**Design and Analysis of Algorithms**

**Assignment 1 Solution**

**Loop invariants**

**Question 1: (Find Min)**

**Answer:**

Loop invariant: At the start of each iteration of the loop, the variable min contains the minimum value seen so far in the array A[1…i-1].

Initialization: Before the first iteration of the loop, we have i=2, which means that the loop invariant holds, since we haven't yet looked at any elements of the array.

Maintenance: Suppose that the loop invariant holds at the start of an iteration of the loop, with some value i. Then, we know that min contains the minimum value seen so far in A[1…i-1]. When we consider the next element A[i], there are two cases:

If A[i] < min, then we update min to A[i]. Now, min contains the minimum value seen so far in A[1…i], which is the desired result.

If A[i] >= min, then min still contains the minimum value seen so far in A[1…i-1], which is consistent with our loop invariant.

Therefore, the loop invariant still holds after the current iteration, and we can move on to the next iteration.

Termination: When the loop terminates, we have i = A.length+1, which means that we have looked at all elements of the array A. By the loop invariant, we know that min contains the minimum value seen so far in A[1…A.length-1]. Since we haven't yet considered A[A.length], we know that min contains the minimum value in the entire array A. Therefore, the algorithm is correct.

Hence, we have proven the correctness of the given algorithm using loop invariants.

**Question 2 : (Bubble sort)**

**Answer:**

Loop Invariant: At the beginning of each iteration of the outer loop (line 1), the sub-array A[1…i-1] contains the i-1 smallest elements of A, in sorted order.

Initialization: Before the first iteration of the outer loop, i=1, and A[1…i-1] is the empty sub-array. The loop invariant holds true, since the empty subarray trivially contains the 0 smallest elements of A, in sorted order.

Maintenance: Assume that the loop invariant holds true before an iteration of the outer loop, for some value of i. Then, during the i-th iteration of the outer loop, the inner loop (line 2) swaps adjacent elements of the sub-array A[i…A.length], if they are not in sorted order. By the end of the i-th iteration of the outer loop, the i-th smallest element of A will have been placed in its correct position in the sub-array A[1…i], and the loop invariant will still hold true for the sub-array A[1…i]. Therefore, the loop invariant still holds true before the (i+1)-th iteration of the outer loop.

Termination: When the outer loop terminates, i=A.length-1, and the loop invariant implies that A[1…A.length-1] contains the first A.length-1 smallest elements of A, in sorted order. Therefore, the last remaining element of A, A[A.length], must be the largest element of A, since all other elements are smaller and have been sorted. Therefore, the entire array A is now sorted in ascending order, as desired.

Therefore, we have shown that the Bubble Sort algorithm correctly sorts an array of elements in ascending order, by using loop invariant to prove its correctness.

**Question 3 : (Selection sort)**

**Answer:**

Loop Invariant: At the beginning of each iteration of the outer loop, the subarray A[1...j-1] is sorted in ascending order.

Initialization: Before the first iteration of the outer loop, j=1, so the subarray A[1...j-1] is empty, and hence sorted. So the loop invariant holds true.

Maintenance: Assuming that the loop invariant is true before the jth iteration, we need to prove that it is true after the jth iteration. During the jth iteration, the algorithm searches for the smallest element in the subarray A[j...n], and swaps it with A[j]. By doing this, the smallest element in the subarray A[j...n] is now in position j, and the subarray A[1...j] is sorted in ascending order. Therefore, the loop invariant is still true after the jth iteration.

Termination: When the outer loop terminates, j=n-1, which means that the subarray A[1...n-2] is sorted in ascending order. The final iteration of the inner loop finds the smallest element in the subarray A[n-1...n], and swaps it with A[n-1]. Therefore, the entire array A is sorted in ascending order.

Since the loop invariant holds true before and after each iteration of the outer loop, and since the loop terminates with a sorted array, we can conclude that the selection sort algorithm is correct.

**Designing Algorithms[40 marks]**

**Question 1:**

Pseudocode:

function replaceWithProduct(arr):

n = length of arr

left\_products = array of n integers

right\_products = array of n integers

result = array of n integers

left\_products[0] = 1

right\_products[n-1] = 1

# Calculate the products of elements to the left of each element

for i from 1 to n-1:

left\_products[i] = left\_products[i-1] \* arr[i-1]

# Calculate the products of elements to the right of each element

for i from n-2 to 0:

right\_products[i] = right\_products[i+1] \* arr[i+1]

# Calculate the product of every other element of the array

for i from 0 to n-1:

result[i] = left\_products[i] \* right\_products[i]

return result

**Question 2:**

Pseudocode:

function removeDuplicates(arr):

n = length of arr

sorted\_arr = mergeSort(arr) # sort the array using merge sort

j = 0

rm\_duplicates=[]

rm\_duplicates[0]=sorted\_arr[0]

# Traverse the sorted array and remove duplicates

for i from 1 to n-1:

if rm\_duplicates[j] != sorted\_arr[i]:

j = j + 1

rm\_duplicates[j] = sorted\_arr[i]

return rm\_duplicates

**Question 3:**

Pseudocode:

function findIndex(arr, low, high):

if low > high:

return -1 # No such index exists

mid = (low + high) / 2

if arr[mid] == mid:

return mid # Index found

if arr[mid] > mid:

return findIndex(arr, low, mid - 1) # Check left half

return findIndex(arr, mid + 1, high) # Check right half

**Question 4**

Pseudocode:

function kth(arr1, arr2, end1, end2,k):

if (arr1 == end1)

return arr2[k];

if (arr2 == end2)

return arr1[k];

int mid1 = (end1 - arr1) / 2;

int mid2 = (end2 - arr2) / 2;

if (mid1 + mid2 < k):

if (arr1[mid1] > arr2[mid2])

return kth(arr1, arr2 + mid2 + 1, end1, end2,k - mid2 - 1);

else

return kth(arr1 + mid1 + 1, arr2, end1, end2, k - mid1 - 1);

else:

if (arr1[mid1] > arr2[mid2])

return kth(arr1, arr2, arr1 + mid1, end2, k);

else

return kth(arr1, arr2, end1, arr2 + mid2, k);