COMP 352 Fall Semester 2017

Assignment 2

Rui Zhao

40018813

Writing questions:

<u>**Question 1:**</u>

a) Pseudo code:

```
Algorithm absoluteX(A,x)

Input: an array A of n integers, an integer value x

Output: the indices and value of the two elements, which subtract exactly to |x|

print ("All pairs of elements of the array that subtract exactly to absolute value of x
are:")

for i←0 to A.length-1 do                              {first pointer loop from the beginning}
    for j←i to A.length-1 do                          {second pointer loop through the array}
        if |a[i]-a[j]|=|x| then                       {check potential qualified elements}
            print (Indices i and j with values a[i] and a[j])    {print indices and values}
    end for
end for
```

b) In order to search two elements whose values subtract to another integer, we make two pointers. Then we fix the first one to the beginning, and use the second one to loop through the array to see if there are qualified elements. When the second pointer reaches the end of the array, increment the first pointer and search the array again.

c) The Big-O is $O(n^2)$, because the total execution time is (n-1)+(n-2)+…+1, which is $O(n^2)$.

d) The Big-Ω is $\Omega(n^2)$, since the total execution time is (n-1)+(n-2)+…+1, which is greater than n^2.

Question 2:

a) **Algorithm absoluteX (A, x)**

```
Input: an array A of n integers, an integer value x
Output: the indices and value of the two elements, which subtract exactly to |x|
Two empty queue q1 and q2                                {create two queues}
for i←0 to A.length-1 do                  {fill these two queues with elements of the array}
    q1.enqueue(A[i])
    q2.enqueue(A[i])
end for
q2.dequeue()               {we start with comparing the first element with the second one}
i←0, j←0                                       {variables used to track the indices}
print ("All pairs of elements that subtract exactly to absolute value of x are:")
while !q1.isEmpty()                              {keeps dequeuing until q1 is empty}
    j←i+1
    while !q2.isEmpty()      {keeps doing comparison and dequeuing until q2 is empty}
        if |q1.front()-q2.front()|=|x| then
            print (Indices i and j with values q1.front() and q2.front())
        q2.dequeue()
        j++
    end while
    q1.dequeue()              {found all qualified element to the front of q1, dequeuer it}
    ++i;
    for temp←i+1 to A.length-1 do
        q2.enqueue(A[temp])       {enqueue the array from the next element of front q1}
    end for
end while
```

b) The algorithm works as follows: store the array in two queues q1 and q2, then compare the elements of q2 with the front of q1 and dequeue them, if qualified, print the indices and values. When q2 is empty, dequeuer q1, store the array once again in q2 starting from the element next to front of q1. Then repeat the process until q1 is empty. I'm using queue because I want to make comparison from the beginning of the array.

c) The Big-O is O(n^2), because the execution time in the while loop is n^2, plus the 2n times for initializing the queue, in total it's about n^2+2n, which is O(n^2).

d) The Big-Ω is Ω(n^2).

e) The space complexity of the utilized queue is O(n), because we created 2 queues, each with n spaces, in total it is 2n.

<u>Question 3:</u>

1) $f(n)=\log n^2=2*\log n \le (\log n)^2 = g(n)$ for $n \ge 4$, therefore, f(n) is O(g(n))

2) $f(n) = n\sqrt{n} + \log n \;\; g(n) = \log n^2 = 2\log n$, since $n\sqrt{n} > \log n$ for $n \ge 1$, therefore, $f(n)$ is $\Omega(g(n))$

3) $f(n) = n \;\; g(n) = (\log n)^2$, since $n \le (\log n)^2$ for $n \ge 4$, therefore, $f(n)$ is $O(g(n))$

4) $\mathrm{f(n)} = \sqrt{\mathrm{n}}\ \ \mathrm{g(n)} = \log 10 \approx 3.322, \text{since } \mathrm{n} > \log 10\ \text{ for } \mathrm{n} \geq 4, \text{therefore, } \mathrm{f(n)} \text{ is } \Omega\big(\mathrm{g(n)}\big)$

5) $\mathrm{f(n)} = 2^{n!}\ \ \mathrm{g(n)} = 3^{n}, \text{since } 2^{n!} > 3^{n} \text{ for } \mathrm{n} \geq 3, \text{therefore, } \mathrm{f(n)} \text{ is } \Omega\big(\mathrm{g(n)}\big)$

6) $\mathrm{f(n)} = 2^{10n}\ \ \mathrm{g(n)} = n^{n}, \text{since } 2^{10n} > n^{n} \text{ for } \mathrm{n} \geq 1, \text{therefore, } \mathrm{f(n)} \text{ is } \Omega\big(\mathrm{g(n)}\big)$

Question 4:

a) The execution time for the first **for** loop is n, for the second one (nested loop) is (n-1)+(n-2)+…+1, which is n^2, for the third one is n. Therefore, we have n^2+2n operations, the algorithm is $O(n^2)$ and $\Omega(n^2)$ in terms of time complexity.

b) The result array M is {04,15,32,45,71,98}.

c) DoSomething will sort the given array in increasing order. It uses a new array Zom to keep track of the count when an element in the array is bigger than the other elements. And then use Zom as the index of M to put the elements in increasing order.

d)
```
Algorithm countingSort(A,n)
Input: an array A of n integers
Output: sorted A
Array result with size n
range ← A[0]
for i ←1 to n-1 do
    if range < A[i]
    range ← A[i]                {find the range of the values of array A}
end for
Array count with size range+1
```

```
for i←0 to range do
    count[A[i]]++          {increment the value of count with index corresponding to A[i]}
end for
for i←1 to range do
    count[i] ← count[i]+ count[i-1]      {shift the index}
end for
for i←0 to n-1 do
    result[count[A[i]-1] ← A[i]
{match the value of A with their index in counter and put the value in result}
     count[A[i]]--       {in case there are duplicate entries}
end for
return result
```

The time complexity of this sorting (counting sort) is O(n).

e) Now the space complexity of DoSomething is O(n), because in the algorithm we create two arrays each with length n. And I don't think this can be improved using this algorithm, because we will need at least one new array as auxiliary memory to finish the job. But for example, if we chose to use bubble sort, then the time complexity will still be O(n^2), but the space complexity will be O(1), since it doesn't require auxiliary memory to do the job.