Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 7

Week: 07

Due: Dec 02, 2019

**Lab 7**

**Problem 1**:

Show that any comparison-based algorithm to sort 4 elements requires at least 5 comparisons.

**Answer**:

In a comparison sort, the sorted order is determined by a sequence of comparison between pairs of elements. Comparison reflects on the use of decision tree.

We know that the depth of a decision tree for a given value of n is Ω(n log n).

That means every comparison sort will require Ω(n log n) comparison in the worst case.

In other words,

* In the worst case of n = 4, we will have 4 log 4 = 8 comparisons being made.
* And in the best case, it would need log(n!) = n log n – O(n) comparison, which is in our case log2(4!) = 4.58 ~ 5 comparisons for an arrays with 4 elements.

**Problem 2**:

Devise an algorithm to sort 4 elements using exactly 5 comparisons.

**Answer**:

Let assume we have an array [10, 5, 10, 1].

If we use the HeapSort, then we will need to define the root, then go left, right and compare with the root of the heap. In this case, we will have as following:

root=10

Check left child larger than root

leftChildIdx=1-heapSize=3 and arrA[leftChildIdx]=5-arrA[largest]=1

Check right child larger than largest element

rightChildIdx=2-heapSize=3 and arrA[rightChildIdx]=10-arrA[largest]=5

Check if largest not root

arrA[i]=1-arrA[largest]=10

root=10

Check left child larger than root

leftChildIdx=1-heapSize=2 and arrA[leftChildIdx]=5-arrA[largest]=1

Check if largest not root

arrA[i]=1-arrA[largest]=5

root=5

root=1

From the above:

* To check left child: 2 comparisons
* To check right child: 1 comparison
* To check root: 2 comparisons

In total, 5 comparisons had been done. This is just a particular array. The main point is the number of comparison of any comparison-based algorithm will be at least nlogn as Lemmas 1, 2 and 3 told us about the depth of nodes and depth of leaf in decision tree.

**Problem 3**:

Devise an algorithm that arranges the elements of a length‐n integer array according to the following scheme:

* position 0: the smallest integer
* position 1: the largest integer
* position 2: the second smallest integer
* position 3: the second largest integer etc.

For example, this algorithm would arrange the input array {1, 2, 17, -4, ‐6, 8} as follows: {-6, 17, -4, 8, 1, 2}. (Notice that –6 is the smallest, 17 the largest, -4 second smallest, 8 second largest, etc.) What is the asymptotic running time of your algorithm? What is the fastest possible asymptotic running time for such an algorithm? Prove your answer.

**Answer**:

Based on the result of the arrangement:

* The array is arranged into pairs like [smallest, largest], [second smallest, second largest], etc.
* The number of pair will be the array length divided by 2.
* If the number of elements is odd, the remaining element will be in the last position of the arranged array.

So, the algorithm could be as following

**Algorithm**: SortIntoPairs (array)

**Input**: array of integers

**Output**: sorted array with pairs

1. Find the number of possible pairs
2. Run loop with the number of possible pairs
   1. Find the smallest element from the array
   2. Move the smallest element from the array to new array (meaning the size of the original array will be reduced by 1)
   3. Find the largest element from the reduced array
   4. Move the largest element from the reduced array to new array (meaning the size of the reduced array will be reduced by 1)
   5. Move to the next pair in the new array

Repeat the loop on the second reduced array

End loop

1. If the reduced array still have element, move that element to the last position of the new array.

**Analysis**:

With the above algorithm, we can see the asymptotic time would be:

Step 1: 1

Step 2: n/2

Step 2.a: n

Step 2.b: n

Step 2.c: n – 1

Step 2.d: n – 1

Step 3: 1

In other words, in the worst case, we will have O(1 + n\*(n + n + (n – 1) + (n-1))/2 ) = O(n^2)

After each loop, the size of the array is reduced by 2, the complexity reduces from O(n) to O(log n).

Therefore, the algorithm time complexity will be O(n^2) in worst case and best case will be O(n log n)

**Implementation**:

public class SortIntoPairs {

  private static int maxIdx = -1;

  private static int minIdx = -1;

  public static void runSortIntoPairs(int[] arr) {

    int n = arr.length;

    int[] resArr = new int[n];

    int pairCount = n/2;

    int cnt = 0;

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

      // find the MINIMUM of the given array and fill into the first position of the pair

      resArr[i + cnt] = findMin(arr);

      // reduce the given array by removing the found MINIMUM element

      arr = reduceArray(arr, minIdx);

      // find the MAXIMUM of the given array and fill into the second position of the pair

      resArr[i + cnt + 1] = findMax(arr);

      // reduce the given array by removing the found MAXIMUM element

      arr = reduceArray(arr, maxIdx);

      // Jump to the next pair

      cnt = cnt + 1;

    }

    if (arr.length > 0) {

      resArr[resArr.length - 1] = arr[arr.length - 1];

    }

  }

  private static int findMax(int[] arr) {

    int max = Integer.MIN\_VALUE;

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

      if (arr[i] > max) {

        max = arr[i];

        maxIdx = i;

      }

    }

    return max;

  }

  private static int findMin(int[] arr) {

    int min = Integer.MAX\_VALUE;

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

      if (arr[i] < min) {

        min = arr[i];

        minIdx = i;

      }

    }

    return min;

  }

  private static int[] reduceArray(int[] arr, int idx) {

    int[] retArr = new int[arr.length - 1];

    int s = 0;

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

      if (i != idx) {

        retArr[s++] = arr[i];

      }

    }

    return retArr;

  }

}

**Test result**:

Array: [1, 2, 17, -4, -6, 8]

Sorted array into pairs: [-6, 17, -4, 8, 1, 2]

Array: [1, 2, 17, -4, -6, 8, 5]

Sorted array into pairs: [-6, 17, -4, 8, 1, 5, 2]

**Problem 4**:

Use RadixSort to sort the following: {80, 1, 46, 53, 28, 55, 32, 6, 9}, with radix = 9.

**Answer**:

Basically the RadixSort depends on the radix. As the elements of the array are integer, that means the base radix = 9 will not satisfy the range of the elements to cover each digit counting bucket, because the integers base on the 10-digit numbers.

Let’s see why the radix = 9 not sufficient for integer sorting.

The given array:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 80 | 1 | 46 | 53 | 28 | 55 | 2 | 6 | 9 |

With radix = 9, for the first pass, we have as below:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bucket | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Counting | 2 | 1 | 1 | 1 | 0 | 1 | 2 | 0 | 1 |
| Result | 80, 9 | 1 | 2 | 53 |  | 55 | 46, 6 |  | 28 |

As we can see the problem with the digit 0 and 9. The digit 0 or 9 divides 9 will result into 0. That means 80 and 9 will be counted in the bucket 0.

Second pass will give us as following:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Bucket | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Counting | 1 | 0 | 1 | 0 | 1 | 2 | 0 | 0 | 1 |
| Result | 9, 1, 2, 6 |  | 28 |  | 46 | 53, 55 |  |  | 80 |

The main problem with the number 9 because it will remain at bucket 0 for all passes. That means the radix base 9 will not satisfy for the integer sorting.

To sort integers, we should use radix = 10, which will cover all digits.