CS-3250

Section 01

-

Homework #4

Patrick Pei

Professor Jeremy Spinrad

Due 3-22-2017

Exercises

1 Expand the recurrence $T(n) = 7T(n/2) + n^3$, 1 step, that is so $T(n/2)$ does not appear in the expression.

$$T(n) = 7T(n/2) + n^3$$

$$T(\frac{n}{2}) = 7T(n/4) + (\frac{n}{2})^3$$

$$T(n) = 7(7T(\frac{n}{4}) + (\frac{n}{2})^3) + n^3$$

2 Suppose that you modify the "big 5"selection algorithm so that your subsets are of size 3 instead of 5. Show your work in getting the correct recursive equation which would govern the running time of the algorithm.

For subsets of size 3, the steps used previously can be minorly modified to fit this new requirement.

1. Divide the n elements of the input into $\lfloor \frac{n}{3} \rfloor$ groups of 3 elements each and at most one group made up of the remaining n mod 3 elements.
2. Find the median of each of the $\lceil \frac{n}{3} \rceil$ groups.
3. Recursively find the median of the $\lceil \frac{n}{3} \rceil$ medians found previously.
4. Partition the elements into less than, equal to, and greater than the median of the medians.
5. If i = k, the result will be found, otherwise, recursively apply the algorithm based on the size of the groups.

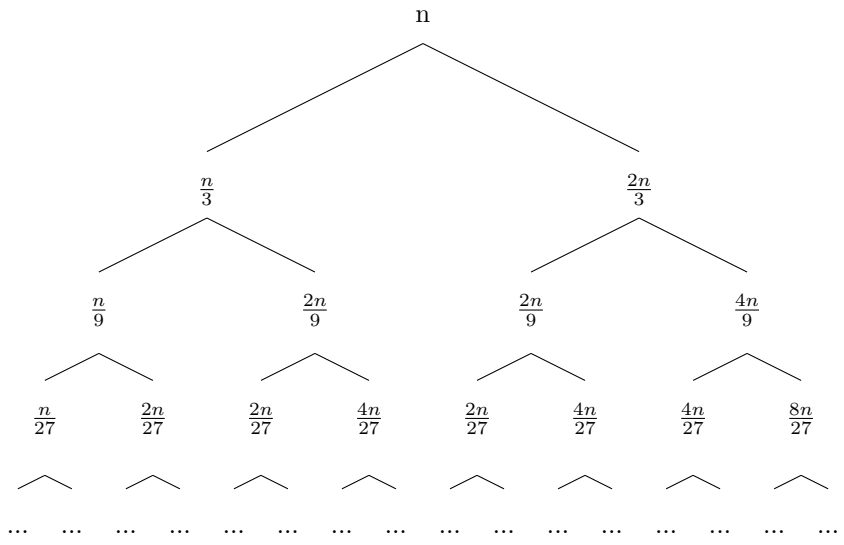
The analysis of this algorithm starts with the observation that for half of the groups, these groups contribute at least 2 elements since half of the medians will be greater than the median of medians (the two exceptions are the group that has fewer than 3 elements if 3 does not exactly divide n and the group containing the median of medians). Thus, in mathematical terms this is

$$2(\frac{1}{2}\lceil \frac{n}{3} \rceil) \geq \frac{n}{3}$$

Furthermore, in the worst case, this many elements are reduced leaving $n - \frac{n}{3} = \frac{2n}{3}$ elements remaining for the recursive procedure. Applying this, the recurrence relation for this algorithm becomes

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + cn$$

since steps 1, 2, and 4 of the algorithm take $O(n)$ time to complete. Step 2 will recursively call the algorithm on a third of the input since the groups are of size 3, and step 5 will have the term $T(\frac{2n}{3})$ since in the very worst case, only $\frac{1}{3}$ of the input elements are eliminated. Finally, by solving this recurrence relation, the overall running time of the algorithm can be found. This can be seen through the first few levels of a recurrence tree representing the recursive equation.



It can be seen that each level of the tree sums to n and for a full tree there are $log_3(n)$ levels. Thus, the runtime complexity of using subsets of size 3 is $O(nlog(n))$.

3

3a) Show that selection sort may repeat a comparison between elements which has already been made.

Proof by example: take the input set [n, n - 1, n - 2, ... 1] Selection sort will identify that the minimum is not in the correct position by comparing and keeping track of a minimum with all elements after index i. In this generalized case, the next step will be [1, n - 1, n - 2, ... n] where comparisons of (n, n - 1), (n - 1, n

- 2), (n - 2, n - 3), etc will have been made. Thus, the next step which results in [1, 2, n -2, ..., n - 1, n] will result comparisons of (n - 1, n - 2), (n - 2, n - 3), (n - 3, n - 4), etc where it is clear that a repeat comparison of (n - 1, n - 2) has been made.

3b) Show that insertion sort will never repeat a comparison between elements which has already been made.

Insertion sort maintains a sorted array at all times and the very idea behind it is to insert an additional element into the correct position to keep the existing array sorted. Moreover, with a starting point of an array of size 1, each element that is inserted is appropriately placed through comparisons with elements that are part of the sorted array. It suffices to show that since it is only finding the correct place in an existing set of numbers, once placed there, the element will no longer be compared with any of the other numbers in the sorted array after being placed (since it is sorted, elements already placed will not need to be compared with each other anymore). Thus, the only comparisons that are possible with the elements already in the sorted set are comparisons with elements that are not yet a part of the sorted set proving that insertion sort will never repeat a comparison between elements which has already been made.

4 Show that mergesort may make a comparison x vs y which has the following property.
For any possible n, there are n sorted orders consistent with previous comparisons, but only 1 of these orders is consistent with x less than y. Thus while mergesort never repeats a comparison, it may make comparisons which are in some sense "very bad". I note that heapsort is even worse, in the sense that heapsort can repeat a comparison which as already been made.

Mergesort may make a comparison x vs y which has the above described property in the case of the penultimate comparison. It is clear that if mergesort is about to make the penultimate comparison, then each of the previously mergesorted subarrays will be sorted at this point and that each of these sorted subarrays implicitly contains rules from previous comparisons. Furthermore, the approach in general can be taken back one step and it can be seen that when analyzing more than just the runtime of these sorting algorithms,

the very comparisons made must be analyzed as well. When taking the approach of focusing more on the comparisons being made throughout the algorithm, it can be seen in general that clearly and obviously, the worst thing possible for a sorting algorithm to do is to repeat a comparison. The entire point is to make comparisons to narrow down the possible orderings and repeating a comparison would not split up the number of outcomes at all. From this, it can be seen that in heapsort, once the root is removed and replaced, repeat comparisons can be made. Whereas, mergesort will never repeat a comparison so the worst thing from here is to make a comparison that does as little as possible for reducing the number of outcomes or sorted orderings, since the entire point as previously stated is to reduce these outcomes. Fundamentally, the best possible outcome for a comparison is to split the number of outcomes in 2 where if a comparison between x and y is made, the outcomes will be organized into x ¡ y and x ¿ y. Thus, in this case, with the property stated in the prompt, the outcomes, if there are k of them, will now be limited to 1 and k - 1, splitting the outcomes as poorly as possible. This can now be seen with a specific example where [x_0, x_2, x_1, x_3]. In the penultimate step, the comparisons up until this point will be of the form (0 ¡ 2), (1 ¡ 3), and finally when the last comparison is made, it can be shown by the following program with rules that all of these k possible outcomes will be reduced by only 1 - giving the very worst case of mergesort.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class Algorithms {
5      public static void main(String[] args) {
6          Integer[] arr = {0, 2, 1, 3};
7          List<List<Integer>> result = getPermutations(arr);
8
9          for (int i = 0; i < result.size(); i++) {
10             System.out.println(result.get(i));
11         }
12     }
```

```
13
14      public static List<List<Integer>> getPermutations(Integer[] arr) {
15          List<List<Integer>> result = new ArrayList<>();
16          getPermutations(arr, 0, result);
17          return result;
18      }
19
20      public static void getPermutations(Integer[] arr, int index, List<List<Integer>> list)
21          if (index >= arr.length - 1) {
22              if (passConditions(arr)) {
23                  ArrayList<Integer> temp = new ArrayList<>();
24                  for (int i = 0; i < arr.length; i++) temp.add(arr[i]);
25                  list.add(temp);
26              }
27          } else {
28              for (int i = index; i < arr.length; i++) {
29                  swap(arr, i, index);
30                  getPermutations(arr, index + 1, list);
31                  swap(arr, i, index);
32              }
33          }
34      }
35
36      public static boolean passConditions(Integer[] arr) {
37          String str = "";
38          for (int i = 0; i < arr.length; i++) {
39              str += arr[i];
```

```
40          }
41
42          return str.indexOf('0') < str.indexOf('2') &&
43                  str.indexOf('1') < str.indexOf('3');
44      }
45
46      public static void swap(Integer[] arr, int i, int j) {
47          Integer temp = arr[i];
48          arr[i] = arr[j];
49          arr[j] = temp;
50      }
51  }
```