# Assignment 1

Name: Mousumi Akter
SID: 904095347
I Discussed with: Alex Knipper and Md Mahadi Hassan

February 12, 2021

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 $z \geq 0$ for which the answer returned, product $\neq z * y$.

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=y 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.

Let's assume the algorithm is incorrect. It's written in the description that z is non-negative and for z=0 and z=1 the algorithm returns the correct answer.

Let z = m for which our algorithm is incorrect and m is a non-negative number. If we can show the final product $\neq m*y$, then we are able to show one instance for which our assumption is true. Let's iterate the algorithm from step 3 to 7:

Iteration 1: z = m, product = y; z = z -1

Iteration 2: z = m-1, product = y+y = 2*y; z = z -2

Iteration 3: z = m-2, product = y+y+y = 3*y; z = z -3

...

Iteration m-1: z = m - (m-2) = 2, product = (m-1)*y; z = 1

Iteration m: z = 1, product = (m)*y; z = 0

So for z=m, the algorithm return correct value of the product. As we fail to show at least one instance for which the algorithm fails. So our initial assumption of the algorithm being incorrect contradicts. Therefore the correctness of the above algorithm is proved by contradiction.

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.

Initially, no part of the array is in descending order because array A is unsorted. So we can say we have an empty set of array elements in descending order.

When i=2, p-2=0, largest numbers in the original array A is in cells A[1]...A[0] arranged in the descending order and the remaining n-(0)=n numbers are in cells A[1]...A[n]

Therefore LI holds true at the beginning of loop as it matches with the initial status of A.

**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.

For the next execution with i=p+1 of the outermost loop, the inner loop will be executed for j=n, n-1, ..., p+1.

When j=n, line 3 checks for A[n]>A[n-1], if so, it swaps A[n] and A[n-1] in line 4 to 6, the larger between these two number will be in A[n-1].

When j=n-1, line 3 checks for A[n-1]>A[n-2], if so, it swaps A[n-1] and A[n-2] in line 4 to 6, the larger between these two number will be in A[n-2].

Therefore, at the end of j=n-1, the largest numbers between A[n], A[n-1] and A[n-2] will be in A[n-2]

Similarly, at the end of j=p+1, the largest numbers between A[n], A[n-1], A[n-2],...,A[p] will be in A[p]

Now in the previous execution with i=p of the outermost loop, the original array A is in cells A[1]...A[p-1] were arranged in the descending order and the remaining numbers were in cells A[p]...A[n] (unsorted part). Therefore, A[p-1]>A[p] because all elements in array are distinct numbers. After execution of i=p+1, from the previous discussion it's clear that A[p] is the largest element of A[p]...A[n]. So this implies after execution of i=p+1, the original array A is in cells A[1], A[2],...,A[p-1]A[p] are arranged in the descending order and the remaining numbers are in cells A[p+1], A[p+2],...,A[n] (unsorted part)

Therefore, we can conclude 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.

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

For the execution with i=k+1 of the outermost loop, the inner loop will be executed for j=n, n-1, ..., k+1.

From the maintenance we have seen, for i=p+1, the original array A is in cells A[1], A[2],...,A[p-1], A[p] are arranged in the descending order.

So, for i=k+1, the original array A is in cells A[1], A[2],...,A[k-1]A[k] are arranged in the descending order. Therefore line 7 will return the kth largest value as A[k]. So the algorithm will produce the correct answer at termination.

3. 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, $1 \leq k \leq n$, 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. (Note that although this question asks about efficiency, you can do it without any knowledge of efficiency analysis.)

```
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
```

Changes are made in Line 2 and Line 11. Modified Algorithm below:

```
KthLargest (A: Array [1..n] of numbers, k)
1       i=1
2       while i≤k
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
11      return A[n-k+1]
```

4. 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 = floor(y/2)
    return result
```

5. 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.

For both question 5(a) and 5(b), assuming array is 0 indexed in the beginning.

```
ReverseString(array A of characters)
1       left = 0
2       right = length(A)-1
3       while right>left:
4               temp=A[left]
5               A[left]=A[right]
6               A[right]=temp
7               left = left + 1
8               right = right – 1
9       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.

```
RecursiveReverseString(array A of characters)
1       if length(A)==1 then
2               return A
3       else
4          mid = floor(length(A)/2)
5          left = RecursiveReverseString(A[0...mid])
6          right = RecursiveReverseString(A[mid+1... length(A)])
7          return right+left
8       end if
```