Paul Laliberte'
CSCI-3104, Algorithms
Written Assignment-1
Recitation TA: Ning Gao

P1:

(A) The number of steps in this nested for loop is

$$1 + 2 + 3 + \cdots + n + (n+1) = (n+1) \cdot \frac{n+2}{2}.$$

Then,

$$\frac{(n+1)(n+2)}{2} = \frac{n^2+3n+2}{2}.$$

Big-Theta only cares about the largest term, ignoring constants and smaller terms. We can also assume that $O(k) = \Omega(k)$, where $k = n^2$. Therefore, a bounds is $\ominus(n^2)$.

(B) For a best case scenario:

$$A_1 = [1, 2, 3, 4, 5]$$
$$B_1 = [1, b_2, b_3, b_4, b_5],$$

where b_2, b_3, b_4, b_5 are all arbitrary integers.

This returns TRUE on the first comparison. In other words, we did not need to loop through the entirety of either array.

For a worst case scenario:

$$A_2 = [1, 2, 3, 4, 5]$$
$$B_2 = [6, 7, 8, 9, 10].$$

This gives a worst case in that for every element $x \in A_2$ it must be compared to every element $y \in B_2$. There are not any equivalent values in either array. Thus, it loops through the entirety of both arrays and returns FALSE.

(C) Pseudocode

```
function checkIfCommonElements(A, B)
    i = 0
    j = 0
    while i < A.length  and i < B.length:
        if A[i] == B[j]:
            output ← True
        else if A[i] < B[j]:
            i = i+1
        else:
            j = j+1
    output ← False
```

P2: We can search for a "peaked" element by adjusting a binary search algorithm.

Pseudocode

```
function recursiveFindMax(a, left, right):
    mid = (left + right) / 2
    if a[mid] > a[mid - 1] and a[mid] < a[mid + 1]:
        output ← a[mid]
        #See note below
    else if a[mid] > a[mid-1] and a[mid] < a[mid+1]:
        recursiveFindMax(a, mid+1, right)
    else:
        recursiveFindMax(a, left, mid-1)
```

Note: For Python it is necessary to **return** the recursive calls on lines 7 and 9, and then you can **return** a[mid]. Failure to do so will result in the output to be "None".
Also one will need to use int(mid), int(mid+1), int(mid-1) in replace of mid, mid+1, mid-1.

Time Complexity: We modeled our algorithm after a binary search algorithm. Thus, we already know it runs in $\ominus(log_2(n))$. We shall somewhat prove it anyways.

Proof. We can split our array in two a times, where a is a positive integer. Then, we can split our array 2^a times. So, say we have an array of size s, where s is a positive integer, such that $s = 2^a$. Therefore,

$$s = 2^a \Rightarrow log_2(s) = log_2(2^a) \Rightarrow log_2(s) = a.$$

□

P3:

(A) Both for loops rely on the n size of the array. A for loop will run less than $\ominus(n^2)$ if, for example, the second for loop runs for a set constant number of times, or the second for loop only runs if a condition is met. This is not the case though, so much like **P1(A)**, we know that the number of steps in the nested for loop is

$$1 + 2 + 3 + \cdots + n + (n+1) = (n+1) \cdot \frac{n+2}{2}.$$

Then,

$$\frac{(n+1)(n+2)}{2} = \frac{n^2 + 3n + 2}{2}.$$

Big-theta only cares about the largest term. Therefore,

$$Time\ Complexity = \ominus(n^2).$$

(B) Given the algorithm from (A), the input arrays of A for which have **constant**, or the same, stock prices will return a profit of zero. For example,

$$A = [210, 210, 210, \ldots].$$

Then A[j] - A[i] is always zero.

(C) Pseudocode

```
function findMinimum(array, left, right):
    b = anArray[]                          #b = [], in python
    array = array[left → right]            #array[left:right], in python
    for i = 0 to array.length:
        if b.length == 0:
            b[0] ← array[i]                #this requires an append in python, i.e. b.append(array[i])
        else:                              #need to put an initial value into b, so can compare with subarray
            if b[0] > array[i]:
                b[0] = array[i]            #check if b[0] is the smaller between the two
    output ← b[0]
```

Time complexity of above algorithm: $\ominus(n)$. It depends on the n size of the array.

(D) We can simply change our algorithm from (C) to find the maximum stock price. Then we subtract the two.

Pseudocode

```
1:  function findMaximumProfit(array):
2:      c = anArray[]
3:      #find maximum
4:      for i = 0 to array.length:
5:          if c.length == 0:
6:              c[0] ← array[i]                #append in python, c.append(array[i])
7:          else:
8:              if c[0] < array[i]:
9:                  c[0] = array[i]
10:     output ← c[0] - findMinimum(array, 0, a.length - 1)
11:     #see note below
```

Note: For line 10, in python we want to return the statement on the right of the arrow. Using our findMinimum function, we can pass the **full** array, using index 0 as the "left", and the largest index, a.length - 1, as the "right". This returns the minimum for which we can subtract the two for a max profit calculation.

Time Complexity: We have two for loops (one is in findMinimum) that both depend on the n size of the array. Hence,

$$\ominus(n) + \ominus(n) = \ominus(2n).$$

Big-Theta ignores the constants. Therefore,

$$Time\ Complexity = \ominus(n).$$

(E) We can unroll the loops, search for a max and min, and then subtract the two.

Pseudocode

```
function maximumProfit(array):
    maximum = 0                        #cannot have negative stock prices, 0 is as low as a stock can go
    for i = 0 to array.length:
        if maximum < array[i]:
            maximum = array[i]
    minimum = maximum                  #upper bound, see note below
    for j = 0 to array.length:
        if minimum > array[j]:
            minimum = array[j]
    output ← maximum - minimum
```

Note: We know the maximum from the first for loop. Then the stock is not going to be higher than the maximum. It also could be the maximum for every price in the array. So, we set the minimum equal to the maximum. This way the minimum is **at least** the maximum if all prices are the same. Otherwise, it will be become a value that is below the maximum value when the second for loop starts.

Time Complexity: Similarly to (D),

$$Time\ Complexity = \ominus(n) + \ominus(n) = \ominus(2n) \Rightarrow \ominus(n).$$

Remark. For **P3: (C), (D), (E)** I made the assumption that $O(n) = \Omega(n)$, and left out the actual statement of this in each answer. I have addressed it here.

Online pdf-version of written assignment write-up, and all pseudo algorithms coded in python are at:
`http://goo.gl/QPFJH5`