Warm-up

Problem 1. The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, where the (i, j) entry of Z is $Z_{ij} = \sum_{k=1}^{n} X_{ik}Y_{kj}$. Suppose that X and Y are divided into four $n/2 \times n/2$ blocks each:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Using this block notation we can express the product of X and Y as follows

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

In this way, one multiplication of $n \times n$ matrices can be expressed in terms of 8 multiplications and 4 additions that involve $n/2 \times n/2$ matrices. Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using this recursive algorithm.

a) Derive the recurrence for $T(n)$. (Assume adding two $k \times k$ matrices takes $O(k^2)$ time.)
b) Solve the recurrence by unrolling it.

Solution 1.

a) Breaking each matrix into four sub-matrices takes $O(n^2)$ time and so does putting together the results from the recursive call. Therefore the recurrence is

$$T(n) = \begin{cases} 8T(n/2) + O(n^2) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

b) $T(n) = c \cdot n^2 + 8 \cdot c \cdot n^2/4 + 64 \cdot c \cdot n^2/16 + ... = O(n^3)$. Therefore, this is no better that the standard algorithm for computing the product of two n by n matrices.

Problem 2. Similar to the integer multiplication algorithm, there are algebraic identities that allow us to express the product of two $n \times n$ matrices in terms of 7 multiplications and $O(1)$ additions involving $n/2 \times n/2$ matrices. Let $T(n)$ be the time complexity of multiplying two $n \times n$ matrices using this recursive algorithm.

a) Derive the recurrence for $T(n)$. (Assume adding two $k \times k$ matrices takes $O(k^2)$ time.)
b) Solve the recurrence by unrolling it.

Solution 2.

a) Breaking each matrix into four sub-matrices takes $O(n^2)$ time and so does putting together the results from the recursive call. Therefore the recurrence is

$$T(n) = \begin{cases} 7T(n/2) + O(n^2) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

b) $T(n) = O(n^{\log_2 7})$, which is slightly better that the standard algorithm for computing the product of two n by n matrices.
The above can be shown using an argument similar to the one we used for $T(n) = 3T(n/2) + O(n)$ during the lecture. When you think of the recursion tree, we get:
Level 1: 1 instance of size n, which takes $c \cdot n^2$ time
Level 2: 7 instances of size $(n/2)^2 = n^2/4$, which take $(7/4) \cdot c \cdot n^2$ time
Level 3: 49 instances of size $(n/4)^2 = n^2/16$, which take $(49/16) \cdot c \cdot n^2$ time
...
Level k: 7^k instances of size $n^2/4^k$, which take $(7/4)^k \cdot c \cdot n^2$ time
Since we halve the size every time, there are $\log n$ levels before we reach the base case. Summing all this up we get

$$\sum_{i=0}^{\log n-1} (7/4)^k \cdot c \cdot n^2 = c \cdot n^2 \cdot \sum_{i=0}^{\log n-1} (7/4)^k.$$

Focussing on everything except the base cases for a moment, when we apply the bound on geometric series with $r = 7/4 > 1$, we get

$$c \cdot n^2 \cdot (7/4)^{\log n}/(7/4 - 1) = (4c/3) \cdot n^2 \cdot (7/4)^{\log n}.$$

Next, we focus on $(7/4)^{\log n}$, which is the same as $n^{\log 7/4}$. Observe that $\log 7/4 = \log 7 - \log 4 = \log 7 - 2$. Plugging this back into the total running time, we get

$$(4c/3) \cdot n^2 \cdot n^{\log 7 - 2} = (4c/3) \cdot n^{\log 7} = O(n^{\log 7}).$$

Finally, we need to consider the number of base cases we have: since the recursion has $\log n$ levels and each level increases the number of subproblems by a factor of 7, this implies we have $7^{\log n} = n^{\log 7}$ base cases (that each take constant time). Adding to this to the $O(n^{\log 7})$ time needed for the rest of the recursion, we conclude that everything together takes $O(n^{\log 7})$ time.

Problem solving

Problem 3. Your friend Alex is very excited because they have discovered a novel algorithm for sorting an array of n numbers. The algorithm makes three recursive calls on arrays of size $\frac{2n}{3}$ and spends only $O(1)$ time per call.

```
1: function NEW-SORT($A$)
2:     if $|A| < 3$ then
3:         Sort $A$ directly
4:     else
5:         NEW-SORT($A[0 : 2n/3]$)
6:         NEW-SORT($A[n/3 : n]$)
7:         NEW-SORT($A[0 : 2n/3]$)
```

Alex thinks their breakthrough sorting algorithm is very fast but has no idea how to analyze its complexity or prove its correctness. Your task is to help Alex:

a) Find the time complexity of NEW-SORT.

b) Prove that the algorithm actually sorts the input array.

Solution 3.

a) Let $T(n)$ be the running time of the algorithm on an array of length n. We observe that the base case is of constant size and thus sorting it takes constant time. For arrays of size at least 3, we recurse three times and each recursion is performed on an array of size $2n/3$. By using the indices during the recursion instead of explicitly copying the array, the divide step takes $O(1)$ time. The algorithm doesn't have a conquer step. Hence, the recursion is

$$T(n) = \begin{cases} 3T(2n/3) + O(1) & \text{for } n \geq 3 \\ O(1) & \text{for } n < 3 \end{cases}$$

which is $O\left(n^{\log_{3/2} 3}\right)$. That's roughly $O(n^{2.71})$. Much worse than the other sorting algorithms we know! Alex is heartbroken...

b) The algorithm, however, is correct. Think of the bottom $\frac{1}{3}$ of the elements in sorted order. After Line 6 is executed we are guaranteed that these element lie in the first $\frac{2}{3}$ of the array and thus Line 7 correctly places them in the first $\frac{1}{3}$ of the array. A similar argument can be made about the top $\frac{1}{3}$ of the elements in sorted order ending in the last $\frac{1}{3}$ of the array. Therefore, the middle $\frac{1}{3}$ of the elements in sorted order are placed in the middle $\frac{1}{3}$ of the array. Within each $\frac{1}{3}$ of the array the elements are sorted, so the array is indeed sorted at the end of the algorithm.

Problem 4. Suppose we are given an array A with n distinct numbers. We say an index i is locally optimal if $A[i] < A[i-1]$ and $A[i] < A[i+1]$ for $0 < i < n-1$, or $A[i] < A[i+1]$ for if $i = 0$, or $A[i] < A[i-1]$ for $i = n-1$.

Design an algorithm for finding a locally optimal index using divide and conquer. Your algorithm should run in $O(\log n)$ time.

Solution 4. First we test whether $i = 0$ or $i = n-1$ are locally optimal entries. Otherwise, we know that $A[0] > A[1]$ and $A[n-2] < A[n-1]$. If $n \leq 4$, it is easy to see that either $i = 1$ or $i = 2$ is locally optimal and we can check that in $O(1)$ time.

Otherwise, pick the middle position in the array (for example $i = \lfloor n/2 \rfloor$) and test whether i is locally optimal. If it is we are done, otherwise $A[i-1] < A[i]$ or $A[i] > A[i+1]$; in the former case we recur on $A[0, \ldots, i]$ and in the latter we recur on $A[i, \ldots, n-1]$. Either way, we reduce the size of the array by half and we maintain that the half that we recurse on contains a locally optimal index (this can be argued similarly to how we argued that binary search recurses on the half that contains the element that we're searching for, if it exists).

The above invariant immediately implies the correctness of our algorithm, since if the half we recurse on contains the index we're searching for and our array gets smaller with each recursive call, we eventually end up in the base case, which is easily checked. Note that our recursion could end earlier, if the middle element happens to be locally optimal, in which case we also found what we were looking for.

Rather than creating a new array each time we recur (which would take $O(n)$ time) we can keep the working subarray implicitly by maintaining a pair of indices (b, e) telling us that we are working with the array $A[b, \ldots, e]$. Thus, each call demands $O(1)$ work plus the work done by recursive calls. This leads to the following recurrence,

$$T(n) = \begin{cases} T(n/2) + O(1) & \text{for } n > 1 \\ O(1) & \text{for } n = 1 \end{cases}$$

which solves to $T(n) = O(\log n)$.

Problem 5. Given two sorted lists of size m and n. Give an $O(\log(m + n))$ time algorithm for finding the kth smallest element in the union of the two lists.

Solution 5. (Sketch.) Suppose A has n element and B and m elements, and $n \geq m$ (otherwise we swap A and B around). For ease of presentation, let us assume that all numbers are distinct.

Let $\text{rank}(x)$ be the rank of x in the union of A and B; namely, $\text{rank}(x)$ is the number of elements in A and B that have value $< x$.

Notice that for each $0 \leq i < n$ we have $i \leq \text{rank}(A[i]) \leq i + m$. This is because if $A[i]$ is smaller than every element in B then its rank would be i, while if $A[i]$ was larger than every element in B then its rank would be $i + m$. Thus, just from the index i, we have some rough estimate of $\text{rank}(A[i])$.

In order to refine this estimate, let us assume that $B[0] < A[i] < B[m-1]$ and therefore $i < \text{rank}(A[i]) < i + m$. Otherwise, if $A[i] < B[0]$ then we know that $\text{rank}(A[i]) = i$, and if $B[m-1] < A[i]$ then we know that $\text{rank}(A[i]) = i + m$.

We can use a pair of indices of B as a pinning interval $I_i = [a, b]$ to narrow the estimate for $\text{rank}(A[i)])$, by keeping the invariant $B[a] < A[i] < B[b]$, which implies $i + a < \text{rank}(A[i]) < i + b$. Initially, $a = 0$ and $b = m - 1$ and the invariant holds from our assumption from above. As long as $b - a > 1$, we can halve the pinning interval by comparing $A[i]$ to $B[\lfloor \frac{a+b}{2} \rfloor]$. Therefore, we can narrow the estimate of $\text{rank}(A[i])$ as much as needed.

Now let us go back too the task of finding kth smallest element. Suppose we use the above estimation on $i = n/2$ and we get the interval $I_{n/2}$. If $k < I_{n/2}$ (i.e., if k is smaller than the left endpoint of $I_{n/2}$) then $k < \text{rank}(A[n/2])$ so we can ignore every item larger than $A[n/2]$ from the large array and search for the kth element in the smaller instance (which is now a constant factor smaller than before). Similarly, if $k > I_{n/2}$ (i.e., if k is larger than the right endpoint of $I_{n/2}$) then $k > \text{rank}(A[n/2])$ so we can ignore everything smaller than $A[n/2]$ from the large array and search for the $(k - n/2)$th element in the smaller instance (which again is a now a constant factor smaller than before).

The only issue is what to do if $k \in I_{n/2}$. To get around this, we find the intervals of two indices: $n/4$ and $3n/4$. Notice that $\text{rank}(A[3n/4]) - \text{rank}(A[n/4]) \geq n/2$. Let $I_{n/4}$ and $I_{3n/4}$ be the respective pinning intervals after one round of pinning. Therefore, $|I_{n/4}|, |I_{3n/4}| \leq m/4$, and since $m \leq n$ it follows that $I_{n/4}$ and $I_{3n/4}$ must be disjoint. Therefore either $k > |I_{n/4}|$ or $k < |I_{3n/4}|$, which means that we can always remove the elements that are either smaller than $A[n/4]$ or larger than $A[3n/4]$.

For the time complexity we note that we do not really need to create a new array each time we recur, we can simply keep two pairs of indices, one for A and one for B, that tell us the parts of the array where we are still searching on. Then, in each iteration we get rid of $n/4 \geq (m + n)/8$ elements in $O(1)$ time. Therefore, the combined length of the intervals is at most $\frac{7}{8}$ their previous combined length. Thus, if we let N measure the combined length of the intervals, we get the recurrence

$$T(n) = \begin{cases} T(7N/8) + O(1) & \text{for } N > 1 \\ O(1) & \text{for } N = 1 \end{cases}$$

which solves to $T(N) = O(\log N)$. Therefore, the algorithm runs in $O(\log(n+m))$ time.

Problem 6. Solve the following recurrences using the Master Theorem or unrolling (the solutions will use the Master Theorem to allow you to practice that). All are $O(1)$ for $n = 1$.

a) $T(n) = 4T(n/2) + O(n^2)$
b) $T(n) = T(n/2) + O(2^n)$
c) $T(n) = 16T(n/4) + O(n)$
d) $T(n) = 2T(n/2) + O(n \log n)$
e) $T(n) = \sqrt{2}T(n/2) + O(\log n)$
f) $T(n) = 3T(n/2) + O(n)$
g) $T(n) = 3T(n/3) + O(\sqrt{n})$

Solution 6. Recall that the Master Theorem applies to recurrences of the following form: $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. There are three cases:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$ with $\varepsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n (this condition will be satisfied for all recurrences you'll encounter during this unit and we'll ignore it).

Also remember that if $f(n)$ is given in big-O notation, we can only use the above to conclude an upper bound on $T(n)$ in big-O notation and not the Θ used in the theorem. This is because Θ implies that we have both an upper and lower bound on the running time and big-O only gives us an upper bound on $f(n)$.

a) $T(n) = 4T(n/2) + O(n^2) \to T(n) = O(n^2 \log n)$ (Case 2 with $a = 4$, $b = 2$ and $f(n) = n^2$)
b) $T(n) = T(n/2) + O(2^n) \to O(2^n)$ (Case 3 with $a = 1$, $b = 2$ and $f(n) = 2^n$)
c) $T(n) = 16T(n/4) + O(n) \to T(n) = O(n^2)$ (Case 1 with $a = 16$, $b = 4$ and $f(n) = n$)
d) $T(n) = 2T(n/2) + O(n \log n) \to T(n) = O(n \log^2 n)$ (Case 2 with $a = 2$, $b = 2$ and $f(n) = n \log n$)
e) $T(n) = \sqrt{2}T(n/2) + O(\log n) \to T(n) = O(\sqrt{n})$ (Case 1 with $a = \sqrt{2}$, $b = 2$ and $f(n) = \log n$)
f) $T(n) = 3T(n/2) + O(n) \to T(n) = O(n^{\log 3})$ (Case 1 with $a = 3$, $b = 2$ and $f(n) = n$)
g) $T(n) = 3T(n/3) + O(\sqrt{n}) \to T(n) = O(n)$ (Case 1 with $a = 3$, $b = 3$ and $f(n) = \sqrt{n}$)