COMP 7270 Assignment 1 No late submissions accepted!

Upload your submission well before this deadline. Even if you are a few minutes late, as a result of which Canvas marks your submission late, your assignment may not be accepted.

Instructions:

1. This is an individual assignment. You should do your own work after discussing with at most two partners. **Write out whom you discussed with. Any evidence of copying will result in a zero grade and additional penalties/actions.**
2. Late submissions **will not** be accepted unless prior permission has been granted or there is a valid and verifiable excuse.
3. **Think carefully; formulate your answers, and then write them out concisely** using English, logic, mathematics and pseudocode (no programming language syntax).
4. **Type your answers in this Word document and submit it. If that is not possible, use a word processor to type your answers as much as possible (you may hand-write/draw equations and figures), turn it into a PDF document and upload**.

NAME: Robin Ward
SID: rcw0024
I DISCUSSED WITH: _____, _____

1. Prove by contradiction that the following algorithm to multiply two integers without using multiplication is correct. Note that your proof must be detailed and that every step of the proof other than the initial assumption must be based on (and justified by) a mathematical or logical fact or a step of the algorithm. Several initial steps of the proof are given as hints. Complete the rest of the proof.

```
function multiply(y,z: non-negative integer)
1 if z==0 then return(0)
2 else if z==1 then return(y)
  else
3       product=0
        repeat
4               product=product+y
5               z=z–1
6       until z==0
7. return product
```

Proof by contradiction with numbered steps:

1. Suppose the algorithm is incorrect.
2. There is some value of z≥0 for which the answer returned, product ≠ z*y.
3. This cannot be z=0 because when z=0 z*y=0 and algorithm step 1 returns the correct answer.
4. This cannot be z= 1 because when z=1 z*y=1 and algorithm step 2 returns the correct answer.
5. If z>1, the condition checks in steps 1 & 2 will fail so steps 3-7 will be executed.

<complete the rest of the proof>

6.This cannot be z=2, y=0 because when z=2 and y=0 step 7 returns the correct answer

7.This cannot be z=2, y=1 because when z=2 and y=1 step 7 returns the correct answer

8. This cannot be z=2, y=2 because when z=2 and y=2 step 7 returns the correct answer

9.This algorithm is correct

2. Consider the Selection Problem (SP) – selecting the k-th largest number from among n distinct numbers, 1≤k≤n. Prove that the algorithm below is correct (i.e., that it will print out the k-th largest number in A at termination) using the **Loop Invariant**:

"Before any execution of the outermost for loop with i=p, the (p-2) largest numbers in the original array A will be in cells A[1]...A[p-2] arranged in the descending order, and the remaining n-(p-2) numbers will be in cells A[p-1]...A[n]."

Note: Your proof must be written clearly, precisely, and at an appropriate level of detail.

```
SelectK-thLargest (A: Array [1…n] of distinct numbers; k: integer such that 1≤k≤n)
1       for i=2 to (k+1)
2               for j=n downto i
3                       if A[j]>A[j-1] then
4                               temp=A[j]
5                               A[j]=A[j-1]
6                               A[j-1]=temp
7       return A[k]
```

Initialization: prove that the LI holds true before the loop begins.

This step usually does not need proof, just validation that it is true. Before the start of iteration of the loop with i=2, the subarray will contain the necessary elements, but in sorted order. It will also contain K, which is true.

Maintenance: prove that if the LI holds true before an execution of the outermost for loop with i=p, it will be true before the next execution with i=p+1.

The point of the nested loop is to swap the largest number with the number to the left of it, only if the number to the left is smaller. It will continue doing this until the array is in descending order. This will hold true. For each round, i will move one position to the right. For each round of the bubble sort, it will produce at least one more correct addition to the front of the array. This is known as the partial solution sort vs sub part of final solution in insertion sort.

Termination: show that given Initialization and Maintenance proofs, the algorithm will produce the correct answer at termination.

For the example array : 5,8,7,6,10

The inner loop will sort in the following rounds:

1: 10,5,8,7,6

2: 10,8,5,7,6

3: 10,8,7,5,6

4: 10,8,7,6,5

As mentioned above, this will add a partial solution with each round/iteration.

Then the outer loop will find the nth value.

Suppose k=1

then for i=2 to (k+1)

would equal for i=2 to 2, which would then return the first value in the array, 10, which is the 1st largest number.

3. Explain in English, using precise language, why it is true that if f(n)=O(g(n)) then g(n) must be Ω(f(n)). Then explain why the converse is also true.

If f(n) = O(g(n)) then

1. there is a constant c > 0, and a constant n0 such that for all n ≥ n0: f(n) ≤ c*g(n)
2. there is a constant c > 0, and a constant n0 such that for all n ≥ n0: g(n) ≥ (1/c)*f(n)
3. there is a constant k > 0, namely k = (1/c), and a constant n0 such that for all n ≥ n0: g(n) ≥ k*f(n) which is the definition of g(n) = Ω (f(n)).
4. If g(n) = Ω (f(n)) then there is a constant k > 0, and a constant n0 such that for all n ≥ n0: g(n) ≥ k*f(n)
5. There is a constant k > 0, and a constant n0 such that for all n ≥ n0: f(n) ≤ (1/k)*g(n)
6. There is a constant c > 0, namely c = (1/k), and a constant n0 such that for all n ≥ n0: f(n) ≤ c*g(n) which is the definition of f(n) = O(g(n)).

4. Understand how this Bubble Sort algorithm works so that you can modify it to solve the Selection Problem: given an Array [1..n] of distinct numbers and an integer k, 1≤k≤n, return the k-th largest number in the array (the 1-st largest number is the largest number). State the modified algorithm as your answer. Make only the minimum needed modifications to obtain a correct and efficient algorithm. The smallest modification is to add a return statement at the end of the algorithm below "return A[the index of the array cell in which the k-th largest number will be after the entire array is sorted]" and while that gets you a correct algorithm, that is not the most efficient way to modify this sorting algorithm to make it do what you want, and therefore not acceptable.

```
Bubble-sort (A: Array [1..n] of numbers)
1       i=1
2       while i≤(n–1)
3               j=1
4               while j≤(n–i)
5                       if A[j]>A[j+1] then
6                               temp=A[j]
7                               A[j]=A[j+1]
8                               A[j+1]=temp
9                       j=j+1
10              i=i+1
```

```
Bubble-sort (A: Array [1..n] of numbers)
Function swap (A[i], A[j])
Integer: I,j,n,k
1       for i=0 to (n-1)
2               for j=0 to n-i-1
3                       if (A[j] > A[j+1])
4                               swap(A[j], A[j+1])
5       return A[k]
```

5. A recursive algorithm to compute the exponent x^y is given below.

```
power-recursive(x,y: non-negative integers)
    if y == 0 then
            return 1
    else
            if odd(y) then
                    return power-recursive(x,(y–1))*x
            else
                    return power-recursive((x*x),floor(y/2))
```

```
            end if
        end if
```

Understand how this algorithm works and then convert it into a non-recursive algorithm. Part of the solution is given below. Fill in the blanks. Note: Your iterative algorithm should use the same strategy as the recursive one. I.e., an iterative algorithm that does the obvious and the inefficient - multiplying x with itself (y-1) times - will get 0 points.

```
power-iterative(x,y: non-negative integers)
    result = 1
    while y > 0
        if odd(y) then
            result = result * x
            y = y - 1
        end if
        x = x * x
        y = y / 2
    return result
```

6. What is the best estimate (tightest upper bound) on the number of times the while loop of power-iterative will execute?

O(n^2)

What is the approximate complexity of power-iterative?

Squaring and multiplication are both = log2b

The running time is O(log b)

7. Read the *MERGE* algorithm on pp.31 of CLRS book for this problem (also shown as follows). Calculate the complexity of the Merge algorithm using (1) the approximate method and (2) the detailed method. You may assume that $n_1=n_2=n/2$. Fill in the table below appropriately. Note: step 3 of the algorithm is not executable.

```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

Approximate analysis:

Step #	Complexity stated as O(_)
1	O(1)
2	O(1)
4	Complexity of # of executions: O(n)
5	O(1)
Loop 4-5	Complexity of entire loop: O(n^2)
6	Complexity of # of executions: O(n)
7	O(1)
Loop 6-7	Complexity of entire loop: O(n^2)
8	O(1)
9	O(1)
10	O(1)
11	O(1)
12	Complexity of # of executions: O(n)
13	O(1)
14	O(1)
15	O(1)
16	O(1)
17	O(1)
13-17	Complexity of single execution of loop body: O(n^2)
12-17	Complexity of entire loop: O(n^2)
1-17	Complexity of algorithm: O(n log(n))

Detailed analysis:

Step #	Cost of single execution	# of times executed
1	$C_1 = 5$	1
2	C2 = 4	1
4	$C_4 = (n_1 + 1)3$	$n_1 + 1$
5	$C_5 = 5$	n_1
6	$C_6 = (n_2 + 1)3$	$n_2 + 1$
7	$C_7 = 4$	n_2
8	$C_8 = 4$	1
9	$C_9 = 4$	1
10	$C_{10} = 2$	1
11	$C_{11} = 2$	1

12	C_{12} =(r + p)3	r + p
13	C_{13} = 3	r
14	C_{14} = 3	r
15	C_{15} = 3	r
16	C_{16} = 3	r
17	C_{17} = 3	n

T(n)= $5+4+((n_1+1)3)+5+((n_2+1)3)+4+4+4+4+2+2+((r+p)3)+3+3+3+3+3+3$

$=9 + ((n_1+1)3)+5+((n_2+1)3)+16+((r+p)3)+15$

$=45+(3n_1+3)+(3n_2+3)+(3r+3p)$

$=51+3n_1+3n_2+3r+3p$

8. Consider the problem of reversing a string provided as an array A[p...r] of characters.
(a) Design an algorithm to this problem that uses the following strategy: iterate through the array from left to right and from right to left, each time swapping the characters in the current leftmost cell and the current rightmost cell, until the middle of the array is reached. State the algorithm precisely using numbered steps that follow the pseudocode conventions that we use.

```
Reverse String(A: array[p..r])
1       p,r: integer
2       p = 0
3       r = length of A
4       function swap (A[p], A[r])
5       while p < r
6               swap(A[p+1], A[r-1])
7       return A
```

(b) Design another algorithm to this problem that uses the following strategy: recursive divide and conquer that splits the array into two halves each time, reverses the two halves and then combines the two half-solutions correctly. State the algorithm precisely using numbered steps that follow the pseudocode conventions that we use.
Because this technique was new to me, I found it easier to program and test this using Python 2.7. I have listed below my program of divide and conquer **as well as the pseudocode conventions that we use.**

Execute | Share main.py STDIN

```
#Divide and Conquer - Robin Ward
A = [1,54, 35, 42, 55, 6]
newA = []
newLeft = []
newRight = []
A_count = len(A)
A_count_half = A_count / 2
i = 0

if A_count < 2:
    print 'less than 2'

while A_count_half > i:
    left = A[int(i)]
    right = A[int(-i-1)]
    right, left = left, right
    newLeft.append(left)
    newRight.append(right)
    i+=1
newA = newLeft + newRight[::-1]

print newA
```

Result

```
$python main.py
[6, 55, 42, 35, 54, 1]
```

```
1       Reverse string(A: Array [p..r], le: array[], ri: array[],nle: array[], nri: array[],  function swap (A[p], A[r]))
2       p,i = 0
3       r = length of array
4       if r < 2
5               return A
6       while (r / 2) > i
7               le = A[i]
8               ri = A[-i-1]
9               swap(le,ri)
10              nle += le
11              nri += ri
12              i += 1
13      A = nle + nri(reversed)
14      return A
```

(c) Which algorithm is more efficient? Justify your answer by providing an appropriate efficiency analysis that clearly supports your claim.

I would have to say that the first approach would be a lot more efficient. The divide and conquer has a lot more assignments that are necessary for it to work.