Lecture 15: Divide and conquer and dynamic programming

We finish the discussion of linear time selection. Recall that we are given a list of unsorted numbers and we want to find the $k$-th smallest number. The main idea is 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. 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 subproblem we are solving recursively will have at most $7n/10 + 6$ elements. So we have: $T(n) \leq T(n/5) + T(7n/10 + 6) + an$ (for some constant $a$) when $n$ 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 sub-problems by finding a vertical line to separate the points. Then, we recursively solve the two sub-problems 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).

Now we start to discuss what is dynamic programming. Dynamic programming is one of the most useful algorithmic techniques. We use an example problem, longest increasing subsequence (LIS), to illustrate the idea. Given a list of $n$ numbers (i.e. sequence, denoted as $A$), we define subsequence as a sub-list of the numbers (with the original order). For example, let the sequence being $4; 2; 7; 5; 3; 5; 9; 6$. A subsequence is $2, 5, 4, 6$. We call a subsequence an increasing subsequence if this subsequence consists of numbers in the increasing order. Example of increasing subsequence of the above example is $4, 7, 9$. The LIS problem is to find the longest increasing subsequence (LIS). We first study how to compute the length of LIS. In this example, the length of LIS is 4.

Let us define $L(i)$ as the length of a LIS that ends at position $i$ (position means index of the array). Then, the length of the LIS of the whole list is simply $\max_{i=1}^n L(i)$ (Why? Make sure you understand this). To see how to compute $L(i)$, we consider a LIS ending at $i$. Its previous number should be some at some $j < i$, where $A[j] < A[i]$. But which one is $j$ if there are multiple such $j$? Let $j$ be the position immediately preceding $i$ in the LIS ending at $i$. Then we know $L(i) = L(j) + 1$. This is because since the
LIS uses \(j\)-th number, the it must take the LIS ending at \(j\) (otherwise it contradicts the assumption of the LIS ending at \(i\)). Since we do not know which \(j\) to use, we take the best: \(L(i) = \max_{j<i, A[j]<A[i]} L(j) + 1\).

**Lecture 16: Dynamic programming: longest increasing subsequence and longest common subsequence**

We continue on the longest increasing subsequence in this lecture. Now to compute a \(L(i)\), we can use the previous recursion: it will then recursively compute \(L(j)\). This works, but with a great cost. The problem is, many subproblems \(L(j)\) are computed multiple times. To see how serious the problem is, we consider the worst case for computing \(L(n)\). Let \(S(i)\) be the number of recursion calls (i.e. nodes in recursion tree) for computing \(L(i)\). Then, in the worst case,

\[
S(n) = 1 + \sum_{i=1}^{n-1} S(i)
\]

We observe \(S(1) = 1, S(2) = 2, S(3) = 4\). Then we guess: \(S(i) = 2^{i-1}\), and this is easily provable using induction. So, \(S(n) = 2^{n-1}\), which implies that the recursion based algorithm runs in exponential time. Now we apply the dynamic programming that will lead to a polynomial time algorithm. The key idea is to memorize the subproblem solutions so that we do not re-compute them, using a bottom-up approach.

```python
1: for i = 1 to n do
2:  \(L[i] = 1\)
3:  for j = 1 to i - 1 do
4:    if \(A[j] < A[i]\) and \(1 + L[j] > L[i]\) then
5:      \(L[i] = 1 + L[j]\)
6:    end if
7:  end for
8: end for
9: return \(\max_{i=1}^{n} L[i]\).
```

Note: when computing \(L(i)\), \(L(j)\) has already been computed. And for each \(L(i)\), it gets updated just once. Running time: there are \(n\) iterations, each iteration taking at most \(O(n)\) time. So the total time is \(O(n^2)\).

The only remaining issue is that we only know the LIS’s length, but what is LIS itself? That is not so hard. We suppose the LIS ending at node \(p\) (i.e. we know the last letter of the LIS, \(A[p]\)). To find the one before \(A[p]\), we perform trace back. In other words, we check which position \(j\) that leading to the current value at \(L[p]\); then we move to this \(j\). We do this recursively (each time figure out its immediate previous node). I suggest you to work out a few examples to better understand this.

The main subject of today’s lecture is the longest common subsequence (LCS) problem. This problem is about comparing two sequences. Our textbook chooses to cover a slightly more complex case of sequence comparison in Section 6.5. That is not exactly the same as the LCS problem, but the two problems are quite related. I would suggest you to first read that section.

We use a two dimensional array: \(LCS[i, j]\), which represents the length of LCS of two prefixes \(X[1..i]\) and \(Y[1..j]\). By this definition, the length of LCS of \(X\) and \(Y\) is equal to \(LCS[n, m]\), where \(n\) and \(m\) are the length of \(X\) and \(Y\) respectively. In class, someone suggested the solution can be found by finding the largest value with \(LCS(i, j)\). This is correct, but checking the entire \(LCS\) array is not necessary because \(LCS[n, m]\) is the largest among \(LCS[i, j]\) (why?).

A key to dynamic programming is how to come up with recurrences of the subproblems. The recurrence allows you to compute the value of a subproblem using the pre-computed values of smaller subproblems. This often involves analyzing the situation to see what you can do to some specific parts of the subproblems. Often you have multiple choices, and you will just take the most desirable one. In
the LCS example, we consider what happens to the last two characters. If these two characters are the
same, then they must be in the LCS (Why? Make sure you understand this).

For the subproblem $LCS(i, j)$, we look at the two cases: $X[i] = Y[j]$ and $X[i] \neq Y[j]$. When $X[i] = Y[j]$, I explained in class that it is always safe to include $X[i]$ in the LCS of $X[1..i]$ and $Y[1..j]$. Make sure you understand why. When $X[i] \neq Y[j]$, clearly either $X[i]$ is not included or $Y[j]$ is not included in the LCS. And one of these two must happen. We then take the better of the two choices.