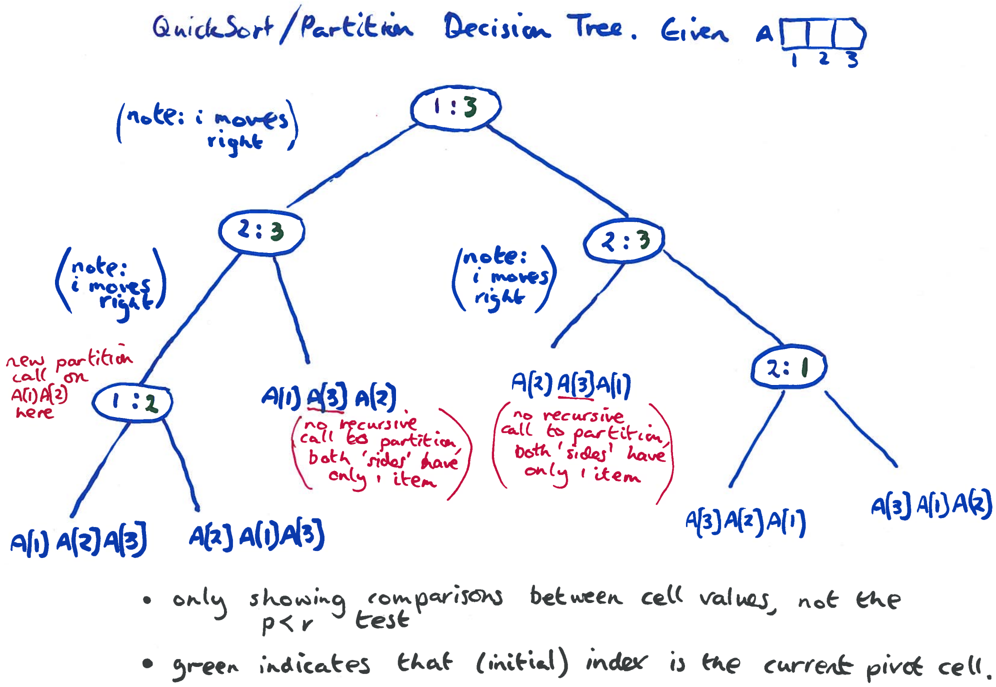
Algorithms and Data Structures 2020/21
Week 6 solutions

1. Draw the decision tree (under the assumption of all-distinct inputs) QUICKSORT for $n = 3$.

   Answer:



2. What is the smallest possible depth of a leaf in a decision tree for a sorting algorithm?

   Answer: The shortest possible depth is $n - 1$. To see this, observe that if we have a root-leaf path (say $p_{r \to \ell}$) with k comparisons, we cannot be sure that the permutation $\pi(\ell)$ at the leaf ℓ is the correct one.

   Proof: To see this consider a graph of n nodes, each node i representing $A[i]$. Draw a (directed) edge from i to j if we compare $A[i]$ with $A[j]$ on the path from root to ℓ. Note that for $k < n-1$, this graph on $\{1, \ldots, n\}$ will *not* be connected. Hence we have two components C_1 and C_2 and we know nothing about the relative order of array elements indexed by C_1 against elements indexed by C_2. Therefore there cannot be a single permutation π that sorts all inputs passing these k tests - so $\pi(\ell)$ is wrong for some arrays which lead to leaf ℓ.

3. **Intuition:** In doing this kind of question, you should always think of choosing comparisons which will carry most information - i.e., the result of the comparison (< or >) will split our current possible permutations as close to half as possible.

   (a) Let the numbers to be sorted be x, y, z, w. Here is the algorithm.

      1. Compare (x, y).
      2. Compare (z, w).
      3. Compare (winner(1), winner (2)).
      4. Compare (loser(1), loser(2)).
      5. Compare (loser(3), winner(4)).

      Output: winner(3), winner(5), loser(5), loser(4).

   (b) Assume wlog that all four inputs are distinct.
   There are $4! = 24$ different permutations of 4 inputs, all are possible outputs. We model this as usual as a binary decision tree with at least 24 leaves (to cover each permutation).

   The length of a root-leaf path in the decision tree corresponds to the number of comparisons done in sorting that particular permutation.

   Suppose that we have a binary tree with height ℓ. Then this tree has at most 2^ℓ leaves. To solve our 4-sort problem, we require $2^\ell \geq 24$, hence we need $\ell \geq \lg 24 > 4$ (to show $\lg 24 > 4$ without an extra calculation, just observe $\lg 16 = 4$).

   Since path-length corresponds to no-of-comparisons, we need a tree which for some inputs does more than 4 comparisons.

4. For this question please follow the exact version of PARTITION from the slides - if you use a different version, you may get not get non-stability (or may get an easier example).

   Example: the array $6_a, 4_a, 6_b, 4_b$.

   At the top-level, 4_b is the pivot.

   Walking from the left, the first $A[j]$ selected for 'swapping' (as <= 4) is $j = 2$ with $A[2] = 4_a$.
   i has been sitting to the left of the array (it did not move during $j = 1$) so it advances to $i \leftarrow 1$.
   $A[1] = 6_a$ and $A[2] = 4_a$ get swapped, to give the new order $4_a, 6_a, 6_b, 4_b$. So far so good.
   Now $j = 3$ has $A[3] = 6_b$ so nothing is done; this is the last index we must consider for j so we exit the loop.
   After exiting loop, $i = 1$, so we swaps $A[2] = 6_a$ and $A[4] = 4_b$ and return the array

$4_a, 4_b, 6_b, 6_a$ with $i + 1 = 2$ as the split point.
So next we have two calls with an 1-element array 4_a, and a 2-element array $6_b, 6_a$.
This version of Partition will end up swapping 6_b with itself on the second call.
So the final output will be $4_a, 4_b, 6_b, 6_a$.

hence not stable.

Your students might find a simpler example.

5. **Intuition:** A good way to first get a feel for this question is to consider the no-of-pivots corresponding to the Best-case (equal splits all the way) and worst-case (array sorted) for Running Time of *non-random quicksort*. In fact these turn out to be best-and-worst cases for pivots also (again in the in non-random quicksort case, which is our question).

   Lemma: We can show that (no matter how we choose the pivots), we use *between* $\lceil (n-1)/2 \rceil$ *and* $\max\{0, n-1\}$ pivots to sort an array of size n (the reason the max is there is to take care of $n = 0$).

   Proof is by induction.

   $n = 1$. We have 0 pivots, with 0 equal to $\lceil (n-1)/2 \rceil$ and $\max\{0, n-1\}$. So OK here.

   $n > 1$. Suppose true for all $k < n$ (I.H.), now we show for n.
   Suppose we split into two partitions of size i and $n - i - 1$, and assume wlog that i is smallest, possibly zero (this guarantees $n - i - 1$ is not zero). Then $piv(n) = piv(i) + 1 + piv(n - i - 1)$.

   For lower bound we know $piv(i) \geq \lceil (i-1)/2 \rceil$, and $piv(n-i-1) \geq \lceil (n-i-2)/2 \rceil$. So
$$piv(n) \geq 1 + \lceil (i-1)/2 \rceil + \lceil (n-i-2)/2 \rceil.$$

   Best way of finishing this is to do case analysis on odd/evenness of n and i. In all 4 cases you will get a lower bound of $\lceil (n-1)/2 \rceil$ (which is only met for n odd, i odd).

   For upper bound, we observe that
$$piv(n) \leq \ 1 + \max\{0, i-1\} + (n - i - 2) \ \leq (n-1).$$

   (we only have one max because we know the rhs has $n - i - 1 > 0$)

   Worst case: Take an array in sorted order $1, 2, 3, \ldots, n$.
   At each step, we will split into a subarray of length $n-1$, then the pivot, and an empty subarray. Hence we use $n - 1$ pivots.

   Best case: take an array of length $2^k - 1$ for some k. The array is arranged so that the final element is 2^{k-1} and such that all elements less than 2^{k-1} are in the first 2^{k-1}

positions, and all elements greater than this are in the last 2^{k-1} positions (also this is true recursively). Then, the first pivot splits the array exactly into two parts of equal size $2^{k-1} - 1$, with the pivot in the middle. Applied recursively, this means we use $2^{k-1} - 1 = \lceil (n-1)/2 \rceil$ calls.

6. Show how to sort n integers in the range $\{1, \ldots, n^2\}$ in $O(n)$ time.

   Answer: This is a simple application of the Radix Sort Theorem of lecture 9. The theorem states that if we have numbers represented by b bits, we can sort in time $\Theta(n\lceil b/lg(n) \rceil)$ time. When our numbers are the integers between 1 and n^2, the numbers of bits needed for the representation is $b = \lceil 2\lg(n) \rceil$.

   Then $\lceil b/lg(n) \rceil \leq 4$. So Radix sort (with bits taken in $\lceil \lg(n) \rceil$ size blocks) runs in $\Theta(4n) = \Theta(n)$.