CS2040S Tutorial 3

AY 24/25 Sem 2 — github/omgeta

Q1. (a.) Array where all values are the same would give $O(n^2)$ time complexity.

(b.) No; we can design a stable QuickSort using $O(n)$ extra memory for an auxiliary array storing the initial ordering of all values. We perform the same swaps in the main array and auxiliary array, and then resolve total ordering of elements from the auxiliary array.

(c.) (i.) $O(n)$

(ii.) $O(nk)$

Q2. (a.) Use an auxiliary array to count if elements have been seen before. If durng traversal, the array has already been seen, return false. $O(n)$

(b.) Same approach as in (a) but also add any elements not seen before to B. If it has been seen before just pass. $O(n)$

(c.) Same approach as in (b) but traverse through both A and B, not resetting the auxiliary array between arrays. $O(n+m)$

(d.) Sort A, creating pointers at the start and end. If the current sum is less than desired, increment the start pointer to increase the sum; else, decrease the end pointer to decrease the sum. Terminate when the sum is found, or when the pointers meet indicating no solution. $O(n \log n)$

Q3. Modified QuickSort Solution, $O(n \log n)$:

1. Choose a random pair of shoes from the pile, and partition the kids into smaller or larger feet groups as well as the exact match child X
2. Partition the pile of shoes about child X, to get two piles of shoes smaller or larger
3. Recurse on the problem pairing up the smaller feet kids with smaller shoes, and the larger feet kids with larger shoes

Q4. (a.) Sort the pivots and for every element not yet searched, use binary search to find where to place the element.

(b.) $O(k \log k)$ (sort pivots) $+O(n \log k)$ (binary search n elements in k pivots)
$= O(n \log k), n \geq k$

(c.) $T(n) = kT(\frac{n}{k}) + O(n \log k)$

(d.) $T(n) = c(n \log k) + kT(\frac{n}{k})$
$= c(n \log k) + k \cdot c(n \log k) + k^2 T(\frac{n}{k^2})$
$= c(n \log k) + k \cdot c(\frac{n}{k} \log k) + k^2 \cdot c(\frac{n}{k^2} \log k) + \cdots + k^{\log_k n} \cdot c(\frac{n}{k^{\log_k n}} \log k)$
$= c(n \log k) \cdot \log_k n$
$= O(n \log k \log_k n) = O(n \log n)$

Q5. (a.) Use two pointers, one traversing from the front and one from the back. If the front hits a 1, stop and wait for the back pointer to hit a 0 then swap, or vice versa. Terminate when both pointers meet. Time complexity $O(n)$, in-place but unstable.

(b.) Counting sort: traverse the array and increment the number of times the element is seen in the auxiliary array. Then based on the frequency count, fill in the values of the array. Space complexity $O(M)$. Time complexity $O(n+M)$

(c.) We can each of the 64 levels once, so time complexity is $O(64n) = O(n)$. It is faster when $64n < n \log n \implies 64 < \log n \implies n > 2^{64}$

(d.) We could use the counting sort in (b) but with 8-bits at a time. This would however take up more memory because we need to create an auxiliary frequency array.