ANARC05B - The Double HeLiX

Khaveesh N IMT2018036, Sai Rithwik IMT2018061

2020-04-24

Contents

Report - ANARC05B - The Double HeLiX

Run

Change your current working directory by going into `DAA-DoubleHelix`

```
cd DAA-DoubleHelix
```

To run both the algorithms and all the tests:

```
make
```

To run only algorithms:

```
make run
```

To run the tests:

```
make test
```

Problem Statement

Source

Two finite, strictly increasing, integer sequences are given. Any common integer between the two sequences constitute an intersection point. Take for example the following two sequences where intersection points are printed in bold:

- First= 3 5 **7** 9 20 **25** 30 40 **55** 56 **57** 60 62
- Second= 1 4 **7** 11 14 **25** 44 47 **55 57** 100

You can *walk* over these two sequences in the following way:

You may start at the beginning of any of the two sequences. Now start moving forward. At each intersection point, you have the choice of either continuing with the same sequence you're currently on, or switching to the other sequence.

The objective is finding a path that produces the maximum sum of data you walked over. In the above example, the largest possible sum is 450, which is the result of adding 3, 5, 7, 9, 20, 25, 44, 47, 55, 56, 57, 60, and 62

Input

The program will be tested on a number of test cases. Each test case will be specified on two separate lines. Each line denotes a sequence and is specified using the following format:

> $n\ v_1\ v_2 \ldots v_n$

where n is the length of the sequence, v_i is the i^{th} element of the sequence.

> **$1 <= n <= 10000$** and **$-10000 <= v_i <= 10000$**

`0` indicates the end of input

Output

For each test case, write on a separate line, the largest possible sum that can be produced.

Conditions

We can traverse the sequence in the following manner:

- We may start at the beginning of any of the two sequences.
- At each intersection point, we have the choice of either continuing with the same sequence, or switching to the other sequence.

SPOJ Submission Status

25840799	2020-04-24 11:47:51	Khaveesh	The Double HeLiX	**accepted** edit ideone it	0.06	11M	PYTHON3

Figure 1: Simple Approach

25840806	2020-04-24 11:48:51	Khaveesh	The Double HeLiX	**accepted** edit ideone it	0.09	11M	PYTHON3

Figure 2: Greedy Approach

Algorithm

Simple Approach

- **Step 1**: Set sum1 & sum2 to be 0.
- **Step 2**: Compare the values at the first index of both lists. Whichever is lesser, add its value to the appropriate sum variable and increment that list's index. Repeat this step until both the values are equal.

- **Step 3**: When they are equal, add the value to both sum variables. Then find the maximum among sum1 & sum2 and assign that value to both of them. Then increment the index of both the lists.
- **Step 4**: When the elements of either one of the list is exhausted, add the remaining values in the other list to its appropriate sum variable.
- **Step 5**: Return the maximum of sum1 & sum2

Greedy Approach

- **Step 1**: We generate the *prefix sum lists* of the given two sequences.
- **Step 2**: We find the point of intersection of both lists. To do this we iterate through every element in a sequence and then perform a *binary search* for the intersection on the second sequence.
- **Step 3**: On finding an intersection we find the difference between the prefix sum at the current intersection and prefix sum at previous intersection for the respective sequences and then greedily select the maximum among both and add it to a *result* variable.
- **Step 4**: Since we have found the maximum till the final intersection, we now need to find the difference between the final element of prefix sum list and the final intersection for the respective sequences and then greedily select the maximum and add it to the *result* variable

Pseudocode

Simple Approach

```
DoubleHelix(A,B,m,n)
  sum1 = 0
  sum2 = 0
  i = 0
  j = 0
  while i < m and j < n
    if A[i] < B[j]
      sum1 += A[i++]
    else if A[i] > B[j]
      sum2 += B[j++]
    else
      sum1 += A[i]
      sum2 += B[j]
      sum1 = sum2 = max(sum1, sum2)
      i++, j++
  while i < m
    sum1 += A[i++]
  while j < n
    sum2 += B[j++]
```

```
20    return max(sum1, sum2)
```

Greedy Approach

```
1  DoubleHelix(A,B,m,n)
2    result, i, flag =0
3    prevIndA, prevIndB = 0
4    prefixA, prefixB
5
6    for i in range(A)
7      bs = binary_search(A[i]
8
9      if(bs != -1):
10       if (flag == 0):
11         flag = 1
12         result += max(prefixA[i], prefixB[bs])
13       else:
14         result += max(prefixA[i]-prefixA[prevIndA]
15                       ,prefixB[bs]-prefixB[prevIndb])
16
17       prevIndA = i
18       prevIndB = bs
19
20   if (flag = 0):
21     result += max(pa1[-1], pa2[-1])
22   else:
23     result += max(prefixA[-1]-prefixA[prevIndA]
24                   ,prefixB[-1]-prefixB[prevIndb])
```

Proof of Correctness

Simple Approach

Let's assume m be the length of list 1, a_i be an element of list 1, n be the length of list 2, b_i be an element of list 2.

We have to prove that by using the following algorithm we get the maximum sum.

Since the elements in both the lists are ordered in strictly increasing order, we can predict where the elements from both the lists will attain equality i.e. intersection points.

- When $a_i < b_i$: This implies that a_i must increase to become equal to b_i. Thus we add a_i to sum1 because it is on the path to the next intersection point and traverse to the next element in the first list.
- When $a_i > b_i$: This implies that b_i must increase to become equal to a_i. Thus we add b_i to sum2 because it is on the path to the next intersection point and traverse to the next element in the

second list.

- When $a_i = b_i$: It is an intersection point. Thus we need to compare and find the maximum of the sum variables in order to maximize the sum.

Hence, we have successfully proved that the path taken by this method gives the maximum sum.

Greedy Approach

Let's assume *m* be the size of list 1, a_i be an element of sequence 1 and pa_i be the difference between prefix sum at index *i* and prefix sum at previous point of intersection(which is initially 0), *n* be the length of list 2 , b_i be an element of sequence 2 and pb_i be the difference between prefix sum at index *i* and prefix sum at previous point of intersection(which is initially 0)

We have to prove that by using the following greedy algorithm we get the maximum sum.

Base Case: We need to show that we get a maximum sum when we have two sequences of length 1.

Since the length is 1, there can be two cases, either a_1 and b_1 same or they aren't same. If a_1 and b_1 are same then the *binary search* determines that there is a point of intersection and then adds as $\max(a_1, b_1)$ which are the same at index 1 to the *result* variable. Then according to *step 4*, the difference generated will be 0 for both the sequences. Hence there won't be any change in result. Therefore we get the maximum in this case. The second case when both are not equal, then we directly go to *step 4* and accordingly maximum is chosen from both the sequences from $\max(pa_1, pb_1)$ which is equivalent to $\max(a_1, b_1)$. Hence the given proposition is true for n = 1 as we get the maximum sum from this single node path.

Induction Step over i: Assuming that our proposition holds true for sequences of length i, 1 respectively, we need to show that it is also true for i+1, 1.

This means that till i there can be two cases, either intersection occurs or intersection doesn't occur.

- Case 1: Intersection doesn't occur till i

We know that we have maximum sum till the element at index i(assumption). Let this maximum sum(which is $\max(pa_i, pb_1)$) be m1. Now at index i+1 there might or might not be be an intersection. If element at i+1 and b_1 intersect then we can tell that we get a maximum sum m1 + $\max(a_{i+1}, b_1)$, as they are identified by the binary search and then added to the result variable. Similar is the case when they don't intersect. Then we directly go to step 4 where we find $\max(pa_{i+1}, pb_1)$ which would obviously give us the maximum sum as the given sequence to us is strictly increasing.

- Case 2: If intersection has occurred at i or before i.

Since the given sequence is strictly increasing, and intersection has already occurred at index i, it means that intersection will never occur at index i+1. Which implies that the maximum would be the prefix sum at index i+1. We have to prove that our algorithm gives that value.

At intersection index k our algorithm chooses max(pa_k, 1) in step 3. Hence, the answer generated will be pa_k. Now after completing to step 4 we will have the result as Prefix Sum till i, since size of i > 1 which obviously is the maximum. Similar is the case when an additional element is added. It will be added in step 4. Since Prefix Sum till i + a_{i+1} is Prefix Sum till i+1. Hence the answer is Prefix Sum till $i+1^{th}$ element, which is greater than b_1 as intersection already occurred.

Basic logic behind reasoning in above case is a+b > a, given b > 0.

Hence proved.

Without loss of generality, we can also prove for induction over j.

Hence by induction, we have successfully proved that the path taken by this method gives the maximum sum.

Complexity

Simple Approach

- **Time Complexity**: `O(m+n)` where `m` is the length of first sequence and `n` is the length of second sequence. This is because the worst case for this algorithm occurs when there are no intersection points and thus each element in both lists will be added to the sum variables once leading to the above time complexity.
- **Space Complexity**: `O(1)`

Greedy Approach

- **Time Complexity**: `O(m*log(n))` where `m` is the length of first sequence and `n` is the length of second sequence, as the worst case will occur when the binary search cannot find an intersection and, cost of each binary search is `O(logn)` which is iterated m times.
- **Space Complexity**: `O(1)`

Data Structures

We have used Python's built-in lists & tuples only.

Side Effects

There are no side-effects during execution of this code.

Bibliography

- **Introduction to Algorithms** by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein
- Multivariable Mathematical Induction

Contributions

Khaveesh Nagappan IMT2018036

- anarc05b_double_helix.py
- test_double_helix.py
- README
- TestCases Format.txt

Sai Rithwik IMT2018061

- anarc05b_double_helix_greedy.py
- Makefile
- correct_tests.txt
- wrong_tests.txt