### **ICS 311, Spring 2021, Problem Set 01, Topic 2**

**Feimei Chen <feimei@hawaii.edu>**

Please include your name in this document as well as in the file name.

### **#1. The correctness of Linear Search**

#### **7 points**

**(a)** Show the pseudocode for Linear Search that you will be analyzing. (It should be code that you understand and believe is correct. You may revise your solution from class, or use the instructor’s solution.) Give each line a number for reference in your analysis.

Linear Search(A, v) // A is a array, V is the value that we look for

1. For i = 0 to n-1 // index form 0 to n-1

2. if A[i] == v

3. return i

4. return NIL

**(b)** Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties (page 19 CLRS).

Loop invariant: in any subArray A[1...i-1] do not contain v value.

**(initialization):**

Trivially true prior to the first iteration of the loop because of the subArray A[0.. i-1]. when i is 0, this subArray an empty array and there is no value that can be found.

**(Maintenance):**

if line 2 is true, so A[i] equal the v value, and the loop is terminated and returns the index of v value. If not, A[i] not equal to V, so increment i for the next iteration of the for loop then preserves the loop invariant

**(Termination)**

In line 1, the loop terminates when i > A.lenth -1 (A.length == n).Because each interaction increment i , so we have i = n in that time, the loop invariant show that in subArray[0... n-1] do not have v value and return NIl that indicates not find v value in the array

In line 3, the loop terminates when in line 2 A[i] equal to v value, then it returns the value and stops to increment i. In other words, it does not found value is subArray[0 ….i-1]

The loop invariant is true.

(Hint: The loop can exist for two reasons. Rather than trying to write and prove an invariant that covers the two cases, use a simpler invariant that deals with only the correctness of the loop completion returned value, as it would be too complex to cover both, and the within-loop returned value is easy to show correctly. This is similar to what we did in class for binary search.)

### **#2. Runtime of BinarySearch**

#### **8 points**

This problem steps you through a recursion tree analysis of BinarySearch to show that it is Θ(lg *n*) in the worst case. Here is a recursive version of BinarySearch:

**BinarySearch**(x, A, low, high)

1 **if** (low > high)

2 **return** “NOT FOUND” // or a sentinel such as -1

3 **else**

4 mid =⎣(low + high) / 2⎦

5 **if** x < A[mid]

6 **return** BinarySearch(x, A, low, mid-1)

7 **else if** x > A[mid]

8 **return** BinarySearch(x, A, mid+1, high)

9 **else**

10 **return** mid

(Comments: The problem involves mathematical equations and drawing diagrams. It is your choice whether to figure out how to do this in Google Docs & Drawing, to do it in your favorite program, or to do your work on paper and scan or photograph it. We prefer that you compile the results in one PDF, but if that is difficult compile them into one zip file.)

**(a) Write the recurrence relation for BinarySearch**, using the formula T(*n*) = *a*T(*n*/*b*) + D(*n*) + C(*n*). (We'll assume T(1) = some constant *c*, and you can use *c* to represent other constants as well, since we can choose *c* to be large enough to work as an upper bound everywhere it is used.)

As we know if x value < A[mid], we will remove A[mid … high] those n/2 value in rigional array[0... mid…. High], which contribute T(n/2). And D(n) = C and C(n) = C and base case also be constant C1

T(n) = C if n =1,

T(n) = T(n/2) +C+C if n>1

T(n) = T(n/2) + C if n>1

**(b) Draw the recursion tree for BinarySearch,** in the style shown in podcast 2E and in Figure 2.5 of CLRS. Don't just copy the example for MergeSort: it will be incorrect. Make use of the recurrence relation you just wrote!

T(n) C

| |

T(n/2) C

| ======> |

T(n/4) C

: :

T(1) C

**(c)** Using a format similar to the counting argument in Figure 2.5 of the text or of podcast 2E, **use the tree to show that BinarySearch is Θ(lg *n*) in the worst case**. Specifically,

1. show what the row totals are, c

2. write an expression for the tree height (justifying it), and

Lg+ 1 because assume n is the exact power of 2, so using (lgn

) times to know how many times n can be dived by 2. for example 8 can be divide by 2 only 3 times(2^3 = 8) so there are lgn + 1 levels(log base on 2)

3. use this information to determine the total computation represented by the tree.

c(lgn +1) = clgn + c = **Θ(lg *n)***

### **#3. Correctness of BubbleSort**

#### **15 points**

BubbleSort is a well known but inefficient sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order, and doing so enough times that there can be no more elements out of order. From CLRS:

BUBBLESORT(A)

1 **for** i = 1 **to** A.length - 1

2 **for** j = A.length **downto** i + 1

3 **if** A[j] < A[j - 1]

4 exchange A[j] with A[j - 1]

Let A' (an array) denote the output of BubbleSort. In order to prove that BubbleSort is correct, we need to show that

1. A' is a permutation of A; that is, A' has the same elements as A.

2. A'[1] ≤ A'[2] ≤ A'[3] ... ≤ A'[n-1] ≤ A'[n]; that is, A’ is sorted

The first part can be proven by showing that BubbleSort never removes or adds items to the array: it merely swaps them in line 4. In this homework, you will use loop invariants to prove the second part, that the resulting array is in order. Since there are nested loops this requires two-loop invariants, handled in (a) and (b).

**(a)** State precisely a loop invariant for the **for** loop in lines 2-4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

**Loop invariant**

At the begion of each interation of inner for loop, A[j] the minimal value in subArray[j...n]

Andj from a.lenth to i

**Initialiazation**: Trivially true prior to the first iteration of the loop because j = a.length, so there are only one element in the subArray[n...n] and it is a smallest value in this subarray.

**Maintenance:**

In line 3-4, when enter the inner for loop, A[j] is smallest value at first of A[j...n], and then if A[j] < A[j-1], then they swap value to keep A[j-1] have smaller value. Therefore, A[j-1] is the smallest value in in subarray[j-1...n]. so decrease j for the next iteration of the for loop then preserves the loop invariant.

Termination:

Inner loop terminates when j equal i, the smallest value occur in A[i] in array[i… j] , then the loop invariant is true.

**(b)** Using the termination condition of the loop invariant proved in part (a), state a loop invariant for the **for** loop in lines 1-4 that will allow you to prove the inequality A'[1] ≤ A'[2] ≤ A'[3] ... ≤ A'[n-1] ≤ A'[n]. Your proof should use the structure of the loop invariant proof presented in Chapter 2 of CLRS.

**Loop invariant**:

the begin of each interation of outer for loop, the subArray[1...i-1] have i -1 smallest value that are sorted.

**Initialization:** Trivially true prior to the first iteration of the loop, because i = 1 so the subArray is empty

**Maintenance: in line 1 to 4** From(a),we know each inner loop find the smallest A[i] in subArray[i...n]. And Suppose that before a given iteration of the loop, the smallest i-1 elements were in the first i-1 places in the array, in sorted order, so we have i smallest elements in the first i place i the array in sorted order. And before this given iteration we can not alter the i-1 elements that are sorted.

At the start of the next iteration, i increments, so smallest i-1 elements are in the first i-1 places in the array, in sorted order. So the loop invariant preserves for the next iteration.

**Termination:** when i = a.length, the outer loop terminates,by the loop invariant, A[1..i-1] includes i-1 smaller elements and in sorted so a.length elements is sorted, in others world, the origin array is sorted.

**(c)** Three parts:

l Give the worst-case running time of BubbleSort, in big-O notation, with justification.

l Give the best-case running time of BubbleSort, in big-O notation, with justification.

l How do these compare to the running time of insertion sort?

Worst-case and best-case running time are same because enter inner loop there is only required constant time for compare and swape, as we can a constant time can be ignore when we have size n. In addition we need run outer loop n-1 times that O(n) and also O(n)times inner loop so the BubbleSort need O(nxn)= O(n^2);

BUBBLESORT(A)

1 **for** i = 1 **to** A.length - 1 n

2 **for** j = A.length **downto** i + 1 n

3 **if** A[j] < A[j - 1] c

4 exchange A[j] with A[j - 1] c

From the learning, I know that it can perform ar O(n) in the best case of insertion sort that is better than the best case of BubbleSort, and BubbleSort and insertion sort both perform same o(n^2)for average and the worst case.