Problem Set 4 Solutions

Calvin Walker

I would like this to be my one allowed late submission

Problem 1:

a)

$$\begin{aligned} T(n) &= T(\frac{n}{2}) + 2n \\ &= T(\frac{n}{4}) + 2n + n \\ &= T(\frac{n}{2^k}) + 2n\Big(\sum_{j=1}^{k} \frac{1}{2^{j-1}}\Big) \end{aligned}$$

Let $k = \log_2(n)$. Then $T(n) = T(1) + 2n\big(\sum_{j=1}^{k} \frac{1}{2^{j-1}}\big) = 1 + 4n$

b)

$$\begin{aligned} T(n) &= 2T(\frac{n}{2}) + 3n^2 \\ &= 4T(\frac{n}{4}) + 3n^2 + 6(\frac{n}{2})^2 \\ &= T(\frac{n}{2^k}) + 3n^2\Big(\sum_{j=1}^{k} \frac{1}{2^{j-1}}\Big) \end{aligned}$$

Let $k = \log_2(n)$. Then $T(n) = nT(1) + 6n^2 = 8n + 6n^2$

c)

$$\begin{aligned} T(n) &= 4T(\frac{n}{2}) + n^2 + 3 \\ &= 16T(\frac{n}{4}) + 4(\frac{n}{2})^2 + n^2 + 4(3) + 3 \\ &= 4^k T(\frac{n}{2^k}) + kn^2 + 4^k - 1 \end{aligned}$$

Let $k = \log_2(n)$. Then $T(n) = n^2T(1) + n^2 log_2(n) + n^2 - 1 = 2n^2 + n^2 \log_2(n) - 1$

Problem 2:

Algorithm: Initialize i_t^A, i_t^B to zero, and j_t^A, j_t^B to the length of $A - 1$ and $B - 1$ respectivley. We will update the pointers at iteration t as follows: let $m_t^A = \lfloor \frac{i_t^A + j_t^A}{2} \rfloor$ and $m_t^B = \lfloor \frac{i_t^B + j_t^B}{2} \rfloor$, and let $k'_t = (m_t^A - i_t^A + 1) + (m_t^B - i_t^B + 1)$ denote the collective number of valid elements prior to positions m_t^A and m_t^B. Now, consider the elements $A[m_t^A]$ and $B[m_t^B]$.

- If $k'_t \leq k_t$:
  - If $A[m_t^A] < B[m_t^B]$, since there are less than k_t valid elements proir to m_t^A and m_t^B, the k'th smallest element cannot be in the first half of A, so set $i_{t+1}^A = m_t^A + 1$, and $k_{t+1} = k_t - (m_A - i_t^A + 1)$ to account for the discarded elements.
  - Otherwise $A[m_t^A] > B[m_t^B]$, so the k'th smallest element cannot be in the first half of B, so set $i_t^B = m_t^B + 1$, and $k_{t+1} = k_t - (m_t^B - i_t^B + 1)$ to account for the discarded elements.
- Otherwise $k'_t > k_t$:
  - If $A[m_t^A] < B[m_t^B]$, since there are collectively more than k valid elements prior to m_t^A and m_t^B, the k'th smallest element cannot be in the second half of B, so set $j_t^B = m_t^B - 1$. As only elements after k were discarded, it is still the k'th element, so $k_{t+1} = k_t$
  - Otherwise $A[m_t^A] > B[m_t^B]$, so similarly the k'th smallest element cannot be in the second half of A, so set $j_t^A = m_t^A - 1$, and leave k_t unchanged.

Runtime: Since half of one of the arrays is discarded at each step, for sorted arrays of length n and m the algorithim has a runtime of $O(\log(n) + \log(m))$ as desired.

Problem 3:

Algorithm: Compute each each sum for all combinations of two elements $x_1 + x_2 \dots x_1 + x_n \dots x_{n-1} + x_n$ and store each sum with their corresponding indexes i, j from the original list of integers in a new array, A. Then, sort A in increacing order of these sums. Initialize a pointer $l = 0$ to the first index of A and r to the last index of A. On each iteration, if $A[r] + A[l] < S$, increment l, if $A[r] + A[l] > S$, decrement r. Otherwise, $A[r] + A[l] = S$, so check if the indicies stored at $A[r], A[l]$ are unique. If so, return these indices, otherwise, increment l and continue the iteration. If $l = r$, terminate, since no four integers at different indecies that sum to S were found.

Runtime: The initial summation of all pairs of elements takes $O(n^2)$ time for an input of n integers. The sorting takes $O(n^2 \log n)$ time, and the comparison of the pairs takes $O(n^2)$ time in the worst case that either l or r traverses the entire array. We can verify that the indices are unique in constant time since we store them alongside the sums in A. So the total runtime is $O(n^2 \log n)$

Explanation: Since we sum all of the possible combinations of elements, and pairs are only discarded from consideration once there is no other pair that could be used to sum to S, the algorithm checks all of the possible combinations of four elements that could sum to S, so the target indecies will always be found, if they exist. In the case that the two pairs sum to S but do not have unique indicies, incrementing l is an arbitrary choice, since changing either of the pointers will suffice in progressing the search.

Problem 4:

Algorithm: Call the algorithm on the two haves of the array: lines 0 through $\lfloor \frac{n}{2} \rfloor - 1$ and lines $\lfloor \frac{n}{2} \rfloor$ through $n - 1$, and merge the results to obtain the full set of visible lines.

Base cases:

- 2 lines: If the lines are of equal slope, then discard the line with a lower intercept, as it's hidden. Otherwise both lines are visible, return them in order of slope.
- 3 lines: Compute the intersection of the two lines with the greatest and least slope. If the other line has a lower y value at the point of intersection, discard it. Otherwise all the lines are visible. Return the valid lines in order of slope.

We can merge two sets of lines A and B sorted by slope into one storted set C as follows: Initialization: take the two smallest lines by slope from A and B and add them in order of slope to C. Iterative step: let c_i be the next element from A or B with the smallest slope. Add it to C. Let x_i be the x-coordinate of the intersection of c_i and c_{i-2}, and x_i' be the x-coordinante of the intersection of c_{i-1} and c_{i-2}

- If $x_i' > x_i$, then c_{i-1} is covered by c_i and c_{i-2}, so we remove c_{i-1} from C, and check the condition again without adding a new line to C
- If $x_i' < x_i$, then the three lines are visible, and we continue to the next iteration.

The merge process terminates once both A and B are empty, and the iteration has concluded.

Runtime: Since finding the intersection of the lines can be found in constant time, and there are at most n deletions during the merge process, the merge process runs in $O(n)$ time, so we have the reccurance relation $T(n) = 2T(\frac{n}{2}) + O(n)$. By the Master theorem this gives a total runtime of $O(n \log n)$

Explanation: The algorithim is essentially a modified version of merge sort, with extra conditions for removing the non visible lines. The base cases are just simple relations between the lines. The merging process leverages the relation between adding another line with greater slope to a set of two visible lines with smaller slope. If it intersects the line with least slope prior to the middle line (lower value of x), we know that the middle line must be concealed. Since a new line could potentially cover multiple lines, we repeatedly check the condition until it no longer holds. While it may seem that this would introduce greater time complexity, these checks are bound by a constant factor of n, since we stop once the next line, c_{i-1} is visible, and lines can only be eliminated once.