Erik Marrero

10/1/2020

Homework2

Professor Neli

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;

import java.io.\*;

import java.util.\*;

// 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 Problem11

{

public static void main(String[] args) {

Random rand = new Random();

try {

PrintWriter fileout = new PrintWriter(new FileWriter("C://Users//erika//Desktop//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//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);

int n = z.length;

Radix ob3 = new Radix();

ob3.radixsort(z, n);

System.out.print("Sorted array through Radix Sort\n");

ob3.printArray(z);

QuickSort1 ob4 = new QuickSort1();

ob4.sort(z, 0, n-1);

System.out.println("Sorted array through QuickSort");

ob4.printArray(z);

}

}

class Radix{

static void countSort(int arr[], int n, int exp)

{

int output[] = new int[n]; // output array

int i;

int count[] = new int[2001];

Arrays.fill(count,0);

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

count[ (arr[i]/exp)%10 ]++;

for (i = 1; i < 10; i++)

count[i] += count[i - 1];

for (i = n - 1; i >= 0; i--)

{

output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];

count[ (arr[i]/exp)%10 ]--;

}

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

arr[i] = output[i];

}

static void radixsort(int arr[], int n)

{

int mx = arr[0];

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

if (arr[i] > mx)

mx = arr[i];

for (int exp = 1; mx/exp > 0; exp \*= 10)

countSort(arr, n, exp);

}

static void printArray(int arr[]) {

int n = arr.length;

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

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

System.out.println();

}

}

class QuickSort1{

static void random(int arr[],int low,int high)

{

Random rand= new Random();

int pivot = rand.nextInt(high-low)+low;

int temp1=arr[pivot];

arr[pivot]=arr[high];

arr[high]=temp1;

}

static int partition(int arr[], int low, int high)

{

// pivot is choosen randomly

random(arr,low,high);

int pivot = arr[high];

int i = (low-1);

for (int j = low; j < high; j++)

{

if (arr[j] < pivot)

{

i++;

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

int temp = arr[i+1];

arr[i+1] = arr[high];

arr[high] = temp;

return i+1;

}

static void sort(int arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

sort(arr, low, pi-1);

sort(arr, pi+1, high);

}

}

static void printArray(int arr[])

{

int n = arr.length;

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

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

System.out.println();

}

}

The assignment made it seem like the table was only necessary for the other question. I included it and put only the formulas. As well as explaining the formulas.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |
| Radix Sort | O(kn) | O(kn) | O(kn) |
| Quicksort | O(n2) | O(n log n) (simple partition) or O(n) | O(n log n) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |
|  | RadixSort | O(kn) | O(kn) | O(kn) |
|  | Quicksort | O(n2) | O(n log n) (simple partition) or O(n) | O(n log n) |

Part 1 Answer:

In the program the inner loop runs n times and y goes down from n to zero. In the outer loop it starts with X=1 and after each iteration, X is doubled X goes 1, 2, 4, 8 and so on. After every iteration, X is 2m where m is the iteration number. The loop will stop when 2M > N2. In the inner loop it executes all the way for each iteration of outer loop. Assuming statements within the ellipses takes unit time, the loop construct time complexity is the order of N\*M. M can be expressed as integer value of log2N2 or 2log2N. So big-O time complexity/efficiency is O(N log2N). we removed 2 from coefficient We do this because the O-efficiency just considers order, not scalar value Statements within Ellipses take 2 millisecond. or 0.002 second The inner loop executes N times so total time taken by inner loop to execute fully is 0.002 \* 1000 = 2 seconds. The outer loop executes till 2M > N2 = 2M > 1000000. This gives M as 19.93. This makes total time to 19.93 x 2 = 39.86 seconds = 39860 miliseconds.

Part 2 Answer:

Yes we can write radix sort for string elements by choosing a base for every letter a to z and selecting an empty character which will fill the places for non uniform strings just like zero. Then we can sort any number of string by using the basic radix sort method. It is a divide and conquer algorithm based sorting. It uses an element as a pivot and sorts the list and partitions it based on the pivot's sorted position. The pivot can be chosen by

Always picking first element as pivot. Always picking last element as pivot,Picking a random element as pivot. Picking median as pivot.

As shown in the program above Radix sort works by sorting each digit from the least significant number to the most significant number. So in the base 10 decimal system, As shown in the program radix sort sorts by the digits in the 1s place, then the 10s place, and so on. To do this, radix sort uses counting sort as a subroutine to sort the digits in each place value. This means that for a three-digit number in base 10, counting sort will be called to sort the 1's place, then it will be called to sort the 10's place, and finally, it will be called to sort the 100's place, resulting in a completely sorted list. Radix sort takes the list of  integers in base *b* and so each number has at most *d* digits where *d*=⌊(log*b*​(*k*)+1)⌋ and *k* is the largest number in the list. For example, three digits are needed to represent decimal 202 (in base 10). It is important that radix sort can work with any base since the running time of the algorithm, *O*(*d*(*n*+*b*), depends on the base it uses. The algorithm runs in linear time when *b* and *n* are of the same size magnitude, so knowing *n*, *b* can be manipulated​ to optimize the running time of the algorithm.

**The program Radix sort** is an integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits that share the same significant position and value. Radix sort uses counting sort class as a subroutine to sort an array of numbers. integers can be used to represent strings by hashing the strings to integers, radix sort works on data types other than just integers. radix sort is not comparison based, it is not bounded by Ω(*n*log*n*) for running time radix sort can perform in linear time. Radix sort incorporates the counting sort algorithm so that it can sort larger, multi-digit numbers without decreasing the efficiency by increasing the range of keys the algorithm must sort. Radix sort complexity is O(kn) for n which are integers of word size k. For all there cases time best , worst and average time complexity is O(kn).

Quicksort is a efficient sorting algorithm. Quicksort is a divide and conquer sorting algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub arrays according to whether they are less than or greater than the pivot. The sub arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved.

On average, the algorithm takes [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*) comparisons to sort *n* items. In the worst case, it makes O(*n*2) comparisons, though this behavior is rare.The best case occurs when the partition process always picks the middle element as pivot. T(n) = 2T(n/2) + \theta(n).The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. T(n) = T(n-1) + \theta(n). The average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy. We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. T(n) = T(n/9) + T(9n/10) + \theta(n)

If we have log2n bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.



