CSCI 421 Design and Analysis of Algorithms Spring 2019

Lecture 2 Activity 2

1. Apply the tree practical improvements on the merge sort algorithm. Compare the algorithms with various cutoff to insertion sort in terms of their running time. Say cutoff is set to 7, 12, 20, 25. You need to run the algorithms against large-size random arrays, say N=1000, 2000, 4000, 8000.

import java.util.Random;

import java.util.Arrays;

import java.util.\*;

@SuppressWarnings("unchecked")

public class Lec2Act2\_1 {

private static final int CUTOFF = 25;

private static void merge(int[] a, int[] aux, int lo, int mid, int hi) {

for (int k = lo; k <= hi; k++)

aux[k] = a[k];

int i = lo, j = mid + 1;

for (int k = lo; k <= hi; k++) {

if (i > mid)

aux[k] = a[j++];

else if (j > hi)

aux[k] = a[i++];

else if (less(aux[j], aux[i]))

aux[k] = a[j++];

else

aux[k] = a[i++];

}

}

private static void sort(int[] a, int[] aux, int lo, int hi) {

if (hi <= lo + CUTOFF - 1) {

insSort(a);

return;

}

int mid = lo + (hi - lo) / 2;

sort(aux, a, lo, mid);

sort(aux, a, mid + 1, hi);

if (!less(a[mid + 1], a[mid]))

return;

merge(aux, a, lo, mid, hi);

}

public static void sort(int[] a) {

int[] aux = new int[a.length];

sort(aux, a, 0, a.length - 1);

}

public static void insSort(int[] a) {

int N = a.length;

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

for (int j = i; j > 0; j--)

if (less(a[j], a[j - 1]))

exch(a, j, j - 1);

else

break;

}

private static void exch(int[] a, int i, int j) {

int swap = a[i];

a[i] = a[j];

a[j] = swap;

}

private static boolean less(Comparable v, Comparable w) {

return v.compareTo(w) < 0;

}

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 static void main(String[] args) {

int size = 8000;

int[] arr = new int[size];

int item = 0;

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

item = (int) (Math.random() \* 100);

arr[i] = item;

}

long startTime, endTime, elapsed;

System.out.println("Merge sort with array size " + size);

System.out.println("Cutoff at: " + CUTOFF);

startTime = System.currentTimeMillis();

Lec2Act2\_1.sort(arr);

endTime = System.currentTimeMillis();

elapsed = endTime - startTime;

System.out.println("Merge Sort Time: " + elapsed);

}

}

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CUTOFF |  |  |  | |
| 7 | 12 | 20 | | 25 |
| A screen shot of a smart phone  Description automatically generated |  | A close up of a screen  Description automatically generated | |  |
|  | A screen shot of a social media post  Description automatically generated |  | | A close up of a screen  Description automatically generated |
| A screen shot of a social media post  Description automatically generated |  | A close up of a screen  Description automatically generated | | A close up of a screen  Description automatically generated |
| A close up of a screen  Description automatically generated | A close up of a screen  Description automatically generated | A close up of a screen  Description automatically generated | | A picture containing device  Description automatically generated |

1. For the quicksort algorithm, its running time is different in best-case and worst- case scenarios depending on the pivot item in the partitioning stage.
2. Show the derivation to obtain the theoretical running time in best-case and worst-case scenarios.
   1. Best case is a completely random array that is shuffled
   2. Worst case would be an array that is already sorted or an array that has all of the same numbers.
3. Construct examples on best-case and worst-case, respectively. Compare the running time in both situations. You need to run the algorithm against large-size arrays, say N=1000, 2000, 4000, 8000.

import java.util.Random;

import java.util.Arrays;

import java.util.\*;

public class Lec2Act2\_2 {

private static int partition(int[] a, int lo, int hi) {

int pivot = a[hi];

int i = (lo - 1);

for (int j = lo; j < hi; j++) {

if (a[j] <= pivot) {

i++;

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

}

int temp = a[i + 1];

a[i + 1] = a[hi];

a[hi] = temp;

return i + 1;

}

public static void sort(int[] a) {

sort(a, 0, a.length - 1);

}

static void sort(int a[], int lo, int hi) {

if (lo < hi) {

int pi = partition(a, lo, hi);

sort(a, lo, pi - 1);

sort(a, pi + 1, hi);

}

}

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 static void main(String[] args) {

int size = 2000;

int[] arr = new int[size];

for (int i = 0; i < arr.length; i++) {

arr[i] = i;

}

Collections.shuffle(Arrays.asList(arr));

int[] arr2 = new int[size];

int item2 = 0;

int n2 = arr2.length;

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

item2 = (int) (i);

arr2[i] = item2;

}

long startTime, endTime, elapsed;

long startTime2, endTime2, elapsed2;

System.out.println("Insertion sort with array size " + size);

System.out.println("=========================WORST CASE==========================");

startTime = System.currentTimeMillis();

sort(arr);

printArray(arr);

endTime = System.currentTimeMillis();

elapsed = endTime - startTime;

System.out.println("WORST CASE Time: " + elapsed);

System.out.println("=========================BEST CASE==========================");

startTime2 = System.currentTimeMillis();

sort(arr2, 0, n2 - 1);

printArray(arr2);

endTime2 = System.currentTimeMillis();

elapsed2 = endTime2 - startTime2;

System.out.println("BEST CASE Time: " + elapsed2);

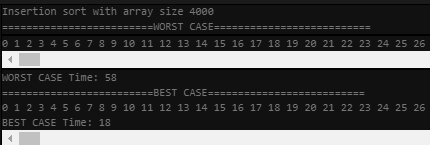
}

}A close up of a screen

Description automatically generated

A close up of a device

Description automatically generated



A screenshot of a cell phone

Description automatically generated