps[n-1] = 0;

    int i, j;

    // Start from the second element, move from right to left
    // and construct the jumps[] array where jumps[i] represents
    // minimum number of jumps needed to reach arr[m-1] from arr[i]
    for (i = n-2; i >=0; i--)
    {
        // If arr[i] is 0 then arr[n-1] can't be reached from here
        if (arr[i] == 0)
            jumps[i] = INT_MAX;

        // If we can directly reach to the end point from here then
        // jumps[i] is 1
        else if (arr[i] >= n - i - 1)
            jumps[i] = 1;

        // Otherwise, to find out the minimum number of jumps needed
        // to reach arr[n-1], check all the points reachable from here
        // and jumps[] value for those points
        else
        {
            min = INT_MAX; // initialize min value

            // following loop checks with all reachable points and
```

```

        // takes the minimum
        for (j = i+1; j < n && j <= arr[i] + i; j++)
        {
            if (min > jumps[j])
                min = jumps[j];
        }

        // Handle overflow
        if (min != INT_MAX)
            jumps[i] = min + 1;
        else
            jumps[i] = min; // or INT_MAX
    }

    return jumps[0];
}

```

Time Complexity:  $O(n^2)$  in worst case.

Thanks to Ashishfor suggesting this solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/minimum-number-of-jumps-to-reach-end-of-a-given-array/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Count smaller elements on right side Populate Inorder Successor for all nodes  
→

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 70

# Minimum number of squares whose sum equals to given number n

A number can always be represented as a sum of squares of other numbers. Note that 1 is a square and we can always break a number as  $(1*1 + 1*1 + 1*1 + \dots)$ . Given a number n, find the minimum number of squares that sum to X.

Examples:

Input: n = 100

Output: 1

100 can be written as 102. Note that 100 can also be written as  $52 + 52 + 52 + 52$ , but this representation requires 4 squares.

Input: n = 6

Output: 3

**We strongly recommend you to minimize your browser and try this yourself first.**

The idea is simple, we start from 1 and go till a number whose square is smaller than or equals to n. For every number x, we recur for n-x. Below is the recursive formula.

If  $n = 1$  and  $x \times x$

Below is a simple recursive solution based on above recursive formula.

```
// A naive recursive C++ program to find minimum
// number of squares whose sum is equal to a given number
#include<bits/stdc++.h>
using namespace std;

// Returns count of minimum squares that sum to n
int getMinSquares(unsigned int n)
{
    // base cases
    if (n <= 3)
        return n;

    // getMinSquares rest of the table using recursive
    // formula
    int res = n; // Maximum squares required is n (1*1 + 1*1 + ..)

    // Go through all smaller numbers
    // to recursively find minimum
    for (int x = 1; x <= n; x++)
    {
        int temp = x*x;
        if (temp > n)
            break;
        else
            res = min(res, 1+getMinSquares(n - temp));
    }
    return res;
}

// Driver program
int main()
{
    cout << getMinSquares(6);
    return 0;
}
```

Output:

3

The time complexity of above solution is exponential. If we draw the complete recursion tree, we can observe that many subproblems are solved again and again. For example, when we start from  $n = 6$ , we can reach 4 by subtracting one 2 times and by subtracting 2 one times. So the subproblem for 4 is called twice.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So min square sum problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, recomputations of same subproblems can be avoided by constructing a temporary array table[] in bottom up manner. Below is Dynamic Programming based solution

```
// A dynamic programming based C++ program to find minimum
// number of squares whose sum is equal to a given number
#include<bits/stdc++.h>
using namespace std;

// Returns count of minimum squares that sum to n
int getMinSquares(int n)
{
    // Create a dynamic programming table
    // to store sq
    int *dp = new int[n+1];

    // getMinSquares table for base case entries
    dp[0] = 0;
    dp[1] = 1;
    dp[2] = 2;
    dp[3] = 3;

    // getMinSquares rest of the table using recursive
    // formula
    for (int i = 4; i <= n; i++)
    {
        // max value is i as i can always be represented
        // as 1*1 + 1*1 + ...
        dp[i] = i;

        // Go through all smaller numbers to
        // to recursively find minimum
        for (int x = 1; x <= i; x++) {
            int temp = x*x;
            if (temp > i)
                break;
            else dp[i] = min(dp[i], 1+dp[i-temp]);
        }
    }
}
```

```

        }
    }

    // Store result and free dp[]
    int res = dp[n];
    delete [] dp;

    return res;
}

// Driver program
int main()
{
    cout << getMinSquares(6);
    return 0;
}

```

Output:

3

Thanks to Gaurav Ahirwar for suggesting this solution in a comment here.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/minimum-number-of-squares-whose-sum-equals-to-given-number-n/>

Category: Misc Tags: Dynamic Programming, MathematicalAlgo

## Chapter 71

# Mobile Numeric Keypad Problem



Given the mobile numeric keypad. You can only press buttons that are up, left, right or down to the current button. You are not allowed to press bottom row corner buttons (i.e. \* and # ). Given a number N, find out the number of possible numbers of given length.

Examples:

For N=1, number of possible numbers would be 10 (0, 1, 2, 3, ..., 9)

For N=2, number of possible numbers would be 36

Possible numbers: 00,08 11,12,14 22,21,23,25 and so on.

If we start with 0, valid numbers will be 00, 08 (count: 2)

If we start with 1, valid numbers will be 11, 12, 14 (count: 3)

If we start with 2, valid numbers will be 22, 21, 23,25 (count: 4)

If we start with 3, valid numbers will be 33, 32, 36 (count: 3)

If we start with 4, valid numbers will be 44,41,45,47 (count: 4)

If we start with 5, valid numbers will be 55,54,52,56,58 (count: 5)

.....



.....

We need to print the count of possible numbers.

**We strongly recommend to minimize the browser and try this yourself first.**

$N = 1$  is trivial case, number of possible numbers would be 10 (0, 1, 2, 3, ..., 9)  
For  $N > 1$ , we need to start from some button, then move to any of the four direction (up, left, right or down) which takes to a valid button (should not go to \*, #). Keep doing this until  $N$  length number is obtained (depth first traversal).

**Recursive Solution:**

Mobile Keypad is a rectangular grid of 4X3 (4 rows and 3 columns)

Lets say  $\text{Count}(i, j, N)$  represents the count of  $N$  length numbers starting from position  $(i, j)$

```
If N = 1
    Count(i, j, N) = 10
Else
    Count(i, j, N) = Sum of all Count(r, c, N-1) where (r, c) is new
                    position after valid move of length 1 from current
                    position (i, j)
```

Following is C implementation of above recursive formula.

```
// A Naive Recursive C program to count number of possible numbers
// of given length
#include <stdio.h>

// left, up, right, down move from current location
int row[] = {0, 0, -1, 0, 1};
int col[] = {0, -1, 0, 1, 0};

// Returns count of numbers of length n starting from key position
// (i, j) in a numeric keyboard.
int getCountUtil(char keypad[][3], int i, int j, int n)
{
    if (keypad == NULL || n <= 0)
        return 0;

    // From a given key, only one number is possible of length 1
    if (n == 1)
```

```

        return 1;

    int k=0, move=0, ro=0, co=0, totalCount = 0;

    // move left, up, right, down from current location and if
    // new location is valid, then get number count of length
    // (n-1) from that new position and add in count obtained so far
    for (move=0; move<5; move++)
    {
        ro = i + row[move];
        co = j + col[move];
        if (ro >= 0 && ro <= 3 && co >=0 && co <= 2 &&
            keypad[ro][co] != '*' && keypad[ro][co] != '#')
        {
            totalCount += getCountUtil(keypad, ro, co, n-1);
        }
    }

    return totalCount;
}

// Return count of all possible numbers of length n
// in a given numeric keyboard
int getCount(char keypad[][3], int n)
{
    // Base cases
    if (keypad == NULL || n <= 0)
        return 0;
    if (n == 1)
        return 10;

    int i=0, j=0, totalCount = 0;
    for (i=0; i<4; i++) // Loop on keypad row
    {
        for (j=0; j<3; j++) // Loop on keypad column
        {
            // Process for 0 to 9 digits
            if (keypad[i][j] != '*' && keypad[i][j] != '#')
            {
                // Get count when number is starting from key
                // position (i, j) and add in count obtained so far
                totalCount += getCountUtil(keypad, i, j, n);
            }
        }
    }
    return totalCount;
}

```

```

}

// Driver program to test above function
int main(int argc, char *argv[])
{
    char keypad[4][3] = {{ '1', '2', '3'},
                        { '4', '5', '6'},
                        { '7', '8', '9'},
                        { '*', '0', '#' } };

    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad, 1));
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad, 2));
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad, 3));
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad, 4));
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad, 5));

    return 0;
}

```

Output:

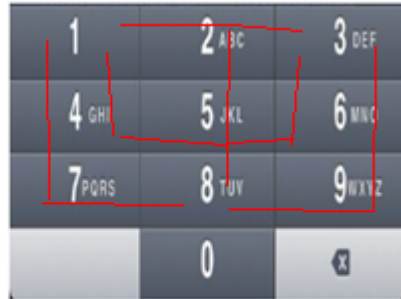
```

Count for numbers of length 1: 10
Count for numbers of length 2: 36
Count for numbers of length 3: 138
Count for numbers of length 4: 532
Count for numbers of length 5: 2062

```

### Dynamic Programming

There are many repeated traversals on smaller paths (traversal for smaller N) to find all possible longer paths (traversal for bigger N). See following two diagrams for example. In this traversal, for  $N = 4$  from two starting positions (buttons '4' and '8'), we can see there are few repeated traversals for  $N = 2$  (e.g. 4 -> 1, 6 -> 3, 8 -> 9, 8 -> 7 etc).

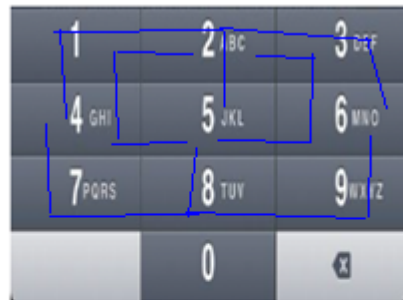


Few traversals starting for button 8 for N = 4

e.g. 8 -> 7 -> 4 -> 1, 8 -> 9 -> 6 -> 3

8 -> 5 -> 4 -> 1, 8 -> 5 -> 6 -> 3

8 -> 5 -> 2 -> 2, 8 -> 5 -> 2 -> 3



Few traversals starting from button 5 for N= 4

e.g. 5 -> 8 -> 7 -> 4, 5 -> 8 -> 9 -> 6

5 -> 4 -> 1 -> 2, 5 -> 6 -> 3 -> 2

5 -> 2 -> 1 -> 4, 5 -> 2 -> 3 -> 6

Since the problem has both properties: Optimal Substructure and Overlapping Subproblems, it can be efficiently solved using dynamic programming.

Following is C program for dynamic programming implementation.

```
// A Dynamic Programming based C program to count number of
// possible numbers of given length
#include <stdio.h>

// Return count of all possible numbers of length n
// in a given numeric keyboard
```

```

int getCount(char keypad[][3], int n)
{
    if(keypad == NULL || n <= 0)
        return 0;
    if(n == 1)
        return 10;

    // left, up, right, down move from current location
    int row[] = {0, 0, -1, 0, 1};
    int col[] = {0, -1, 0, 1, 0};

    // taking n+1 for simplicity - count[i][j] will store
    // number count starting with digit i and length j
    int count[10][n+1];
    int i=0, j=0, k=0, move=0, ro=0, co=0, num = 0;
    int nextNum=0, totalCount = 0;

    // count numbers starting with digit i and of lengths 0 and 1
    for (i=0; i<=9; i++)
    {
        count[i][0] = 0;
        count[i][1] = 1;
    }

    // Bottom up - Get number count of length 2, 3, 4, ... , n
    for (k=2; k<=n; k++)
    {
        for (i=0; i<4; i++) // Loop on keypad row
        {
            for (j=0; j<3; j++) // Loop on keypad column
            {
                // Process for 0 to 9 digits
                if (keypad[i][j] != '*' && keypad[i][j] != '#')
                {
                    // Here we are counting the numbers starting with
                    // digit keypad[i][j] and of length k keypad[i][j]
                    // will become 1st digit, and we need to look for
                    // (k-1) more digits
                    num = keypad[i][j] - '0';
                    count[num][k] = 0;

                    // move left, up, right, down from current location
                    // and if new location is valid, then get number
                    // count of length (k-1) from that new digit and
                    // add in count we found so far
                    for (move=0; move<5; move++)

```

```

        {
            ro = i + row[move];
            co = j + col[move];
            if (ro >= 0 && ro <= 3 && co >= 0 && co <= 2 &&
                keypad[ro][co] != '*' && keypad[ro][co] != '#')
            {
                nextNum = keypad[ro][co] - '0';
                count[num][k] += count[nextNum][k-1];
            }
        }
    }
}

// Get count of all possible numbers of length "n" starting
// with digit 0, 1, 2, ..., 9
totalCount = 0;
for (i=0; i<=9; i++)
    totalCount += count[i][n];
return totalCount;
}

// Driver program to test above function
int main(int argc, char *argv[])
{
    char keypad[4][3] = {{ '1', '2', '3' },
                        { '4', '5', '6' },
                        { '7', '8', '9' },
                        { '*', '0', '#' } };

    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad, 1));
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad, 2));
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad, 3));
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad, 4));
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad, 5));

    return 0;
}

```

Output:

```

Count for numbers of length 1: 10
Count for numbers of length 2: 36

```

```

Count for numbers of length 3: 138
Count for numbers of length 4: 532
Count for numbers of length 5: 2062

```

### A Space Optimized Solution:

The above dynamic programming approach also runs in  $O(n)$  time and requires  $O(n)$  auxiliary space, as only one for loop runs  $n$  times, other for loops runs for constant time. We can see that  $n$ th iteration needs data from  $(n-1)$ th iteration only, so we need not keep the data from older iterations. We can have a space efficient dynamic programming approach with just two arrays of size 10. Thanks to Nik for suggesting this solution.

```

// A Space Optimized C program to count number of possible numbers
// of given length
#include <stdio.h>

// Return count of all possible numbers of length n
// in a given numeric keyboard
int getCount(char keypad[][3], int n)
{
    if(keypad == NULL || n <= 0)
        return 0;
    if(n == 1)
        return 10;

    // odd[i], even[i] arrays represent count of numbers starting
    // with digit i for any length j
    int odd[10], even[10];
    int i = 0, j = 0, useOdd = 0, totalCount = 0;

    for (i=0; i<=9; i++)
        odd[i] = 1; // for j = 1

    for (j=2; j<=n; j++) // Bottom Up calculation from j = 2 to n
    {
        useOdd = 1 - useOdd;

        // Here we are explicitly writing lines for each number 0
        // to 9. But it can always be written as DFS on 4X3 grid
        // using row, column array valid moves
        if(useOdd == 1)
        {
            even[0] = odd[0] + odd[8];

```

```

        even[1] = odd[1] + odd[2] + odd[4];
        even[2] = odd[2] + odd[1] + odd[3] + odd[5];
        even[3] = odd[3] + odd[2] + odd[6];
        even[4] = odd[4] + odd[1] + odd[5] + odd[7];
        even[5] = odd[5] + odd[2] + odd[4] + odd[8] + odd[6];
        even[6] = odd[6] + odd[3] + odd[5] + odd[9];
        even[7] = odd[7] + odd[4] + odd[8];
        even[8] = odd[8] + odd[0] + odd[5] + odd[7] + odd[9];
        even[9] = odd[9] + odd[6] + odd[8];
    }
    else
    {
        odd[0] = even[0] + even[8];
        odd[1] = even[1] + even[2] + even[4];
        odd[2] = even[2] + even[1] + even[3] + even[5];
        odd[3] = even[3] + even[2] + even[6];
        odd[4] = even[4] + even[1] + even[5] + even[7];
        odd[5] = even[5] + even[2] + even[4] + even[8] + even[6];
        odd[6] = even[6] + even[3] + even[5] + even[9];
        odd[7] = even[7] + even[4] + even[8];
        odd[8] = even[8] + even[0] + even[5] + even[7] + even[9];
        odd[9] = even[9] + even[6] + even[8];
    }
}

// Get count of all possible numbers of length "n" starting
// with digit 0, 1, 2, ..., 9
totalCount = 0;
if(useOdd == 1)
{
    for (i=0; i<=9; i++)
        totalCount += even[i];
}
else
{
    for (i=0; i<=9; i++)
        totalCount += odd[i];
}
return totalCount;
}

// Driver program to test above function
int main()
{
    char keypad[4][3] = {'1','2','3'},
        {'4','5','6'},

```



```

        {'7','8','9'},
        {'*','0','#'}
    };
    printf("Count for numbers of length %d: %d\n", 1, getCount(keypad, 1));
    printf("Count for numbers of length %d: %d\n", 2, getCount(keypad, 2));
    printf("Count for numbers of length %d: %d\n", 3, getCount(keypad, 3));
    printf("Count for numbers of length %d: %d\n", 4, getCount(keypad, 4));
    printf("Count for numbers of length %d: %d\n", 5, getCount(keypad, 5));

    return 0;
}

```

Output:

```

    Count for numbers of length 1: 10
    Count for numbers of length 2: 36
    Count for numbers of length 3: 138
    Count for numbers of length 4: 532
    Count for numbers of length 5: 2062

```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Source

<http://www.geeksforgeeks.org/mobile-numeric-keypad-problem/>

Category: Misc Tags: Dynamic Programming, Matrix

Post navigation

← Comparison of Autoboxed Integer objects in Java Arista Networks Interview  
| Set 3 →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 72

# Number of paths with exactly k coins

Given a matrix where every cell has some number of coins. Count number of ways to reach bottom right from top left with exactly k coins. We can move to (i+1, j) and (i, j+1) from a cell (i, j).

Example:

```
Input:  k = 12
        mat[] [] = { {1, 2, 3},
                      {4, 6, 5},
                      {3, 2, 1}
                    };

Output: 2
There are two paths with 12 coins
1 -> 2 -> 6 -> 2 -> 1
1 -> 2 -> 3 -> 5 -> 1
```

**We strongly recommend you to minimize your browser and try this yourself first.**

The above problem can be recursively defined as below:

```
pathCount(m, n, k):  Number of paths to reach mat[m][n] from mat[0][0]
                      with exactly k coins
```

```

If (m == 0 and n == 0)
    return 1 if mat[0][0] == k else return 0
Else:
    pathCount(m, n, k) = pathCount(m-1, n, k - mat[m][n]) +
                        pathCount(m, n-1, k - mat[m][n])

```

Below is C++ implementation of above recursive algorithm.

```

// A Naive Recursive C++ program to count paths with exactly
// 'k' coins
#include <bits/stdc++.h>
#define R 3
#define C 3
using namespace std;

// Recursive function to count paths with sum k from
// (0, 0) to (m, n)
int pathCountRec(int mat[][C], int m, int n, int k)
{
    // Base cases
    if (m < 0 || n < 0) return 0;
    if (m==0 && n==0) return (k == mat[m][n]);

    // (m, n) can be reached either through (m-1, n) or
    // through (m, n-1)
    return pathCountRec(mat, m-1, n, k-mat[m][n]) +
           pathCountRec(mat, m, n-1, k-mat[m][n]);
}

// A wrapper over pathCountRec()
int pathCount(int mat[][C], int k)
{
    return pathCountRec(mat, R-1, C-1, k);
}

// Driver program
int main()
{
    int k = 12;
    int mat[R][C] = { {1, 2, 3},
                      {4, 6, 5},
                      {3, 2, 1}

```

```

        };
    cout << pathCount(mat, k);
    return 0;
}

```

Output:

2

The time complexity of above solution recursive is exponential. We can solve this problem in Pseudo Polynomial Time (time complexity is dependent on numeric value of input) using Dynamic Programming. The idea is to use a 3 dimensional table  $dp[m][n][k]$  where  $m$  is row number,  $n$  is column number and  $k$  is number of coins. Below is Dynamic Programming based C++ implementation.

```

// A Dynamic Programming based C++ program to count paths with
// exactly 'k' coins
#include <bits/stdc++.h>
#define R 3
#define C 3
#define MAX_K 1000
using namespace std;

int dp[R][C][MAX_K];

int pathCountDPRecDP(int mat[][C], int m, int n, int k)
{
    // Base cases
    if (m < 0 || n < 0) return 0;
    if (m==0 && n==0) return (k == mat[m][n]);

    // If this subproblem is already solved
    if (dp[m][n][k] != -1) return dp[m][n][k];

    // (m, n) can be reached either through (m-1, n) or
    // through (m, n-1)
    dp[m][n][k] = pathCountDPRecDP(mat, m-1, n, k-mat[m][n]) +
        pathCountDPRecDP(mat, m, n-1, k-mat[m][n]);

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

```

```

// This function mainly initializes dp[][][] and calls
// pathCountDPRecDP()
int pathCountDP(int mat[][C], int k)
{
    memset(dp, -1, sizeof dp);
    return pathCountDPRecDP(mat, R-1, C-1, k);
}

// Driver Program to test above functions
int main()
{
    int k = 12;
    int mat[R][C] = { {1, 2, 3},
                      {4, 6, 5},
                      {3, 2, 1}
                    };
    cout << pathCountDP(mat, k);
    return 0;
}

```

Output:

2

Time complexity of this solution is  $O(m*n*k)$ .

Thanks to Gaurav Ahirwar for suggesting above solution.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/number-of-paths-with-exactly-k-coins/>

Category: Arrays Tags: Dynamic Programming, Matrix

Post navigation

← Treap | Set 2 (Implementation of Search, Insert and Delete) Iterative Depth First Traversal of Graph →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 73

# Program for Fibonacci numbers

The Fibonacci numbers are the numbers in the following integer sequence.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, .....

In mathematical terms, the sequence  $F_n$  of Fibonacci numbers is defined by the recurrence relation

$$F_n = F_{n-1} + F_{n-2}$$

with seed values

$$F_0 = 0 \text{ and } F_1 = 1.$$

Write a function *int fib(int n)* that returns  $F_n$ . For example, if  $n = 0$ , then *fib()* should return 0. If  $n = 1$ , then it should return 1. For  $n > 1$ , it should return  $F_{n-1} + F_{n-2}$

Following are different methods to get the  $n$ th Fibonacci number.

### **Method 1 ( Use recursion )**

A simple method that is a direct recursive implementation mathematical recurrence relation given above.

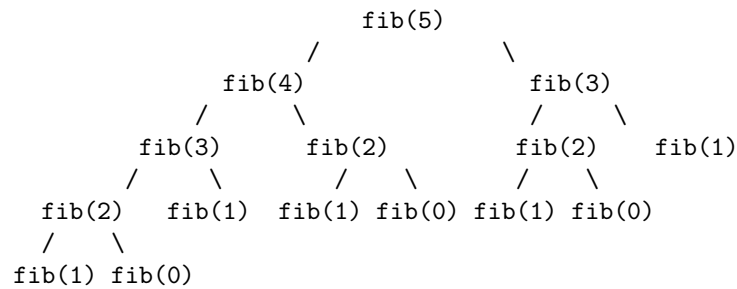
```

#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

*Time Complexity:*  $T(n) = T(n-1) + T(n-2)$  which is exponential. We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.



*Extra Space:*  $O(n)$  if we consider the function call stack size, otherwise  $O(1)$ .

## Method 2 ( Use Dynamic Programming )

We can avoid the repeated work done in the method 1 by storing the Fibonacci numbers calculated so far.

```

#include<stdio.h>

```

```

int fib(int n)
{
    /* Declare an array to store Fibonacci numbers. */
    int f[n+1];
    int i;

    /* 0th and 1st number of the series are 0 and 1*/
    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++)
    {
        /* Add the previous 2 numbers in the series
           and store it */
        f[i] = f[i-1] + f[i-2];
    }

    return f[n];
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

*Time Complexity:*  $O(n)$

*Extra Space:*  $O(n)$

### **Method 3 ( Space Optimized Method 2 )**

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibonacci number in series.

```

#include<stdio.h>
int fib(int n)
{
    int a = 0, b = 1, c, i;
    if( n == 0)
        return a;
    for (i = 2; i <= n; i++)

```



```

    {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

*Time Complexity:* O(n)

*Extra Space:* O(1)

#### **Method 4 ( Using power of the matrix $\begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}$ )**

This another O(n) which relies on the fact that if we n times multiply the matrix  $M = \begin{Bmatrix} 1 & 1 \\ 1 & 0 \end{Bmatrix}$  to itself (in other words calculate  $\text{power}(M, n)$ ), then we get the (n+1)th Fibonacci number as the element at row and column (0, 0) in the resultant matrix.

The matrix representation gives the following closed expression for the Fibonacci numbers:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}.$$

```
#include <stdio.h>
```

```
/* Helper function that multiplies 2 matrices F and M of size 2*2, and
   puts the multiplication result back to F[] [] */
void multiply(int F[2][2], int M[2][2]);
```

```
/* Helper function that calculates F[] [] raise to the power n and puts the
   result in F[] []
   Note that this function is designed only for fib() and won't work as general
   power function */
```

```

void power(int F[2][2], int n);

int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);

    return F[0][0];
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

void power(int F[2][2], int n)
{
    int i;
    int M[2][2] = {{1,1},{1,0}};

    // n - 1 times multiply the matrix to {{1,0},{0,1}}
    for (i = 2; i <= n; i++)
        multiply(F, M);
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}

```

*Time Complexity:*  $O(n)$

*Extra Space:*  $O(1)$

#### **Method 5 ( Optimized Method 4 )**

The method 4 can be optimized to work in  $O(\text{Log}n)$  time complexity. We can do recursive multiplication to get  $\text{power}(M, n)$  in the previous method (Similar to the optimization done in thispost)

```
#include <stdio.h>

void multiply(int F[2][2], int M[2][2]);

void power(int F[2][2], int n);

/* function that returns nth Fibonacci number */
int fib(int n)
{
    int F[2][2] = {{1,1},{1,0}};
    if (n == 0)
        return 0;
    power(F, n-1);
    return F[0][0];
}

/* Optimized version of power() in method 4 */
void power(int F[2][2], int n)
{
    if( n == 0 || n == 1)
        return;
    int M[2][2] = {{1,1},{1,0}};

    power(F, n/2);
    multiply(F, F);

    if (n%2 != 0)
        multiply(F, M);
}

void multiply(int F[2][2], int M[2][2])
{
    int x = F[0][0]*M[0][0] + F[0][1]*M[1][0];
    int y = F[0][0]*M[0][1] + F[0][1]*M[1][1];
    int z = F[1][0]*M[0][0] + F[1][1]*M[1][0];
    int w = F[1][0]*M[0][1] + F[1][1]*M[1][1];
```

```

    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}

/* Driver program to test above function */
int main()
{
    int n = 9;
    printf("%d", fib(9));
    getchar();
    return 0;
}

```

***Time Complexity:***  $O(\text{Log}n)$

*Extra Space:*  $O(\text{Log}n)$  if we consider the function call stack size, otherwise  $O(1)$ .

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

#### **References:**

[http://en.wikipedia.org/wiki/Fibonacci\\_number](http://en.wikipedia.org/wiki/Fibonacci_number)

<http://www.ics.uci.edu/~eppstein/161/960109.html>

#### **Source**

<http://www.geeksforgeeks.org/program-for-nth-fibonacci-number/>

## Chapter 74

# Program for nth Catalan Number

Catalan numbers are a sequence of natural numbers that occurs in many interesting counting problems like following.

- 1) Count the number of expressions containing n pairs of parentheses which are correctly matched. For n = 3, possible expressions are (((())), ()(()), ()()(), (()()), (()()).
- 2) Count the number of possible Binary Search Trees with n keys (See this)
- 3) Count the number of full binary trees (A rooted binary tree is full if every vertex has either two children or no children) with n+1 leaves.

See this for more applications.

The first few Catalan numbers for n = 0, 1, 2, 3, ... are **1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ...**

### Recursive Solution

Catalan numbers satisfy the following recursive formula.

$$C_0 = 1 \quad \text{and} \quad C_{n+1} = \sum_{i=0}^n C_i C_{n-i} \quad \text{for } n \geq 0;$$

Following is C++ implementation of above recursive formula.

```
#include<iostream>
using namespace std;

// A recursive function to find nth catalan number
```

```

unsigned long int catalan(unsigned int n)
{
    // Base case
    if (n <= 1) return 1;

    // catalan(n) is sum of catalan(i)*catalan(n-i-1)
    unsigned long int res = 0;
    for (int i=0; i<n; i++)
        res += catalan(i)*catalan(n-i-1);

    return res;
}

// Driver program to test above function
int main()
{
    for (int i=0; i<10; i++)
        cout << catalan(i) << " ";
    return 0;
}

```

Output :

```
1 1 2 5 14 42 132 429 1430 4862
```

Time complexity of above implementation is equivalent to nth catalan number.

$$T(n) = \sum_{i=0}^{n-1} T(i) * T(n-i) \quad \text{for } n \geq 0;$$

The value of nth catalan number is exponential that makes the time complexity exponential.

### Dynamic Programming Solution

We can observe that the above recursive implementation does a lot of repeated work (we can the same by drawing recursion tree). Since there are overlapping subproblems, we can use dynamic programming for this. Following is a Dynamic programming based implementation in C++.

```

#include<iostream>
using namespace std;

```

```

// A dynamic programming based function to find nth
// Catalan number
unsigned long int catalanDP(unsigned int n)
{
    // Table to store results of subproblems
    unsigned long int catalan[n+1];

    // Initialize first two values in table
    catalan[0] = catalan[1] = 1;

    // Fill entries in catalan[] using recursive formula
    for (int i=2; i<=n; i++)
    {
        catalan[i] = 0;
        for (int j=0; j<i; j++)
            catalan[i] += catalan[j] * catalan[i-j-1];
    }

    // Return last entry
    return catalan[n];
}

// Driver program to test above function
int main()
{
    for (int i = 0; i < 10; i++)
        cout << catalanDP(i) << " ";
    return 0;
}

```

Output:

```
1 1 2 5 14 42 132 429 1430 4862
```

Time Complexity: Time complexity of above implementation is  $O(n^2)$

#### Using Binomial Coefficient

We can also use the below formula to find nth catalan number in  $O(n)$  time.

$$C_n = \frac{1}{n+1} \binom{2n}{n}$$

We have discussed a  $O(n)$  approach to find binomial coefficient  $nCr$ .

```
#include<iostream>
using namespace std;

// Returns value of Binomial Coefficient C(n, k)
unsigned long int binomialCoeff(unsigned int n, unsigned int k)
{
    unsigned long int res = 1;

    // Since C(n, k) = C(n, n-k)
    if (k > n - k)
        k = n - k;

    // Calculate value of [n*(n-1)*---*(n-k+1)] / [k*(k-1)*---*1]
    for (int i = 0; i < k; ++i)
    {
        res *= (n - i);
        res /= (i + 1);
    }

    return res;
}

// A Binomial coefficient based function to find nth catalan
// number in  $O(n)$  time
unsigned long int catalan(unsigned int n)
{
    // Calculate value of  $2nCn$ 
    unsigned long int c = binomialCoeff(2*n, n);

    // return  $2nCn/(n+1)$ 
    return c/(n+1);
}

// Driver program to test above functions
int main()
{
    for (int i = 0; i < 10; i++)
        cout << catalan(i) << " ";
    return 0;
}
```



Output:

1 1 2 5 14 42 132 429 1430 4862

Time Complexity: Time complexity of above implementation is  $O(n)$ .

We can also use below formula to find nth catalan number in  $O(n)$  time.

$$C_n = \frac{(2n)!}{(n+1)! n!} = \prod_{k=2}^n \frac{n+k}{k} \quad \text{for } n \geq 0$$

**References:**

[http://en.wikipedia.org/wiki/Catalan\\_number](http://en.wikipedia.org/wiki/Catalan_number)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Source**

<http://www.geeksforgeeks.org/program-nth-catalan-number/>

## Chapter 75

# Remove minimum elements from either side such that $2 \times \text{min}$ becomes more than max

Given an unsorted array, trim the array such that twice of minimum is greater than maximum in the trimmed array. Elements should be removed either end of the array.

Number of removals should be minimum.

Examples:

```
arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200}  
Output: 4  
We need to remove 4 elements (4, 5, 100, 200)  
so that  $2 \times \text{min}$  becomes more than max.
```

```
arr[] = {4, 7, 5, 6}  
Output: 0  
We don't need to remove any element as  
 $4 \times 2 > 7$  (Note that min = 4, max = 8)
```

```
arr[] = {20, 7, 5, 6}  
Output: 1  
We need to remove 20 so that  $2 \times \text{min}$  becomes
```

more than max

```
arr[] = {20, 4, 1, 3}
```

Output: 3

We need to remove any three elements from ends

like 20, 4, 1 or 4, 1, 3 or 20, 3, 1 or 20, 4, 1

### Naive Solution:

A naive solution is to try every possible case using recurrence. Following is the naive recursive algorithm. Note that the algorithm only returns minimum numbers of removals to be made, it doesn't print the trimmed array. It can be easily modified to print the trimmed array as well.

```
// Returns minimum number of removals to be made in
// arr[l..h]
minRemovals(int arr[], int l, int h)
1) Find min and max in arr[l..h]
2) If 2*min > max, then return 0.
3) Else return minimum of "minRemovals(arr, l+1, h) + 1"
   and "minRemovals(arr, l, h-1) + 1"
```

Following is C++ implementation of above algorithm.

```
#include <iostream>
using namespace std;

// A utility function to find minimum of two numbers
int min(int a, int b) {return (a < b)? a : b;}

// A utility function to find minimum in arr[l..h]
int min(int arr[], int l, int h)
{
    int mn = arr[l];
    for (int i=l+1; i<=h; i++)
        if (mn > arr[i])
            mn = arr[i];
    return mn;
}

// A utility function to find maximum in arr[l..h]
```

```

int max(int arr[], int l, int h)
{
    int mx = arr[l];
    for (int i=l+1; i<=h; i++)
        if (mx < arr[i])
            mx = arr[i];
    return mx;
}

// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovals(int arr[], int l, int h)
{
    // If there is 1 or less elements, return 0
    // For a single element, 2*min > max
    // (Assumption: All elements are positive in arr[])
    if (l >= h) return 0;

    // 1) Find minimum and maximum in arr[l..h]
    int mn = min(arr, l, h);
    int mx = max(arr, l, h);

    //If the property is followed, no removals needed
    if (2*mn > mx)
        return 0;

    // Otherwise remove a character from left end and recur,
    // then remove a character from right end and recur, take
    // the minimum of two is returned
    return min(minRemovals(arr, l+1, h),
               minRemovals(arr, l, h-1)) + 1;
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << minRemovals(arr, 0, n-1);
    return 0;
}

```

Output:

Time complexity: Time complexity of the above function can be written as following

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

An upper bound on solution of above recurrence would be  $O(n \times 2^n)$ .

### Dynamic Programming:

The above recursive code exhibits many overlapping subproblems. For example `minRemovals(arr, l+1, h-1)` is evaluated twice. So Dynamic Programming is the choice to optimize the solution. Following is Dynamic Programming based solution.

```
#include <iostream>
using namespace std;

// A utility function to find minimum of two numbers
int min(int a, int b) {return (a < b)? a : b;}

// A utility function to find minimum in arr[l..h]
int min(int arr[], int l, int h)
{
    int mn = arr[l];
    for (int i=l+1; i<=h; i++)
        if (mn > arr[i])
            mn = arr[i];
    return mn;
}

// A utility function to find maximum in arr[l..h]
int max(int arr[], int l, int h)
{
    int mx = arr[l];
    for (int i=l+1; i<=h; i++)
        if (mx < arr[i])
            mx = arr[i];
    return mx;
}

// Returns the minimum number of removals from either end
```

```

// in arr[l..h] so that 2*min becomes greater than max.
int minRemovalsDP(int arr[], int n)
{
    // Create a table to store solutions of subproblems
    int table[n][n], gap, i, j, mn, mx;

    // Fill table using above recursive formula. Note that the table
    // is filled in diagonal fashion (similar to http://goo.gl/PQqoS),
    // from diagonal elements to table[0][n-1] which is the result.
    for (gap = 0; gap < n; ++gap)
    {
        for (i = 0, j = gap; j < n; ++i, ++j)
        {
            mn = min(arr, i, j);
            mx = max(arr, i, j);
            table[i][j] = (2*mn > mx)? 0: min(table[i][j-1]+1,
                                              table[i+1][j]+1);
        }
    }
    return table[0][n-1];
}

// Driver program to test above functions
int main()
{
    int arr[] = {20, 4, 1, 3};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << minRemovalsDP(arr, n);
    return 0;
}

```

Time Complexity:  $O(n^3)$  where  $n$  is the number of elements in `arr[]`.

Further Optimizations:

The above code can be optimized in many ways.

1) We can avoid calculation of `min()` and/or `max()` when min and/or max is/are not changed by removing corner elements.

2) We can pre-process the array and build segment tree in  $O(n)$  time. After the segment tree is built, we can query range minimum and maximum in  $O(\log n)$  time. The overall time complexity is reduced to  $O(n^2 \log n)$  time.

### A $O(n^2)$ Solution

The idea is to find the maximum sized subarray such that  $2*\min > \max$ . We run two nested loops, the outer loop chooses a starting point and the inner loop

chooses ending point for the current starting point. We keep track of longest subarray with the given property.

Following is C++ implementation of the above approach. Thanks to Richard Zhang for suggesting this solution.

```
// A O(n*n) solution to find the minimum of elements to
// be removed
#include <iostream>
#include <climits>
using namespace std;

// Returns the minimum number of removals from either end
// in arr[l..h] so that 2*min becomes greater than max.
int minRemovalsDP(int arr[], int n)
{
    // Initialize starting and ending indexes of the maximum
    // sized subarray with property 2*min > max
    int longest_start = -1, longest_end = 0;

    // Choose different elements as starting point
    for (int start=0; start<n; start++)
    {
        // Initialize min and max for the current start
        int min = INT_MAX, max = INT_MIN;

        // Choose different ending points for current start
        for (int end = start; end < n; end ++ )
        {
            // Update min and max if necessary
            int val = arr[end];
            if (val < min) min = val;
            if (val > max) max = val;

            // If the property is violated, then no
            // point to continue for a bigger array
            if (2 * min <= max) break;

            // Update longest_start and longest_end if needed
            if (end - start > longest_end - longest_start ||
                longest_start == -1)
            {
                longest_start = start;
                longest_end = end;
            }
        }
    }
}
```

```

    }
}

// If not even a single element follow the property,
// then return n
if (longest_start == -1) return n;

// Return the number of elements to be removed
return (n - (longest_end - longest_start + 1));
}

// Driver program to test above functions
int main()
{
    int arr[] = {4, 5, 100, 9, 10, 11, 12, 15, 200};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << minRemovalsDP(arr, n);
    return 0;
}

```

This article is contributed by **Rahul Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/remove-minimum-elements-either-side-2min-max/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Fab.com Pune Interview | Set 2 Amazon Interview | Set 80 →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.



## Chapter 76

# Shortest Common Supersequence

Given two strings str1 and str2, find the shortest string that has both str1 and str2 as subsequences.

Examples:

```
Input:  str1 = "geek",  str2 = "eke"
Output: "geeke"
```

```
Input:  str1 = "AGGTAB", str2 = "GXTXAYB"
Output: "AGXGTXAYB"
```

**We strongly recommend you to minimize your browser and try this yourself first.**

This problem is closely related to longest common subsequence problem. Below are steps.

- 1) Find Longest Common Subsequence (lcs) of two given strings. For example, lcs of “geek” and “eke” is “ek”.
- 2) Insert non-lcs characters (in their original order in strings) to the lcs found above, and return the result. So “ek” becomes “geeke” which is shortest common supersequence.

Let us consider another example, str1 = “AGGTAB” and str2 = “GXTXAYB”. LCS of str1 and str2 is “GTAB”. Once we find LCS, we insert characters of both strings in order and we get “AGXGTXAYB”

### How does this work?

We need to find a string that has both strings as subsequences and is shortest such string. If both strings have all characters different, then result is sum of lengths of two given strings. If there are common characters, then we don't want them multiple times as the task is to minimize length. Therefore, we first find the longest common subsequence, take one occurrence of this subsequence and add extra characters.

$$\text{Length of the shortest supersequence} = (\text{Sum of lengths of given two strings}) - (\text{Length of LCS of two given strings})$$

Below is C implementation of above idea. The below implementation only finds length of the shortest supersequence.

```
/* C program to find length of the shortest supersequence */
#include<stdio.h>
#include<string.h>

/* Utility function to get max of 2 integers */
int max(int a, int b) { return (a > b)? a : b; }

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

// Function to find length of the shortest supersequence
// of X and Y.
int shortestSuperSequence(char *X, char *Y)
{
    int m = strlen(X), n = strlen(Y);

    int l = lcs(X, Y, m, n); // find lcs

    // Result is sum of input string lengths - length of lcs
    return (m + n - l);
}

/* 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];
}

/* Driver program to test above function */
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    printf("Length of the shortest supersequence is %d\n",
           shortestSuperSequence(X, Y));
    return 0;
}

```

Output:

Length of the shortest supersequence is 9

Below is **Another Method** to solve the above problem.  
A simple analysis yields below simple recursive solution.

Let  $X[0..m-1]$  and  $Y[0..n-1]$  be two strings and  $m$  and  $n$  be respective lengths.

```

if (m == 0) return n;
if (n == 0) return m;

// If last characters are same, then add 1 to result and
// recur for X[]
if (X[m-1] == Y[n-1])
    return 1 + SCS(X, Y, m-1, n-1);

// Else find shortest of following two
// a) Remove last character from X and recur
// b) Remove last character from Y and recur
else return 1 + min( SCS(X, Y, m-1, n), SCS(X, Y, m, n-1) );

```

Below is simple naive recursive solution based on above recursive formula.

```

/* A Naive recursive C++ program to find length
   of the shortest supersequence */
#include<bits/stdc++.h>
using namespace std;

int superSeq(char* X, char* Y, int m, int n)
{
    if (!m) return n;
    if (!n) return m;

    if (X[m-1] == Y[n-1])
        return 1 + superSeq(X, Y, m-1, n-1);

    return 1 + min(superSeq(X, Y, m-1, n),
                  superSeq(X, Y, m, n-1));
}

// Driver program to test above function
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    cout << "Length of the shortest supersequence is "
         << superSeq(X, Y, strlen(X), strlen(Y));
}

```

```

        return 0;
    }

```

Output:

```

Length of the shortest supersequence is 9

```

Time complexity of the above solution exponential  $O(2^{\min(m, n)})$ . Since there are overlapping subproblems, we can efficiently solve this recursive problem using Dynamic Programming. Below is Dynamic Programming based implementation. Time complexity of this solution is  $O(mn)$ .

```

/* A dynamic programming based C program to find length
   of the shortest supersequence */
#include<bits/stdc++.h>
using namespace std;

// Returns length of the shortest supersequence of X and Y
int superSeq(char* X, char* Y, int m, int n)
{
    int dp[m+1][n+1];

    // Fill table in bottom up manner
    for (int i = 0; i <= m; i++)
    {
        for (int j = 0; j <= n; j++)
        {
            // Below steps follow above recurrence
            if (!i)
                dp[i][j] = j;
            else if (!j)
                dp[i][j] = i;
            else if (X[i-1] == Y[j-1])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
                dp[i][j] = 1 + min(dp[i-1][j], dp[i][j-1]);
        }
    }

    return dp[m][n];
}

```

```
// Driver program to test above function
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";
    cout << "Length of the shortest supersequence is "
          << superSeq(X, Y, strlen(X), strlen(Y));
    return 0;
}
```

Output:

```
Length of the shortest supersequence is 9
```

Thanks to Gaurav Ahirwar for suggesting this solution.

#### **Exercise:**

Extend the above program to print shortest supersequence also using function to print LCS.

#### **References:**

[https://en.wikipedia.org/wiki/Shortest\\_common\\_supersequence](https://en.wikipedia.org/wiki/Shortest_common_supersequence)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### **Source**

<http://www.geeksforgeeks.org/shortest-common-supersequence/>

Category: Strings Tags: Dynamic Programming

Post navigation

← Count BST subtrees that lie in given range Shortest Superstring Problem →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

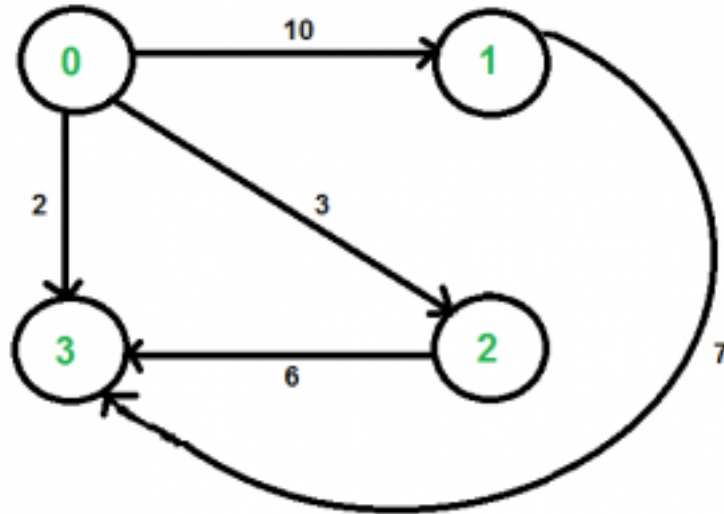
## Chapter 77

# Shortest path with exactly k edges in a directed and weighted graph

Given a directed and two vertices 'u' and 'v' in it, find shortest path from 'u' to 'v' with exactly k edges on the path.

The graph is given as adjacency matrix representation where value of  $\text{graph}[i][j]$  indicates the weight of an edge from vertex i to vertex j and a value INF(infinite) indicates no edge from i to j.

For example consider the following graph. Let source 'u' be vertex 0, destination 'v' be 3 and k be 2. There are two walks of length 2, the walks are {0, 2, 3} and {0, 1, 3}. The shortest among the two is {0, 2, 3} and weight of path is  $3+6 = 9$ .



The idea is to browse through all paths of length  $k$  from  $u$  to  $v$  using the approach discussed in the previous post and return weight of the shortest path. A **simple solution** is to start from  $u$ , go to all adjacent vertices and recur for adjacent vertices with  $k$  as  $k-1$ , source as adjacent vertex and destination as  $v$ . Following is C++ implementation of this simple solution.

```

// C++ program to find shortest path with exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and infinite value
#define V 4
#define INF INT_MAX

// A naive recursive function to count walks from u to v with k edges
int shortestPath(int graph[][V], int u, int v, int k)
{
    // Base cases
    if (k == 0 && u == v) return 0;
    if (k == 1 && graph[u][v] != INF) return graph[u][v];
    if (k <= 0) return INF;

    // Initialize result
    int res = INF;

    // Go to all adjacents of u and recur

```



```

    for (int i = 0; i < V; i++)
    {
        if (graph[u][i] != INF && u != i && v != i)
        {
            int rec_res = shortestPath(graph, i, v, k-1);
            if (rec_res != INF)
                res = min(res, graph[u][i] + rec_res);
        }
    }
    return res;
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 10, 3, 2},
                        {INF, 0, INF, 7},
                        {INF, INF, 0, 6},
                        {INF, INF, INF, 0}
                      };

    int u = 0, v = 3, k = 2;
    cout << "Weight of the shortest path is " <<
         shortestPath(graph, u, v, k);
    return 0;
}

```

Output:

Weight of the shortest path is 9

The worst case time complexity of the above function is  $O(V^k)$  where  $V$  is the number of vertices in the given graph. We can simply analyze the time complexity by drawing recursion tree. The worst occurs for a complete graph. In worst case, every internal node of recursion tree would have exactly  $V$  children. We can optimize the above solution using **Dynamic Programming**. The idea is to build a 3D table where first dimension is source, second dimension is destination, third dimension is number of edges from source to destination, and the value is count of walks. Like other Dynamic Programming problems, we fill the 3D table in bottom up manner.

```

// Dynamic Programming based C++ program to find shortest path with
// exactly k edges
#include <iostream>
#include <climits>
using namespace std;

// Define number of vertices in the graph and inifinite value
#define V 4
#define INF INT_MAX

// A Dynamic programming based function to find the shortest path from
// u to v with exactly k edges.
int shortestPath(int graph[][V], int u, int v, int k)
{
    // Table to be filled up using DP. The value sp[i][j][e] will store
    // weight of the shortest path from i to j with exactly k edges
    int sp[V][V][k+1];

    // Loop for number of edges from 0 to k
    for (int e = 0; e <= k; e++)
    {
        for (int i = 0; i < V; i++) // for source
        {
            for (int j = 0; j < V; j++) // for destination
            {
                // initialize value
                sp[i][j][e] = INF;

                // from base cases
                if (e == 0 && i == j)
                    sp[i][j][e] = 0;
                if (e == 1 && graph[i][j] != INF)
                    sp[i][j][e] = graph[i][j];

                //go to adjacent only when number of edges is more than 1
                if (e > 1)
                {
                    for (int a = 0; a < V; a++)
                    {
                        // There should be an edge from i to a and a
                        // should not be same as either i or j
                        if (graph[i][a] != INF && i != a &&
                            j != a && sp[a][j][e-1] != INF)
                            sp[i][j][e] = min(sp[i][j][e], graph[i][a] +
                                                    sp[a][j][e-1]);
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
return sp[u][v][k];
}

// driver program to test above function
int main()
{
    /* Let us create the graph shown in above diagram*/
    int graph[V][V] = { {0, 10, 3, 2},
                        {INF, 0, INF, 7},
                        {INF, INF, 0, 6},
                        {INF, INF, INF, 0}
                    };

    int u = 0, v = 3, k = 2;
    cout << shortestPath(graph, u, v, k);
    return 0;
}

```

Output:

Weight of the shortest path is 9

Time complexity of the above DP based solution is  $O(V^3K)$  which is much better than the naive solution.

This article is contributed by **Abhishek**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/shortest-path-exactly-k-edges-directed-weighted-graph/>

Category: Graph Tags: Dynamic Programming

Post navigation

← Connect n ropes with minimum cost Tarjan's Algorithm to find Strongly Connected Components →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 78

# Tiling Problem

Given a “2 x n” board and tiles of size “2 x 1”, count the number of ways to tile the given board using the 2 x 1 tiles. A tile can either be placed horizontally i.e., as a 1 x 2 tile or vertically i.e., as 2 x 1 tile.

Examples:

Input n = 3

Output: 3

Explanation:

We need 3 tiles to tile the board of size 2 x 3.

We can tile the board using following ways

- 1) Place all 3 tiles vertically.
- 2) Place first tile vertically and remaining 2 tiles horizontally.
- 3) Place first 2 tiles horizontally and remaining tiles vertically

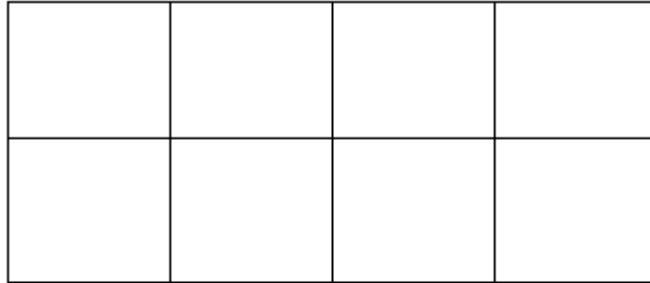
Input n = 4

Output: 5

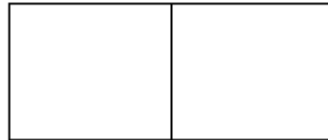
Explanation:

For a 2 x 4 board, there are 5 ways

- 1) All 4 vertical
- 2) All 4 horizontal
- 3) First 2 vertical, remaining 2 horizontal
- 4) First 2 horizontal, remaining 2 vertical
- 5) Corner 2 vertical, middle 2 horizontal



Board



Tile

**We strongly recommend you to minimize your browser and try this yourself first.**

Let “count(n)” be the count of ways to place tiles on a “2 x n” grid, we have following two ways to place first tile.

- 1) If we place first tile vertically, the problem reduces to “count(n-1)”
- 2) If we place first tile horizontally, we have to place second tile also horizontally. So the problem reduces to “count(n-2)”

Therefore, count(n) can be written as below.

$$\begin{aligned}\text{count}(n) &= n \text{ if } n = 1 \text{ or } n = 2 \\ \text{count}(n) &= \text{count}(n-1) + \text{count}(n-2)\end{aligned}$$

The above recurrence is nothing but Fibonacci Number expression. We can find n'th Fibonacci number in  $O(\log n)$  time, see below for all method to find n'th Fibonacci Number.

Different methods for n'th Fibonacci Number.

This article is contributed by Saurabh Jain. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/tiling-problem/>

Category: Misc Tags: Dynamic Programming, MathematicalAlgo

## Chapter 79

# Total number of non-decreasing numbers with n digits

A number is non-decreasing if every digit (except the first one) is greater than or equal to previous digit. For example, 223, 4455567, 899, are non-decreasing numbers.

So, given the number of digits n, you are required to find the count of total non-decreasing numbers with n digits.

Examples:

```
Input:  n = 1  
Output: count = 10
```

```
Input:  n = 2  
Output: count = 55
```

```
Input:  n = 3  
Output: count = 220
```

**We strongly recommend you to minimize your browser and try this yourself first.**

One way to look at the problem is, count of numbers is equal to count n digit number ending with 9 plus count of ending with digit 8 plus count for 7 and

so on. How to get count ending with a particular digit? We can recur for n-1 length and digits smaller than or equal to the last digit. So below is recursive formula.

$$\begin{aligned} \text{Count of } n \text{ digit numbers} = & (\text{Count of } (n-1) \text{ digit numbers Ending with digit } 9) + \\ & (\text{Count of } (n-1) \text{ digit numbers Ending with digit } 8) + \\ & \dots\dots\dots + \\ & \dots\dots\dots + \\ & (\text{Count of } (n-1) \text{ digit numbers Ending with digit } 0) \end{aligned}$$

Let count ending with digit 'd' and length n be count(n, d)

count(n, d) = (count(n-1, i)) where i varies from 0 to d

Total count = count(n-1, d) where d varies from 0 to n-1

The above recursive solution is going to have many overlapping subproblems. Therefore, we can use Dynamic Programming to build a table in bottom up manner. Below is Dynamic programming based C++ program.

```
// C++ program to count non-decreasing number with n digits
#include<bits/stdc++.h>
using namespace std;

long long int countNonDecreasing(int n)
{
    // dp[i][j] contains total count of non decreasing
    // numbers ending with digit i and of length j
    long long int dp[10][n+1];
    memset(dp, 0, sizeof dp);

    // Fill table for non decreasing numbers of length 1
    // Base cases 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
    for (int i = 0; i < 10; i++)
        dp[i][1] = 1;

    // Fill the table in bottom-up manner
    for (int digit = 0; digit <= 9; digit++)
    {
```



```

        // Compute total numbers of non decreasing
        // numbers of length 'len'
        for (int len = 2; len <= n; len++)
        {
            // sum of all numbers of length of len-1
            // in which last digit x is <= 'digit'
            for (int x = 0; x <= digit; x++)
                dp[digit][len] += dp[x][len-1];
        }
    }

    long long int count = 0;

    // There total nondecreasing numbers of length n
    // will be dp[0][n] + dp[1][n] ..+ dp[9][n]
    for (int i = 0; i < 10; i++)
        count += dp[i][n];

    return count;
}

// Driver program
int main()
{
    int n = 3;
    cout << countNonDecreasing(n);
    return 0;
}

```

Output:

220

Thanks to Gaurav Ahirwar for suggesting above method.

**Another method is based on below direct formula**

Count of non-decreasing numbers with n digits =  

$$N*(N+1)/2*(N+2)/3* \dots *(N+n-1)/n$$
  
 Where N = 10

Below is a C++ program to compute count using above formula.

```
// C++ program to count non-decreasing numner with n digits
#include<bits/stdc++.h>
using namespace std;

long long int countNonDecreasing(int n)
{
    int N = 10;

    // Compute value of N*(N+1)/2*(N+2)/3* ....*(N+n-1)/n
    long long count = 1;
    for (int i=1; i<=n; i++)
    {
        count *= (N+i-1);
        count /= i;
    }

    return count;
}

// Driver program
int main()
{
    int n = 3;
    cout << countNonDecreasing(n);
    return 0;
}
```

Output:

220

Thanks to Abhishek Somani for suggesting this method.

**How does this formula work?**

$$N * (N+1)/2 * (N+2)/3 * \dots * (N+n-1)/n$$

Where  $N = 10$

Let us try for different values of n.

For n = 1, the value is N from formula.  
Which is true as for n = 1, we have all single digit numbers, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

For n = 2, the value is  $N(N+1)/2$  from formula  
We can have N numbers beginning with 0, (N-1) numbers beginning with 1, and so on.  
So sum is  $N + (N-1) + \dots + 1 = N(N+1)/2$

For n = 3, the value is  $N(N+1)/2(N+2)/3$  from formula  
We can have  $N(N+1)/2$  numbers beginning with 0,  $(N-1)N/2$  numbers beginning with 1 (Note that when we begin with 1, we have N-1 digits left to consider for remaining places),  $(N-2)(N-1)/2$  beginning with 2, and so on.  
 $\text{Count} = N(N+1)/2 + (N-1)N/2 + (N-2)(N-1)/2 + \dots + 3 + 1$   
[Combining first 2 terms, next 2 terms and so on]  
 $= 1/2[N^2 + (N-2)N + \dots + 4]$   
 $= N*(N+1)*(N+2)/6$  [Refer this , putting  $n=N/2$  in the even sum formula]

For general n digit case, we can apply Mathematical Induction. The count would be equal to count n-1 digit beginning with 0, i.e.,  $N*(N+1)/2*(N+2)/3* \dots*(N+n-1-1)/(n-1)$ . Plus count of n-1 digit numbers beginning with 1, i.e.,  $(N-1)*(N)/2*(N+1)/3* \dots*(N-1+n-1-1)/(n-1)$  (Note that N is replaced by N-1) and so on.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Source

<http://www.geeksforgeeks.org/total-number-of-non-decreasing-numbers-with-n-digits/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Barracuda Networks Interview Experience HP R&D Interview Experience (On-Campus, full time) →

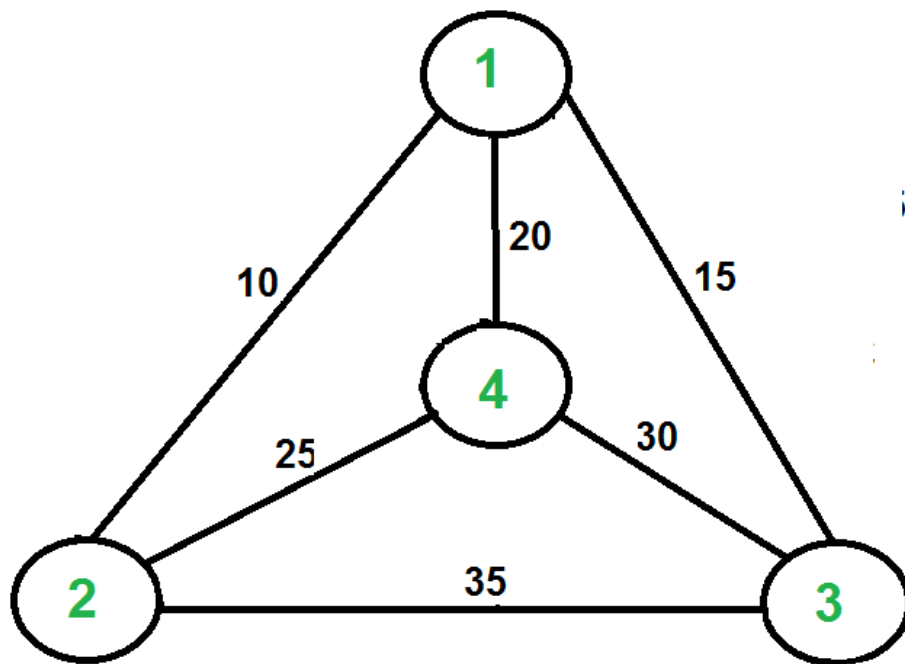
Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

## Chapter 80

# Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming)

**Travelling Salesman Problem (TSP):** Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exist a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is  $10+25+30+15$  which is 80.

The problem is a famous NP hardproblem. There is no polynomial time know solution for this problem.

Following are different solutions for the traveling salesman problem.

#### Naive Solution:

- 1) Consider city 1 as the starting and ending point.
- 2) Generate all  $(n-1)!$  Permutationsof cities.
- 3) Calculate cost of every permutation and keep track of minimum cost permutation.
- 4) Return the permutation with minimum cost.

Time Complexity:  $\Omega(n!)$

#### Dynamic Programming:

Let the given set of vertices be  $\{1, 2, 3, 4, \dots, n\}$ . Let us consider 1 as starting and ending point of output. For every other vertex  $i$  (other than 1), we find the minimum cost path with 1 as the starting point,  $i$  as the ending point and all vertices appearing exactly once. Let the cost of this path be  $\text{cost}(i)$ , the cost of corresponding Cycle would be  $\text{cost}(i) + \text{dist}(i, 1)$  where  $\text{dist}(i, 1)$  is the distance from  $i$  to 1. Finally, we return the minimum of all  $[\text{cost}(i) + \text{dist}(i, 1)]$  values. This looks simple so far. Now the question is how to get  $\text{cost}(i)$ ?

To calculate  $\text{cost}(i)$  using Dynamic Programming, we need to have some recur-

sive relation in terms of sub-problems. Let us define a term  $C(S, i)$  be the cost of the minimum cost path visiting each vertex in set  $S$  exactly once, starting at 1 and ending at  $i$ .

We start with all subsets of size 2 and calculate  $C(S, i)$  for all subsets where  $S$  is the subset, then we calculate  $C(S, i)$  for all subsets  $S$  of size 3 and so on. Note that 1 must be present in every subset.

```
If size of S is 2, then S must be {1, i},
    C(S, i) = dist(1, i)
Else if size of S is greater than 2.
    C(S, i) = min { C(S-{i}, j) + dis(j, i) } where j belongs to S, j != i and j != 1.
```

For a set of size  $n$ , we consider  $n-2$  subsets each of size  $n-1$  such that all subsets don't have  $n$ th in them.

Using the above recurrence relation, we can write dynamic programming based solution. There are at most  $O(n \cdot 2^n)$  subproblems, and each one takes linear time to solve. The total running time is therefore  $O(n^2 \cdot 2^n)$ . The time complexity is much less than  $O(n!)$ , but still exponential. Space required is also exponential. So this approach is also infeasible even for slightly higher number of vertices.

We will soon be discussing approximate algorithms for travelling salesman problem.

### References:

<http://www.lsi.upc.edu/~mjserna/docencia/algofib/P07/dynprog.pdf>  
<http://www.cs.berkeley.edu/~vazirani/algorithms/chap6.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

### Source

<http://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>

## Chapter 81

# Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3 or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... shows the first 11 ugly numbers. By convention, 1 is included. Write a program to find and print the 150'th ugly number.

### METHOD 1 (Simple)

Thanks to Nedyлко Draganov for suggesting this solution.

#### Algorithm:

Loop for all positive integers until ugly number count is smaller than n, if an integer is ugly then increment ugly number count.

To check if a number is ugly, divide the number by greatest divisible powers of 2, 3 and 5, if the number becomes 1 then it is an ugly number otherwise not.

For example, let us see how to check for 300 is ugly or not. Greatest divisible power of 2 is 4, after dividing 300 by 4 we get 75. Greatest divisible power of 3 is 3, after dividing 75 by 3 we get 25. Greatest divisible power of 5 is 25, after dividing 25 by 25 we get 1. Since we get 1 finally, 300 is ugly number.

#### Implementation:

```
# include<stdio.h>
# include<stdlib.h>

/*This function divides a by greatest divisible
   power of b*/
int maxDivide(int a, int b)
{
```

```

        while (a%b == 0)
            a = a/b;
        return a;
    }

    /* Function to check if a number is ugly or not */
    int isUgly(int no)
    {
        no = maxDivide(no, 2);
        no = maxDivide(no, 3);
        no = maxDivide(no, 5);

        return (no == 1)? 1 : 0;
    }

    /* Function to get the nth ugly number*/
    int getNthUglyNo(int n)
    {
        int i = 1;
        int count = 1;    /* ugly number count */

        /*Check for all integers untill ugly count
        becomes n*/
        while (n > count)
        {
            i++;
            if (isUgly(i))
                count++;
        }
        return i;
    }

    /* Driver program to test above functions */
    int main()
    {
        unsigned no = getNthUglyNo(150);
        printf("150th ugly no. is %d ", no);
        getchar();
        return 0;
    }

```

This method is not time efficient as it checks for all integers until ugly number count becomes n, but space complexity of this method is  $O(1)$



## METHOD 2 (Use Dynamic Programming)

Here is a time efficient solution with  $O(n)$  extra space. The ugly-number sequence is 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...

because every number can only be divided by 2, 3, 5, one way to look at the sequence is to split the sequence to three groups as below:

(1)  $1 \times 2, 2 \times 2, 3 \times 2, 4 \times 2, 5 \times 2, \dots$

(2)  $1 \times 3, 2 \times 3, 3 \times 3, 4 \times 3, 5 \times 3, \dots$

(3)  $1 \times 5, 2 \times 5, 3 \times 5, 4 \times 5, 5 \times 5, \dots$

We can find that every subsequence is the ugly-sequence itself (1, 2, 3, 4, 5, ...) multiply 2, 3, 5. Then we use similar merge method as merge sort, to get every ugly number from the three subsequence. Every step we choose the smallest one, and move one step after.

### Algorithm:

```
1 Declare an array for ugly numbers:  ugly[150]
2 Initialize first ugly no:  ugly[0] = 1
3 Initialize three array index variables i2, i3, i5 to point to
  1st element of the ugly array:
    i2 = i3 = i5 = 0;
4 Initialize 3 choices for the next ugly no:
    next_multitple_of_2 = ugly[i2]*2;
    next_multitple_of_3 = ugly[i3]*3;
    next_multitple_of_5 = ugly[i5]*5;
5 Now go in a loop to fill all ugly numbers till 150:
For (i = 1; i
Example:
```

Let us see how it works

```
initialize
    ugly[] = | 1 |
    i2 = i3 = i5 = 0;
```

First iteration

```
    ugly[1] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(2, 3, 5)
              = 2
    ugly[] = | 1 | 2 |
    i2 = 1, i3 = i5 = 0 (i2 got incremented )
```

Second iteration

```
    ugly[2] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(4, 3, 5)
```

```

        = 3
    ugly[] = | 1 | 2 | 3 |
    i2 = 1, i3 = 1, i5 = 0 (i3 got incremented )

```

Third iteration

```

    ugly[3] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(4, 6, 5)
              = 4
    ugly[] = | 1 | 2 | 3 | 4 |
    i2 = 2, i3 = 1, i5 = 0 (i2 got incremented )

```

Fourth iteration

```

    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(6, 6, 5)
              = 5
    ugly[] = | 1 | 2 | 3 | 4 | 5 |
    i2 = 2, i3 = 1, i5 = 1 (i5 got incremented )

```

Fifth iteration

```

    ugly[4] = Min(ugly[i2]*2, ugly[i3]*3, ugly[i5]*5)
              = Min(6, 6, 10)
              = 6
    ugly[] = | 1 | 2 | 3 | 4 | 5 | 6 |
    i2 = 3, i3 = 2, i5 = 1 (i2 and i3 got incremented )

```

Will continue same way till I

Program:

```

#include<stdio.h>
#include<stdlib.h>
#define bool int

/* Function to find minimum of 3 numbers */
unsigned min(unsigned , unsigned , unsigned );

/* Function to get the nth ugly number*/
unsigned getNthUglyNo(unsigned n)
{
    unsigned *ugly =
        (unsigned *) (malloc (sizeof(unsigned)*n));
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned i;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;

```

```

*(ugly+0) = 1;

for(i=1; i<n; i++)
{
    next_ugly_no = min(next_multiple_of_2,
                       next_multiple_of_3,
                       next_multiple_of_5);
    *(ugly+i) = next_ugly_no;
    if(next_ugly_no == next_multiple_of_2)
    {
        i2 = i2+1;
        next_multiple_of_2 = *(ugly+i2)*2;
    }
    if(next_ugly_no == next_multiple_of_3)
    {
        i3 = i3+1;
        next_multiple_of_3 = *(ugly+i3)*3;
    }
    if(next_ugly_no == next_multiple_of_5)
    {
        i5 = i5+1;
        next_multiple_of_5 = *(ugly+i5)*5;
    }
} /*End of for loop (i=1; i<n; i++) */
return next_ugly_no;
}

/* Function to find minimum of 3 numbers */
unsigned min(unsigned a, unsigned b, unsigned c)
{
    if(a <= b)
    {
        if(a <= c)
            return a;
        else
            return c;
    }
    if(b <= c)
        return b;
    else
        return c;
}

/* Driver program to test above functions */
int main()
{

```

```
    unsigned no = getNthUglyNo(150);  
    printf("%dth ugly no. is %d ", 150, no);  
    getchar();  
    return 0;  
}
```

**Algorithmic Paradigm:** Dynamic Programming

**Time Complexity:**  $O(n)$

**Storage Complexity:**  $O(n)$

Please write comments if you find any bug in the above program or other ways to solve the same problem.

## Source

<http://www.geeksforgeeks.org/ugly-numbers/>

Category: Misc Tags: Dynamic Programming

Post navigation

← Output of C Programs | Set 4 Little and Big Endian Mystery →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.

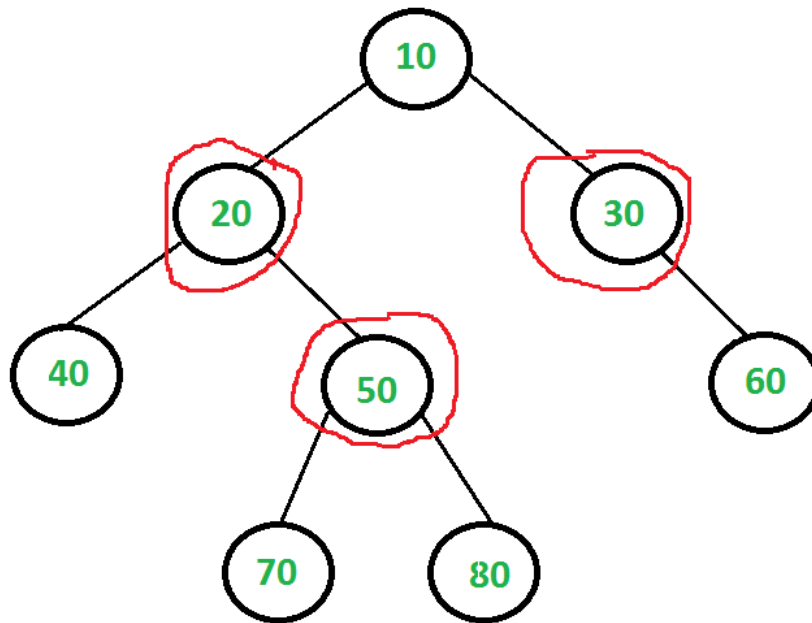
## Chapter 82

# Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree)

A vertex cover of an undirected graph is a subset of its vertices such that for every edge  $(u, v)$  of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph.

The problem to find minimum size vertex cover of a graph is NP complete. But it can be solved in polynomial time for trees. In this post a solution for Binary Tree is discussed. The same solution can be extended for n-ary trees.

For example, consider the following binary tree. The smallest vertex cover is {20, 50, 30} and size of the vertex cover is 3.



The idea is to consider following two possibilities for root and recursively for all nodes down the root.

**1) Root is part of vertex cover:** In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).

**2) Root is not part of vertex cover:** In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

Below is C implementation of above idea.

```
// A naive recursive C implementation for vertex cover problem for a tree
#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
```

```

        int data;
        struct node *left, *right;
};

// The function returns size of the minimum vertex cover
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or there
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    // Calculate size of vertex cover when root is not part of it
    int size_excl = 0;
    if (root->left)
        size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);
    if (root->right)
        size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

    // Return the minimum of two sizes
    return min(size_incl, size_excl);
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);

```

```

        root->left->right->right = newNode(14);
        root->right              = newNode(22);
        root->right->right        = newNode(25);

    printf ("Size of the smallest vertex cover is %d ", vCover(root));

    return 0;
}

```

Output:

```

    Size of the smallest vertex cover is 3

```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, vCover of node with value 50 is evaluated twice as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Vertex Cover problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming (DP) problems, re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'vc' is added to tree nodes. The initial value of 'vc' is set as 0 for all nodes. The recursive function vCover() calculates 'vc' for a node only if it is not already set.

```

/* Dynamic programming based program for Vertex Cover problem for
   a Binary Tree */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;

```



```

        int vc;
        struct node *left, *right;
    };

// A memoization based function that returns size of the minimum vertex cover.
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or there
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // If vertex cover for this node is already evaluated, then return it
    // to save recomputation of same subproblem again.
    if (root->vc != 0)
        return root->vc;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    // Calculate size of vertex cover when root is not part of it
    int size_excl = 0;
    if (root->left)
        size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);
    if (root->right)
        size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

    // Minimum of two values is vertex cover, store it before returning
    root->vc = min(size_incl, size_excl);

    return root->vc;
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->vc = 0; // Set the vertex cover as 0
    return temp;
}

// Driver program to test above functions

```

```

int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left              = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left    = newNode(10);
    root->left->right->right   = newNode(14);
    root->right              = newNode(22);
    root->right->right        = newNode(25);

    printf ("Size of the smallest vertex cover is %d ", vCover(root));

    return 0;
}

```

Output:

Size of the smallest vertex cover is 3

#### References:

<http://courses.csail.mit.edu/6.006/spring11/lectures/lec21.pdf>

#### Exercise:

Extend the above solution for n-ary trees.

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<http://www.geeksforgeeks.org/vertex-cover-problem-set-2-dynamic-programming-solution-tree/>

## Chapter 83

# Weighted Job Scheduling

Given N jobs where every job is represented by following three elements of it.

- 1) Start Time
- 2) Finish Time.
- 3) Profit or Value Associated.

Find the maximum profit subset of jobs such that no two jobs in the subset overlap.

Example:

Input: Number of Jobs  $n = 4$

Job Details {Start Time, Finish Time, Profit}

Job 1: {1, 2, 50}

Job 2: {3, 5, 20}

Job 3: {6, 19, 100}

Job 4: {2, 100, 200}

Output: The maximum profit is 250.

We can get the maximum profit by scheduling jobs 1 and 4.

Note that there is longer schedules possible Jobs 1, 2 and 3  
but the profit with this schedule is  $20+50+100$  which is less than 250.

A simple version of this problem is discussed herewhere every job has same profit or value. The Greedy Strategy for activity selection doesn't work here as the longer schedule may have smaller profit or value.

The above problem can be solved using following recursive solution.

- 1) First sort jobs according to finish time.

```

2) Now apply following recursive process.
// Here arr[] is array of n jobs
findMaximumProfit(arr[], n)
{
    a) if (n == 1) return arr[0];
    b) Return the maximum of following two profits.
        (i) Maximum profit by excluding current job, i.e.,
            findMaximumProfit(arr, n-1)
        (ii) Maximum profit by including the current job
}

```

How to find the profit including current job?

The idea is to find the latest job before the current job (in sorted array) that doesn't conflict with current job 'arr[n-1]'. Once we find such a job, we recur for all jobs till that job and add profit of current job to result.

In the above example, "job 1" is the latest non-conflicting for "job 4" and "job 2" is the latest non-conflicting for "job 3".

The following is C++ implementation of above naive recursive method.

```

// C++ program for weighted job scheduling using Naive Recursive Method
#include <iostream>
#include <algorithm>
using namespace std;

// A job has start time, finish time and profit.
struct Job
{
    int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool myfunction(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]. If there is no compatible job,
// then it returns -1.

```

```

int latestNonConflict(Job arr[], int i)
{
    for (int j=i-1; j>=0; j--)
    {
        if (arr[j].finish <= arr[i-1].start)
            return j;
    }
    return -1;
}

// A recursive function that returns the maximum possible
// profit from given array of jobs. The array of jobs must
// be sorted according to finish time.
int findMaxProfitRec(Job arr[], int n)
{
    // Base case
    if (n == 1) return arr[n-1].profit;

    // Find profit when current job is included
    int inclProf = arr[n-1].profit;
    int i = latestNonConflict(arr, n);
    if (i != -1)
        inclProf += findMaxProfitRec(arr, i+1);

    // Find profit when current job is excluded
    int exclProf = findMaxProfitRec(arr, n-1);

    return max(inclProf, exclProf);
}

// The main function that returns the maximum possible
// profit from given array of jobs
int findMaxProfit(Job arr[], int n)
{
    // Sort jobs according to finish time
    sort(arr, arr+n, myfunction);

    return findMaxProfitRec(arr, n);
}

// Driver program
int main()
{
    Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "The optimal profit is " << findMaxProfit(arr, n);
}

```

```

        return 0;
    }

```

Output:

```

    The optimal profit is 250

```

The above solution may contain many overlapping subproblems. For example if `lastNonConflicting()` always returns previous job, then `findMaxProfitRec(arr, n-1)` is called twice and the time complexity becomes  $O(n \cdot 2^n)$ . As another example when `lastNonConflicting()` returns previous to previous job, there are two recursive calls, for  $n-2$  and  $n-1$ . In this example case, recursion becomes same as Fibonacci Numbers.

So this problem has both properties of Dynamic Programming, Optimal Substructure and Overlapping Subproblems.

Like other Dynamic Programming Problems, we can solve this problem by making a table that stores solution of subproblems.

Below is C++ implementation based on Dynamic Programming.

```

// C++ program for weighted job scheduling using Dynamic Programming.
#include <iostream>
#include <algorithm>
using namespace std;

// A job has start time, finish time and profit.
struct Job
{
    int start, finish, profit;
};

// A utility function that is used for sorting events
// according to finish time
bool myfunction(Job s1, Job s2)
{
    return (s1.finish < s2.finish);
}

// Find the latest job (in sorted array) that doesn't
// conflict with the job[i]
int latestNonConflict(Job arr[], int i)
{

```

```

        for (int j=i-1; j>=0; j--)
        {
            if (arr[j].finish <= arr[i].start)
                return j;
        }
        return -1;
    }

    // The main function that returns the maximum possible
    // profit from given array of jobs
    int findMaxProfit(Job arr[], int n)
    {
        // Sort jobs according to finish time
        sort(arr, arr+n, myfunction);

        // Create an array to store solutions of subproblems. table[i]
        // stores the profit for jobs till arr[i] (including arr[i])
        int *table = new int[n];
        table[0] = arr[0].profit;

        // Fill entries in M[] using recursive property
        for (int i=1; i<n; i++)
        {
            // Find profit including the current job
            int inclProf = arr[i].profit;
            int l = latestNonConflict(arr, i);
            if (l != -1)
                inclProf += table[l];

            // Store maximum of including and excluding
            table[i] = max(inclProf, table[i-1]);
        }

        // Store result and free dynamic memory allocated for table[]
        int result = table[n-1];
        delete[] table;

        return result;
    }

    // Driver program
    int main()
    {
        Job arr[] = {{3, 10, 20}, {1, 2, 50}, {6, 19, 100}, {2, 100, 200}};
        int n = sizeof(arr)/sizeof(arr[0]);
        cout << "The optimal profit is " << findMaxProfit(arr, n);
    }

```

```
        return 0;
    }
```

Output:

```
The optimal profit is 250
```

Time Complexity of the above Dynamic Programming Solution is  $O(n^2)$ . Note that the above solution can be optimized to  $O(n \log n)$  using Binary Search in `latestNonConflict()` instead of linear search. Thanks to Garvit for suggesting this optimization.

#### References:

<http://courses.cs.washington.edu/courses/cse521/13wi/slides/06dp-sched.pdf>

This article is contributed by Shivam. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

#### Source

<http://www.geeksforgeeks.org/weighted-job-scheduling/>

Category: Arrays Tags: Dynamic Programming

Post navigation

← Optimal read list for given number of days Housing.com Interview Experience  
| Set 3 (On-Campus) →

Writing code in comment? Please use [code.geeksforgeeks.org](http://code.geeksforgeeks.org), generate link and share the link here.