Homework Assignment 1

Due: Friday Jan. 21, 2022 (11:59pm)

(Total 40 marks)

CMPUT 204
Department of Computing Science
University of Alberta

Problem 1. (10 marks)

Consider the following definition of Fibonacci numbers:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n \geq 2 \end{cases}$$

a. (3 marks) Give the set of all natural numbers q for which $F(q) \geq 1.6^{q-2}$. Justify briefly.

Solution: The answer is $\mathbb{N}^+$, *i.e.*, the set of all the positive integers. Note $F(1) = 1 \geq 1.6^{1-2} = 1.6^{-1}$, and $F(2) = F(0) + F(1) = 1 \geq 1.6^{2-2} = 1.6^0 = 1$. Then for all $t > 2$, $F(t) = F(t-1) + F(t-2) \geq 1.6^{t-3} + 1.6^{t-4} = 1.6^{t-4} \cdot (1.6 + 1) \geq 1.6^{t-4} \cdot (1.6^2) = 1.6^{t-2}$.

Note: (a) Strong induction is used here, though we didn't explicitly state it (for "justify briefly", a sketch of a proof is just fine). (b) You need two base cases to apply I.H.

b. (3 marks) Does your answer to **a** imply the following assertion: $\exists c, n_0 > 0$, such that $0 \leq c \cdot 1.6^n \leq F(n)$, $\forall n \geq n_0$. Explain briefly.

Solution: Yes. By taking $c = 1.6^{-2}$, and $n_0 = 1$ for example, we have $0 \leq c \cdot 1.6^n = 1.6^{n-2} \leq F(n)$, $\forall n \geq n_0$. There are infinitely many other possibilities. You can verify that any $0 < c \leq 1.6^{-2}$ and any $n_0 \geq 1$ will work.

c. (4 marks) Consider the following recursive implementation of function $F(n)$ (which differs slightly from the one given in the lecture notes).

```
procedure fib(n)
if (n = 0) then
     return 0
else if (n = 1 or n = 2) then
     return 1
else
     return fib(n − 1) + fib(n − 2)
```

Prove by induction that the procedure $\texttt{fib}(n)$ is correct.

Proof.

Base case: For $n = 0$, $\text{fib}(0) = F(0) = 0$; for $n = 1$, $\text{fib}(1) = F(1) = 1$, and for $n = 2$, $\text{fib}(2) = \text{fib}(1) + \text{fib}(0) = 1$ while $F(2) = F(1) + F(0) = 1$.

Induction Step: Let $k \geq 2$. Assume for all $0 \leq i \leq k$, $\text{fib}(i) = F(i)$ and we show $\text{fib}(k+1) = F(k+1)$. According to the code, since $k+1 > 2$, $\text{fib}(k+1)$ invokes $\text{fib}(k)+\text{fib}(k-1)$, which, by induction hypothesis (IH), returns the value of $F(k) + F(k-1) = F(k+1)$. The last step of derivation is by the definition of function F.

Problem 2. (10 marks) Consider the following recursive version of `InsertionSort`:

procedure InsertionSort(A, n)
    `**Sorts array` A `of size` n
    if $n > 1$ **then**
        InsertionSort($A, n-1$)
        $x \leftarrow A[n]$
        PutInPlace($A, n-1, x$)
    end if

procedure PutInPlace(A, j, x)
    if $(j = 0)$ **then**
        $A[1] \leftarrow x$
    else if $x > A[j]$ **then**
        $A[j+1] \leftarrow x$
    else $**$ i.e., $x \leq A[j]$
        $A[j+1] \leftarrow A[j]$
        PutInPlace($A, j-1, x$)
    end if

a. (6 marks)

- [3 marks] First, prove (using induction) the correctness of `PutInPlace` by showing that:

  For any array A, and natural number j such that (i) A has (at least) $j+1$ cells, and (ii) the subarray $A[1..j]$ is sorted, when `PutInPlace(`A, j, x`)` terminates, the first $j+1$ cells of A contain all the elements that were originally in $A[1..j]$ plus x in sorted order.

  We prove the predicate by induction. The base case is when $j = 0$, for which the claim is true as `PutInPlace` just puts x in the first cell of A.

  Induction step: Let $j \geq 1$ be an arbitrary integer, and assuming the claim holds for $j-1$ we show it also holds for j. If x is greater than $A[j]$, then x is greater than all elements in $A[1..j]$, so by putting x in the $j+1$-th cell the array $A[1..j+1]$ satisfies the required: sorted, and contain all the required elements. Otherwise $A[j] \geq x$. Then $A[j]$ is the max-element of $A[1..j]$ and x. So we put $A[j]$ in the $j+1$-th cell. Then by invoking `PutInPlace` on $A[1..j-1]$, by IH, the result is that $A[1..j]$ contains the elements of $A[1..j-1]$ and x, sorted. This means that altogether $A[1..j+1]$ is sorted.

- [3 marks] Use induction and the correctness of `PutInPlace()` to prove the correctness of `InsertionSort(`A, n`)`.

  We show the following predicate by induction: "For any array A containing n pair-wise comparable elements, `InsertionSort(`A, n`)` correctly sorts A." We use induction on the number of elements of A.

  Base case: `InsertionSort(`$A, 1$`)` correctly sorts any array of size 1 as the code does nothing and an array of size 1 is trivially sorted.

  Induction Step: Consider an arbitrary integer $n \geq 1$ and assuming `InsertionSort(`A, n`)` correctly sorts any array of size n, we show `InsertionSort(`$A, n+1$`)` also correctly sorts any array of size $n+1$. Since $n+1 > 1$ then we first invoke a recursive call to sort the first n elements of the array. By IH, the recursive call indeed correctly sorts the first n elements. We then use the function `PutInPlace` to put the last element (the $n+1$th-element) in its right place. Since `PutInPlace(`$A, n, A[n+1]$`)` places $x = A[n+1]$ in its right place, making the whole $n+1$ elements of the array sorted.

b. (4 marks) For each state of array A below, indicate whether it ever occurs during the call, `InsertionSort(`A, n`)`, where $A = [4, 7, 2, 8, 6, 5]$ and $n = 6$.

i. $A = [4, 2, 7, 8, 6, 5]$ Not occur

ii. $A = [2, 4, 7, 8, 6, 5]$ Yes

iii. $A = [2, 4, 7, 6, 8, 5]$ No

iv. $A = [2, 4, 6, 7, 8, 5]$ Yes

v. $A = [2, 4, 6, 7, 5, 8]$ No

vi. $A = [2, 4, 6, 5, 7, 8]$ No

vii. $A = [2, 4, 5, 6, 7, 8]$ Yes

Problem 3. (8 marks) Consider the following code, which takes as input an array A of n integers, where $n > 0$.

```
procedure Unknown(A, n)
i ← 1
p ← A[1] + 1
while (i < n) do
    j ← i + 1
    p ← p + (A[j] + j)
    i ← i + 1
return p
```

(i) Suppose the input array is $A = [2, 5, 3, 6]$. What does the procedure return?

(ii) Give a loop invariant (LI) for the loop in the code.

Hint: Here is a partial analysis: At the start of each iteration i ($1 \le i \le n$), we have

- when $i = 1$, the value of p is $A[1] + 1$;
- when $i = 2$, the value of p is $(A[1] + 1) + (A[2] + 2)$;
- when $i = 3$, the value of p is $(A[1] + 1) + (A[2] + 2) + (A[3] + 3)$;
- ......
- when $i = n$, the loop terminates.

(iii) Prove the the properties of Initialization, Maintenance, and Termination (including both Termination #1 and Termination #2)

Solution (for (ii) and (iii))

* LI: At the start of each iteration i ($1 \le i \le n$), $p = (A[1] + 1) + ... + (A[i] + i)$ (You may also express this by $p = \sum_{k=1}^{i} (A[k] + k)$).

* Initialization: At the start of the first iteration, $i = 1$ and $p = A[1] + 1$. LI holds.

* Maintenance: For a variable v, we will write v^{new} to express its value after the code in the loop executes and just write v to denote its value before the code execution.

  Suppose at the start of iteration i ($1 \leq i < n$), LI holds, i.e., $p = \sum_{k=1}^{i}(A[k]+k)$. By the execution of the code, $p^{new} = p + (A[i+1] + (i+1))$, which, by IH, equals $\sum_{k=1}^{i}(A[k]+k) + (A[i+1]+(i+1)) = \sum_{k=1}^{i+1}(A[k]+k) = \sum_{k=1}^{i^{new}}(A[k]+k)$. This shows that LI holds at the start of the next iteration.

* Termination 1: The variable i starts with value 1, is incremented by 1 in each iteration, and nothing else changes its value. This guarantees that its value will eventually reaches n upon which the loop terminates.

* Termination 2: When the loop terminates, $i = n$. The LI implies $p = \sum_{k=1}^{n}(A[k]+k)$. Since the question does not specify what the procedure Unknow(.) intends to compute, there is nothing further to prove.

Problem 4. (12 marks) Describe an algorithm for finding both the minimum and maximum of n numbers using fewer than $3n/2$ key comparisons.

Hints:

- The case of $n = 1$ or 2 is trivial, at most 1 key comparision is needed. When $n > 2$, one can use a loop to examine each element after the first two and conduct two key comparisons to determine whether it is a new candidate minimum/maximum. This algorithm requires $1 + 2(n-2)$ key comparisons, which does not satisfy the condition given in the problem.

- To reduce key comparisons, proccess each pair of elements (after the first two) using a loop and determine if the smaller of the pair is a new candidate minimum and if the larger one is a new candidate maximum. You need to handle both cases when n is even or odd.

Here are what you need to do for this problem.

- Write a pseudocode for your algorithm, which must use exactly one loop (as sketched above, you don't need multiple loops).

- Explain why your algorithm uses fewer than $3n/2$ key comparisons.

- Write a loop invariant that your algorithm maintains and show the properties of Initialization, Maintenance, and Termination (including both Termination #1 and Termination #2)

Solution: The pseudo-code for the algorithm:

Sorting $A[1 \dots n]$
**Precondition: $A[1 \dots n]$ is an array of numbers, $n \geq 1$.
**Postcondition: Returns the minimum and maximum elements of $A[1 \dots n]$.
if $n = 1$ **then**
    $a = A[1]$
    $b = A[1]$
    return a, b
else
    if $A[1] < A[2]$ **then**
        $a = A[1]$

```
            b = A[2]
        else
            a = A[2]
            b = A[1]
        end if
        t = ⌊n/2⌋
        for i from 2 to t
            if A[2i − 1] < A[2i] then
                min = A[2i − 1]
                max = A[2i]
            else
                min = A[2i]
                max = A[2i − 1]
            end if
            if min < a then
                a = min
            end if
            if max > b then
                b = max
            end if
        end for
        if n > 2t then
            if A[n] < a then
                a = A[n]
            end if
            if A[n] > b then
                b = A[n]
            end if
        end if
        return a, b
    end if
```

If n is even, then the total number of comparisons are $1 + 3 \cdot \frac{n-2}{2} = \frac{3n}{2} - 2$. If n is odd, then the total number of comparisons are $1 + 3 \cdot \frac{n-3}{2} + 3 = \frac{3n}{2} - \frac{1}{2}$. (Here we count $n > 2t$ as one key comparison. Since it is entirely clear whether a "key" is involved in this comparison, it is fine if you don't count it, in which case,"+3" should be "+2".)

Proposed loop invariant: a is the minimum value of $A[1..2(i-1)]$, and b is the maximum value of $A[1..2(i-1)]$.

Initialization: For $i = 2$, we have $A[1..2(i-1)] = A[1..2]$. The loop invariant holds since we set a as the smaller one and b as the larger one of $A[1]$ and $A[2]$.

Maintenance: Suppose a is the minimum value of $A[1..2(k-1)]$, and b is the maximum value of $A[1..2(k-1)]$, we want to show a will be the minimum value of $A[1..2k]$, and b will be the maximum value of $A[1..2k]$ after the k_{th} loop iteration. For a, It suffices to prove that $a \leq A[2k-1]$ and $a \leq A[2k]$. This holds because $a \leq min$, and min is the minimum value selected from $A[2k-1]$ and $A[2k]$. For b, since $b \geq max$, and max is the maximum value selected from $A[2k-1]$ and $A[2k]$, we have $b \geq A[2k-1]$ and $b \geq A[2k]$.

Termination: (Termination 1 can be argued easily - we omit it here.) After the ending of the for loop, $i = t + 1 = \lfloor \frac{n}{2} \rfloor + 1$, we have $A[1..2(i-1)] = A[1..(2 \cdot \lfloor \frac{n}{2} \rfloor)]$. If n is even, then $2 \cdot \lfloor \frac{n}{2} \rfloor = n$, which means a is the minimum value of $A[1..n]$, and b is the maximum value of $A[1..n]$, as wanted. If n is odd, then $2 \cdot \lfloor \frac{n}{2} \rfloor = n - 1$, we actually have a is the minimum value of $A[1..(n-1)]$, and b is the maximum value of $A[1..(n-1)]$. So in this case we need another if clause to compare a and b with $A[n]$.