CSC373 – Problem Set 3

Remember to write your **full name(s)** and **student number(s)** prominently on your submission. To avoid suspicions of plagiarism: at the beginning of your submission, **clearly state any resources (people, print, electronic) outside of your group, the course notes, and the course staff, that you consulted**.

Remember that you are required to submit your problem sets as both LaTeX `.tex` source files and `.pdf` files. There is a 10% penalty on the assignment for failing to submit both the `.tex` and `.pdf`.

Due Oct 10, 2020, 22:00; required files: ps3.pdf, ps3.tex

Answer each question completely, always justifying your claims and reasoning. Your solution will be graded not only on correctness, but also on clarity. Answers that are technically correct that are hard to understand will not receive full marks. Mark values for each question are contained in the [square brackets].

You may work in groups of up to THREE to complete these questions.

Please see the course information sheet for the late submission policy.

[15 points]

You are given an array S of n distinct numbers (not necessarily integers) in sorted order such that all numbers in S are non-negative, i.e. $a \geqslant 0$ for all $a \in S$. You are also given a target value V. Your goal is to determine the number of pairs (a, b) such that $a, b \in S$ and

$$a^3 + b^3 + 3ab = V.$$

We will consider (a, b) to be the same pair as (b, a). e.g. Given $S = [4, 6, 7, 9, 11]$, and $V = 685$, the answer is 1 (consider the pair $(6, 7)$), and if $V = 863$, the answer is 0.

Systematically design an algorithm for this problem that has a worst-case time complexity of $O(n)$. Note: you're not allowed to use a formula for solving the cubic equation.

1. **(3 points)** Clearly describe all the subproblems that will be solved by your algorithm, and the values that will be stored in all the data-structure being used by your algorithm.

   The subproblem is **"Can the current choice of a and b make the statement $a^3 + b^3 + 3ab = V$ valid"**.

   We will be using 2 variables to store the index of our current choice of a and b.

2. **(5 points)** Write your algorithm in pseudo-code and analyze its complexity.

```
1 def findPair(S, V):
2
3     def validate(a,b):
4         return a**3 + b**3 + 3*a*b
5
6     if len(S) < 1:
7         return 0
8     elif len(S) == 1:
```

```
9           return 1 if validate(S[0],S[0]) == V else 0
10      # Set pointer_a and pointer_b to index of the first/last element of S
11      index_a, index_b = 0, len(S)-1
12      total = 0
13
14      # Check for self-pairs
15      for i in S:
16          if validate(i,i) == V:
17              total += 1
18
19      # Check for all pairs
20      while index_a != index_b:
21          if validate(S[index_a], S[index_b]) < V:
22              # estimate too small, increase value of a by moving right
23              move index_a to the right by 1
24          elif validate(S[index_a], S[index_b]) > V:
25              # estimate too large, decrease value of b by move left
26              move index_b to the left by 1
27          else:
28              total += 1
29              move index_a to the right by 1
30
31      return total
```

3. **(7 points)** Give a convincing proof of correctness for your algorithm. Remember that if your algorithm is greedy, you will not get any marks for the previous parts without a proof of correctness.

   We will prove the correctness of algorithm by proving all paths of the algorithm will return a correct output (the post-condition holds).

   path 1(line 6): This path is entered when S is an empty array, in which case the we will return 0 (since there is no pair for us to choose). Hence, this path holds the post-condition.

   path 2(line 8): This path is entered when S is of length 1, in which case the we will check if the only element in S can satisfy the condition as a self-pair, and return 1 if it does, or 0 if it doesn't. Hence, this path also holds the post-condition and gives the correct output.

   path 3(line 29): Line 9,10 are used to initialize our 2 index variables and the total number of pairs that satisfies the condition.

   self-pair for loop (line 13):

   For this loop, we will be looping through all the elements in S and check to see if any of them satisfies the condition $i^3 + i^3 + 3i^2 = V$. If the current element satisfies the conditoin, then we will add 1 to the total count.

   Since this is a for loop traversing through the entire array S without altering any of the element in S, it will eventually terminate, and when it does terminate, we

should have found all the "self-pairs" found and added to our total count variable.

while loop (line 18):

In this loop, we will be checking whether the 2 elements S[index_a] and S[index_b] satisfies the condition.

case 1: When validate(S[index_a], S[index_b]) ¡ V, then we will move index_a to the right by 1. Since the array S is already sorted and has no negative number, choosing S[index_a+1] as our new a will guarantee us to have a larger result than before, so our result will be closer to V.

case 2: When validate(S[index_a], S[index_b]) ¿ V, then we will move index_b to the left by 1. Since the array S is already sorted and has no negative number, choosing S[index_b-1] as our new b will guarantee us to have a smaller result than before, so our result will be closer to V.

case 3: When validate(S[index_a], S[index_b]) = V, then we found a pair that satisfies the condition, therefore we will add 1 to the total, and move index_a to the right by 1(or we can move index_b to the left by 1, this step is arbitrary).

The loop will eventually terminate when index_a = index_b, this is the case where we have traverse through the entire array. Then, by this time, we should have found all the possible pairs that satisfies the condition. And because of the "already-sorted" nature of the array, we won't miss any pair due to our a being too small and b is large enough to be a correct pair (in this case, we will be moving index_a to larger values until a is large enough).

Time complexity: The time complexity of our algorithm is O(n).

Path 1 and path 2 both take O(1) time to check.

For the first loop, we are looking at every element in S, and checking whether it satisfies the condition(which takes constant time to calculate).

For the while loop, think of array S as a doubly linked list(with head being index_a and tail being index_b) and we are traversing through it and checking for the condition at each iteration (also constant time). So, the worst case is one of the pointer stays where it is during the entire traversal, and the other pointer keeps updating itself until both pointers meet, this step takes O(n).

In conclusion, we have O(1) + O(n) + O(n), giving us O(n) in total, which satisfies the requirement.

Space complexity: the space complexity if O(1), since we are only using 3 variables in total, 2 index variables to keep track of current a and b, and 1 to keep track of the total number of pairs that satisfies the condition.