Marco Zuniga

COMP 3270

HW 1

June 1, 2018

1. Having a machine running at 4GHZ which requires 200 clock cycles to execute one computation step, the machine can then run 2e7 steps in each second.

$$\frac{4e9\ clock\ cycles\ per\ second}{2e2\ clock\ cycles\ per\ step} = 2e7\ steps\ per\ second$$

Based on the algorithm being used, one would divide the number of steps needed to be computed by the number solved for above.

A1: $\frac{200\ million\ steps}{2e7\ steps\ per\ second} = 10\ s$

A2: $\frac{200\ million \log(200\ million)\ steps}{2e7\ steps\ per\ second * \log 2} = 275.754\ s$

A3: $\frac{(200\ million)(200\ million)\ steps}{2e7\ steps\ per\ second} = 2e9\ s = 23148.1\ days$

2. Create an efficient way for a driver to pick up customers at different locations in the timeliest manner with a target destination in the end.

(1) The solution begins by creating objects of all customers that have an input of name, latitude, and longitude.
(2) The application will use a List as the data structure to hold the customer objects.
(3) The desired output will just be the correct customer that needs to be picked up next.

Algorithm/Strategy:
The application will use an algorithm to find the nearest customer to the driver. A for-loop can be used to search through all customers. The first customer in the List will be considered the "key" and will be compared to the following customer in the List. If the following customer is closer to the driver then the "key" customer, then the following customer will now become the "key" customer. Once a customer is picked up, they will be removed from the List. This will be repeated until all customers have been picked up.

3. To find the i largest numbers in each set of n numbers, one could first use Mergesort to sort the numbers from least to greatest. Prior to sorting the set of n numbers, one could remove the beginning and last numbers to reduce the number of steps Mergesort will have to perform. Next, the two values that were removed in the beginning can be added to the end of the sorted set of numbers [2..n – 1]. Quicksort can then be used to put the

last two numbers in their appropriate positions. After this has been completed, one can just return the i largest numbers. Mergesort has a time complexity of O(nlog(n)). Since we're removing two values from the set of n numbers, the number of steps for using Mergesort will be (n – 2)*log(n – 2). The time complexity of Quicksort will be O(nlog(n)) because all but two of the numbers in the set are already sorted. The number of steps for Quicksort will be nlog(n). We want to use Mergesort the first time because we don't know how well the numbers will be sorted originally. Since we know that Quicksort will only have to sort two numbers, its time complexity will be the same but the space complexity will be smaller as opposed to using Mergesort. Adding these two up will give you nlog(n – 2) – 2log(n – 2) + nlog(n) steps.

4. The problem required the student to use the following algorithm below for a given set of input values and state the output.

```
Algorithm Mystery (A:array[1..n] of integer)
   sum, max: integer
1  sum = 0
2  max = 0
3  for i = 1 to n
4    sum = 0
5    for j = i to n
6       sum = sum + A[j]
7       if sum > max then
8          max = sum
9  return max
```

(a) A: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
Output: 55
(b) A: [-1, -2, -3, -4, -5, -6, -7, -8, -9, -10];
Output: 0
(c) A: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
Output: 0
(d) A: [-1, 2, -3, 4, -5, 6, 7, -8, 9, -10];
Output: 14

What does the algorithm return when the input array contains all negative integers?
Answer: 0. Adding negative numbers won't make sum greater than the original max – 0.
What does the algorithm return when the input array contains all non-negative integers?
Answer: All values added together.

5.

Step	Big-Oh Complexity
1	O(1)
2	O(1)
3	O(n)
4	O(1)
5	O(n)
6	O(1)
7	O(1)
8	O(1)
9	O(1)
Complexity of the algorithm	$O(n^2)$

6.

Step	Cost of each execution	Total # of times executed
1	1	1
2	1	1
3	1	n + 1
4	1	n
5	1	$\sum_{i=1}^{n} i + 1$
6	6	$\sum_{i=1}^{n} i$
7	3	$\sum_{i=1}^{n} i$
8	2	$\sum_{i=1}^{n} i$
9	1	1

$T(n) = 12n^2 + 18n + 8$

7. The student was required to prove whether the algorithm below is correct or incorrect.

```
Count(f: input file)
count, i, j : integer
count = 0
while (end-of-file(f) = false)
   i = read-next-integer(f)
   if (end-of-file(f) = false) then
      j = read-next-integer(f)
      if i=j then count = count+1
return count
```

The student chose to use proof by counterexample to see if the algorithm is incorrect.
Solution:
Assuming a set of numbers '1112' were passed into the Count method, i will hold the first number in the sequence; 1. Because there are still more values in the input file, j will then hold the number 1, the second number in the sequence. The method will check if these two are identical, in which they are, and increment count. Because there are more numbers in the input file, the loop will continue. The value of i will now be 1, the third number in the sequence, and the value of j will now be 2, the last value in the sequence. The method will now see if the third '1' and 2 are equal, in which they aren't. The loop now ends. The value of count is only 1 when it should be 3. This algorithm is incorrect because it only checks if two identical numbers appear consecutively in pairs. When the loop executes for the second time, i will obtain the third value and compare it with the fourth value, disregarding whether the value before i was the same.

8. Suppose that *A* does not collect enough information to determine which elements are greater than the kth largest element and which are elements are less than it. Because the kth largest number is a sequence of numbers n, partitioning around the kth largest element one can look to the left or to the right of the kth largest element to see which elements are greater or less than it, which is a contradiction to our assumption that *A* does not collect enough information to determine which elements are greater and less than the kth largest element.

9.

```
Function g(n: nonnegative integer)
  if n <= 1 then return(n)
  else
  return(5*g(n-1)-6*g(n-2))
```

The student will use proof by induction to show that the algorithm *g* is correct.
Base Case: When n is equal to 0, $3^0 – 2^0 = 0$. When n is equal to 1, $3^1 – 2^1 = 1$. This shows that the base case holds.
Inductive Hypothesis: Function *g*(n) is equal to $3^n – 2^n$ for $0 \leq n \leq k$ for some k.
Inductive Step:
Now we must show that Function *g*(n + 1) generates the correct output.
For this to be true, the last return statement of 5*g(n + 1 – 1) – 6*g(n + 1 – 2) must return $3^{n+1} – 2^{n+1}$.
This can also be written as 5*g(n) – 6*g(n -1). Since we already proved that the algorithm holds for values of 0 (an even number) and 1 (an odd number) for n in our base case, entering a number n that is greater than or equal to 2 will hold the condition that the function must take in a nonnegative number, therefore the algorithm holds true.

10.

```
Algorithm Max(A: Array[1..n] of integer)
begin
  max = A[1]
  for i=2 to n
    if A[i] > max then max = A[i]
  return max
end
```

Loop-Invariant:
Before the start of the i^{th} iteration, where $2 \leq i \leq n$, ‘max’ will hold the first value in the Array passed into the Max method.
Initialization:
Before the start of the first iteration of the loop, max holds A[1] which is the first value in the Array passed into the Max method.
Maintenance:

- Suppose that before the i^{th} execution of the loop, ‘max’ will hold the largest value found in Array[1..i-1].
- During the i^{th} iteration of the loop, the for-loop compares the previous ‘max’ value to the i^{th} value in the Array.
- The first case would be if the i^{th} value is greater than the current value in ‘max’, Array[i] will then be set to ‘max’.

- The second case would be if the i^{th} value is less than or equal to the current value in 'max', the value in 'max' will then remain unchanged.
- During the $(i + 1)^{th}$ iteration, the value in Array[i + 1] will still be compared to 'max' which will hold the previous maximum value found in Array[1..i], perform the same comparisons, and make the appropriate changes.

Termination:

- Initialization showed that the LI will be true before the loops begins.
- Maintenance showed that if the LI was true before an iteration, it would still be true before the next iteration.
- LI will be true before any iteration of the loop.
- The loop ends after the n^{th} iteration, which is before the $(n + 1)^{th}$ iteration.
- The LI must be true at that time.
- Before the start of the $(n + 1)^{th}$ iteration, the value of Array[n] was compared to the maximum value found in Array[1..n – 1].
- Thus, the algorithm will return the maximum value found in Array[1..n].

11.

```
Algorithm Convert(n: positive integer)
output: b (an array of bits corresponding to the binary representation of n)
begin
  t = n
  k = 0
  while (t > 0)
    k = k + 1
    b[k] = t mod 2
    t = t div 2   (div refers to integer division)
end
```

Use the following **loop invariant**: If m is the integer represented by the binary array $b[1..k]$ then $n = t2^k + m$

Initialization:

Before the start of the while-loop, t is equal to n and k is equal to 0. Therefore $n = n2^0 + 0 = n$ and the LI holds true. The value of 'm' is 0 because no values have been inputted into the array 'b'.

Maintenance:

- Suppose that before the first execution, the loop invariant $n = t2^k + m$ holds.
- Once the loop starts, k will be equal to 1 and the first index in the array 'b' will now hold either a '1' or '0'. Since we know 'k' is some integer value and 't' was initialized to 'n', we also know that 'm' will represent some integer – in this case either a '0' or '1' after the first iteration.
- The first case would be if the initial value passed into the method was '1', then the loop will execute once, insert the value 1 in the array 'b' and terminate.
- The other case would be if $n > 2$.

- Before the start of the $(t - 1)^{th}$ execution, 'k' was incremented by 1 to insert a new value. Therefore, the value of 'm' will continue to change after each iteration.
- Since anything divided by 2 will mod a 0 or 1, the new element inserted into the array will be a valid integer. 't' will continue to get smaller as it is divided by 2 during each iteration.

Termination:

- Initialization showed that the LI will be true before the loops begins.
- Maintenance showed that if the LI was true before an iteration, it would still be true before the next iteration.
- The LI will remain true before any iteration of the loop.
- The loop terminates after the n^{th} iteration and before the $(n + 1)^{th}$ execution, or once t is decremented to 0. The LI must be true at that time.
- Prior to the final iteration of the loop, the array 'b' will hold b[1..k-1]
- Once t decrements to 0, the array 'b' will hold b[1..k–1+1] = b[1..k].
- 'm' will still represent the integer represented by the binary array b[1..k].
- 't' will now be equal to 0 and $n = (0)*(2)^k + m$, resulting in m being equal to the initial positive integer passed into the Convert method. Therefore, the LI still holds and the algorithm works correctly.

12. The student was required to describe a recursive algorithm to reverse a string that uses the strategy of swapping the first and last characters and recursively reversing the rest of the string.

(a)

```
Algorithm ReverseString(A[p..q]: array of string characters)
output: A[p..q] in reverse order
begin
1   if A[p..q] is empty return A[p..q]
2     *Base Case
3   i = 1
4   j = q – p + 1
5   if j = i return A[p..q]
6     *Will only happen when input string array has 1 character
7   else
8   swap(A[i], A[j])
9   x ← first-character(A[p..q])
10  y ← last-character(A[p..q)
11  delete-first-character(A[p..q])
12  delete-last-character(A[p..q])
13  temp ← concatenate(x, ReverseString(A[p..q]))
14    *The strings between x and y will eventually return reversed. Just attach x back to
      front of the string and attach y to the end of the string.
15  return concatenate(temp, y)
```

(b) Algorithm's recursive tree for input string "i<33270!".