CSC 225 FALL 2015
ALGORITHMS AND DATA STRUCTURES I
ASSIGNMENT 3
UNIVERSITY OF VICTORIA

(Question 1) Let A be the array to sort, and we use a modified Merge-Sort on A.

To get the total number of inversions by the end of a Merge-Sort of A we count the inversions fixed in each merge. So, we count the number of inversions which are fixed while merging two successive subarrays A_1 and A_2 into a new subarray B of A. Note that trying to count inversions individually could never work, because there could be $\mathcal{O}(n^2)$ inversions.

As we compare elements during Merge-Sort, A_1 is already sorted from smallest to largest, as is A_2. Suppose we compare $a_k \in A_1$ with $a_j \in A_2$. If $a_k < a_j$, and a_j is inserted into B first, then a_k is smaller than all remaining elements in both A_1 and A_2 and no inversions are fixed. If $a_k > a_j$, then a_j is inserted into B and a_j is smaller than all remaining elements in both A_1 and A_2. In this case, the number of inversions fixed is the number of elements remaining in A_1.

It is actually a bit tricky to get the return values set up properly to count all the inversions as we merge, but it should look something like this:

```
1 ) CountInversionsMergeSort(A, low, high) {
2 )   n ← high − low + 1;
3 )   if (n < 2) {
4 )     return 0;
5 )   }
6 )   mid ← low + n/2;
7 )   // note that the recursive calls must be done in this order
8 )   left ← CountInversionsMergeSort(A, low, mid − 1);
9 )   right ← CountInversionsMergeSort(A, mid, high);
10 )  both ← CountInversionsMerge(A, low, mid, high);
11 )  return left + right + both;
12 ) }
```

where the `CountInversionsMerge` function is implemented as described in the previous paragraph, and returns the number of inversions counted when merging the successive subarrays in A from indices low to $mid - 1$ and mid to $high$.

Since these extra steps in Merge-Sort are each constant time, we can count the total number of inversions in time $\mathcal{O}(n \log n)$.

comments

Many people tried to run a linear algorithm, which is a commendable effort, but the operation of removing an element from an array is, unfortunately, not constant time. Some

linear number of values remaining in the array may have to be shifted back one index to maintain the access properties that ensure a correct algorithm.

Some students were misunderstanding how recursive code deals with arrays, and trying to return the array, or return both the array and the count of inversions. Most languages when passed the array as a parameter will not copy the entire array as its own local variable, it will pass a reference to the array, so there is no need to return the array A. Somewhere near the end of the `CountInversionsMerge` function should copy the sorted values back into A.

(Question 2) There are actually a number of possible ways to arrange the integers 1 to 7 in an array so that Quick-Sort takes $\frac{1}{2}(7)(7-1) = 21$ steps, i.e. when there are i elements left to sort during an iteration, it partitions $i-1$ elements into one subarray, and the pivot by itself into another. Therefore, Quick-Sort must choose either the smallest or largest of the remaining i elements as pivot.

The trouble is that there are a couple different ways to deal with partitioning the array. One way is to create new arrays at each recursion step for each partition. The second way is to swap elements within the array itself as we partition.

Let's deal with the partition strategy that creates new arrays first. Working backward to build the array from size i to $i+1$ is then simply a matter of inserting either the next $(i+1)^{th}$ largest or smallest element among those already inserted, into the chosen pivot index calculated for an array of size $i+1$. Note that we can start with any value we wish, and our first choice will determine the remaining minimum or maximum elements, but there is always at least either a next minimum or maximum element for each insertion we make no matter which element is inserted first.

The sequence of index positions where we make an insertion so as to place the worst pivots is: 0, 1, 1, 2, 2, 3, 3. Any elements already at the position of insertion or larger are each moved one index higher. So, for example, we can build a worst-case array in the most straight-forward way, by always inserting the next largest element at every iteration, as opposed to sometimes inserting the next smallest element:

```
[1] (insert 1 at index 0)

[1 2] (insert 2 at index 1)

[1 3 2] (insert 3 at index 1)

[1 3 4 2] (insert 4 at index 2)

[1 3 5 4 2] (insert 5 at index 2)

[1 3 5 6 4 2] (insert 6 at index 3)

[1 3 5 7 6 4 2] (insert 7 at index 3)
```

and it should be clear that Quick-Sort performs the worst for the array [1 3 5 7 6 4 2].

The second partition strategy swaps the pivot with the element in the array that occupies the pivot's final position once the array is completely sorted. Constructing a worst-case array is not so easy in this strategy, but [6 5 4 7 3 2 1] is one example. If you want to convince yourself that the successive pivots are the worst choice at each stage, here is a step-by-step trace of this example, but to keep things short, the partitioning comparison steps are written in one line:

```
[6 5 4 7 3 2 1]

(7 chosen as pivot)

[6 5 4 1 3 2] [7] (7 swaps with 1)

(1 chosen as pivot)

[1] [5 4 6 3 2] [7] (1 swaps with 6)

(6 chosen as pivot)

[1] [5 4 2 3] [6] [7] (6 swaps with 2)

(2 chosen as pivot)

[1] [2] [4 5 3] [6] [7] (2 swaps with 5)

(5 chosen as pivot)

[1] [2] [4 3] [5] [6] [7] (5 swaps with 3)

(3 chosen as pivot)

[1] [2] [3] [4] [5] [6] [7] (3 swaps with 4)

[1 2 3 4 5 6 7] (exit from all recursive calls)
```

As explained in your textbook, during a recursive partition split, there are two indices, *low* and *high*, and they sweep through the array until meeting, at which point the pivot is swapped with whatever value happens to be at $low = high$. Unfortunately, the numbers we are using might confuse you into conflating the value of the pivot with the final index it is sorted to—they are not the same thing. I'm not reproducing the textbook here, so look up Quick-Sort if you need to convince yourself.

(Question 3) Suppose it was possible to implement a Priority Queue ADT so that both Insert and RemoveMin ran in $\mathcal{O}(\log \log n)$ steps when the Queue was filled with n elements. Then we could take some array A with n elements and insert each element of A into the priority queue. Then repeatedly remove the minimum element from the priority queue until all the elements of A are placed back in order (just like you did for the programming part of this assignment!). The number of steps to sort would be $\mathcal{O}(n \log \log n)$, which is impossible

because the fastest runtime for a sorting algorithm is $\mathcal{O}(n \log n)$.

(Question 4) To sort S in linear time we must find a way to view its n integers that somehow depends on n. One way to change the representation of an integer is to change its base, so consider base n instead of base 10. Since the upper bound on any integer x in S is $n^2 - 1$, we have that

$$b = \log_n(x) < log_n(n^2) = 2$$

and note that for values of b larger than 1, we only need at most 2 base-n digits to represent the corresponding integer x. Therefore, a Radix-Sort with bins labelled from 0 to $n-1$ needs to only check through all the elements of S twice.

The first pass of the Radix-Sort, say for some element $x \in S$, decides to place x at the end of the bin that matches the first-digit base n representation of x, i.e. the bin i where $i \equiv x \mod n$. If we are sorting from smallest to largest, then the second pass of the Radix-Sort will iterate through bins from the previous pass in the order $0, 1, 2, \ldots, n-1$ and place an element x into one of a new set of n bins, again labelled 0 to $n-1$. This time an element is placed at the end of the bin that matches the second-digit base n representation of x, i.e. i where $i \equiv \frac{x}{n} \mod n$. Finally, the elements in the second-pass bins are placed in order from bin 0 to bin $n-1$, and the final result is an array of sorted elements of S.

All loops in the above description iterate n times and there are a constant number of loops, so altogether, an algorithm based on this strategy takes $\mathcal{O}(n)$ steps.

comments

There is at least one issue that would be nice to see resolved. Many students in their pseudocode (which you did not have to write: read the question) decided to re-assign each original array value to hold a pair of values (the quotient and the remainder when divided by n). Among all these students, only one went on to actually try to implement Radix Sort with this data structure choice, and this made for a more complex implementation than they probably at first imagined. Good for them for tackling something complex, but please, everyone, in the future—a rule of thumb I use is to just write enough so that *someone else* could easily code it, and no more. Beware awkward data structure choices that try to mush steps together for you, since it is pretty clear for most people that this choice cornered them into a situation where they did not want to continue explaining.

Many students tried to avoid having to describe more details, so they simply called Radix Sort as if it were already implemented in the way they wanted. Part of the issue with this is that there are different implementations of Radix Sort, and the students that avoided describing any implementation of Radix Sort will probably have a difficult time describing how it works in the future. This is where most of the marks came off if you chose to do this.

Some wanted to convert the integers in S to strings first, but this just made their descriptions of code more complicated than it needed to be and led to bugs. For example, if base ten numbers were used for the Radix Sort, and the numbers converted to strings were not padded with zeros, then unfortunately, lexicographic ordering is *not* what is wanted

to sort the array properly, since this will lead to mixed significant digits sorted into the same buckets. Otherwise, the numbers to be sorted should each have the exact same number of digits.

Some wanted to use a recursion implementation, but it gave no advantage in runtime over the iterative approach, and it is arguably more difficult to expose the runtime issues using recursive techniques, so a few convinced themselves they had linear runtime, when in fact they did not.

Feedback doesn't typically feel good, and I don't mean to belittle anyone's efforts. It is awesome to see the variety that everyone came up with, you are all a truly creative class, and I hope everyone continues to improve as well as continue to be curious about the topics in the course. I think the students that make many mistakes trying to arrive at a respectable answer and are willing to adjust based on feedback can surpass even their own goals for improvement.

(HeapSort) There was only a handful of students that did not get linearithmic runtime for their code. I'm not sure any solution written up here will help much, since the code is basically written in the textbook, and at some point a person that wants to learn needs to attempt implementing it on their own, but I'll give a couple basic pieces of advice:

- try writing on paper in pseudocode first and test your pseudocode on small examples
- if you cannot run pseudocode on small examples, then you haven't described steps in enough detail
- other than for basic indices, use variable and method names that make sense, not just random single letters
- break up your code into methods
- consider the runtime of each of your methods individually
- make sure your code compiles before submitting it
- test your code on some small example files you make up yourself
- test your code on the sample input files given under the Resources tab of the course page on conneX

I hope everyone is starting to see the connection between being able to write code that not only works, but works fast, and the ability to communicate in concise terms a description of an algorithm.