Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 5

Week: 05

Due: Nov 18, 2019

**Lab 5**

**Problem 1**:

**Answer**:

Based on Stability Sorting, an algorithm is stable if the algorithm maintains the relative order of elements with equal keys. In other words, stable sorting algorithm is said when the original order of equal keys is maintained.

Bubble Sort is a sorting algorithm in which two adjacent elements of an array are compared and swapped if they are in wrong order. This way repeats until the array is sorted. In this way, the order is maintained if the two elements are equal, that means it is stable algorithm.

Insertion Sort works similarly, where the second element is compared with the first element and is swapped if it is not in order. Next, we place the third element in the subarray of the first and second elements. We repeat this process until our array gets sorted. In other word, the order of the elements is maintained.

Selection Sort works differently. First, it finds the smallest element in the array. This smallest element is swapped with the first element. After this, search for the smallest element in the subarray formed by excluding the first element and compare it with the first element of the subarray. This process repeats until the array is sorted. As we can see, the order of the elements if they are equal, would be swapped, which means unstable.

**Problem 2**:

**Answer**:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | 7 | 6 | 5 | 4 | 3 | 2 | 1 |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 7 | 6 | 5 |  |  |  | 4 | 3 | 2 | 1 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  | 6 | 5 |  |  | 4 | 3 |  |  | 2 | 1 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  | 6 |  | 5 |  | 4 |  | 3 |  | 2 |  | 1 |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| 7 |  | 5 | 6 |  |  | 3 | 4 |  |  | 1 | 2 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  | 5 | 6 | 7 |  |  |  | 1 | 2 | 3 | 4 |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  |  |  |

**Problem 3**:

**Answer**:

Algorithm: MergeInsertionSort(A, min, max)

Input: array of integers

Output: sorted array

If max – min > threshold

Get middle point = (min + max) / 2

Mergeinsertionsort(A, min, middle)

Mergeinsertionsort(A, middle, max)

Then merge all sorted sides

Merge(A, min, middle, max)

Else

Call insertionsort(A, min, max)

The implementation of the above algorithm.

**import** java.util.\*;

**public** **class** MergeInsertionSort {

**public** **static** **int** *threshold* = 5;

**public** **static** **int**[] generateArray(**int** n, **int** range) {

Random rnd = **new** Random();

**int**[] arr = **new** **int**[n];

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

arr[i] = rnd.nextInt(range);

}

**return** arr;

}

**public** **static** **void** printArray(**int**[] arr) {

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

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

}

System.***out***.println();

}

**public** **static** **void** merge(**int**[] arr, **int** min, **int** mid, **int** max) {

**int** n1 = mid - min + 1;

**int** n2 = max - mid;

**int**[] leftArr = Arrays.*copyOfRange*(arr, min, mid + 1);

**int**[] rightArr = Arrays.*copyOfRange*(arr, mid + 1, max + 1);

**int** rIdx = 0;

**int** lIdx = 0;

**for** (**int** i = min; i < max - min + 1; i++) {

**if** (rIdx == n2) {

arr[i] = leftArr[lIdx];

lIdx++;

} **else** **if** (lIdx == n1) {

arr[i] = rightArr[rIdx];

rIdx++;

} **else** **if** (rightArr[rIdx] > leftArr[lIdx]) {

arr[i] = leftArr[lIdx];

lIdx++;

} **else** {

arr[i] = rightArr[rIdx];

rIdx++;

}

}

}

**public** **static** **void** insertionSort(**int**[] arr, **int** min, **int** max) {

// we loop through all elements in the original array from the min + 1 element

**for** (**int** i = min; i < max; i++) {

// store the current element as the key

**int** tempVal = arr[i + 1];

**int** j = i + 1;

// loop through all elements from the key to the min element

// check if the current element is smaller than the key

**while** (j > min && arr[j - 1] > tempVal) {

// we move the current element backward

arr[j] = arr[j - 1];

j--;

}

// we finally move the key

arr[j] = tempVal;

}

// Debug purpose

// int[] temp = Arrays.copyOfRange(arr, min, max +1);

// Arrays.stream(temp).forEach(i -> System.out.print(i + " "));

// System.out.println();

}

**public** **static** **void** mergeSort(**int**[] arr, **int** min, **int** max) {

**if** (max - min > *threshold*) {

// get the middle point

**int** mid = (min + max) / 2;

// apply merge sort to both parts of this

*mergeSort*(arr, min, mid);

*mergeSort*(arr, mid + 1, max);

// and finally merge all that sorted stuff

*merge*(arr, min, mid, max);

} **else** {

*insertionSort*(arr, min, max);

}

}

**public** **static** **void** main(String[] args) {

**int** n = 10;

**int** range = 100;

**int**[] arr = *generateArray*(n, range);

System.***out***.println("Before sorting");

*printArray*(arr);

*mergeSort*(arr, 0, arr.length - 1);

System.***out***.println("After sorting");

*printArray*(arr);

}

}

After testing and comparing this MergeInsertionSort (MergeSortPlus) with standard MergeSort, I can see the performance improved due to the overhead of many recursive merging calls is reduced significantly.

**Problem 4**:

**Answer**:

Basically, the iteration of the given Binary Search algorithm ends after k iterations.

At each iteration, the array is split into half, its length is reduced by half at each iteration.

Iteration 1: length of array = n

Iteration 2: length of array =

Iteration 3: length of array =

Hence, at iteration k, length of array is , and subsequently, it becomes 1. In other word, , or

Therefore, the time complexity of Binary search is

**Problem 5**:

**Answer**:

|  |  |
| --- | --- |
|  |  |
|  |  |

In the case of complete tree, all leaves are filled. As we counted, there are 8 leaves with the height is 3. Therefore, the statement is true.

If the height is 2, the complete tree will have 4 leaves.

In other words, the maximum number of leaves is , where k is the height of the tree and n is the maximum number of leaves.

**Problem 6**:

**Answer**:

1. To check the input array is self-aware.

Algorithm: check array self-aware

Input: an array

Output: true for self-aware or false for not self-aware

Counter 🡨 0

For i from 0 to array length – 2

If array[i] != 0

For j from i + 1 to array length – 1 then

If array[i] == array[j] then

Counter++

array[j] 🡨 0 // to skip the already counted element

If counter != array[i] return false

Return true

The asymptotic running time is , because we need to run through the array twice to count the occurrences for each non-zero element in the array.

**public** **static** **boolean** checkSelfAwareArray(**int**[] arr) {

**int** counter = 0;

**boolean** selfAwareFlag = **true**;

**for** (**int** i = 0; i < arr.length - 1; i++) {

**if** (arr[i] != 0) {

counter = 1;

**for** (**int** j = i + 1; j < arr.length; j++) {

**if** (arr[i] == arr[j]) {

counter++;

arr[j] = 0;

}

}

**if** (arr[i] != counter) {

selfAwareFlag = **false**;

**break**;

}

}

}

**return** selfAwareFlag;

}

Testing:

**int**[] arr = {2, 0, 2, 0};

System.***out***.println(*checkSelfAwareArray*(arr));

**int**[] arr1 = {2, 1, 2, 0};

System.***out***.println(*checkSelfAwareArray*(arr1));

**int**[] arr2 = {2, 1, 2, 3, 0};

System.***out***.println(*checkSelfAwareArray*(arr2));

**int**[] arr3 = {5, 2, 1, 5, 2, 5, 5, 0, 5, 0};

System.***out***.println(*checkSelfAwareArray*(arr3));

**int**[] arr4 = {5, 2, 1, 5, 2, 5, 5, 0, 5, 1};

System.***out***.println(*checkSelfAwareArray*(arr4));

Result:

true

true

false

true

false

1. To print out a list of self-aware arrays.

As definition, each element array[i] is the exact number of occurrences of i in array, that means:

* Array[i] value is not more than the array length
* Subsequently, we can generate multiple list by starting the first element by the array length, then decrease by 1 for each starting first element of the generated array. For example: array length = 5, we can ignore the order of the elements.

{5, 5, 5, 5, 5}

{4, 4, 4, 4, 0} or {4, 4, 4, 4, 1}

{3, 3, 3, 2, 2} or {3, 3, 3, 2, 1} or {3, 3, 3, 2, 0} or {3, 3, 3, 1, 1} or {3, 3, 3, 1, 0} or {3, 3, 3, 0, 0}

{2, 2, 1, 1, 1} or {2, 2, 1, 1, 0} or {2, 2, 1, 0, 0} or {2, 2, 0, 0, 0}

{1, 0, 0, 0, 0}

{0, 0, 0, 0, 0}

The asymptotic running time would be because we have to run n loop on top of n elements.

Algorithm: generate list of self-aware array

Input: an integer n

Output: list of self-aware arrays

k 🡨 n

While k >= 0 then

Array 🡨 new array with n length

For i 🡨 0 to k do

Arr[i] 🡨 k

If n – k < k then

For i 🡨 k to n then

**public** **static** **void** generateSelfAwareArray(**int** n) {

**int** k = n;

**while**(k >= 0) {

**int**[] arr = **new** **int**[n];

**for** (**int** i = 0; i < k; i++) {

arr[i] = k;

}

**if** (n - k < k) {

**for** (**int** i = k; i < n; i++) {

arr[i] = n - k;

}

}

k--;

}

}

Result:

5 5 5 5 5

4 4 4 4 1

4 4 4 4 0

3 3 3 2 2

3 3 3 2 1

3 3 3 2 0

3 3 3 1 1

3 3 3 1 0

3 3 3 0 0

2 2 1 1 1

2 2 1 1 0

2 2 1 0 0

2 2 0 0 0

1 0 0 0 0

0 0 0 0 0