(CSCI 251) Activity One and Exam One Review

Student Name \_\_\_\_\_Dustin Riley\_\_\_\_\_\_\_\_

***Instruction:*** *The students are encouraged to type the answer use WORD and submit the word file through blackboard. To learn how to type math notation in word, please watch video* [*https://www.youtube.com/watch?v=SRGaW3maK38*](https://www.youtube.com/watch?v=SRGaW3maK38)*. You may search other videos to learn how to do this faster.*

*However, if the student feels it takes too much time to type the answer, then the student can use handwriting to write down the answers on paper. The students then scan the paper into pdf or image file then upload the file to blackboard. In this case, any unclear handwriting may result 0 points to the problem.*

*Exam One consists of two parts. Part One is online multiple choices/fill in blanks. The students should review the embedded homework problems to prepare Part One. This activity is Practice Problem Set for Exam One Part Two review. The questions in real Exam One may not be the same questions listed here. However, the testing concepts will be the same or similar.*

Problem One (8 points) In your textbook, the binary search is presented in iteration format. Please rewrite the binary algorithm in recursive format. You should write the algorithm in the follow format. The italic parts are the parts you need to modify.

Input: *array of numbers, low index, high index, key(number searched for)*

Output: *if the key is not found return -1, if key is found return index location*

Procedure

BinarySearchRecursive(*numbers, low, high, key*){

If(low > high) //each time binarysearch is recalled it checks if key exists

return -1

mid = (low + high)/2 //finds midpoint in each binary list created

//from switching low/high around

if(numbers[mid] < key){ //if midpoint of current binary list is <key

recall search and move low to mid + 1

return BinarySearchRecursive(numbers, mid + 1, high, key)

}

else if(numbers[mid] < key){ //if midpoint of current binary list is >key

recall search and move high to mid - 1

return BinarySearchRecursive(numbers, low, mid - 1, key)

}

return mid //if key is found

} *Algorithm details*

End Procedure

Note: You are not required to write code. In fact, you should not write code here. You should just write the algorithm. Also, you may need another algorithm so called the driver for BinarySearchDriver which simply call BinarySearchRecursive appropriately to search the given array for given key.

Problem Two (8 points, 4 points each) Consider the following array:

12, 13, 15, 15, 16, 16, 16, 19, 23, 24, 25, 26, 29, 30, 31 ,32

1. If use linear search algorithm from your textbook with search key 16, what is the return value? 4
2. If use binary search algorithm from your textbook with search key 16, what is the return value? 5

Problem Three (8 points, 4 points each)

You shall have studied this one in CSCI 241. If you have forgot, please review corresponding materials in Chapter Six Additional Materials

1. Order the following big O notation, from the fastest running time to slowest running time.
2. Determine big O notation for function

O(n²)

Problem Four (10 points, 5 points each) Write the following algorithms. The algorithms are all named isSorted. We assume an array object know its size,

1. The first algorithm returns true if the array is sorted either in ascend order or in descend order; returns false otherwise. The algorithm takes an int array as argument.

isSorted(numbers, arraySize){

lowcount = 0

highcount = 0

for(int i = 0; i + 1 < arraySize; i++){

if(numbers[i] <= numbers[i+1]){

lowcount++

}

else if(numbers[i] >= numbers[i+1]){

highcount++

}

}

if(lowcount == arraySize){

return true

}

else if(highcount == arraySize){

return true

}

else {

return false

}

}

1. The second algorithm takes an int array and a Boolean value as arguments. If the Boolean value is true, then the algorithm returns true if the array is sorted in ascend order; If the Boolean value is false, then the algorithm returns true if the array is sorted in descend order; The algorithm returns false otherwise

isSorted(numbers,bool){

lowcount = 0

highcount = 0

for(int i = 0; i + 1 < arraySize; i++){

if(numbers[i] <= numbers[i+1]){

lowcount++

}

else if(numbers[i] >= numbers[i+1]){

highcount++

}

}

If(bool == true){

if(lowcount == arraySize){

return true

}

else {

return false

}

}

else if(bool == false){

if(highcount == arraySize){

return true

}

else {

return false

}

}

}

Problem Five (16 points, 8 points each) Do NOT write code, just algorithm.

1. Modify the insertion sort algorithm with an extra argument, named isAscend, of Boolean type. If isAscend is true, then the array will be sorted in ascend order; otherwise, the array will be sorted in descend order.

InsertionSort(numbers, numbersSize, isAscend) {

i = 0

j = 0

temp = 0

for (i = 1; i < numbersSize; ++i) {

j = i

if(isAscend == true){

while (j > 0 && numbers[j] < numbers[j - 1]) {

temp = numbers[j]

numbers[j] = numbers[j - 1]

numbers[j - 1] = temp

--j

}

}

else if(isAscend == false){

while (j > 0 && numbers[j] < numbers[j - 1]) {

temp = numbers[j - 1]

numbers[j - 1] = numbers[j]

numbers[j] = temp

--j

}

}

1. Do the same problem to select sort algorithm.

SelectionSort(numbers, numbersSize, isAscend) {

i = 0

j = 0

indexSmallest = 0

indexLargest = 0

temp = 0

for (i = 0; i < numbersSize - 1; ++i) {

if(isAscend == true){

indexSmallest = i

for (j = i + 1; j < numbersSize; ++j) {

if ( numbers[j] < numbers[indexSmallest] ) {

indexSmallest = j

}

}

temp = numbers[i]

numbers[i] = numbers[indexSmallest]

numbers[indexSmallest] = temp

}

else if(isAscend != true){

indexLargest = i

for (j = i + 1; j < numbersSize; ++j) {

if ( numbers[j] > numbers[indexLargest] ) {

indexLargest = j

}

}

temp = numbers[i]

numbers[i] = numbers[indexLargest]

numbers[indexLargest] = temp

}

}

Problem Six (Total 18 points, 3 points each for first 4 sub-problems, 6 points for number five) Answer the questions based on book’s Quick Sort algorithm.

1. What is the running time of partition? N
2. What is the running time of quick sort? O(NlogN) to O(N²)
3. How many times the partition algorithm will be executed in a quick sort? Assume the array size is n. Justify your answer.

log₂N if split in half each time

N^2 if smallest is

1. How many comparisons are needed to sort a list of 2048 elements?

log₂2048 \* 2048 = 22528 to 4194304 = 2048²

1. Given array 8, 5, 3, 7, 1, 6, 4, 2. List array to show the array changes in memory. You may list out array after each partition call is terminated.

Low = 0 high = 7 pivot = 7

[7,5,3,1,6,4,2] [8]

Low = 0 high = 6 pivot = 1

[1] [5,3,7,6,4,2] [8]

Low = 1 high = 6 pivot = 7

[1][5,3,2,6,4][7][8]

Low = 1 high = 5 pivot = 2

[1][2][3,5,6,4][7][8]

Low = 2 high = 5 pivot = 5

[1][2][3,5,4][6][7][8]

Low = 2 high = 4 pivot = 5

[1][2][3,4][5][6][7][8]

Low = 2 high = 3 pivot = 3

[1][2][3][4][5][6][7][8]

[1,2,3,4,5,6,7,8]

Problem Seven (Total 18 points, 3 points for first 5 sub-problems; 6 points for number five) Answer the questions based on book’s merge sort algorithm

1. What is the running time of merger? N
2. What is the running time of merger sort? O(NlogN)
3. How many times the merger algorithm will be executed in a merger sort? N
4. How many comparisons are needed to sort a list of 2048 elements?

2048 \* log₂2048 = 22528

1. Given array 8, 5, 3, 7, 1, 6, 4, 2. List array to show the array changes in memory. You may list out array after each merge call is terminated.

I = 0 k = 7 j = 3

[8,5,3,7][1,6,4,2]

I = 0 k = 3 j = 1 | I = 4 k = 7 j = 5

[8,5][3,7] | [1,6][4,2]

I = 0 k = 1 j = 0 | I = 2 k = 3 j = 2 | I = 4 k = 5 j = 4 | I = 6 k = 7 j = 6

[8][5][3][7][1][6][4][2]

[5,8][3,7] [1,6][2,4]

[3,5,7,8] [1,2,4,6]

[1,2,3,4,5,6,7,8]

Problem Eight (7 points) According to your book, “…To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The pivot can be any value within the array being sorted, commonly the value of the middle array element.” Now rewrite the partition algorithm by using the last element of the subarray as the pivot.

Partition(numbers, lowIndex, highIndex) {

pivot = numbers[highindex]

done = false

while (!done) {

while (numbers[lowIndex] < pivot) {

lowIndex += 1

}

while (pivot < numbers[highIndex]) {

highIndex -= 1

}

// If zero or one elements remain, then all numbers are

// partitioned. Return highIndex.

if (lowIndex >= highIndex) {

done = true

}

else {

// Swap numbers[lowIndex] and numbers[highIndex]

temp = numbers[lowIndex]

numbers[lowIndex] = numbers[highIndex]

numbers[highIndex] = temp

// Update lowIndex and highIndex

lowIndex += 1

highIndex -= 1

}

}

return highIndex

}

Problem Nine (7 points) What is the difference between stable and unstable sorting algorithms? For the sorting algorithms you studied in this chapter, which are stable? Which are unstable?

Stable sorting algoritms keep equal values in order i.e. [1,8,9,8] -> [1,8,8,9]

Unstable ones do not [1,8,9,8] -> [1,8,8,9]

|  |  |
| --- | --- |
| Stable | Unstable |
| Merge sort | Quick sort |
| Insertion sort | Selection sort |
| Radix sort | Shell sort |