

# Longest Bitonic Subsequence

We'll cover the following



- Problem Statement
- Basic Solution
  - Code
- Top-down Dynamic Programming with Memoization
  - Code
- Bottom-up Dynamic Programming
  - Code

## Problem Statement #

Given a number sequence, find the length of its Longest Bitonic Subsequence (LBS). A subsequence is considered bitonic if it is monotonically increasing and then monotonically decreasing.

### Example 1:

Input: {4,2,3,6,10,1,12}  
Output: 5  
Explanation: The LBS is {2,3,6,10,1}.

### Example 2:

Input: {4,2,5,9,7,6,10,3,1}  
Output: 7  
Explanation: The LBS is {4,5,9,7,6,3,1}.

## Basic Solution #

A basic brute-force solution could be to try finding the Longest Decreasing Subsequences (LDS), starting from every number in both directions. So for every index 'i' in the given array, we will do two things:

1. Find LDS starting from 'i' to the end of the array.
2. Find LDS starting from 'i' to the beginning of the array.

LBS would be the maximum sum of the above two subsequences.

### Code #

Here is the code:

JavaJSPython3C++

⚙️📋

```
1 def find_LBS_length(nums):
2     maxLength = 0
3     for i in range(len(nums)):
4         c1 = find_LDS_length(nums, i, -1)
5         c2 = find_LDS_length_rev(nums, i, -1)
6         maxLength = max(maxLength, c1 + c2 - 1)
7     return maxLength
8
9 # find the longest decreasing subsequence from currentIndex till the end of the array
10
11
12 def find_LDS_length(nums, currentIndex, previousIndex):
13     if currentIndex == len(nums):
14         return 0
15
16     # include nums[currentIndex] if it is smaller than the previous number
17     c1 = 0
18     if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
19         c1 = 1 + find_LDS_length(nums, currentIndex + 1, currentIndex)
20
21     # excluding the number at currentIndex
22     c2 = find_LDS_length(nums, currentIndex + 1, previousIndex)
23
24     return max(c1, c2)
25
26 # find the longest decreasing subsequence from currentIndex till the beginning of the array
27
28
29 def find_LDS_length_rev(nums, currentIndex, previousIndex):
30     if currentIndex < 0:
31         return 0
32
33     # include nums[currentIndex] if it is smaller than the previous number
34     c1 = 0
35     if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
36         c1 = 1 + find_LDS_length_rev(nums, currentIndex - 1, currentIndex)
37
38     # excluding the number at currentIndex
39     c2 = find_LDS_length_rev(nums, currentIndex - 1, previousIndex)
40
41     return max(c1, c2)
42
43
44 def main():
45     print(find_LBS_length([4, 2, 3, 6, 10, 1, 12]))
46     print(find_LBS_length([4, 2, 5, 9, 7, 6, 10, 3, 1]))
47
48
49 main()
```

⏮️📄⏭️

The time complexity of the above algorithm is exponential  $O(2^n)$ , where 'n' is the lengths of the input array. The space complexity is  $O(n)$  which is used to store the recursion stack.

## Top-down Dynamic Programming with Memoization #

To overcome the overlapping subproblems, we can use an array to store the already solved subproblems.



We need to memoize the recursive functions that calculate the longest decreasing subsequence. The two changing values for our recursive function are the current and the previous index. Therefore, we can store the results of all subproblems in a two-dimensional array. (Another alternative could be to use a hash-table whose key would be a string (currentIndex + "|" + previousIndex)).

Code #

Here is the code:

Java

JS

Python3

C++

```
1 def find_LBS_length(nums):
2     n = len(nums)
3
4     lds = [[-1 for _ in range(n+1)] for _ in range(n)]
5     ldsRev = [[-1 for _ in range(n+1)] for _ in range(n)]
6
7     maxLength = 0
8     for i in range(n):
9         c1 = find_LDS_length(lds, nums, i, -1)
10        c2 = find_LDS_length_rev(ldsRev, nums, i, -1)
11        maxLength = max(maxLength, c1 + c2 - 1)
12
13    return maxLength
14
15    # find the longest decreasing subsequence from currentIndex till the end of the array
16
17
18    def find_LDS_length(dp, nums, currentIndex, previousIndex):
19        if currentIndex == len(nums):
20            return 0
21
22        if dp[currentIndex][previousIndex + 1] == -1:
23            # include nums[currentIndex] if it is smaller than the previous number
24            c1 = 0
25            if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
26                c1 = 1 + find_LDS_length(dp, nums, currentIndex + 1, currentIndex)
27
28            # excluding the number at currentIndex
29            c2 = find_LDS_length(dp, nums, currentIndex + 1, previousIndex)
30
31            dp[currentIndex][previousIndex + 1] = max(c1, c2)
32
33        return dp[currentIndex][previousIndex + 1]
34
35    # find the longest decreasing subsequence from currentIndex till the beginning of the array
36
37
38    def find_LDS_length_rev(dp, nums, currentIndex, previousIndex):
39        if currentIndex < 0:
40            return 0
41
42        if dp[currentIndex][previousIndex + 1] == -1:
43            # include nums[currentIndex] if it is smaller than the previous number
44            c1 = 0
45            if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
46                c1 = 1 + find_LDS_length_rev(dp, nums,
```

```

46         dp[currentIndex - 1, currentIndex)
47         currentIndex - 1, currentIndex)
48
49     # excluding the number at currentIndex
50     c2 = find_LDS_length_rev(dp, nums, currentIndex - 1, previousIndex)
51
52     dp[currentIndex][previousIndex + 1] = max(c1, c2)
53     return dp[currentIndex][previousIndex + 1]
54
55
56 def main():
57     print(find_LBS_length([4, 2, 3, 6, 10, 1, 12]))
58     print(find_LBS_length([4, 2, 5, 9, 7, 6, 10, 3, 1]))
59
60
61 main()

```



## Bottom-up Dynamic Programming #

The above algorithm shows us a clear bottom-up approach. We can separately calculate LDS for every index i.e., from the beginning to the end of the array and vice versa. The required length of LBS would be the one that has the maximum sum of LDS for a given index (from both ends).

Code #

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

Java

JS

Python3





C++



```

1  def find_LBS_length(nums):
2      n = len(nums)
3      lds = [0 for _ in range(n)]
4      ldsReverse = [0 for _ in range(n)]
5
6      # find LDS for every index up to the beginning of the array
7      for i in range(n):
8          lds[i] = 1 # every element is an LDS of length 1
9          for j in range(i-1, -1, -1):
10             if nums[j] < nums[i]:
11                 lds[i] = max(lds[i], lds[j] + 1)
12
13     # find LDS for every index up to the end of the array
14     for i in range(n-1, -1, -1):
15         ldsReverse[i] = 1 # every element is an LDS of length 1
16         for j in range(i+1, n):
17             if nums[j] < nums[i]:
18                 ldsReverse[i] = max(ldsReverse[i], ldsReverse[j]+1)
19
20     maxLength = 0
21     for i in range(n):
22         maxLength = max(maxLength, lds[i] + ldsReverse[i]-1)
23
24     return maxLength
25
26
27 def main():
28     print(find_LBS_length([4, 2, 3, 6, 10, 1, 12]))
29     print(find_LBS_length([4, 2, 5, 9, 7, 6, 10, 3, 1]))
30


```


```
30
31
32  main()
33
```





The time complexity of the above algorithm is  $O(n^2)$  and the space complexity is  $O(n)$ .


 **Back**


**Next** 

Subsequence Pattern Matching

Longest Alternating Subsequence

 **Mark as Completed**

 **Report an Issue**

 **Ask a Question**  
([https://discuss.educative.io/tag/longest-bitonic-subsequence\\_\\_pattern-5-longest-common-substring\\_\\_grokking-dynamic-programming-patterns-for-coding-interviews](https://discuss.educative.io/tag/longest-bitonic-subsequence__pattern-5-longest-common-substring__grokking-dynamic-programming-patterns-for-coding-interviews))

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.