Jane Downer
CS 430

Homework 3

1. (4 pts) Stack depth for QUICKSORT - The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After the call to PARTITION, the left sub array is recursively sorted and then the right sub array is recursively sorted. The second recursive call in QUICKSORT is not really necessary; it can be avoided by using an iterative control structure. Good compilers provide this technique, called tail recursion. Consider the following version of quick sort, which simulates tail recursion.

```
QUICKSORT'(A, p, r)
1  while p < r
2        do    // Partition and sort left sub array.
3              q = PARTITION (A, p, r)
4              QUICKSORT'(A, p, q - 1)
5              p = q + 1
```

1a) (2pts) Argue that QUICKSORT '(A, 1, length [A]) correctly sorts the array A.

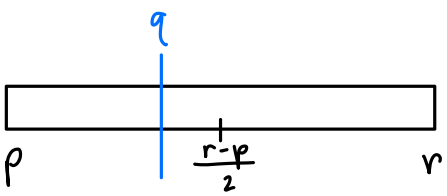
In the Partition algorithm, q is the index that divides the array into two subarrays — one with elements greater than A[r], and one with elements less than or equal to A[r]. By calling Quicksort on (A, p, q-1), the "less-than" subarray gets sorted. By then setting p equal to q + 1, the next iteration of the while loop recursively sorts the "greater than" subarray, which begins at the newly defined p, at index i + 1. Therefore, after partitioning A into two subarrays — one with values greater than those in the other — the two subarrays are recursively sorted. This yields a fully sorted array.

1b) (1pt) Compilers usually execute recursive procedures by using a stack that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. When a procedure is invoked, its information is pushed onto the stack; when it terminates, its information is popped. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires O(1) stack space. The stack depth is the maximum amount of stack space used at any time during a computation. Describe a scenario in which the stack depth of QUICKSORT' is $\Theta(n)$ on an n-element input array.

The worst-case time complexity for Quicksort is generally $\Theta(n^2)$. However, in the case that the original array A is already sorted, then the right subarray will always be of size zero (and therefore, never needs to be sorted) – i.e., $q = r$ with every partition. In this case, Quicksort will be recursively called $n-1$ times before it is fully sorted.

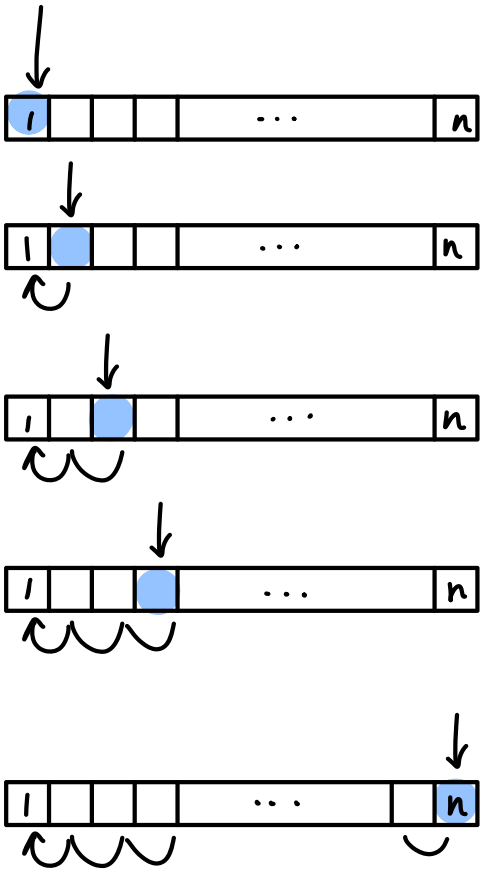
1c) (1pt) Modify the code for QUICKSORT' so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

```
Quicksort"(A, p, r)
    While p < r:
        q = Partition(A, p, r)
        if q < ⌊(r-p)/2⌋:
            Quicksort"(A, p, q-1)
            p = q + 1
        else:
            Quicksort"(A, q + 1, r)
            r = q - 1
```



if q is between p and $\frac{p-q}{2}$, call Quicksort" on the smaller (left) subarray. If you always choose the smaller subarray to recurse on, then the array size is reduced by at least half with each recursion.

2. (2 pts) What is the biggest and smallest possible depth of a leaf in a decision tree for a comparison sort? Explain why in detail.



Index #	# comparisons: best-case	# comparisons: worst-case
1	0	0
2	1	1
3	1	2
4	1	3
⋮	⋮	⋮
n	1	n-1

Each comparison corresponds to another level of the decision tree.

In the best case scenario, the new element will only have to be compared to the element immediately before it in the array (if the new element ≥ prior element, it can stay where it is). So in the best-case scenario, there will be n-1 comparisons (the first element has nothing to be compared against). In the worst case, each new element at position i will need to be compared against all $i-1$ already-sorted elements. Therefore, the worst-case # comparisons is $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$.

Smallest possible depth of leaf: $n-1$

Biggest possible depth of leaf: $\frac{n(n-1)}{2}$

3. (1pt) If the input array is {-29864, 9, 1, 0, 89926187}. Is counting sort a best choice to sort the array? Why?

No. Counting sort assumes that each integer in the array is greater than or equal to zero.

4. (3pts) Randomized median finding algorithm to make your pseudo code generic enough, so that it can answer order statistics for any kth biggest element from an n-element array. For e.g. k = n/2 gives the median and k=1 gives the max. (You may like to invoke some algorithms from lectures.)

```
Random-select (A, k)

    pivot = random element in A
    initialize empty arrays A1, A2
    n = length(A)

    for i from 1 to n:
        if A[i] < pivot:
            append A[i] to A1
        else if A[i] > pivot:
            append A[i] to A2
        else
            pass
    if k ≥ n - length(A2)
        return Random-select (A1, k)
    else if k ≤ length(A2)
        return Random-select (A2, k)
    else:
        return pivot
```