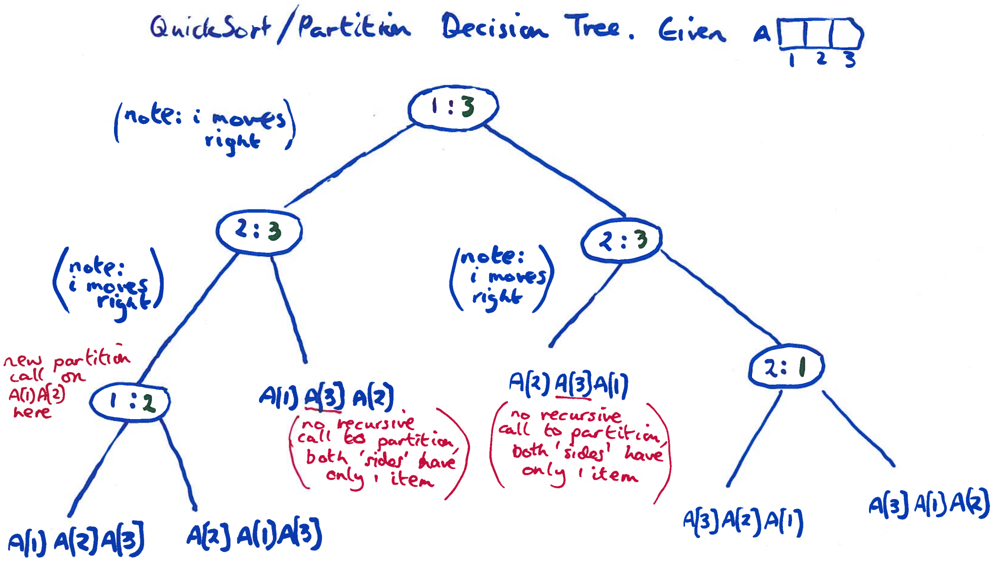
Algorithms and Data Structures 2016
Week 5 solutions (Tues 9th - Fri 12th February)

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

   answer: (of course you should also *explain* why to students)



- only showing comparisons between cell values, not the p<r test
- green indicates that (initial) index is the current pivot cell.

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 - ie 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 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 calc, 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. Show that there is no comparison sort whose running time linear for at least half of $n!$ inputs of length n. What about a fraction of $1/n$ of the inputs of length n? What about a fraction of $1/2^n$?

   Answer:

   1st case: First case (1/2 of all $n!$ inputs) is not much different from what's in the notes. Have a try, and if stuck, ask me.

   2nd case: Take the second case - we would like an algorithm S which sorts in cn time, for some $c > 0$, on at least a $1/n$ fraction of all $n!$ input permutations.

   For 2nd case, I will prove by contradiction. So suppose there *is* some sorting algorithm S which sorts in cn time every permutation in some set $P \subset S_n$, such that $|P| \geq n!/n$. Consider the decision tree of S, and restrict it to the leaves which are labelled by elements of P - by this, I mean that we remove every leaf not in P, every internal node which has no elements of P below it, and every edge which has no elements of P below it. Let h_P denote the height of the restricted tree T_P. We will now derive a lower bound on h_P.

Observe that as it stands, the restricted tree T_P on P is not a binary tree, as there may be many vertices which have just one child. We take care of this by contracting any degree-2 vertices (except the root), ie by identifying the edges passing through such a vertex. Finally, if the root node has degree 1 in T_P , we will delete it and its outgoing edge. These contractions and prunings create a binary tree T_P' (though *not* necessarily a complete or near-complete binary tree). The point to note is that T_P' will have height h_P' such that $h_P' \leq h_P$.

Now we lower bound h_P'. *By* $h_P' \leq h_P$ *this will automatically give us the exact same lower bound on* h_P. We have a binary tree of height h_P', hence it can have at most $2^{h_P'}$ leaves. Since $|P| \geq n!/n$, it must contain at least $n!/n$ leaves. Hence we require $2^{h_P'} \geq n!/n$, and by $h_P \geq h_P'$, we certainly require $2^{h_P} \geq n!/n = (n-1)!$. This is equivalent to $h_P \geq \lg((n-1)!)$. Then by $(n-1)! \geq (n-1)^{(n-1)/2}$, we require $h_P \geq \lg((n-1)^{(n-1)/2}) = ((n-1)/2)\lg(n-1)$.

note: I think it is clear that $(n-1)/2\lg(n-1)$ is non-linear (ie is *not* $O(n)$). To prove rigorously, imagine I am comparing against cn, for *any* given c (I'm doing this because in the definition of $O(\cdot)$, we have the power to choose c). I will now show that *regardless of which* $c > 0$ *we are working with*, that for sufficiently large n, we have $((n-1)/2)\lg(n-1) > cn$. Take $n_0 = 2^{2c+2} + 1$. Then for all $n > n_0$,

$$\begin{aligned}
((n-1)/2)\lg(n-1) &\geq ((n-1)/2)\lg(n_0-1) \\
&= \frac{n-1}{2}(2c+2) \\
&\geq c(n-1) + (2^{2c+2}) \\
&= cn + (2^{2c+1} - c) \\
&> cn.
\end{aligned}$$

Therefore, regardless of which $c > 0$ we chose, our corresponding definition of n_0 gives $h_P \geq h_P' \geq ((n-1)/2)\lg(n-1) > cn$ for all $n \geq n_0$. Hence we have a contradiction.

So no sorting algorithm can sort $1/n$th of its inputs in linear time.

3rd case:

For the case when we ask about $1/2^n$, the answer is *still* no (no sorting alg takes just linear time on this fraction).

It's a bit harder though. When we are working with h_P', our assumption changes, and now we have the condition

$$h_P \geq h_P' \geq \lg(n!/2^n),$$

since the size of the set P is now only required to be a 2^n fraction. We then apply our usual formula $n! \geq n^{n/2}$. Therefore it must be the case that

$$h_P \geq h'_P \geq \lg(n^{n/2}/2^n).$$

Now observe that $2^n = 4^{n/2}$, so the condition is exactly equivalent to asking for

$$\begin{aligned} h_P \geq h'_P &\geq \lg((\frac{n}{4})^{n/2}). \\ &= \frac{n}{2}\lg(\frac{n}{4}) \\ &= \frac{n}{2}(\lg(n) - \lg(4)) \\ &= \frac{n}{2}(\lg(n) - 2) \end{aligned}$$

As in the solution to the *2nd case* I'd say it was obvious that this is non-linear...

You might or might-not want to do a rigorous proof of non-linearity in relation to some arbitrary cn. An interesting thing is that actually the n_0 of *2nd case* will work here. This is just luck, maybe helped a bit by the fact that although $\frac{n}{2}(\lg(n) - 2)$ is (a bit) smaller than the value for *2nd case*, it is nevertheless neater (in terms of working with lg etc).

5. Tutor, 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).

   I tried to achieve this with just three items but could not see how ...

   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.

6. **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*, I don't have time to write details). 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 - I'm not going to prove this, but check $n = 15$ as an example.

7. 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 $\Theta(4n) = \Theta(n)$.

Mary Cryan