Name: Kevina Wong

ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**

Problem Set 1 – Due Jan 24 11:55pm **Spring 2020, CU-Boulder**

Advice 1: For every problem in this class, you must justify your answer: show how you arrived at it and why it is correct. If there are assumptions you need to make along the way, state those clearly.

Advice 2: Informal reasoning is typically insufficient for full credit. Instead, write a logical argument, in the style of a mathematical proof.

Instructions for submitting your solutions:

- The solutions **should be typed** and we cannot accept hand-written solutions. Here's a short intro to LaTeX.
- You should submit your work through the **class Canvas page** only.
- You may not need a full page for your solutions; pagebreaks are there to help Gradescope automatically find where each problem is. Even if you do not attempt every problem, please submit this template of at least 9 pages (or Gradescope has issues with it).

Quicklinks: 1 2a 2b 2c 3 4a 4b 4c 4d

---

1. *What are the three components of a loop invariant proof? Write a 1–2-sentence description for each one.*

   The three components of a loop variant proof are:

   - **Initialization:** This steps makes sure that the condition(s) is/are true prior to the first iteration of the loop. This step is similar to a base case in induction proofs where we make sure that the condition holds true for the initial cases.
   - **Maintenance:** If a condition holds true before the loop iterates, then it will continue to be true for future iterations. This step is similar to the inductive step in induction proofs.
   - **Termination:** When the loop terminates, the loop invariant shows that the algorithm is correct by giving us a useful property

Name: Kevina Wong
ID: 109179049

2. *Identify and state a useful loop invariant in the following algorithms. You* do not *need to prove anything about it.*

   (a)
```
FindMinElement(A) : //array A is not empty
    ret = A[length(A)]
    for i = 1 to length(A)-1 {
        if A[length(A)-i] < ret{
            ret = A[length(A)-i]
    }}
    return ret
```

   Answer: At the start of each iteration of the for loop, the variable ret holds the smallest element in the subset A[length(A)−i+1:length(A)].

(b)
```
LinearSearch(A, v) : //array A is not empty and has no duplicates
    ret = -1 //index -1 implies the element haven't been found yet
    for i = 1 to length(A) {
        if A[i] == v{
            ret = i
    }}
    return ret
```

Answer: At the start of each iteration of the for loop, if ret $= -1$, v is not found in the subset A[1:i], or if ret $\neq -1$, then v is found in the subset A[1:i] with the index A[ret].

(c)
```
ProductArray(A) : //array A is not empty
        product = 1
        for i = 1 to length(A) {
            product = product * A[i]
        }
        return product
```

Answer: At the start of each iteration of the loop, the variable product holds the product of all of the elements in the subarray A[1:i].

Name: Kevina Wong
ID: 109179049

3. *The algorithm 2a is a standard find-min operation: it is supposed to return the element of minimum value in A. Use a loop invariant proof to show the algorithm 2a from the preceding question is correct.*

   Here is a scaffold of the proof to get you started.

   We will use the following as our loop invariant: At the start of each iteration of the for loop, the variable ret holds the smallest element in the subset A[length(A)−i+1:length(A)].

   Initialization: We need to show that the loop invariant holds before the first loop. In this case, it is when i=1. In the beginning of the pseudocode, ret is set to A[length(A)]. The subset A[length(A)−i+1:length(A)] only holds one element, so naturally, it is trivial that ret is the minimum element in its subset, which is A[length(A)]. This shows that the loop invariant holds true prior to the first iteration of the loop.

   Maintenance: Due to the if statement, this loop invariant is able to hold true. This is because every time the loop iterates, the if loops checks the value and replaces ret with a new minimum value if there is one for that iteration. Therefore, if the loop invariant holds true for the i^{th} iteration, then the loop invariant will hold true for the $i+1^{th}$ iteration.

   Termination: The loop will terminate when i > length(A)−1. The first instance it terminates is when i = length(A). The subset of this array will be A[length(A)-length(A)+1:length(A)] which is the same as A[1:length(A)] (which is the entire sub-array). Therefore, it means every element of the original array will be checked, and when the loop terminates, ret holds the minimum element of the A[1:length(A)], and the function returns the correct minimum element of the array. Thus, the function gives a "useful property" and shows the correctness of this algorithm.

4. *Let $A = [a_1, a_2, \ldots, a_n]$ be an array of numbers. Let's define a 'reverse' as a pair of distinct indices $i, j \in \{1, 2, \ldots, n\}$ such that $i < j$ but $a_i > a_j$; i.e., a_i and a_j are out of order.*

   For example - In the array A = [1, 3, 5, 2, 4, 6], (3, 2), (5, 2) and (5, 4) are the only reverses i.e. the total number of reverses is 3.

   (a) *Let A be an arbitrary array of length n. At most, how many reverses can A contain in terms of the array size n? Explain your answer with a short statement.*

   Answer: At most, the number of reverses that can be contained in an array of size n is

$$\frac{(n)(n-1)}{2}$$

   . An example worst case array for this problem is B = [6,5,4,3,2,1] (n=6). Here is a list of reverses for this case:

   (6,5), (6,4), (6,3), (6,2), (6,1)
   (5,4), (5,3), (5,2), (5,1)
   (4,3), (4,2), (4,1),
   (3,2), (3,1),
   (2,1)

   Therefore, I concluded that at most, the maximum number of reverses for array B is $5 + 4 + 3 + 2 + 1$. Thus, an overall general representation of the maximum number of reverses in any array with size n is

$$\frac{(n)(n-1)}{2}$$

   .

(b) *We say that A is sorted if A has no reverses. Design a sorting algorithm that, on each pass through A, examines each pair of consecutive elements. If a consecutive pair forms a reverse, the algorithm swaps the elements (to fix the out of order pair). For instance, if your array A was [4,2,7,3,6,9,10], your first pass should swap 4 and 2, then compare (but not swap) 4 and 7, then swap 7 and 3, then swap 7 and 6, etc. Formulate pseudo-code for this algorithm, using nested for loops.* ***Hint:*** *After the first pass of the outer loop think about where the largest element would be. The second pass can then safely ignore the largest element because it's already in it's desired location. You should keep repeating the process for all elements not in their desired spot.*

Answer: This description is essentially a bubble sort.

```
sortArray(A){
    for i = 1 to length(A){
        for j = 1 to length(A) - 1 {
            if A[j] > A[j+1]{
                swap A[j] and A[j+1]
            }
        }
    }
    return A
}
```

(c) *Your algorithm has an inner loop and an outer loop. Provide the "useful" loop invariant (LI) for the inner loop.You don't need to show the complete LI proof.*

Answer: At the start of each iteration of the loop, the subarray A[1:j] will contain elements from the original array A, and subarray A[j] will the the largest element in the subset A[1:j].

(d) *Assume that the inner loop works correctly. Using a loop-invariant proof for the outer loop, formally prove that your pseudo-code correctly sorts the given array. Be sure that your loop invariant and proof covers the initialization, maintenance, and termination conditions.*

Loop Invariant (LI): At the start of each iteration of the loop, the subarray A[length(A)-i+1:length(A)] contains element from the original array A, but sorted.

- **Initialization:** We need to show that the loop invariant holds before the first iteration of the loop, which is when i = 1. When i = 1, the subarray consists of A[length(A)-1+1:length(A)] = A[length(A)]. Since the subarray only consists of one element, it is trivial that this array is already sorted. This shows that the loop invariant holds prior to the first iteration of this loop.
- **Maintenance:** Due to the inner for loop and the if statement, the loop invariant is able to hold true for every iteration of the outer loop. The inner loop and the if statement ensures that the largest element swaps to the right as long as A[j]>A[j+1]. It keeps swapping right until it is placed in its proper position. The subarray A[length(A)-1:length(A)] consists of elements in the original array A, but in sorted order. Furthermore, due to the inner for loop and the if statement, if the loop holds for the i^{th} iteration, it will be true for the $i+1^{th}$ iteration. Thus, the loop invariant is preserved.
- **Termination:** The loop terminates when i < length(A). This can be when i = length(A). Therefore, the subarray A[length(A)-length(A)+1] = A[1:length(A)] (which is the entire array) will be sorted when the loop terminates. Because the entire sorted array A is returned, we can conclude that the algorithm is correct.

Name: Kevina Wong
ID: 109179049

CSCI 3104, Algorithms **Profs. Chen & Grochow**
Problem Set 1 – Due Jan 24 11:55pm **Spring 2020, CU-Boulder**

References:

- Prof. Chen's Week 1 Lecture Notes
- Proof.pdf on canvas posted by Prof Grochow
- HackerEarth Bubble Sort Visualizer (For refereshser on bubble sort)