STUDENT NAME: _____

STUDENT ID: _____

McGill University
Faculty of Science
School of Computer Science

MIDTERM EXAMINATION - Sample solutions

COMP-250: Introduction to Computer Science

March 12, 2014

Examiner: Prof. Doina Precup

Write your name at the top of this page. Answer directly on exam paper (use both sides of the paper). There are 3 questions worth 100 points in total. Partial credit will be given for incomplete or partially correct answers.

SUGGESTIONS: READ ALL THE QUESTIONS BEFORE YOU START!
GOOD LUCK!

1. [40 points] **An array problem**

   You are given an array a of integers of size n, whose elements are all distinct numbers (no two elements are equal) and a value x.

   (a) [30 points] Write an algorithm CountPairs (either in pseudocode or in Java) that returns the number of distinct pairs of elements in a have the sum x.

   For example, on the array: 1 4 3 2 5, calling with $x = 5$ would return 2, calling with $x = 1$ would return 0 and calling with $x = 3$ would return 1.

   You are allowed to rearrange the elements of the array. Make your algorithm as efficient as possible. You may call any of the algorithms we discussed in class (sorting, max, search etc), and you do NOT need to reproduce their code here.

   Solution: The idea is to sort the array, then for each element $a[i]$ we will compute $x - a[i]$ and do binary search to see if this number is in the array. We will need two extra small checks: one to avoid pairing the element with itself, and one to avoid double counting. The solution in pseudocode is as follows:

   Algorithm CountPairs (a, n, x)
   Input: An array a of integers of size n with distinct elements, and a value x
   Output: The number of distinct pairs in a whose sum is x

   int $p \leftarrow 0$; //This will hold the result
   Sort(a,n);
   first we sort the array
   //Next we will loop over the elements to count
   for (**int** $i \leftarrow 0$; $i < n$; $i++$)
   if ($2 * a[i] \neq x$) **then**
   if BinarySearch(a,n,$x - a[i]$) **then** $p \leftarrow p + 1$
   end for
   return $p/2$; //We would have double-counted each pair, once for each of its members, so we need to correct for that

   (b) [10 points] Give the $O()$ for the running time of your algorithm in terms of the size of the array n, and explain your answer in 2-3 sentences. No formal proof is necessary.

   Solution: Sorting can be done in $O(n \log n)$ (e.g. using Mergesort). The for loop runs n times and inside, at worst we would call binary search every time, and it takes $O(\log n)$, which give us $O(n \log n)$ time for the loop. Hence, overall the algorithm is $O(n \log n)$.

2. [20 points] **Short questions**

   (a) [5 points] Consider the following Java method:

   ```
   public static int mystery(int n) {
   int s = 0;
   for (int i = 1; i < n; i = i * 2)
      s++;
   return s;
   }
   ```

   What function of n does the code compute?

   Solution: The code returns $\log_2 n$

   (b) [5 points] The Stack data structure has three main operations: push, pop and top. As discussed in class, stacks can be implemented using either arrays or lists. Assuming in the array implementation, enough memory is allocated for the stack to not exceed capacity, is there any difference in terms of $O()$ for these operations in the two implementations?

   Solution: No, in both implementations these operations are $O(1)$.

(c) [10 points] Consider the following Java code:

```
public class Test{
  int n;

  public Test() {
    n = 0;
  }

  public void incr(int i) {
    i++;
    n = n + i;
  }

  public int get() {
    return n;
  }

  public static void main(String[] args) {
    Test x = new Test();
    int i = 3;
    x.incr(i);
    System.out.println("i="+i);
    System.out.println("n="+x.get());
  }
}
```

What will the code print? (no explanation necessary)

Solution: The code will print:
i=3
n=4

Note that i is a primitive type, so it is passed by value. Hence, even though it is incremented inside the method, this change is not visible once the method is finished.

3. [40 points] **Recursion**

   You are given an array a of integers of size n.

   (a) [25 points] Write *a recursive algorithm*, called IsSumConstant, which returns true if the sum of all pairs of elements whose indices sum to $n-1$ is the same, i.e. $a[0]+a[n-1] == a[1]+a[n-2] == a[2]+a[n-3] == ....$ If the array is of odd size, you will add the middle element to itself and compare that sum to the rest.

   For example, on the array: 1 3 4 2, the algorithm should return false (1+2 is not the same as 3+4).

   On the array: 5, the algorithm should return true (we only have one sum).

   On the array: 3 1 4 2 the algorithm should return true (because 3+2 is the same as 4+1).

   On the array: 1 2 3 4 5 the algorithm should return true (because 1+5, 2+4 and 3+3 are all the same).

   You are allowed to write more than one method as part of the algorithm. You may write either pseudocode or Java, as you prefer.

   Solution: Note that this problem is a variation on the palindromes problem. We will write an algorithm which calls a helper function working on a part of the array, and this function will be recursive.

   Algorithm IsSumConstant(a,n)
   Input: An array a of size n
   Output: True if the sum of all pairs of elements whose indices sum to $n-1$ is the same, false otherwise
   if ($n \leq 2$) **then return true**;
   return IsSumConstantRec(a, 1, $n-2$, $a[0]+a[n-1]$)
   Algorithm IsSumConstantRec(a, i, j, x)
   Input: An array a, two indices i and j and a number x
   Output: True if all elements whose sum of indices is equal to $i+j$ is x and false otherwise
   if $(i = j$ **or** $i+1 = j)$ **then**
   // Base case, we are left with 1 or 2 elements
   if $(a[i]+a[j] \neq x)$ **then return false**
   else return true
   return $(a[i]+a[j] = x)$ **and** IsSumConstantRec(a, $i+1$, $j-1$, x)

   (b) [10 points] Write a recurrence for the running time of your algorithm.
   Solution: $T(n) = c + T(n-2)$ because every time we recurse, we eliminate two elements from consideration

   (c) [5 points] Based on this recurrence, what is the $O()$ of your algorithm?
   Solution: $O(n)$ (you can expand the recurrence like we did in class for the recursive max). Note that we would not be able to do any better with another algorithm, as we need to look at all elements in the array to answer correctly, so this is also a lower bound.