CS 3310: Data and File Structures

Instructor: Dr. Ajay K. Gupta, Western Michigan University

Lab TA: Yu Guo

Matt Peter

**Software Life Cycle Report – Assignment 1**

**Phase 1: Specification**

Objectives:

1. Write a program that uses multiple techniques to search in sorted arrays
2. Find the solution to the problem using both iterative and recursive methods
3. Compare the results you get in terms of theoretical and empirical complexities

Write a program that:

1. Generates n random floating-point numbers from 1 to m
2. Stores the randomly generated values in an array and sorts the array
3. Searches for a random value in the array using binary, ternary, and quad search
4. Calculates the number of values in the array that lie between two random values

**Phase 2: Design**

2.1 Modules and their Basic Structure

1. Module 1: Class Hw1Main contains:
   1. public static void main (String[] args) – Gets information from the user, generates array, and calls search methods
2. Module 2: Class IterativeSearch contains:
   1. public int binarySearch (float[] nums, float searchVal) – Searches the array for a value using a loop and a single search point
   2. public int ternarySearch (float[] nums, float searchVal) – Searches the array for a value using a loop and two search points
   3. public int quadSearch (float[] nums, float searchVal) – Searches the array for a value using a loop and three search points
   4. public int rangeQuery (float[] nums, float searchVal1, float searchVal2) – Searches the array for the number of values that lie between two values using a single search point
3. Module 3: Class RecursiveSearch contains:
   1. public int binarySearch (float[] nums, float searchVal, int first, int last) – Searches the array for a value using recursion and a single search point
   2. public int ternarySearch (float[] nums, float searchVal, int first, int last) – Searches the array for a value using recursion and two search points
   3. public int quadSearch (float[] nums, float searchVal, int first, int last) – Searches the array for a value using recursion and three search points
   4. public int rangeQuery (float[] nums, float searchVal1, float searchVal2, int first, int last) – Calls rangeQueryFindBound to find the positions of searchVal1 and searchVal2 and return the difference
   5. private int rangeQueryFindBound (float[] nums, float searchVal, int first, int last, boolean findingLower) – Searches the array for a value using recursion and a single search point and returns its position (or closest position)

2.2 Pseudocode for the Modules

2.2.1 Pseudocode for Hw1Main

1a. Hw1Main Pseudocode Refinement #1:

// Declare and Instantiate Scanner and Random objects

// Declare and Instantiate IterativeSearch and RecursiveSearch objects

// Declare and Instantiate variables

// Gather input from the user

// Generate and sort array

// Generate search values

// Call search methods using IterativeSearch and RecursiveSearch objects

// Print out resulting data

2.2.2 Pseudocode for IterativeSearch

2a. IterativeSearch Pseudocode Refinement #1:

// Method binarySearch – Searches the array for a value using a loop and a single search point

// Method ternarySearch – Searches the array for a value using a loop and two search points

// Method quadSearch – Searches the array for a value using a loop and three search points

// Method rangeQuery – Searches the array for the number of values that lie between two values using a single search point

2b. IterativeSearch Pseudocode Refinement #2:

// Method binarySearch – Searches the array for a value using a loop and a single search point

// Declare and Instantiate variables to hold positions in array

// Enter loop that will continue until there are no values left to check

// If current position is equal to value, return it

// If value is less, set last to current value

// If value is more, set first to current value

// If value isn’t found, return length of array

// Method ternarySearch – Searches the array for a value using a loop and two search points

// Declare and Instantiate variables to hold positions in array

// Enter loop that will continue until there are no values left to check

// If one of two current positions is equal to value, return that position

// If not, set bounds around where value is

// If value isn’t found, return length of array

// Method quadSearch – Searches the array for a value using a loop and three search points

// Declare and Instantiate variables to hold positions in array

// Enter loop that will continue until there are no values left to check

// If one of three current positions is equal to value, return that position

// If not, set bounds around where value is

// If value isn’t found, return length of array

// Method rangeQuery – Searches the array for the number of values that lie between two values using a single search point

// Declare and Instantiate variables to hold positions in array and to hold bounds of value range

// Enter loop that will continue until there are no values left to check

// If current position is equal to value, set as lower bound

// If value is less, set last to current value

// If value is greater, set first to current value

// If value isn’t found, set first as lower bound

// Reset first and last values

// Enter loop that will continue until there are no values left to check

// If current position is equal to value, set as upper bound

// If value is less, set last to current value

// If value is greater, set first to current value

// If value isn’t found, set last as upper bound

// Return the difference between upper and lower bound plus 1

2.2.3 Pseudocode for RecursiveSearch

3a. RecursiveSearch Pseudocode Refinement #1:

// Method binarySearch – Searches the array for a value using recursion and a single search point

// Method ternarySearch – Searches the array for a value using recursion and two search points

// Method quadSearch – Searches the array for a value using recursion and three search points

// Method rangeQuery – Calls rangeQueryFindBound to find the positions searchVal1 and searchVal2 and return the difference

// Method rangeQueryFindBound – Searches the array for a value using recursion and a single search point and returns its position (or closest position)

3b. RecursiveSearch Pseudocode Refinement #2:

// Method binarySearch – Searches the array for a value using recursion and a single search point

// Calculate middle position of array

// If value is at position, return position

// If value is less, recursively call method with middle as last

// If value is greater, recursively call method with middle as first

// If value isn’t found, return length of array

// Method ternarySearch – Searches the array for a value using recursion and two search points

// Calculate 1/3 and 2/3 positions of array

// If value is at one of those positions, return position

// Else, find what range value is in and recursively call method with respective bounds

// If value isn’t found, return length of array

// Method quadSearch – Searches the array for a value using recursion and three search points

// Calculate 1/4, 1/2, and 3/4 positions of array

// If value is at one of those positions, return position

// Else, find what range value is in and recursively call method with respective bounds

// If value isn’t found, return length of array

// Method rangeQuery – Calls rangeQueryFindBound to find the positions searchVal1 and searchVal2 and return the difference

// Call rangeQueryFindBound with smaller searchVal and store as lower bound

// Call rangeQueryFindBound with larger searchVal and store as upper bound

// Return difference between upper and lower bounds plus 1

// Method rangeQueryFindBound – Searches the array for a value using recursion and a single search point and returns its position (or closest position)

// Calculate middle position of array

// If value is at position, return position

// If value is less, recursively call method with last as middle

// If value is greater, recursively call method with first as middle

// If value isn’t found, check to see if we’re looking for upper or lower bound

// Upper bound = first

// Lower bound = last

**Phase 3: Risk Analysis**

There are no known risks associate with this application.

**Phase 4: Verification**

All the steps of the algorithm were checked to ensure correct results in all circumstances. A variety of tests with a wide range of values were also done to check for correct output.

**Phase 5: Coding**

5a. Code Refinement #1:

File Hw1Main.java

package edu.wmich.cs3310.MPeter.hw1;

import java.util.\*;

public class Hw1Main {

public static void main(String[] args) {

// Declare and Instantiate Scanner and Random objects

// Declare and Instantiate IterativeSearch and // RecursiveSearch objects

// Declare and Instantiate variables

// Gather input from the user

// Generate and sort array

// Generate search values

// Call search methods using IterativeSearch and // Recursive Search objects

// Print out resulting data

}

}

File IterativeSearch.java

package edu.wmich.cs3310.MPeter.hw1;

public class IterativeSearch {

public int binarySearch(float[] nums, float searchVal) {

// Declare and Instantiate variables to hold positions // in array

// Enter loop that will continue until there are no values // left to check

// If current position is equal to value, return it

// If value is less, set last to current value

// If value is more, set first to current value

// If value isn’t found, return length of array

}

public int ternarySearch(float[] nums, float searchVal) {

// Declare and Instantiate variables to hold positions // in array

// Enter loop that will continue until there are no values // left to check

// If one of two current positions is equal to value, // return that position

// If not, set bounds around where value is

// If value isn’t found, return length of array

}

public int quadSearch(float[] nums, float searchVal) {

// Declare and Instantiate variables to hold positions // in array

// Enter loop that will continue until there are no values // left to check

// If one of three current positions is equal to // value, return that position

// If not, set bounds around where value is

// If value isn’t found, return length of array

}

public int rangeQuery(float[] nums, float searchVal1, float searchVal2) {

// Declare and Instantiate variables to hold positions in // array and to hold bounds of value range

// Enter loop that will continue until there are no values // left to check

// If current position is equal to value, set as // lower bound

// If value is less, set last to current value

// If value is greater, set first to current value

// If value isn’t found, set first as lower bound

// Reset first and last values

// Enter loop that will continue until there are no values // left to check

// If current position is equal to value, set as // upper bound

// If value is less, set last to current value

// If value is greater, set first to current value

// If value isn’t found, set last as upper bound

// Return the difference between upper and lower bounds // plus 1

}

}

File Recursive Search.java

package edu.wmich.cs3310.MPeter.hw1;

public class RecursiveSearch {

public int binarySearch(float[] nums, float searchVal, int first, int last) {

// Calculate middle position of array

// If value is at position, return position

// If value is less, recursively call method with middle as // last

// If value is greater, recursively call method with middle // as first

// If value isn’t found, return length of array

}

public int ternarySearch(float[] nums, float searchVal, int first, int last) {

// Calculate 1/3 and 2/3 positions of array

// If value is at one of those positions, return position

// Else, find what range value is in and recursively call // method with respective bounds

// If value isn’t found, return length of array

}

public int quadSearch(float[] nums, float searchVal, int first, int last) {

// Calculate 1/4, 1/2, and 3/4 positions of array

// If value is at one of those positions, return position

// Else, find what range value is in and recursively call // method with respective bounds

// If value isn’t found, return length of array

}

public int rangeQuery(float[] nums, float searchVal1, float searchVal2, int first, int last) {

// Call rangeQueryFindBound with smaller searchVal and // store as lower bound

// Call rangeQueryFindBound with larger searchVal and // store as upper bound

// Return difference between upper and lower bounds plus 1

}

private int rangeQueryFindBound(float[] nums, float searchVal, int first, int last, boolean findingLower) {

// Calculate middle position of array

// If value is at position, return position

// If value is less, recursively call method with last as // middle

// If value is greater, recursively call method with first // as middle

// If value isn’t found, check to see if we’re looking for // upper or lower bound

// Upper bound = first

// Lower bound = last

}

}

5b. Code Refinement #2:

File Hw1Main.java

package edu.wmich.cs3310.MPeter.hw1;

import java.util.\*;

public class Hw1Main {

public static void main(String[] args) {

// Declare and Instantiate several objects and variables

Scanner kbrd = new Scanner(System.in);

Random rand = new Random();

IterativeSearch is = new IterativeSearch();

RecursiveSearch rs = new RecursiveSearch();

float[] numbers;

int n = 0;

int m = 0;

// Gather input from user

System.out.println(“Please enter the amount of numbers you’d like generated: “);

n = kbrd.nextInt();

System.out.println(“Please enter the upper bound you’d like for the numbers generated: “);

m = kbrd.nextInt();

// Generate random values and adjust to fit range

numbers = new float[n];

for (int i = 0; i < n; i++) {

numbers[i] = (rand.nextFloat() \* (m – 1.0f)) + 1.0f;

}

// Sort array

Arrays.sort(numbers);

// Generate random values to search for in array

float searchValue1 = (rand.nextFloat() \* (m – 1.0f)) + 1.0f;

float searchValue2 = (rand.nextFloat() \* (m – 1.0f)) + 1.0f;

float searchValue3 = (rand.nextFloat() \* (m – 1.0f)) + 1.0f;

// Adjust range values if necessary

if (searchValue2 > searchValue3) {

float temp = searchValue2;

searchValue2 = searchValue3;

searchValue3 = temp;

}

// Print array information

System.out.println(“\nIterative Binary Search”);

System.out.println(“------------------------“);

System.out.println(is.binarySearch(numbers, searchValue1));

System.out.println(“\nRecursive Binary Search”);

System.out.println(“------------------------“);

System.out.println(rs.binarySearch(numbers, searchValue1, 0, n - 1)):

System.out.println(“\nIterative Ternary Search”);

System.out.println(“-------------------------“);

System.out.println(“is.ternarySearch(numbers, searchValue1));

System.out.println(“\nRecursive Ternary Search”);

System.out.println(“-------------------------“);

System.out.println(rs.ternarySearch(numbers, searchValue1, 0, n – 1));

System.out.println(“\nIterative Quad Search”);

System.out.println(“----------------------“);

System.out.println(is.quadSearch(numbers, searchValue1));

System.out.println(“\nRecursive Quad Search”);

System.out.println(“----------------------“);

System.out.println(rs.quadSearch(numbers, searchValue1, 0, n – 1));

System.out.println(“\nIterative Range Query”);

System.out.println(“----------------------“);

System.out.println(is.rangeQuery(numbers, searchValue2, searchValue3));

System.out.println(“\nRecursive Range Query”);

System.out.println(“----------------------“);

System.out.println(rs.rangeQuery(numbers, searchValue2, searchValue3, 0, n – 1));

// Close Scanner object

kbrd.close();

}

}

File IterativeSearch.java

package edu.wmich.cs3310.MPeter.hw1;

public class IterativeSearch {

public int binarySearch(float[] nums, float searchVal) {

// Declare and Instantiate variables to hold the first, last, and middle positions of the array

int first = 0;

int last = nums.length – 1;

int middle = 0;

// Calculate the position of the middle of the array

// Check if the value is at the calculated location

// Otherwise, adjust first or last accordingly

while (first <= last) {

middle = (int)((first + last) / 2.0f);

if (searchVal == nums[middle]) {

return middle;

} else if (searchVal < nums[middle]) {

last = middle – 1;

} else {

First = middle + 1;

}

}

return nums.length;

}

public int ternarySearch(float[] nums, float searchVal) {

// Declare and Instantiate variables to hold the first, last, and two middle positions of the array

int first = 0;

int last = nums.length – 1;

int firstMid = 0;

int secondMid = 0;

// Calculate the 1/3 and 2/3 positions of the array

// Check if the value is at one of the calculated locations

// Otherwise, adjust first and/or last accordingly

while (first <= last) {

firstMid = (int)((first \* 2.0f + last) / 3.0f);

secondMid = (int)((first + last \* 2.0f) / 3.0f);

if (searchVal == nums[firstMid]) {

return firstMid;

} else if (searchVal == nums[secondMid]) {

return secondMid;

} else if (searchVal < nums[firstMid]) {

last = firstMid – 1;

} else if (searchVal > nums[secondMid]) {

first = secondMid + 1;

} else {

first = firstMid + 1;

last = secondMid – 1;

}

}

return nums.length;

}

public int quadSearch(float[] nums, float searchVal) {

// Declare and Instantiate variables to hold the first, last, and three middle positions of the array

int first = 0;

int last = nums.length – 1;

int firstMid = 0;

int secondMid = 0;

int thirdMid = 0;

// Calculate the 1/4, 1/2, and 3/4 positions of the array

// Check if the value is at one of the calculated locations

// Otherwise, adjust first and/or last accordingly

while (first <= last) {

firstMid = (int)((first \* 3.0f + last) / 4.0f);

secondMid = (int)((first + last) / 2.0f);

thirdMid = (int)((first + last \* 3.0f) / 4.0f);

if (searchVal == nums[firstMid]) {

return firstMid;

} else if (searchVal == nums[secondMid]) {

return secondMid;

} else if (searchVal == nums[thirdMid]) {

return thirdMid;

} else if (searchVal < nums[firstMid]) {

last = firstMid – 1;

} else if (searchVal > nums[thirdMid]) {

first = thirdMid + 1;

} else if (searchVal > nums[firstMid} && searchVal < nums[secondMid]) {

first = firstMid + 1;

last = secondMid – 1;

} else {

first = secondMid + 1;

last = thirdMid – 1;

}

}

return nums.length;

}

public int rangeQuery(float[] nums, float searchVal1, float searchVal2) {

// Declare and Instantiate variables to hold the first, last and middle positions of the array as well as the upper and lower bounds

int lowerBound = 0;

int upperBound = 0;

int first = 0;

int last = nums.length – 1;

int middle = 0;

// Calculate the position of the middle of the array

// Check if the value is at the calculated location

// Otherwise, adjust first or last accordingly

while (first <= last) {

middle = (int)((first + last) / 2.0f);

if (searchVal1 == nums[middle]) {

lowerBound = middle;

break;

} else if (searchVal < nums[middle]) {

last = middle – 1;

} else {

first = middle + 1;

}

}

// If value does not exist, set the lower bound to first

if (first > last) {

lowerBound = first;

}

// Reset the values of first and last

first = 0;

last = nums.length – 1;

// Calculate the position of the middle of the array

// Check if the value is at the calculated location

// Otherwise, adjust first or last accordingly

while (first <= last) {

middle = (int)((first + last) / 2.0f);

if (searchVal2 == nums[middle]) {

upperBound = middle;

break;

} else if (searchVal2 < nums[middle]) {

last = middle – 1;

} else {

first = middle + 1;

}

}

// If value does not exist, set upper bound to last

if (first > last) {

upperBound = last;

}

// Calculate and return the number of values within the bounds (inclusive)

return (upperBound – lowerBound + 1);

}

}

File Recursive Search.java

package edu.wmich.cs3310.MPeter.hw1;

public class RecursiveSearch {

public int binarySearch(float[] nums, float searchVal, int first, int last) {

// Calculate the position of the middle of the array

int middle = (int)((first + last) / 2.0f);

// Check if the value is at the calculated location

// Otherwise, recursively call this method with new boundaries

if (first <= last) {

if (searchVal == nums[middle]) {

return middle;

} else if (searchVal < nums[middle]) {

return binarySearch(nums, searchVal, first, middle – 1);

} else {

return binarySearch(nums, searchVal, middle + 1, last);

}

}

Return nums.length;

}

public int ternarySearch(float[] nums, float searchVal, int first, int last) {

// Calculate the positions of the 1/3 and 2/3 positions in the array

int firstMid = (int)((first \* 2.0f + last) / 3.0f);

int secondMid = (int)((first + last \* 2.0f) / 3.0f);

// Check if the value is at one of the calculated locations

// Otherwise, recursively call this method with new boundaries

if (first <= last) {

if (searchVal == nums[firstMid]) {

return firstMid;

} else if (searchVal == nums[secondMid]) {

return secondMid;

} else if (searchVal < nums[firstMid]) {

return ternarySearch(nums, searchVal, first, firstMid – 1);

} else if (searchVal > nums[secondMid]) {

return ternarySearch(nums, searchVal, secondMid + 1, last);

} else {

return ternarySearch(nums, searchVal, firstMid + 1, secondMid – 1);

}

}

return nums.length;

}

public int quadSearch(float[] nums, float searchVal, int first, int last) {

// Calculate the positions of the 1/4, 1/2, and 3/4 positions in the array

int firstMid = (int)((first \* 3.0f + last) / 4.0f);

int secondMid = (int)((first + last) / 2.0f);

int thirdMid = (int)((first + last \* 3.0f) / 4.0f);

// Check if the value is at one of the calculated locations

// Otherwise, recursively call this method with new boundaries

if (first <= last) {

if (searchVal == nums[firstMid]) {

return firstMid;

} else if (searchVal == nums[secondMid]) {

return secondMid;

} else if (searchVal == nums[thirdMid]) {

return thirdMid;

} else if (searchVal < nums[firstMid]) {

return quadSearch(nums, searchVal, first, firstMid – 1);

} else if (searchVal > nums[thirdMid]) {

return quadSearch(nums, searchVal, thirdMid + 1, last);

} else if (searchVal > nums[firstMid] && searchVal < nums[secondMid) {

return quadSearch(nums, searchVal, firstMid + 1, secondMid – 1);

} else {

return quadSearch(nums, searchVal, secondMid + 1, thirdMid – 1);

}

}

return nums.length;

}

public int rangeQuery(float[] nums, float searchVal1, float searchVal2, int first, int last) {

int lowerBound = rangeQueryFindBound(nums, searchVal1, first, last, true);

int upperBound = rangeQueryFindBound(nums, searchVal2, first, last, false);

return (upperBound – lowerBound + 1);

}

private int rangeQueryFindBound(float[] nums, float searchVal, int first, int last, boolean findingLower) {

// Calculate the position of the middle of the array

int middle = (int)((first + last) / 2.0f);

// Check if the value is at the calculated location

// Otherwise, recursively call this method with new boundaries

if (first <= last) {

if (searchVal == nums[middle]) {

return middle;

} else if (searchVal < nums[middle]) {

return rangeQueryFindBound(nums, searchVal, first, middle – 1, findingLower);

} else {

return rangeQueryFindBound(nums, searchVal, middle + 1, last, findingLower);

// If value does not exist, return respective boundary position

} else {

if (findingLower) {

return first;

}

}

return last;

}

}

**Phase 6: Testing**

|  |  |  |
| --- | --- | --- |
| Input/Output Analysis | | |
|
| **Method** | **Input Size** | **Output Size** |
| Hw1Main | 0 | 0 |
| binarySearch (Iterative) | n + 1 | 1 |
| ternarySearch (Iterative) | n + 1 | 1 |
| quadSearch (Iterative) | n + 1 | 1 |
| rangeQuery (Iterative) | n + 2 | 1 |
| binarySearch (Recursive) | n + 3 | 1 |
| ternarySearch (Recursive) | n + 3 | 1 |
| quadSearch (Recursive) | n + 3 | 1 |
| rangeQuery (Recursive) | n + 4 | 1 |
| rangeQueryFindBound | n + 4 | 1 |

The table above shows an analysis of the input and output for each method.

The table below shows an analysis of the space used within each method as well as the big O notation for each of the resulting equations.

|  |  |  |
| --- | --- | --- |
| Space Complexity Analysis | | |
|
| **Method** | **Space Usage** | **Space Complexity** |
| Hw1Main | n + 11 | O(n) |
| binarySearch (Iterative) | 3 | O(1) |
| ternarySearch (Iterative) | 4 | O(1) |
| quadSearch (Iterative) | 5 | O(1) |
| rangeQuery (Iterative) | 5 | O(1) |
| binarySearch (Recursive) | 1 | O(1) |
| ternarySearch (Recursive) | 2 | O(1) |
| quadSearch (Recursive) | 3 | O(1) |
| rangeQuery (Recursive) | 2 | O(1) |
| rangeQueryFindBound | 1 | O(1) |

|  |  |  |
| --- | --- | --- |
| Time Complexity Analysis | | |
|
| **Method** | **Instruction Count** | **Time Complexity** |
| Hw1Main | (164 + n)\*log(n) + 8n + 123 | O(n\*log(n)) |
| binarySearch (Iterative) | 10\*log(n) + 6 | O(log(n)) |
| ternarySearch (Iterative) | 20\*log(n) + 7 | O(log(n)) |
| quadSearch (Iterative) | 29\*log(n) + 8 | O(log(n)) |
| rangeQuery (Iterative) | 20\*log(n) + 18 | O(log(n)) |
| binarySearch (Recursive) | 10\*log(n) + 1 | O(log(n)) |
| ternarySearch (Recursive) | 18\*log(n) + 1 | O(log(n)) |
| quadSearch (Recursive) | 27\*log(n) + 1 | O(log(n)) |
| rangeQuery (Recursive) | 20\*log(n) + 11 | O(log(n)) |
| rangeQueryFindBound | 10\*log(n) + 3 | O(log(n)) |

All the search methods as well as the range query ended up having time complexities of O(log(n)) through the theoretical complexity analysis of the code. This ended up holding true through the empirical complexity analysis as well. The graph below shows the average number of nanoseconds (out of 20 tests) it took to do the searches and range queries using a variety of sizes of arrays. The result appears fairly linear when the x-axis is set to a logarithmic scale of base 10, showing that it can be closely related to a time complexity of O(log(n)).

**Phase 7: Refining the Program**

All required features are included in the program so no refinements are needed.

**Phase 8: Production**

A zip file containing source files, a Javadoc, and test data have been submitted.

**Phase 9: Maintenance**

Any maintenance can be performed once feedback has been obtained.