EECS 3101 - B Tutorial 5 Notes

Shahin Kamali

York University

September 2022

Note for lower-bound proofs

The first two questions in this tutorial concern lower bounds like the one we saw for comparison-based sorting. In approaching these questions, we take the following steps:

1. Counting the number of possible inputs. In the case of Sorting, the number of possible inputs is $n!$ (each possible permutation of items forms an input instance for sorting).

2. Forming a decision tree. Each decision tree bijects to an algorithm, and each possible input would be a leaf of the tree. Each action by the algorithm has a certain number of outcomes, which indicates the number of children of nodes in the tree. In the case of sorting, each action (comparison) has two outcomes [we assume all items are distinct]. So, a decision tree for sorting is a binary tree with $n!$ leaves. The tree's height would define the worst-case running time of the algorithm it bijects to.

3. Find the height of the tree. You can do it by looking at the degree of the tree and the number of its children. In the case of sorting, the tree is binary, and the number of leaves is $n!$; therefore, the running time is at least $\log_2 n! = \Omega(n \log n)$. Note that the height is minimized when the tree is fully balanced. Unbalanced trees represent worse algorithms; here, we are interested in a lower bound for the running time of the *best* algorithm.

Lower Bound Questions

1. Recall that in a sorted array of n distinct comparable items, we can use *binary search* to search for a given item in $O(\log n)$. Prove that binary search is the optimal searching algorithm in a sorted array. You need to use a decision tree approach to show that no search algorithm can search in a sorted array in time less than $O(\log n)$.

   Answer: We take the three steps mentioned above as follows. First, we need to count the possible inputs when searching for x. The first input is an array that finds x in its first index (note that if two different arrays both find x at index 1, they are considered the same because the way they are treated by an algorithm may be similar when searching for x). Similarly, the second input array finds x in its second position, and more generally, the i'th input finds x in its i'th index (for $1 \leq i \leq n$). The $(n+1)$'th input is formed by an array that does not contain x. Therefore, we have $n+1$ possible inputs for binary search, each requiring a separate treatment and output. Hence, each of these inputs is associated with a different leaf in the decision tree of a search algorithm. In the second step, we observe that there are two outcomes

for each action (comparison) in any search algorithm, that is, $<$ and $>$. Therefore, the decision tree of any search algorithm is a binary tree with $n+1$ leaves. In the third step, we note that the height of such a tree is at least $\log_2(n+1) = \Omega(\log n)$. Note that the height is minimized when the tree is fully balanced (an unbalanced tree will have a higher height and bijects to a search algorithm with a larger number of comparisons in the worst-case). In summary, we showed that any search algorithm requires $\Omega(\log n)$ comparison. Given that binary search runs in $O(\log n)$, we conclude that binary search is an asymptotically optimal search algorithm (in sorted arrays).

2. We say an array of numbers is *almost-sorted* if at least half of the elements appear in their right positions in the sorted array.

   - For example, array $A = \{2, 1, 3, 4, 6, 5, 7, 8\}$ is almost-sorted because $3, 4, 7$, and 8 are in their correct position.

   Provide a *tight lower bound* for sorting any almost sorted array of n numbers using a comparison-based sorting. A complete answer includes an algorithm whose running time is asymptotically equal to your lower bound.

   Answer: . We take the three steps mentioned above as follows. First, we need to count the number of almost-sorted arrays. For that, consider arrays in which the first $n/2$ items appear in sorted order and the remaining $n/2$ items appear in an arbitrary order. These arrays are almost-sorted, and there are $(n/2)!$ such arrays. We can conclude that the number of almost-sorted arrays is *at least* $(n/2)!$. In the second step, we note that our decision tree has two outcomes ($<$ and $>$) per action (comparison) because our items are distinct (otherwise, three outcomes $<$, $=$, $>$ were possible). Therefore, the decision tree is a binary tree with at least $(n/2)!$ leaves. In the third step, we observe that the height of a fully balanced binary tree with $(n/2)!$ is $\log_2(n/2)!$. Increasing the number of leaves may increase this height, but we can surely state that the number of actions (comparisons) needed for sorting an almost-sorted array is at least $\log_2(n/2)! = \Omega(n \log n)$.

   Clearly, the lower bound of $\Omega(n \log n)$ is asymptotically tight since any array can be sorted in time $O(n \log n)$.

Note for dynamic programming

To solve a problem using a dynamic programming approach, we take the following four steps:

1. Form sub-problems for the given problem in a way that the solution for each sub-problem can be found by looking at the solution for smaller sub-problems. This step is often the most demanding part of designing dynamic programming algorithms. For the *rod cutting* problem, we were looking at the maximum profit for cutting a rod of length n, and the sub-problem was asking for the maximum profit for cutting a rod of length $j \leq n$ (when $j = n$, the sub-problem was the actual problem we wanted to solve). For the *longest common subsequence (LCS)* problem, the sub-problem was asking for the length of the LCS of the prefix of X of length i (for $1 \leq i \leq m$) and prefix of Y of length j (for $1 \leq j \leq n$). When $i = m$ and $j = n$, the subproblem becomes the actual problem we want to solve. For the *knapsack* problem, the subproblems asked for the maximum profit for a knapsack of capacity B when the input items are $a_1, \ldots, a_k$ (we have $B \leq C$ and $k \leq n$). The subproblem with $B = C$ and $k = n$ is the actual problem we want to solve.

2. Write down the value/cost of the optimal solution for each subproblem as a recursive function of the optimal solution for smaller subproblems. If you define sub-problem properly, this step follows immediately from your definition of subproblems. See Slides 8 for rod-cutting, Slide 23 for LCS, and Slide 31 for knapsack. Like any recursive function, you must set the base cases appropriately.

3. Fill out a dynamic programming table in a bottom-up approach. There is an entry for each sub-problem in the table, and you set its value by looking at the optimal solution for smaller subproblems (by looking at their respective entries in the table). The first entries to fill are the base-case entries. Other entries must be filled in appropriately to ensure the optimal solutions for smaller subproblems are computed first (so that the optimal value for larger subproblems can be computer by looking at those entries). Sometimes we store "flags" in this step to allow fast recovery of the optimal solution (more on flags below). By the end of this step, the *value* of the optimal solution for the actual problem can be found in its appropriate entry in the DP table.

4. Starting from the entry in the table associated with the actual problem, we can use the flags to recover the actual solution. Examples of such flags include array s in the rod-cutting problem (see Slide 13) and the arrows in the LCS problem. Sometimes we can compute the solution by only looking at the values stored in the DP table (without flags). For example, in the case of the knapsack problem, an item was optimal if its entry was not equal to the entry above it in the DP table.

Dynamic Programming Question

1. A string is "palindrome" if it reads the same backward as forward. Given a string S, we want to find the longest subsequences of S that is also a palindrome. For example, when $S = ABBDCAB$, the longest palindromic subsequence (LPS) of S is $ABBA$. Devise a dynamic programming algorithm to find the LPS of S.

   Answer:

   1 *Step 1:* Recall that we need to define subproblems in a way that we can devise the optimal solution value for each subproblem using the value of the optimal solutions for smaller subproblems.
   Let $S[i, j]$ indicate the substring of S starting at index i and ending at index j. Each subproblem (i, j) asks for the longest palindrome subsequence (LPS) of the substring of $S[i, j]$. Note that if $S[i] = S[j] = x$, we can find the LPS $S[i + 1, j - 1]$ and add x to the endpoints of that LPS. For example, if $S[i, j] = BBDCAB$, then we can find the LPS of $BDCA$ (which is either of "B", "D", "C" or "A") and add 'B' to its endpoints, to get either of "BBB", "BDB", "BCB" or "BAB". If $S[i] \neq S[j]$, then we know either $S[i]$ or $S[j]$ (or possibly both) are not a part of the LPS of $S[i, j]$. If $S[i]$ is not in the LPS, the LPS of $S[i, j]$ is the same as the LPS of $S[i + 1, j]$. Otherwise, when $S[j]$ is not in the LPS, the LPS of $S[i, j]$ is the same as the LPS of $S[i, j - 1]$. For example, if $S[i, j] = ABBDCAB$, the LPS of $S[i, j]$ is either the LPS of $S[i + 1, j] = BBDCAB$ (which is BBB) or the LPS of $S[i, j - 1] = ABBDCA$ (which is $ABBA$). Since we are looking for a longer LPS, we have to take the longer of the two possible candidates.

   2 *Step 2:* Recall that in step 2, we write down a recursive formula for the value of optimal solutions.
   From the discussion above, we get the following recursive formula:

$$LPS[i,j] = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{if } i = j \\ LPS[i+1, j-1] + 2 & \text{if } i < j \text{ and } S[i] = S[j] \\ \max\{LPS[i+1,j], LPS[i,j-1]\} & \text{otherwise} \end{cases}$$

Note that $i > j$, then $S[i,j]$ will be empty, and its LSP has length 0. If $i = j$, then $S[i,j]$ contains only one character, and its length is 1. Otherwise, if $S[i] = S[j] = x$, the LPS of $S[i,j]$ is formed by adding x to the endpoints of the LPS of $S[i+1, j-1]$. Note that this addition contributes two extra characters to the LPS. Finally if $S[i] \neq S[j]$, the LPS of $S[i,j]$ is formed by either that of $S[i+1,j]$ or $S[i,j-1]$.

3 *Step 3:* fill up the dynamic programming table recursively. From the above formula, it is clear that the value of $LPS[i,j]$ is calculated from those of $S[i+1,j-1]$ (the entry on south-west), $S[i,j-1]$ (the west entry), and $S[i+1,j]$ (the south entry). Therefore, after setting the base cases (when $i \geq j$), we need to fill the DP table row-by-row, from the lowest row to the highest, and within each row, we fill the table from left to right. This way, the entries on west/south and southwest are calculated earlier for each entry. The code and completed table look as follows.

i	j	1	2	3	4	5	6	7
		A	B	B	D	C	A	B
1	A	1	←1	←1	↓2	←2	↙4	←4
2	B	0	1	↙2	←2	←2	←2	↙3
3	B	0	0	1	←1	←1	←1	↙3
4	D	0	0	0	1	←1	←1	←1
5	C	0	0	0	0	1	←1	←1
6	A	0	0	0	0	0	1	←1
7	B	0	0	0	0	0	0	1

```
LPS-Fill-DP-Table(S, n)
1.  for i = 1 to n
2.      for j = 1 to n
3.          if i = j
4.              LPS[i][j] = 1
5.          else
6.              LPS[i][j] = 0
7.  for i = n − 1 downto 1
8.      for j = i + 1 to n
9.          if S[i] = S[j]
10.             LPS[i, j] = LPS[i + 1, j − 1]
11.             flag[i, j] = ↙
12.         else
13.             LPS[i, j] = max{LPS[i + 1, j], LPS[i, j − 1]}
14.             if LPS[i + 1, j] > LPS[i, j − 1]
15.                 flag[i, j] = ↓
16.             else
17.                 flag[i, j] = ←
```

4 *Step 4:* Recall that in this step, we retrieve the actual LPS by moving backwards in the table. The flags can help us follow the entries that give us the LPS. Those entries are indices at which the flags is set to ↙. In the example below, ABBA is printed, using a stack. See the code below.

i	j	1	2	3	4	5	6	7
		A	B	B	D	C	A	B
1	A	1	←1	←1	↓2	← 2	↙4	← 4
2	B	0	1	↙2	← 2	← 2	← 2	↙3
3	B	0	0	1	←1	←1	←1	↙3
4	D	0	0	0	1	←1	←1	←1
5	C	0	0	0	0	1	←1	←1
6	A	0	0	0	0	0	1	←1
7	B	0	0	0	0	0	0	1

```
LPS-Retrieve-DP-Table(S, n, LPS, flag)
1.   i ← 1
2.   j ← n
3.   Stack ← an empty stack
4.   while i ≤ j
5.       if flag[i, j] =↙
6.           printS[i]
7.           Stack.push(S[i])
8.           i ← i + 1
9.           j ← j − 1
10.      else if flag[i, j] =↓
11.          i ← i + 1
12.      else
13.          j ← j − 1
```