Longest Bitonic Subsequence

We'll cover the following

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:

Java JS Python3 C++

```python
def find_LBS_length(nums):
  maxLength = 0
  for i in range(len(nums)):
    c1 = find_LDS_length(nums, i, -1)
    c2 = find_LDS_length_rev(nums, i, -1)
    maxLength = max(maxLength, c1 + c2 - 1)
  return maxLength

# find the longest decreasing subsequence from currentIndex till the end of the array


def find_LDS_length(nums,  currentIndex, previousIndex):
  if currentIndex == len(nums):
    return 0

  # include nums[currentIndex] if it is smaller than the previous number
  c1 = 0
  if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
    c1 = 1 + find_LDS_length(nums, currentIndex + 1, currentIndex)

  # excluding the number at currentIndex
  c2 = find_LDS_length(nums, currentIndex + 1, previousIndex)

  return max(c1, c2)

# find the longest decreasing subsequence from currentIndex till the beginning of the arra


def find_LDS_length_rev(nums,  currentIndex,  previousIndex):
  if currentIndex < 0:
    return 0

  # include nums[currentIndex] if it is smaller than the previous number
  c1 = 0
  if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
    c1 = 1 + find_LDS_length_rev(nums, currentIndex - 1, currentIndex)

  # excluding the number at currentIndex
  c2 = find_LDS_length_rev(nums, currentIndex - 1, previousIndex)

  return max(c1, c2)


def main():
  print(find_LBS_length([4, 2, 3, 6, 10, 1, 12]))
  print(find_LBS_length([4, 2, 5, 9, 7, 6, 10, 3, 1]))


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++

```python
def find_LBS_length(nums):
  n = len(nums)

  lds = [[-1 for _ in range(n+1)] for _ in range(n)]
  ldsRev = [[-1 for _ in range(n+1)] for _ in range(n)]

  maxLength = 0
  for i in range(n):
    c1 = find_LDS_length(lds, nums, i, -1)
    c2 = find_LDS_length_rev(ldsRev, nums, i, -1)
    maxLength = max(maxLength, c1 + c2 - 1)

  return maxLength

# find the longest decreasing subsequence from currentIndex till the end of the array


def find_LDS_length(dp,  nums,  currentIndex,  previousIndex):
  if currentIndex == len(nums):
    return 0

  if dp[currentIndex][previousIndex + 1] == -1:
    # include nums[currentIndex] if it is smaller than the previous number
    c1 = 0
    if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
      c1 = 1 + find_LDS_length(dp, nums, currentIndex + 1, currentIndex)

    # excluding the number at currentIndex
    c2 = find_LDS_length(dp, nums, currentIndex + 1, previousIndex)

    dp[currentIndex][previousIndex + 1] = max(c1, c2)

  return dp[currentIndex][previousIndex + 1]

# find the longest decreasing subsequence from currentIndex till the beginning of the arra


def find_LDS_length_rev(dp,  nums,  currentIndex,  previousIndex):
  if currentIndex < 0:
    return 0

  if dp[currentIndex][previousIndex + 1] == -1:
    # include nums[currentIndex] if it is smaller than the previous number
    c1 = 0
    if previousIndex == -1 or nums[currentIndex] < nums[previousIndex]:
      c1 = 1 + find_LDS_length_rev(dp, nums,
```

```
                                    currentIndex - 1, currentIndex)

    # excluding the number at currentIndex
    c2 = find_LDS_length_rev(dp, nums, currentIndex - 1, previousIndex)

    dp[currentIndex][previousIndex + 1] = max(c1, c2)
  return dp[currentIndex][previousIndex + 1]


def main():
  print(find_LBS_length([4, 2, 3, 6, 10, 1, 12]))
  print(find_LBS_length([4, 2, 5, 9, 7, 6, 10, 3, 1]))


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++

```
def find_LBS_length(nums):
  n = len(nums)
  lds = [0 for _ in range(n)]
  ldsReverse = [0 for _ in range(n)]

  # find LDS for every index up to the beginning of the array
  for i in range(n):
    lds[i] = 1  # every element is an LDS of length 1
    for j in range(i-1, -1, -1):
      if nums[j] < nums[i]:
        lds[i] = max(lds[i], lds[j] + 1)

  # find LDS for every index up to the end of the array
  for i in range(n-1, -1, -1):
    ldsReverse[i] = 1  # every element is an LDS of length 1
    for j in range(i+1, n):
      if nums[j] < nums[i]:
        ldsReverse[i] = max(ldsReverse[i], ldsReverse[j]+1)

  maxLength = 0
  for i in range(n):
    maxLength = max(maxLength, lds[i] + ldsReverse[i]-1)

  return maxLength


def main():
  print(find_LBS_length([4, 2, 3, 6, 10, 1, 12]))
  print(find_LBS_length([4, 2, 5, 9, 7, 6, 10, 3, 1]))
```

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

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

Back

Subsequence Pattern Matching

Next

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)