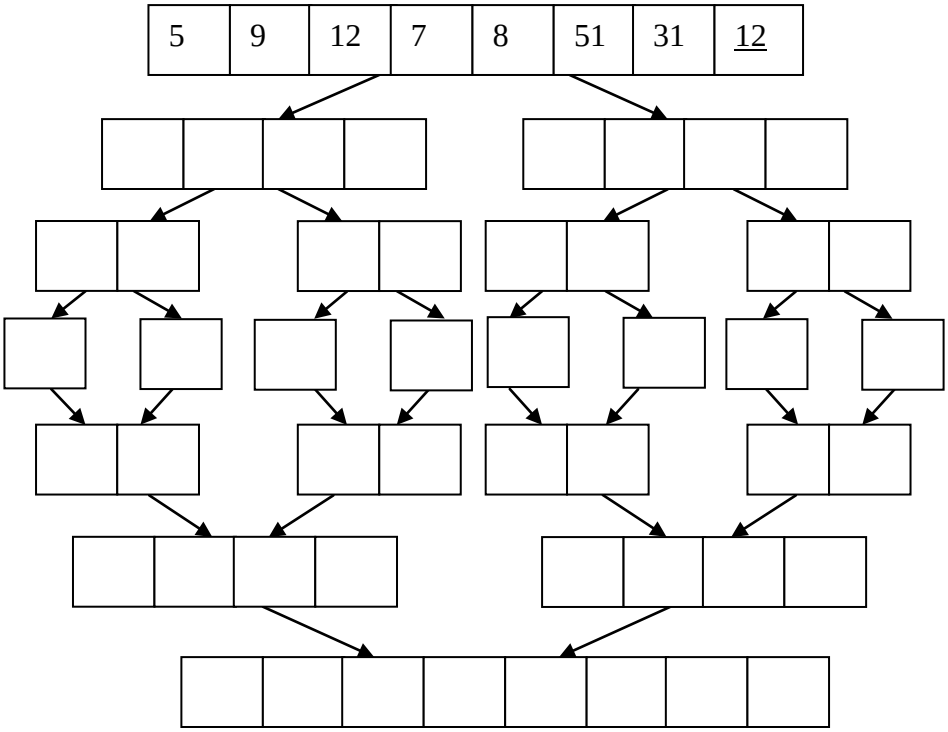
CS1102: Data Structure and Algorithms

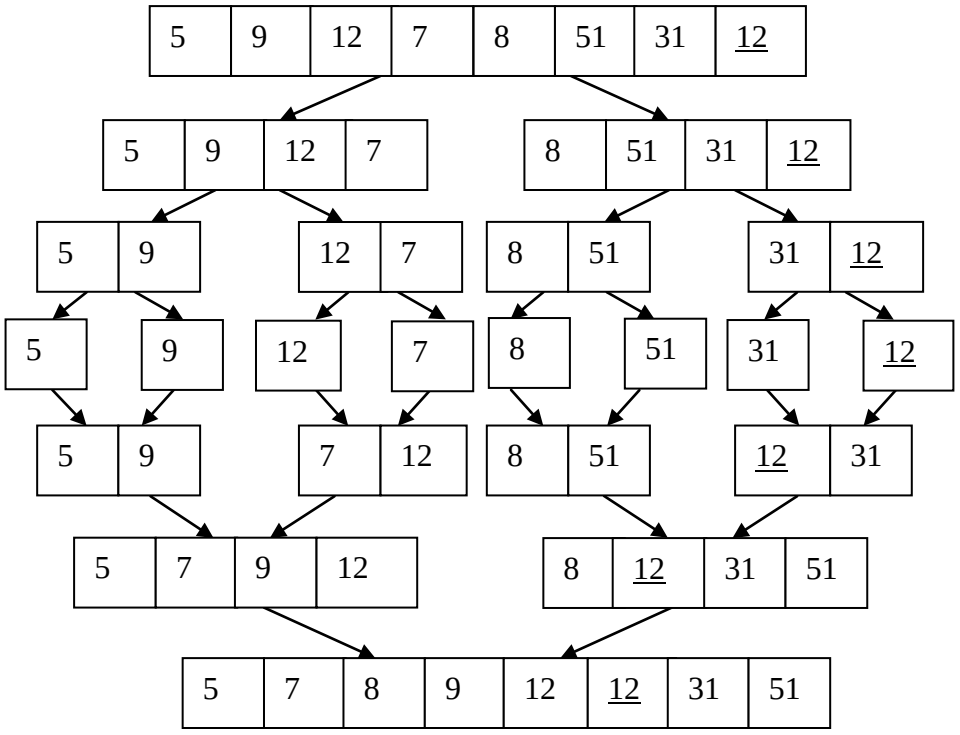
Tutorial 7: Sorting (Solutions)

Week of 15 March 2010

1. a. Perform merge sort on the following array:



Answer:



b. Merge sort has the following recursive structure:

```
1. public static void mergeSort(int[] a, int i, int j){
2. if (i < j) {
3.        int mid = (i+j)/2;
4.        mergeSort(a,i,mid);
5.        mergeSort(a,mid+1,j);
6.        merge(a,i,mid,j);
7.        }
8. }
```

Suppose line 6 is removed, what is the time complexity of the modified program?

Answer:

Let $n = 2^k$ for some value of k where n is the length of our array.
Each method call takes constant time (since we remove merge step) and mergeSort(a, i, j) terminates when i = j.

Rename the modified program as modifiedMergeSort(). From the code, we have:

$O(\text{modifiedMergeSort}(a, 0, 2^k - 1)) =$
$= 1 + O(\text{modifiedMergeSort}(a, 0, 2^{k-1} - 1)) +$
$O(\text{modifiedMergeSort}(a, 2^{k-1}, 2^k - 1))$
(i,e. the sum of the complexity of line 3,4 and 5)
$=1 + 2 * O(\text{modifiedMergeSort}(a, 0, 2^{k-1} - 1))$
(since the complexity of the first half and of the second half are the same)
$=1 + 2*(1 + 2* O(\text{modifiedMergeSort}(a, 0, 2^{k-2} - 1)))$
$=1 + 2 + 2^2* O(\text{modifiedMergeSort}(a, 0, 2^{k-2} - 1)$
$= \ldots$
$=1 + 2 + 2^2 + \ldots + 2^k * O(\text{modifiedMergeSort}(a, 0, 2^{k-k} - 1))$
$=1 + 2 + 2^2 + \ldots + 2^k * O(\text{modifiedMergeSort}(a, 0, 0))$
$=1 + 2 + 2^2 + \ldots + 2^k$
$= 2^{k+1} - 1$
$=2n - 1$
$= O(n)$

Hence the complexity is O(n).

2. Based on the Big-O notation, which of these sorting algorithms would you use to sort the following data stored in an array: (1) Merge Sort; (2) Quick Sort with first element pivot; (3) Insertion Sort; (4) Selection Sort; (5) Bubble Sort (improved version) and (6) Radix Sort? Explain your answer.

a. 1,000,000 distinct integers from 0 to 999,999 in reverse order.
b. 1,000,000 distinct real numbers from 0.0 to 1.0 in random order.
c. 1,000,000 distinct integers from 0 to 999,999 with only one element out of place, i.e. the array is sorted without this element.
d. 1,000,000 distinct real numbers from 0.0 to 1.0, where all the elements are at most 5 places away from their proper position.

Answer:

a. Radix Sort is best, if space is not a factor.
(Also requires that the number of digits is fixed)
1) Merge Sort: always O(n log n)
2) Quick Sort: $O(n^2)$ – this is the worst case for Quick Sort
3) Insertion Sort: $O(n^2)$ – also worst case
4) Selection Sort: always $O(n^2)$
5) Bubble Sort: $O(n^2)$ – also worst case
6) Radix Sort: 6n operations, or O(6n) = O(n). The maximum number of digits is 6, so there are 6 iterations.

b. Quick Sort is best, but Merge Sort is also good if space is not a factor.
1) Merge Sort: always O(n log n)
2) Quick Sort: O(n log n) in average case
3) Insertion Sort: $O(n^2)$
4) Selection Sort: always $O(n^2)$
5) Bubble Sort: $O(n^2)$
6) Radix Sort: not appropriate for real numbers.

c. Insertion Sort performs best. If element is before its proper place with respect to the "bubbling" direction then Bubble Sort (improved version) is also a good choice;
1) Mergesort: always O(n log n)
2) Quick Sort: $O(n^2)$ – close to the worst case
3) Insertion Sort: O(n) – only O(n + d) comparisons and shifts are needed, where d is the distance between the current position and its proper position of the misplacing element. Note that the out of place element can be either before or after its proper position.
4) Selection Sort: always $O(n^2)$

5) Bubble Sort: "before its proper place" – O(n)
   "after its proper place" – $O(n^2)$
   For example, 2,3,4,5,6,7,8,9,1 can only be sorted after 8 iterations plus one final iteration to make sure no more bubbling; in each iteration, we need to do 8 comparisons.
6) Radix Sort: 6n operations, or O(6n) = O(n).

d. Insertion Sort is best. Bubble Sort is almost as good.
1) Merge Sort: always O(n log n) ≈ 1,000,000 x 20 = 20,000,000 operations.
2) Quick Sort: $O(n^2)$ – close to worst case
3) Insertion Sort: At most 5 shifts in each iteration, so O(5n) = O(n).
4) Selection Sort: always $O(n^2)$.
5) Bubble Sort: Each element shifts at most 5 times plus one final iteration to make sure no more bubbling, so O(5n+n) = O(6n) = O(n).
6) Radix Sort: Not appropriate for real numbers.

3. Write Java methods to achieve the following tasks. Your methods should have the complexity of O (n log n):

a. Find the median of an integer array (Recall that the median of a list of n number is defined as its $\lceil n/2 \rceil$ smallest element). Assume our array is not empty.

Answer:

The problem can be solved by applying the following steps:

i. Sorting the array using merge sort
ii. Retrieve the value at the index position of $\lceil n/2 \rceil$ - 1.

Since the complexity of the first step is O(n log n) where the complexity of the second step is O(1), the total complexity is O(n log n).

```
public static int median (int[] a){
      mergeSort(a, 0, a.length - 1);
      return a[(int)Math.ceil(a.length/2) - 1];

}
```

b. Check whether an integer array contains duplicates.

Answer:

Similar to part a, we firstly sort the array using merge sort. If the array contains duplicates, they should be next to each other after sorted. Hence by a single traversing, we can detect the duplicates in our array. Using merge sort costs O(n log n) while a single traversing costs O(n), therefore the complexity is O(n log n).

```
public static boolean hasDuplicates(int[] a){

      //If the array is empty or contains only 1 element,
      //return false.
      if(a.length <= 1) return false;
      else{
            //Sort the array
            mergeSort(a, 0, a.length - 1);

            //Traverse the array, return true if there
            //are duplicates.
            for(int i = 0 ; i < a.length – 1; i++){
                  if(a[i] == a[i + 1]) return true;
            }
            //if no duplicate is found, return false
            return false;
      }
}
```

c. Find the distance between the two closest numbers in an integer array.

Answer:

Here is the algorithm to achieve the task:

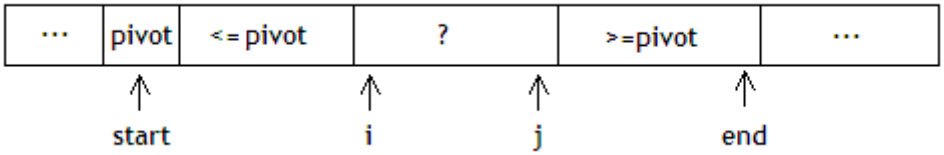
i. Sort the array using merge sort.
ii. Traverse the array, calculate the distance between two consecutive numbers and return the smallest distance.

The time complexity is O(n log n).

```
public static int smallestDistance(int[] a){

      //If our array is empty or only contains 1 element,
      //-1 will be returned as the result because we need 2
      //elements to calculate the distance.
      if(a.length <= 1) return -1;

      else{
            //Sort the array
            mergeSort(a, 0, a.length - 1);

            int distance = a[1] – a[0]; //The initial value of
                                        //distance.
            //Traverse the array, calculate the distance
            //between two consecutive numbers
            for(int i = 1 ; i < a.length – 1; i++){
                  distance = Math.min(distance,a[i + 1] – a[i]);
                  //if the distance is 0, which means that there
                  //are duplicates, we will immediately
                  // terminate the loop and return distance
                  //because 0 is the smallest distance we can
                  //find
                  if(distance == 0) return 0;
            }

            //return the smallest distance
            return distance;
      }
}
```

CS1102: Data Structure and Algorithms

4. This question will refer to the *QuickSort* algorithm stated in your lecture notes.
   a. Implement the *QuickSort* algorithm by using two indices ***i*** and ***j*** as shown in the diagram below to iterate through the subarray during each recursive call to find the partition index.




```
void QuickSort(int[]A, int start, int end)
{
   ...
}
```

Answer:

```
void QuickSort(int[]A, int start, int end)
{
   if(start < end)
   {
    int pivot = A[start];
    int i = start+1;
    int j = end;
    while(true)
    {
      while(i<end && A[i]<= pivot) //increment i till A[i] > pivot or
                                   till the end of the array
        i++;
      while(j>start && A[j]>=pivot) //decrement j till A[j] < pivot
                                   or till the beginning of the array
        j--;
      if(i<j) {
        swap(A, i, j); //swap A[i] and A[j] so that after swapping,
                        A[i]<pivot<=A[j]
        i++;
        j--;
      }
      else break;
    }
    swap(A, j, start); //j is the partition index, move pivot to
                        position j, which is its final position so
                        that all the elements on the left of the
                        pivot are smaller than the pivot and all
                        those on the right are greater or equal to
                        the pivot
    QuickSort (A, start, j-1);
    QuickSort (A, j+1, end);
   }
}

//This method will swap a[i] and a[j]
void swap (int[] a, int i, int j){
```

```
  int temp = a[i];
  a[i] = a[j];
  a[j] = temp;

}
```

b. Since *QuickSort* has a time complexity of O(n^2) in the worst case, and O(*n log n*) in the best/average case, while merge sort has a time complexity of O(*n log n*) under all cases, explain why we still use *QuickSort* instead of *MergeSort*.

Answer:

First, for Quick Sort, we can store the pivot (for integers and real numbers) in *registers* and therefore avoid frequent reads and writes from the memory during comparison; while for Merge Sort, each time we are comparing two different elements which cannot be stored in registers at all times, therefore frequent reads and writes are inevitable.

Second, Merge Sort using array needs to create *an additional array* to store the output during each iteration, which takes time as well as space. In addition, when too many arrays are created, there may not be large enough contiguous memory spaces to store those new arrays at one place. The garbage collection process has to reclaim spaces occupied by those inaccessible arrays and thus will be time consuming.

Finally, for Quick Sort, in theory, when *randomized pivots* are chosen, there will be some good choices and some bad choices, and eventually in the entire recursive tree, this balances out.