CSI 2110 Computer Science Fall 2016 University of Ottawa

Assignment #1 solutions and mark breakdown

1. (2 marks) Given an n-element array A, Algorithm X executes an $O(n)$-time computation for each even number in A and an $O(\log n)$-time computation for each odd number in A.

   (a) What is the best-case running time of Algorithm X?

Answer: O(n log n) (1 mark). Additional info: The best-case running time will occur when all the n elements of the array are odd, as for each of these n numbers an O(log n)-time computation will be performed instead of the O(n) one.

   (b) What is the worst-case running time of Algorithm X?

Answer: $O(n^2)$ (1 mark). Additional info: The worst-case running time will occur when all the n elements of the array are even, as for each of these n numbers an O(n)-time computation will be performed.

2. (9 marks) Use the definition of "$f(n)$ is $O(g(n))$" to prove the following statements.

(a) $f(n)=7n^3+3n^2-2n+100$ is $O(n^3)$. (1.5 mark) Many different answers are possible; here is one of them: The witness we chose are c=110 and n0= 1, since for all n ≥1, the following holds $f(n)= 7n^3+3n^2-2n+100 \leq 7n^3+3n^2+100 \leq 7n^3+3n^3+100n^3 \leq 110n^3$. So f(n)≤ 110 n^3, for all n≥1.

(b) $f(n) = (n^2+1)/(n+1)$ is $O(n)$. (1.5 marks) The witness we chose are c=2 and n0=1. For all n ≥1, the following holds (n^2+1) ≤ n^2 +2n+1 = $(n+1)^2$. So we get $f(n)= (n^2+1)/(n+1) \leq (n+1)^2/(n+1)= n+1 \leq n+n = 2n$. So $f(n) \leq 2n$ for all $n \geq 1$.

(c) $f(n)=n!$ is $O(n^n)$. (1.5 marks) The witness we chose are c=1 and n0=1, since for all n≥1 we have f(n)=n!=n*(n-1)*…*2*1 ≤ n*n*…*n*n = n^n. So, f(n) ≤ n^n, for all n≥1. (Some students may want to use the product notation, making more explicit that both are products of n numbers each of which is smaller than n)

(d) $f(n) = \log_2 n$ is $O(\log_{10} n)$. (1.5 marks) The witness we chose are c= $\log_2 10$ and n0=1. For this question we use the change of basis formula $= \log_b a = \log_c a / \log_c b$. So we get

for all n, f(n)= $\log_2 n = \log_{10} n / \log_{10} 2 = (\log_2 10) \log_{10} n$*, for all n>=1.*
So $f(n) \leq (\log_2 10) \log_{10} n$*, for all* $n \geq 1$*.*

(e) $f(n) = n^3$ **is not** $O(100n^2)$. (1.5 marks) Suppose by contradiction that $f(n) = n^3$ **is** $O(100n^2)$, i.e. there exist c >0 and n0≥1 such that f(n) = $n^3 \leq c*100*n^2$ for all n>=n0. Now pick any n1 > max{c*100, n0}; therefore for all $n \geq$n1, we obtain $c*100*n^2 < n1*n^2 \leq n*n^2 = n^3$. So, we proved that, for all n>n1, $f(n) = n^3 \leq c*100*n^2 < n^3$ implying $n^3 < n^3$, which is a contradiction. Therefore our assumption that $f(n) = n^3$ **is** $O(100n^2)$, *must be false*.

(f) $f(n) = 2^{n+1}$ is $\Theta(2^n)$ (1.5 marks) This consists of two parts. First the big-Oh: $f(n) = 2^{n+1} = 2*2^n$, so using c=2 and $n0$=1, we get $f(n) = 2^{n+1} \leq 2*2^n = c\, 2^n$, for all $n \geq n0$. Thus, $f(n) = 2^{n+1}$ is $O(2^n)$. Now proving the Omega: $f(n) = 2^{n+1} = 2*2^n >= 2^n$, so using c=1 and n0=1, we obtain $f(n) = 2^{n+1} \geq c\, 2^n$ for all $n \geq n0$. Thus, $f(n) = 2^{n+1}$ is $\Omega(2^n)$.

3. Given an array, A, of n integers, give an $O(n)$-time algorithm that finds the longest subarray of A such that all the numbers in that subarray are in sorted order. Your algorithm outputs two integers: the initial and final indices of the longest subarray.

(a) (4 marks) Give the algorithm pseudocode.

```
if (n=0) return "empty subarray"
else {
    longest= currentLongest=1; maxini=ini=0; // A[0..0] is the longest so far
    for (i=1; i<n; i++) {
        If A[i-1]<=A[i] then { // one more item in current subarray
            currentLongest++;  // increment size
            If currentLongest > longest then { //must update the longest so far
                longest=currentLongest
                maxini=ini;
            }
        }
        else {   // new subarray starts and size is current 1 : A[i..i]
            currentLongest=1;
            ini=i;
        }
    return (maxini, maxini+longest-1) // initial and final indices of longest subarray
}
```

(b) (1 mark) Justify your big-Oh (1 mark). For all n>=1 we go to the first else, and the loop is executed n-1 times. Inside the loop we have no more than a constant number of steps (steps in the if-the-else). So the algorithm runs in O(n). The given solution uses only a constant amount of extra variables; if your solution uses another array of length n then your solution to the question will be worth 80% (4/5).

4. Suppose you are given a sorted array, A, of n distinct integers in the range from 1 to n+1, so there is exactly one integer in this range missing from A. Give an O(log n)-time algorithm for finding the integer in this range that is not in A. Hint: the algorithm resembles binary search The key to this solution is that the missing element will be (considering indexes 1..n) the first index such that A[i] not=i, and therefore A[i]>i(since the array is sorted and only one element is missing). Ex: A[1]=1 A[2]=2 A[3]=4 A[4]=5, here we conclude the missing element is i=3 is the lowest i such that A[i]>i

(a) (4 marks) Give the algorithm pseudocode.

```
Algorithm Find_missing(A, n) {
 // We give an iterative version of binary search since this is easier to analyse
// We also assume array index from 1..n to make it more clear (it can easily be
// changed to work for array with index 0..(n-1)

begin=1; end=n;
while (begin <=end) {
    i=(begin+end) div 2;  // this is the same as floor( (begin+end)/2)
    if (A[i]=i) then  begin=i+1  // everyone up to i is present in the array
    else  // here we know A[i]>i since A[i]<i never happens
            end=i-1
}
// we reach here when begin=end+1
// here we conclude A[i]=i for i=1.. begin-1  and A[begin]=A[end+1]>end+1=begin
// therefore begin is the missing integer
return begin
```

(b) (1 mark) Justify your big-Oh (1 mark).

A careful justification of why this algorithm, like binary search, takes *O(log n)* would require a careful induction proof which is not required here. We would accept that you quote that this has the same running time as binary search so it is *O(log n).* Alternatively, you could give a not so formal justification as follows. At the beginning, the number of candidates to be the missing number is *n+1*; at each iteration, the number of candidates is reduced by half, until we have only one candidate. The number of iterations is at most k= $log_2(n+1)$*, as by the definition of logarithm base 2, k is the number of times we can divide n+1 by 2 until we obtain the number 1 or less, which causes the loop to end.*

5. (4 marks) Fill a table showing a series of following queue operations and their effects on an initially empty queue *Q* of integer objects. Here *Q* is implemented with an Array of size 7.

Operation	**Output Q**
enqueue (4)	Q: 4,-,-,-,-,-,-, f=0, r=1
dequeue ()	Q: -,-,-,-,-,-,-, f=r=1 Output(return): 4
dequeue ()	Q: -,-,-,-,-,-,-, f=r=1 Output: ERROR/empty queue exception
enqueue (44)	Q: -,44,-,-,-,-,-, f=1, r=2
enqueue (7)	Q: -,44,7,-,-,-,-, f=1, r=3
enqueue (6)	Q: -,44,7,6,-,-,-, f=1, r=4
dequeue ()	Q: -,-,7,6,-,-,-, f=2, r=4 Output : 44
isEmpty()	Q: -,-,7,6,-,-,-, f=2, r=4 Output: false
enqueue (3)	Q: -,-,7,6,3,-,-, f=2 r=5
enqueue(5)	Q: -,-,7,6,3,5,-, f=2,r=6
dequeue ()	Q: -,-,-,6,3,5,-, f=3, r=6 Ouput: 7
dequeue ()	Q: -,-,-,-,3,5,-, f=4, r=6 Ouput: 6
dequeue ()	Q: -,-,-,-,-,5,- f=5 r=6 Ouput: 3
dequeue ()	Q: -,-,-,-,-,-,-, f=6 r=6 Ouput: 5
enqueue(32)	Q: -,-,-,-,-,-,32, f=6, r=0
enqueue(39)	Q: 39,-,-,-,-,-,32, f=6, r=1
enqueue(9)	Q: 39, 9,-,-,-,-,32, f=6 r=2
size()	Q: 39,9,-,-,-,-,32, f=6 r=2 Output: 3
enqueue (32)	Q: 39, 9,32,-,-,-,32, f=6,r=3
size()	Q: 39, 9,32,-,-,-,32, f=6,r=3 Output: 4
dequeue ()	Q: 39,9,32,-,-,-,-, f=0,r=3 Output: 32
enqueue (6)	Q: 39, 9,32,6,-,-,-, f=0,r=4
enqueue (5)	Q: 39,9,32,6,5,-,-, f=0,r=5
dequeue ()	Q: - ,9,32,6,5,-,-, f=1,r=5 Output 39

front()	Q: -,9,32,6,5,-,-, f=1,r=5 Output 9
size()	Q: -,9,32,6,5,-,-, f=1,r=5 Output 4
enqueue (9)	Q: -,9,32,6,5,9,-, f=1,r=6

6. (2 marks) Give an example of a positive function *f(n)* such that *f(n)* is neither $O(n^2)$ nor $\Omega(n^2)$. Explain both assertions.

f (n) = n^3 (1+sin(n))
Note that:
$-1 \leq \sin(n) \leq 1$ and (sin(n) repeatedly oscilates between -1 and 1)
$0 \leq 1+\sin(n) \leq 2$ and 1+sin(n) repeatedly oscilates between 0 and 2)
$0 \leq n^3*(1+\sin(n)) \leq 2n^3$ and $n^3*(1+\sin(n))$ repeatedly oscilates between 0 and $2n^3$)

Suppose by contradiction f(n) is $O(n^2)$, thus there exist c>0 and $n_0 \geq 1$ such that f (n)$\leq$ c. n^2 for all $n \geq n_0$. However, for infinitely many values n' larger than n_0, we have $f(n')=2(n')^3 \leq c.(n')^2$.Using similar arguments as in question 2-e we get a contradiction.

Now, suppose by contradiction that f(n) is $\Omega(n^2)$, thus there exist c>0 and $n_0 \geq 1$ such that f (n) $\geq$ c. n^2 for all $n \geq n_0$. However, for infinitely many values n' larger than n_0 we have $f(n')=0$. Using $0=f(n') \geq c.(n')^2 \geq c.n_0^2 \geq c > 0$, *we get the contradiction that* $0 > 0$.

7. (3 marks) Give a big-Oh characterization, in terms of *n*, of the running time of the following method. Show your analysis!

```
1. public void Ex(int n)
2. int a = 1;
3. for (int i = 0 ; i < n*n ; i++)
4.   for (int j = 0; j <= i; j++)
5.     if( a <= j)
6.        a = i;
7. }
```

The lines are number to facilitate reference. First, you need to be convinced that the big-Oh will be determined by the number times the loop runs, or the number of times lines 5 to 6 are executed, given that lines 5-6 run in constant time.
The first loop runs with i from 0 to n^2 and the second loop with j=0 to i. So the total number of times that lines 5-6 are executed will be 1+2+3 + … + n^2 which equals $n^2(n^2+1)/2= (n^4+ n^2)/2 < (n^4+ n^4)/2 = n^4$ which is $O(n^4)$.

8. Give a big-Oh characterization (in terms of the number *n* of elements stored in the queue) of the running time of the following methods. Show your analysis!

(a) (4 marks) Describe how to implement the queue ADT using two stacks. That is: write pseudocode algorithms which implement the *enqueue()* and *dequeue()* methods of the queue using the methods of the stack.

Assume that we have two stacks named ***S1*** and ***S2***.
S1 will store the queue elements assuming the top element in the stack is the front of the queue. ***S2*** will be used as temporary storage when enqueuing new elements.

Explanation:
dequeue
When we want to dequeue an element, we can just perform a pop on ***S1***.
enqueue
While when we want to enqueue an element ***o***, we should pop all the elements from ***S1*** and push them in ***S2***, then push the element ***o*** in ***S1*** and then pop all the elements from ***S2*** and push them back in ***Q1***.

The code follows:

```
Algorithm dequeue()
If S1.isEmpty()  then
       ERROR
Return S1.pop()


Algorithm enqueue(object o)
While  not S1.isEmpty
      S2.push(S1.pop())

S1.push(o)

While not S2.isEmpty
      S1.push(S2.pop())
```

SOLUTION 2: Note it is also possible to make enqueue simply one S2.push consequently making dequeue() to do more work doing the swing of elements from S2 to S1 then pop hen swing the elements back from S1 to S2.

(b) (1 mark) What are the running times of your *dequeue()* and *enqueue()* algorithms?

The running times will be as a function of n, the number of elements in the queue.
Answer for solution 1: (can simply say the big-Oh; here we give more explanations)
dequeue() runs in constant time, regardless of the queue size so it is in O(1).

enqueue() will perform the following tasks:

1) the first while loop does n times S1.pop, and n times S2.push, so it runs in O(n)
2) one S1.push runs in O(1)
3) the second while loop does n times S2.pop and n times S1.push, so it runs in O(n)
The tree steps together run in O(n)+O(1)+O(n) which is O(n).

SOLUTION 2: dequeue() runs in O(n) and enqueue runs in O(1)

Marking scheme: 0.5 to say correct running time for dequeue() and 0.5 to say correct running time for enqueue;
If code has different big-Oh as above, the answers for running time should match what the student programmed.