Lecture 14: Divide and conquer

We first finish the divide and conquer approach for integer multiplication.

Our next problem is on multiplying two \( n \times n \) square matrices \( A \) and \( B \). We apply divide and conquer. See Section 4.2 for more details. One comment about analyzing the first divide and conquer algorithm. The combine step takes \( \Theta(n^2) \). This is because it involves adding up a constant number of \( n/2 \times n/2 \) matrices (which takes \( \Theta(n^2) \)) time, and then copy these four resulting \( n/2 \times n/2 \) matrices \( C_{i,j} \) into the \( n \times n \) matrix \( C \), which also take \( \Theta(n^2) \) since there are \( \Theta(n^2) \) cells in \( C \) and each copying takes \( O(1) \) time. Thus, \( T(n) = 8T(n/2) + \Theta(n^2) \). We use the Master theorem to find out \( T(n) = \Theta(n^3) \), which is no better than the original naive algorithm. Surprisingly, Strassen found a way to improve the algorithm. See the posted document that explains Strassen’s method, which reduces the number of subproblems, and how to get better running time.

The next problem is the linear time selection problem. We are given a list of unsorted numbers and we want to find the k-th smallest number. The simplest way is to sort these numbers but this takes \( O(n \log n) \) time, and also it seems a waste of time since we only just want to find one number. The main idea is to choose a value called pivot, which can be used to split the list into two parts: one part contains all the values larger than the pivot and the other contains all the values smaller than the pivot. This allows us to recursively find the k-th smallest number in one of the parts. The key for efficiency is that we want more or less balanced partitions. The best case is each time we split the list into half and this will lead to linear time. The worst case is that the list is partitioned into two very unbalanced parts. This can lead to \( O(n^2) \) time.

The main difficulty of Selection is finding a good pivot which allows relatively even partition. For this, we first arbitrarily divide the \( n \) numbers into groups of 5 elements (and so there are \( n/5 \) groups). I ignore the remainders for now (do not matter for asymptotic behavior: see the textbook for more precise version). Then we find the median of the \( n/5 \) groups using brute force (simple: only 5 elements) and thus \( O(1) \) time each and total time for this step is \( O(n) \). We then recursively find (using exactly the same worst case \( O(n) \) algorithm by making the algorithm finding the median) the median of the medians of the \( n/5 \) groups; let \( x \) be the median of medians. We then partition the original \( n \) numbers with \( x \) as the pivot. The rest of algorithm is just as the randomized version: it issues recursively call (if needed) to one of the subproblems. The key question is why this algorithm runs in \( O(n) \) time. This is because \( x \) is ensured to be not too unbalanced. Why? Note that at least \( 0.5 \times (n/5) - 2 \) groups with median \( x \) or smaller and with 5 elements in each group. So those groups have at least 3 elements larger than its median and so is also at least \( x \). So we have at least \( 3 \times (0.5 \times (n/5) - 2) = 3n/10 - 6 \) elements larger than \( x \). We can perform the same logic to the other side of the partition. This implies that the sub problem we are solving recursively will have at most \( 7n/10 + 6 \) elements. So we have:

\[
T(n) \leq T(n/5) + T(7n/10 + 6) + an \quad \text{for some constant} \ a \quad \text{when} \ n \ \text{is relatively large, and this shows the time is} \ O(n). \]

Why? We can prove by induction. Assume assume \( T(n') \leq cn' \) for all \( n' < n \). Then, \( T(n) \leq c \times n/5 + c(7n/10 + 6) + a = 0.9cn + 7c + an = cn + ((-c/10 + a)n + 7c) \leq cn \), since the last inequality holds when \( c \) is large enough.

Our last problem is the closest point problem, as we discussed before (where we say there is a \( O(n^2) \) algorithm). The details can be found in Section 5.4. Briefly, we divide the points into two subproblems by finding a vertical line to separate the points. Then, we recursively solve the two subproblems and get smallest distances \( d_1 \) and \( d_2 \) respectively. Let \( d = \min(d_1, d_2) \). We now need to consider a point to the left but within \( d \) distance from the central vertical line, and a point to the right within \( d \) from the center. We then consider each of these points within this band from top to bottom, and calculate the distance between this point with each point below it. The key observation is that there can be no
more than 7 points to consider for each point. See the textbook for more details. The running time is then reduced to $O(n \log n)$ (if we ignore the time for sorting, which can be taken care with proper implementation).