CS311: Homework #1

Due on September 11, 2013 at 4:30pm

Professor Lathrop Section 3

Josh Davis

Problem 1

Demonstrate that the quicksort implementation given is incorrect by providing an input on which the algorithm does not produce correctly sorted input. Briefly explain why the input fails.

Solution
The following input $list = [3, 3]$, $start = 0$, and $end = 1$, will cause the given implementation of quicksort to loop infinitely.

Explanation
The reason this input fails is because when the first call to *partition* happens, $left = start + 1 = 1$ and $right = 1$. The *while* loop on line 3 waits until $left \leq right$.

However, $left$ and $right$ can never advance past each other because neither line 4 or 7 take into the account the possibility when the pivot value equals either $list[left]$ or $list[right]$. To fix this, all we need to do is change line 4 to be less than **or equal**, $list[left] \leq list[start]$. Now $left$ can be incremented and cause the *while* loop to finish.

There is one last issue. When *partition* returns, it will set $mid = 1$. The problem though is that we recursively call *quicksort* on $start = 0$ and $end = 1$. Which is the exact same as the first call that we made to *quicksort*. To fix this, the first call to *quicksort* should pass $mid - 1$ instead of mid.

This is what the final code should look like:

```
1   function QUICK-SORT($list, start, end$)
2       if $start \geq end$ then
3           return
4       end if
5       $mid \leftarrow$ PARTITION($list, start, end$)
6       QUICK-SORT($list, start, mid - 1$)
7       QUICK-SORT($list, mid + 1, end$)
8   end function
```

and:

```
1   function PARTITION($list, start, end$)
2       $left \leftarrow start + 1$
3       $right \leftarrow end$
4       while $left \leq right$ do
5           while $left \leq end \land list[left] \leq list[start]$ do
6               $left \leftarrow left + 1$
7           end while
8           while $right \geq start \land list[right] > list[start]$ do
9               $right \leftarrow right - 1$
10          end while
11          if $left \leq right$ then
12              SWAP($list, left, right$)
13          end if
14      end while
15      SWAP($list, start, right$)
16      return $right$
17  end function
```

Problem 2

Part A

Proof. □

Part B

Proof. □

Part C

Proof. □

Part D

Proof. □

Problem 3

Using the implemenation of insertion sort given, solve the following problems.

Part A
Identify a useful invariant I_0 for the outer loop, that is, a loop invariant which will allow you to complete parts b through d.

Solution
The loop invariant will be the following:

When looping, all of the original elements in $list[0 \dots i-1]$ will still be there but will be in sorted order.

Part B
Prove that if I_0 holds after the last iteration of the outer loop completes, then the algorithm sorts $list$ in ascending order.

Proof. We will prove that I_0 holds after the last iteration and that the array is sorted.

The for loop in the code iterates from $i = 1$ all the way up to $list.length$. When the for loop terminates, $i = list.length$, which will mean that $list[0 \dots i-1]$ will consist of the first i elements in the array in sorted order.

Since the range contains the whole array, it is obvious that all of A will now be sorted, which concludes the proof and the **Termination** part of the loop invariant. □

Part C
Prove that I_0 holds before the loop is executed for the first time.

Proof. We will prove that I_0 holds before the loop is executed for the first time.

The for loop in the code iterates from $i = 1$ all the way up to $list.length$. When the for loop starts, $i = 1$, which will mean that $list[0 \dots i-1]$ will consist of just $list[0 \dots 0]$ elements, or one element.

This is trivial because any range with one element is always sorted. This concludes the proof and the **Initialization** part of the loop invariant. □

Part D
Prove that I_0 is an invariant for the outer loop.

Proof. To prove that I_0 is an invariant for the **Maintenance** step, we'll have to prove that lines 4–7 are also correct by using another loop invariant.

The inner loop invariant, I_1, will be as follows:

Each iteration the elements in the range $list[j \dots i]$ will be greater or equal to the value v.

Initialization
At the beginning of the loop, $j = i$ (line 3). Which means that the range consists of the elements $list[i \dots i]$, or just one element. Since v is initialized to be equal to $list[i]$, trivially, any value is always greater or equal to itself. Thus the invariant holds.

Maintenance

Problem 3 continued on next page. . .

During each iteration of the loop, the value of $list[j-1]$ is moved to the position $list[j]$ because of line 5. The value of j is then decremented by one because of line 6.

This now means that $list[j]$ and $list[j-1]$ contain the same value. This is important because it maintains the loop invariant, I_1. Since $list[j]$ and $list[j-1]$ are in ascending order the loop invariant holds because all the elements in the range $list[j \ldots i]$ are in ascending order as well. Thus the loop invariant holds.

Termination
The while loop ends either when $j \leq 0$ or $list[j-1] \leq v$.

In the first case, when $j = 0$, this would mean that every value in $list[0 \ldots i]$ is greater than or equal to v. Therefore there can be no element that could be less than v since 0 is the beginning of the list. Thus the loop invariant holds.

In the second case, when $list[j-1] \leq v$, it means that $list[j \ldots i]$ is all greater than or equal to v because if the value $list[j-1]$ were to be in the range, the loop invariant would no longer hold. Thus the loop invariant holds.

Now that we can see that the loop invariant I_1 is correct we can now continue with the I_0 invariant.

According to the loop invariant I_1, we can see that every value $list[j \ldots i]$ is greater or equal to v. We know that all values from $list[0 \ldots j-1]$ are the original elements but in sorted order. By inserting v into the position $list[j]$ on line 8, we are placing v into a spot such that all values $list[0] \leq list[1] \ldots list[j-2] \leq list[j-q] \leq v$ and $v < list[j+1] \leq list[j+2] \ldots list[i-1] \leq list[i]$.

Therefore we have proven the **Maintenance** step of the loop invariant I_0 with the help of the other loop invariant, I_1. Thus this concludes our proof. $\square$

Problem 4

Prove or disprove $(A - B) \cup C = (A \cup B) - C$.

Counterexample
Let $A = \emptyset$, $B = \emptyset$ and $C = \{1\}$. Which gives this for the left hand side:

$$(A - B) \cup C = (\emptyset - \emptyset) \cup \{1\} = \emptyset \cup \{1\} = \{1\}$$

And the right hand side:

$$(A \cup B) - C = (\emptyset \cup \emptyset) - \{1\} = \emptyset - \{1\} = \emptyset$$

Since the left doesn't equal the right, the proposition has been disproven.

Problem 5

Given an instance of the longest common subsequence problem.
Solution
Given two strings, $s =$ "$abcdefg$" and $t =$ "$biczdxyg$", the longest common subsequence between the two is "$bcdg$".

Problem 6

One hundred quarters lie scattered about on a table before you. Some known n of them are heads up, while the rest are tails up. You cannot in any way observe the configuration of individual quarters (perhaps you are blindfolded). You can, however, pick them up and place them back on the table in the opposite configuration, ie flip them.

Your objective is to divide the one hundred quarters into two groups such that each group has the same number of heads. The two groups need not have the same number of quarters. How do you do this?

Solution
Given the number of coins is 100, we can get the same number of quarters in both groups by splitting the quarters into two groups of 50. Since there are n heads in both groups, there is i number of heads from 0–n in the first pile, and then j number of heads in the second such that $j = n - i$.

Given we know the number of n, there are two possibilities for n. Either $n \leq 50$ or $n > 50$. If the number of heads is greater than 50, then the first thing we do is flip all the coins so that $n = 100 - n$.

Next all we do is take n number of coins from the pile and flip each one once and place it into a separate pile. When selecting the coin from the pile of 100, there are two possibilities. If the coin we select is tails, we will flip it resulting in a heads added to the other pile and bringing the two piles closer to the same number of heads. If the coin we select is a heads, then we will be reducing the number of heads in the first pile and still bringing the number of heads in the first pile closer to the number in the second pile. By performing this n times, we will end up with the same number of heads in both piles.