Problem Set 3: Due Friday, March 16th at 10:10am

Please follow the instructions given on the previous homework sheets.

[20] Problem 1. Selection Problem

Consider the following problem:
Given an unsorted list of \( n \) distinct numbers \( x_1, x_2, \ldots, x_n \), and an integer \( B \), find a maximum cardinality subset of the list which sums to at most \( B \). For example, given the list \{35, 27, 22, 55, 87, 74\} and \( B = 100 \), a maximum cardinality set has three elements: \{35, 22, 27\}; since \( 35 + 22 + 27 \leq 100 \) and any four elements of the original list sums to more than 100.

(a) [5pts] Give a simple \( O(n \log n) \) algorithm to solve this problem.

(b) [15pts] Describe (and provide pseudocode for) how to use the Selection algorithm (discussed in class) to solve the problem in expected \( O(n) \) time. Justify your time bound. Note you may use the Selection algorithm as a black box routine which given any list, and a value \( k \) returns the \( k \)th smallest element in expected linear time. (Hint: start by finding the median of the list, and testing if all the numbers smaller than the median sum to \( B \) or less). Justify the \( O(n) \) expected run time.

[20] Problem 2. Selection Problem

In our discussion of the Selection algorithm we assumed there were no duplicate items in our input list. We now consider the more general case where our input list of numbers, \( x_1, x_2, \ldots, x_n \) may have multiple copies of the same item. Note that even with duplicates, our notion of the \( k \)th smallest element remains the same: the value of the \( k \)th element of a sorted list. For example, for the list \{1,3,3,3,3,4,4,6,6,7\}, we would return 3 if \( k = 2, 3, 4, 5 \), and return 4 if \( k = 6, 7 \).

(a) [7 pts] A slight modification of the \textit{RandSelect} algorithm described in class will make it work even if there are many duplicates: in Step 2 when we partition the list, we let \( L \) be the set of elements \( \leq p \) (as opposed to strictly \( < p \)). Although this modified algorithm works, it may be much slower. Analyze the running time (in terms of big-Oh) of this modified algorithm to find the median element when the list (of size \( n \)) consists of \( n \) copies of the same item. Obviously which element you choose as the pivot element doesn’t matter in this case, so the best case, average case, and worst case are all the same.

(b) [13 pts] Modify the Randomized-Selection algorithm so it has \( O(n) \) expected performance even when there may be many duplicates (HINT: modify the partition routine along with the method of deciding which sublist to continue searching). Provide pseudocode. Justify your time bound.

[25] Problem 3. Assembly Line

Consider a variation of the assembly line scheduling problem. We now have a third line \( C \), but this line is different, it has only odd numbered stations \( (1,3,5, \ldots, n) \) and each station \( C_i \) in line \( C \) does the operations of both the \( i \)th and \( (i + 1) \)th stations of lines A and B (for simplicity, assume \( n \) is even). As before, if we are on line C, we can continue on line C with no delay, or switch to line A or B (but after completing \( C_i \) we would switch to \( A_{i+2} \) or \( B_{i+2} \)). Similarly, after completing an operation on line A or B we can switch to line C for the next operation. However, we assume that we can switch to line C only after completing an \textit{even} (not odd) operation on A or B. For simplicity we will assume it takes time \( d \) to switch to line C (from A or B) or to switch from line C to line A or B. Assume it takes time \( c_i \) for \( C_i \) to complete the \( i \)th and \( i + 1 \) operations.

Modify the dynamic programming formulation given in class to find the fastest time using a combination of lines A,B and C by providing the new DP formulation (you don’t have to output the actual sequence or give any pseudocode but you should show the new DP formulation).
(a) [10 pts] Apply the algorithm we discussed to fill in the following table and find the length of the LCS of B A C B A D and A B A Z D C (do not worry about finding the actual LCS or that corresponding array).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>B</th>
<th>A</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b) [32 pts] Now consider a new problem: Given an array $A$ of $n$ integers $a[1], a[2], \ldots, a[n]$ in unsorted order, we say that $a[j_1], a[j_2], \ldots, a[j_t]$ is an increasing subsequence of $A$ of length $t$, if the following conditions are met:

- $j_1 < j_2 < \cdots < j_t$
- $a[j_1] \leq a[j_2] \leq \cdots \leq a[j_t]$

For example, given the array $[1, 4, 7, 3, 11, 2, 5, 18, 6]$, $[3, 5]$ is one possible increasing subsequence of length 2, while $[1, 4, 7, 11, 18]$ is an increasing subsequence of length 5. We want to find an efficient algorithm that, given $A$, finds the length of the longest increasing sequence of $A$.

Describe an efficient dynamic programming algorithm (running time $O(n^2)$ is enough to get full credit) that, given $A$, finds the length of the longest increasing subsequence of $A$. The presentation of your solution should be structured as follows:

- [7 pts] Specify what the dynamic programming matrix (or array) is supposed to contain; its size; and how to retrieve the answer to the subsequence problem once the matrix (or array) is filled.
- [12 pts] Specify which entries can be filled at the beginning, and with which values (i.e. the base cases); and give a recursive formula for the other entries. You do not need to provide pseudocode, but be sure to give the recursive dynamic programming formulation.
- [5 pts] Estimate the time needed to fill one entry, and give the total time needed to fill the whole matrix (or array) and to find the answer to the problem.
- [8 pts] Provide pseudocode (about 5-7 lines at most) to find the actual Longest Increasing Sequence once the matrix (or array) is filled.

Hint: You can solve this problem using a 1-D array.