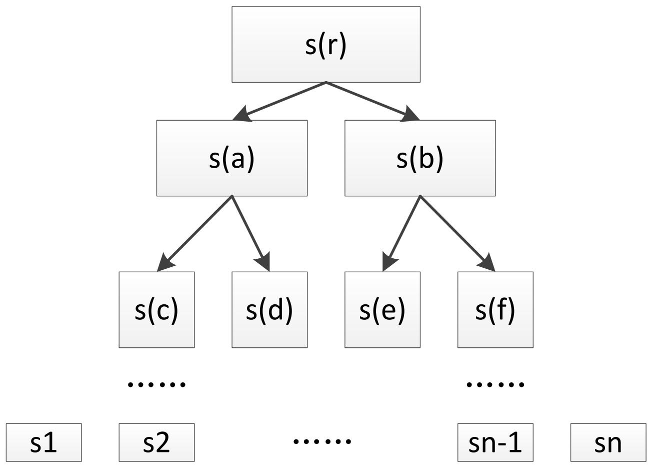
R-4.9

Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at rank (index)$\lfloor n/2 \rfloor$, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted?



We select the element in the middle of this sequence, which means we divide every sequence into two parts which have the same number of elements, so the above tree will have a $\log n$ depth. At every level, although there is no need to change the position of any element, we must still traverse every sequence and the total time is $O(n)$ at every level. As a result, the total time for this algorithm is $O(n \log n)$.

R-4.14

Which, if any, of the algorithms bubble-sort, heap-sort, merge-sort, and quick-sort are stable?

In bubble-sort, we just swap the neighboring two elements. If they are equal, we don't need to swap them. So bubble-sort is stable.

Heap-sort: we can assume that there are two equal elements at the same level in the tree, and the forward one is in the left subtree and another one is in the right subtree. After updating, the left one may have no change, and another one maybe swap with its parent node, which result in the back one become into the forward one. So heap-sort is not stable.

In merge-sort, at the bottom, if there is only one element, we don't need to do anything. If there are two elements and they are equal, we also don't need to swap them. And if we merge two sequence and there are two equal elements, we can save the forward one first, and then save the next one, which don't change the order. So merge-sort is stable.

In quick-sort, we can assume that there is a list: 3, 3, 1, and if we choose the second 3 as pivot, we will swap the first 3 and 1, which causes disorder. So quick-sort is not stable.

In conclusion, bubble-sort and merge-sort are stable.

R-4.16

Is the bucket-sort algorithm in-place? Why or why not?

No. Bucket-sort must use additional storage to save the bucket, which means that if there are n elements and m buckets, there will be $O(n+m)$ space.

C-4.1

Show how to implement method equals(B) on a set A, which tests whether A=B in $O(|A|+|B|)$ time by means of a variation of the generic merge algorithm assuming A and B are implemented with sorted sequences.

We can add a new method into merge-sort. If the element in A is equal to the element in B, we will add this element to a new sequence C. After that, we will check C to see that if it has the same size as B. Like merge-sort, this method will take $O(|A|+|B|)$ time.

C-4.4

Let A be a collection of objects. Describe an efficient method for converting A into a set. That is, remove all duplicates from A. What is the running time of this method?

First, we can use merge-sort to sort all the objects in A, which takes $O(n\log n)$ time. Then, we can delete all the duplicates in A, which takes $O(n)$ time. As a result, the total time is $O(n\log n)$.

C-4.9

Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your apporoach to three colors?

For two colors, we can assume that we have two indexes, one starts from the beginning of the array to the end, named blue index, and the other starts from the end to the beginning, named red index. If the blue index is at the blue elements or the red index is at the red elements, they will continue to move. If the blue index is at the red elements and the red index is at the blue elements, we will swap these two elements. Continue to do this until they meet. After that, we can get the ordered sequence.

For three colors, we can follow this algorithm twice. We assume that there are three colors: blue, red and green. At the first time, we can assume the red and green as the same color. After that, we can move all the blue elements to beginning and other elements to end. At the second time, we can sort the rest red and green elements and ignore the blue elements. After that, we can get the ordered sequence.