Solutions -- Class 0224, Topic 10A: Quicksort

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. Nonunique 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.

$\Theta(n^2)$: Line 4 test includes equality, so will always succeed. Everything in the partition will be placed to the left of (smaller side of) the pivot. Thus the returned pivot will be the last element of the array, so the recursion will be on n-1 items, leading to cost of n + n-1 + n-2 + … + 1 = $O(n^2)$ just as in the worst case analyzed in the lecture notes.

```
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
```

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) time (without sorting). *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 and then generalize for calls to any region of an array, e.g., A[37,1044].

a. Strategy described in English (simpler version):

- Call partition on A(p,r) and get the returned pivot position q.
- If q = i, we are done: return A[q]. It must be the ith 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 ith 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 ith largest element is in the upper half of the partitioned array, so we we recurse on A[q+1,p].
- This will work if the original call was to A[1,n]. However, if the lowest index was not originally 1, we need to adjust i to find the (p+i-1)th element in the lower partition or the (i-q)th element in the upper partition.

b. Pseudocode:

A good student response for the simpler case (Table 3):

```
Find-ith-element(A, p, r, i)
1 j = PARTITION(A, p, r)
2 if j == i
3    return A[j]
4 else if j > i
5    Find-ith-element(A, p, j-1, i)
6 else
7    Find-ith-element(A, j+1, r, i)
```

For the more general case: (Different procedure name)

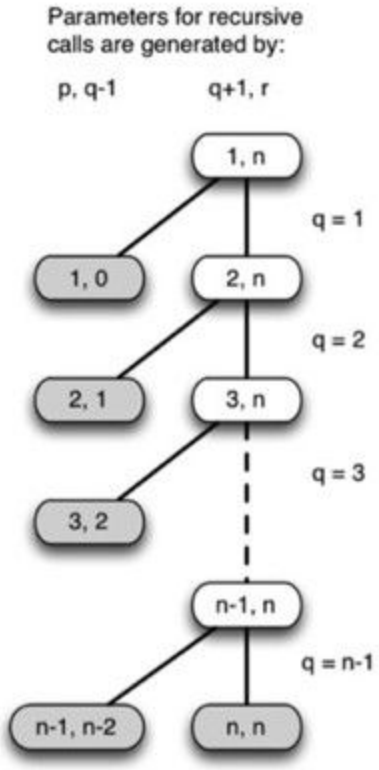
```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2       return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k            // the pivot value is the answer
6       return A[q]
7  elseif i < k
8       return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

2. (Extra Credit!) 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.

(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 $\Theta(f(n))$, where your job is to identify f. 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.

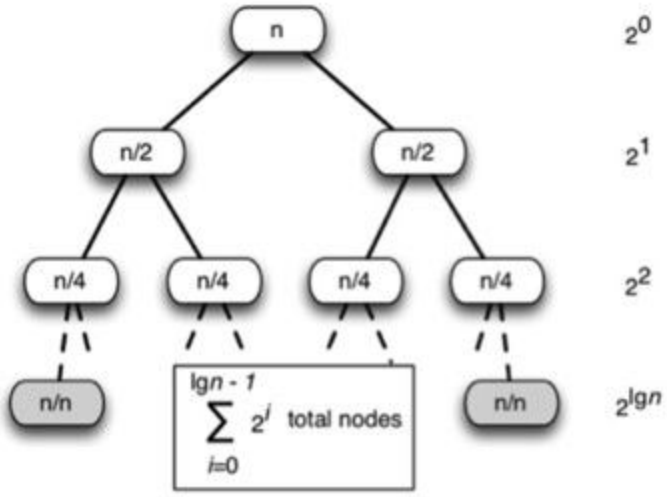


(b) Asymptotically, how many calls to Partition are made in the **best case runtime** (when the pivot is always the median key)? Answer with $\Theta(f(n))$, where your job is to identify f. Justify your conclusion, for example by using recurrence relations or reasoning about the recursion tree.

$T(n) = T(n/2) + T((n/2)-1) + c \leq 2T(n/2) + c$

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. $\Theta(n)$



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