Topic 10A: Quicksort -- Solutions

Note: for simplicity we will use the nonrandomized version of Quicksort in the following problems. However, the results also apply to the randomized version.

1. Non-unique keys (warmup problem)

What is the running time of Quicksort (randomized or non-randomized) when all elements of the input array A have the same value? Justify your conclusion with reference to relevant lines of code.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Θ(n^2): Line 4 test of Partition includes equality, so will always succeed and execute lines 5 and 6. Everything in the partition will be placed to the left of (smaller side of) the pivot by lines 5 and 6. Thus the pivot position returned by line 8 will be the last element of the array.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Back in Quicksort, the recursion in line 3 will be on n-1 items (just removing the pivot) until the pivot is all that is left, and the recursion in line 4 will always be on 0 items (constant time each level of recursion).

Thus the sizes of the original call plus the recursive calls in line 3 will be on n + n-1 + n-2 + … + 1 + 0 = O(n^2) items, just as in the worst case analyzed in the lecture notes. In terms of recurrence relations, T(n) for a call to Quicksort is O(n) to partition in line 2, T(n-1) for recursion in line 3, and c or O(1) for recursion in line 4 (as well as the test of line 1). Putting it together and dropping the constant, the recurrence is T(n) = T(n-1) + O(n), which we have previously analyzed to be O(n^2)

2. Finding the *i*th smallest element with Partition

Use Quicksort's Partition procedure to write an algorithm for finding the i^{th} smallest element of an unsorted array in O(n) expected time (without sorting it). *Hint: what does the returned value of Partition tell you about the rank ordering of the pivot?*

- Describe the strategy in English
- Then write pseudocode for an algorithm.
- The solution is simpler if you assume arrays of A[1..n], so in the initial call p=1 and r=n. Solve the simpler version first.

- Then if you have time generalize for calls to any region of an array, e.g., the 5th largest element in A[37,1044].
- You do not need to prove O(n) expected time, but don't do something that takes longer such as sorting

a. Strategy described in English:

- Call partition on A(p,r) and get the returned pivot position q.
- If q = i and the original call was to A[1,n], we are done: return A[q]. The q^{th} position must be the i^{th} largest element because all smaller elements have been placed to the left of A[q] and there are exactly i-1 smaller elements.
- If q > i then the i^{th} largest element is in the lower half of the partitioned array: recursively find the ith smallest item in A[p,q-1].
- If q < i then the i^{th} largest element is in the upper half of the partitioned array, so we recursively find q = i in A[q+1,r]. This will work if the original call was to A[1,n], because we are assuming i is relative to the initial set of n elements, not the recursive set, and is therefore the same as the array index.
- However, if the lowest index was not originally 1, we need to adjust to convert between positional and absolute indexing, as well as to take into account the number of smaller elements discarded.

Conceptually, we are partially sorting the array only up to the point of knowing what would have ended up at the i^{th} position in a full sort.

b. Pseudocode, assuming initial call p=1:

```
Find-ith-element(A, p, r, i)
// Find i-th element of array A.
// we might insert error checking first: p <= i <= r?
1 q = PARTITION(A, p, r)
2 if q == i
3    return A[q]
4 else if q > i // search in lower partition
5    return Find-ith-element(A, p, q-1, i)
6 else          // search in upper partition
7    return Find-ith-element(A, q+1, r, i)
```

c. Pseudocode allowing any range of the array:

```
Find-ith-element(A, p, r, i)
// Find i-th element of subarray A[p,...,r]
1 q = PARTITION(A, p, r)
```

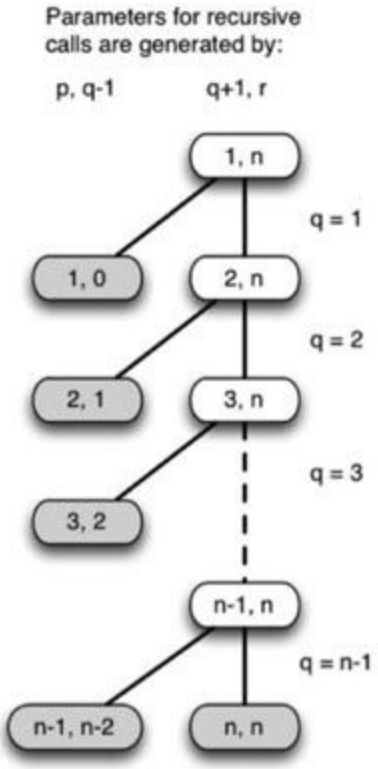
```
2 k = q - p + 1  // k is ordinal position of q in A[p,r]
2 if k == i      // is it the ith position?
3    return A[q] // then return the item at this index
4 else if k > i  // search in lower partition; i still holds
5    return Find-ith-element(A, p, q-1, i)
6 else           // adjust for discarded; search in upper
7    return Find-ith-element(A, q+1, r, i-k) // k were discarded
```

3. Challenge Problem: Calls to Partition in Worst and Best Case

We have seen that Quicksort requires $\Theta(n^2)$ comparisons in the worst case when the pivot is always chosen to be the smallest or largest element, and $\Theta(n \lg n)$ comparisons in the best case when the pivot is always the median key (or expected case for the randomized version). Here we examine the number of calls to Partition made in each of these cases.
In the following, do NOT assume that the number of calls to partition is the same as the overall runtime of quicksort. That is the point!

(a) Asymptotically, how many calls to Partition are made in the **worst case runtime** (when the pivot is always chosen to be the smallest or largest element)? Answer with Θ. Justify your conclusion, for example by using recurrence relations or reasoning about the recursion tree.

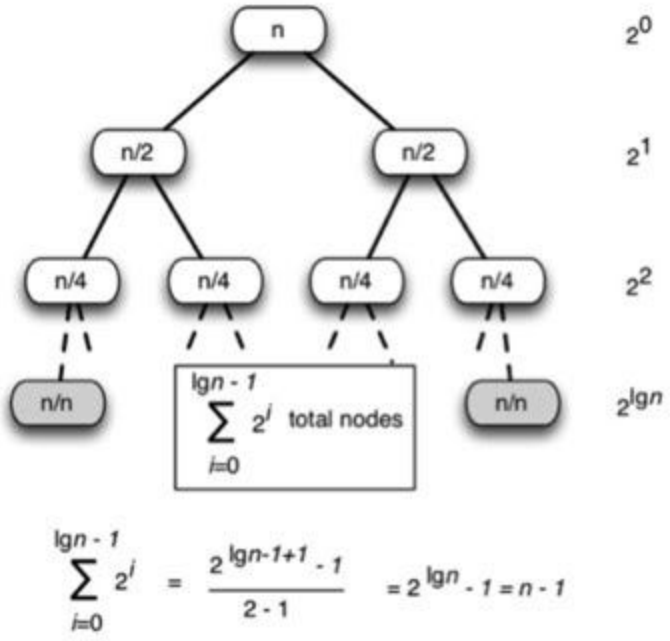
T(n) = T(n-1) + c = cn, so $\Theta(n)$ calls are made.



(b) Asymptotically, how many calls to Partition are made in the **best case runtime** (when the pivot is always the median key)? Answer with Θ. Justify your conclusion, for example by using recurrence relations or reasoning about the recursion tree.

$T(n) = T(n/2) + T((n/2)-1) + c \leq 2T(n/2) + c$
By the master theorem, a=2, b=2, $f=c=cn^0$.
$\log_2 2 = 1$ so subtract epsilon = 1 to get n^0: Case 1. $\Theta(n)$

The recursion tree:



$$\sum_{i=0}^{\lg n - 1} 2^i = \frac{2^{\lg n - 1 + 1} - 1}{2 - 1} = 2^{\lg n} - 1 = n - 1$$

A MUCH SIMPLER ARGUMENT: No matter how the recursion breaks down, it must proceed until every item has been 'picked off' as a pivot. So, n items must be picked off in n calls to Partition.