Erik Marrero

9/20/2020

Hemework 1

CS-253

Professor Neli

Problem 1:

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.Random;

import java.io.FileWriter;

import java.io.PrintWriter;

import java.nio.file.Path;

// Java program for implementation of Selection Sort

class SelectionSort {

void sort(int arr[]) {

int n = arr.length;

// One by one move boundary of unsorted subarray

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

// Find the minimum element in unsorted array

int min\_idx = i;

for (int j = i + 1; j < n; j++)

if (arr[j] < arr[min\_idx])

min\_idx = j;

// Swap the found minimum element with the first

// element

int temp = arr[min\_idx];

arr[min\_idx] = arr[i];

arr[i] = temp;

}

}

// Prints the array

void printArray(int arr[]) {

int n = arr.length;

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

System.out.print(arr[i] + " ");

System.out.println();

}

}

// Java program for implementation of Bubble Sort

class BubbleSort {

void bubbleSort(int arr[]) {

int n = arr.length;

for (int i = 0; i < n - 1; i++)

for (int j = 0; j < n - i - 1; j++)

if (arr[j] > arr[j + 1]) {

// swap arr[j+1] and arr[i]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

/\* Prints the array \*/

void printArray(int arr[]) {

int n = arr.length;

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

System.out.print(arr[i] + " ");

System.out.println();

}

}

// Java program for implementation of Insertion Sort

class InsertionSort {

/\*Function to sort array using insertion sort\*/

void sort(int arr[]) {

int n = arr.length;

for (int i = 1; i < n; ++i) {

int key = arr[i];

int j = i - 1;

/\* Move elements of arr[0..i-1], that are

greater than key, to one position ahead

of their current position \*/

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

/\* A utility function to print array of size n\*/

static void printArray(int arr[]) {

int n = arr.length;

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

System.out.print(arr[i] + " ");

System.out.println();

}

}

public class Problem1

{

public static void main(String[] args) {

// create instance of Random class

Random rand = new Random();

try {

PrintWriter fileout = new PrintWriter(new FileWriter("C:/Users/erika/Desktop/Homework1\_neli/random1.txt"));

for (int i = 1; i < 2001; i++) {

int ran = rand.nextInt(2001);

fileout.println(ran);

}

fileout.close();

}

catch (Exception e) {

System.out.println(e);

}

int[] z=new int[2000];

for(int i=0;i<2000;i++) {

try {

String line = Files.readAllLines(Paths.get("C://Users//erika//Desktop//Homework1\_neli//random1.txt")).get(i);

z[i]=Integer.parseInt(line);

} catch (IOException e) {

System.out.println(e);

}

}

SelectionSort ob = new SelectionSort();

ob.sort(z);

System.out.println("Sorted array through selection sort");

ob.printArray(z);

BubbleSort ob1 = new BubbleSort();

ob1.bubbleSort(z);

System.out.println("Sorted array through bubble sort");

ob1.printArray(z);

InsertionSort ob2 = new InsertionSort();

ob2.sort(z);

System.out.println("Sorted array through insertion sort");

ob2.printArray(z);

}

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program run  N = 10 | Algorithm  Type | Best Case | Worst case | Average case |
| BubbleSort | O(n)  sorted array  1,2,3,4....  9 comparisons  0exchanges | O(n2)  if sorting is decreasing then array is increasing.  10,9,8,7,......,3,2,1  45 comparisons  45 exchanges | O(n2)  anything but sorted array.  3,4,7,3,2......  42 comparisons  21 exchanges |
| InsertionSort | n-1 --> O(n)  sorted array  1,2,3,4....  36 comparisons  9 exchanges | (n\*(n-1))/2 --> O(n2)  strictly decreasing values.  10,9,8.....,2,1  46 comparisons  54 exchanges | n2/4 --> O(n2)  anything but sorted array  4,7,3,8,2,6......  N comparisons  N exchanges |
| seletionSort | O(n2)  any format it will run for  (n\*(n-1))/2 times 45 comparisons  9 exchanges | O(n2)  any format it will run for  (n\*(n-1))/2 times 45 comparisons  9 exchanges | O(n2)  any format it will run for  (n\*(n-1))/2 times 45 comparisons  9 exchanges |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program run  N = 2000 | Algorithm  Type | Best Case | Worst case | Average case |
| BubbleSort | O(n)  sorted array  1,2,3,4....  1999 comparisons  0 exchanges | O(n2)  if sorting is decreasing then array is increasing.  2000,1999,1998,......,3,2,1  1999000 comparisons  1999000 exchanges | O(n2)  anything but sorted array.  1363,45,75,333,237......  1432998 comparisons  945768 exchanges |
| InsertionSort | n-1 --> O(n)  sorted array  1,2,3,4....  1997001 comparisons  2000999 exchanges | (n\*(n-1))/2 --> O(n2)  strictly decreasing values.  2000,1999,1998.....,2,1  1999 comparisons  998002 exchanges | n2/4 --> O(n2)  anything but sorted array  1363,45,75,333,237.....  1999 comparisons  N exchanges |
| seletionSort | O(n2)  any format it will run for  (n\*(n-1))/2 times  1999000 comparisons  1999 exchanges | O(n2)  any format it will run for  (n\*(n-1))/2 times  1999000 comparisons  1999exchanges | O(n2)  any format it will run for  (n\*(n-1))/2 times 1999000 comparisons  1999 exchanges |

NOW COMPARING THEIR TIME COMPLEXITIES:

BEST CASE WROST CASE AVERAGE CASE

1. SELECTION SORT O(n^2)   O(n^2) O(n^2)

2. BUBBLE SORT   O(n) O(n^2) O(n^2)

3.INSERTION SORT   O(n) O(n^2) O(n^2)

1. SELECTION SORT :

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the left most element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

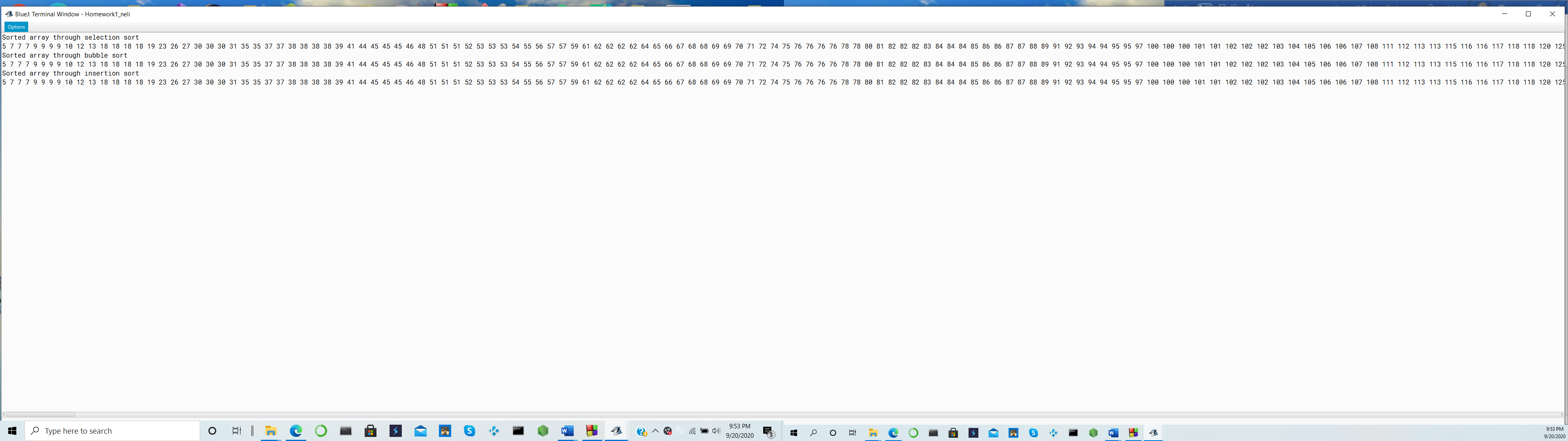
This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where **n** is the number of items.

2. BUBBLE SORT

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n** is the number of items. Bubble sorting time complexity is input-dependent. Comparisons and exchanges are done withing an inner loop, so you would expect the minimum time complexity to be O(N^2). However, because we are setting our inner index variable j to do one less operation each pass, total operations withing because 1+2…+N, which is the summation 1 to , or (N(N+1))/2. So N= 10 would give a maximum time complexity of 45 comparisons and 45 exchanges, and N=2000 would give us 1,999,000 comparisons and exchanges. This is very inefficient but it is the best choice.

3.INSERTION SORT

It is applied to a list of n elements, assumed to be all different and initially in random order. On average, half the elements in a list A1 ... Aj are less than elementAj+1, and half are greater. Therefore, the algorithm compares the (j + 1)th element to be inserted on the average with half the already sorted sub-list, so tj = j/2. Working out the resulting average-case running time yields a quadratic function of the input size, just like the worst-case running time.



Problem 2:

public class problem2 {

public static int operations = 0;

public static int comparisons = 0;

public static int[] numbers = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

public static int fact(int i) {

comparisons++;

if (i <= 1) {

return 1;

} else {

operations++;

return i \* fact(i - 1);

}

}

public static boolean binary(int[] list, int goal, int low, int high) {

operations++;

int middle = (low + high) / 2;

if (list[middle] == goal) {

return true;

} else {

if (low > high) {

return false;

} else {

comparisons++;

if (list[middle] < goal) {

return binary(list, goal, middle + 1, high);

} else {

return binary(list, goal, low, middle + 1);

}

}

}

}

public static int fib(int i) {

comparisons++;

if (i == 0) {

return 0;

} else if (i == 1) {

return 1;

} else {

operations++;

return fib(i - 1) + fib (i - 2);

}

}

public static void main(String[] args) {

System.out.println("fact of 5: " + fact(5));

System.out.println("Number of operations are: " + operations);

System.out.println("Number of comparisons are: " + comparisons);

operations = 0;

comparisons = 0;

System.out.println();

System.out.println("Recursive binary");

binary(numbers, 8, 0, numbers.length);

System.out.println("The Number of operations are : " + operations);

System.out.println("The Number of comparisons are : " + comparisons);

operations = 0;

comparisons = 0;

System.out.println();

System.out.println("5th fib number: " + fib(5));

System.out.println("The Number of operations are : " + operations);

System.out.println("The Number of comparisons are : " + comparisons);

operations = 0;

comparisons = 0;

}

}

**(A) Computing the nth Fibonacci number**

Problem size: n

At each step the original problem is divided into two part where size of one subproblem is one less than the previous and the size of other subproblem is two less than the previous. As you can see with Fibonacci’s time complexity depends on the input n that is entered. fib(0)and fib(1) always have O(1)time complexity since they return one value. If we set n to 6 and follow the chart you will see that you get. Fib(2) = fib(1)+fib(0), or O(1),Fib(3) = fib(2)+fib(1), or O(3),Fib(4) = fib(3)+fib(2), or O(5),Fib(5) = fib(4)+fib(3), or O(7)The values in order are 1,1,2,3,4,5,6 These are the first values of the Fibonacci sequence, and if you continue the to follow the examples and the chart bellow it will continue to follow the pattern. The time it takes is O9fib(N-1)).The growth of the Fibonacci sequence eventually converges on growing at an approximate slope of 1.618033, so you can also approximate the big O notation at O(1.618…n) which is in linear category(if you ignore the constant of proportionality). **The** value of Fib(**n**) is sum of all values returned by **the** leaves in **the** recursion tree which is equal to **the** count of leaves. Since each leaf will take **O**(1) to compute, T(**n**) is equal to Fib(**n**) x **O**(1) . Consequently, **the** tight bound for this **function** is **the Fibonacci sequence** itself (~ θ(1.6 **n** ) )

**(b) Computing the factorial of n**

Problem size: n

At each step the size of the subproblem becomes one less than the previous and finally the size of the subproblem becomes 1.As you can The complexity factor of recursive time is O (n ) calculated through recurrence equations space complexity is also O (n). In the non- recursive implementation, the space complexity is O (1). If you treat the input x as a number, then the runtime is indeed a polynomial in x. However, polynomial time is formally defined such that the runtime of the algorithm must be a polynomial with respect to the number of bits used to specify the input to the problem.

**(c) Recursive binary search**

Problem size: n

At each step the size of the subproblem(array) becomes half and finally the size of the subproblem becomes 1. Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

