CSCE 750, Homework 5

This assignment covers material from the lectures on Chapters 14, 16, and 17 [3rd ed. Chapters 14, 15, and 17], in preparation for Quiz 5.

Page 485, Exercise 17.1-1 [3rd ed. Page 344, Ex 14.1-1]: Show how OS-SELECT($T.root, 10$) operates on the red-black tree T shown in Figure 17.1. [The fact that Figure 17.1 is a red-black tree is not important.]

Page 485, Exercise 17.1-2 [3rd ed. Page 344, Ex 14.1-2]: Show how OS-RANK(T, x) operates on the red-black tree T shown in Figure 17.1 and the node x with $x.key = 35$. [The fact that Figure 17.1 is a red-black tree is not important.]

Pages 485, Exercise 17.1-3 [3rd ed. Page 344, Ex 14.1-3]: Write a nonrecursive version of OS-SELECT.

Page 486, Exercise 17.1-7 [3rd ed. Page 345, Ex 14.1-7]: Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4 on Page 47) in an array of n distinct elements in $O(n \lg n)$ time.

Pages 372–373, Exercise 14.1-2 [3rd ed. Page 370, Ex 15.1-2]: Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the *density* of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then countinues by applying the greedy strategy to the remaining piece of length $n - i$.

Page 373, Exercise 14.1-3 [3rd ed. Page 370, Ex 15.1-3]: Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

Page 381, Exercise 14.2-1 [3rd ed. Page 378, Ex 15.2-1]: Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$. [Construct the m and s tables, then use them to find the optimal parenthesization.]

Page 381, Exercise 14.2-2 [3rd ed. Page 378: Ex 15.2-2]: Give a recursive algorithm MATRIX-CHAIN-MULTIPLY(A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call is MATRIX-CHAIN-MULTIPLY($A, s, 1, n$).) Assume that the call RECTANGULAR-MATRIX-MULTIPLY(A, B) returns the product of matrices A and B.

Page 393, Exercise 14.3-2 [3rd ed. Page 389, Ex 15.3-2]: Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divided-and-conquer algorithm such as MERGE-SORT.

Page 393, Exercise 14.3-3 [3rd ed. Pages 389–390, Ex 15.3-3]: Consider the antithetical variant of the matrix-chain multiplication problem where the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure? [Write pseudocode for a DP algorithm for this variant problem, or explain why DP won't work.]

Page 393, Exercise 14.3-4 [3rd ed. Page 390, Ex 15.3-4]: As stated, in dynamic programming, you first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that she does not always need to solve all the subproblems in order to find an optimal solution. She suggests that she can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

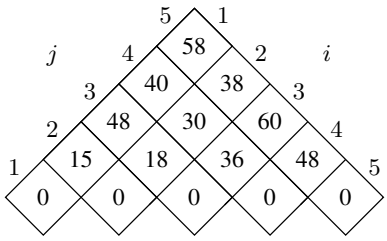
Page 407, Problem 14-2 [3rd ed. Page 405, Problem 15-2]: ***Longest palindrome subsequence***

A *palindrome* is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, `civic`, `racecar`, and `aibohphobia` (fear of palindromes).

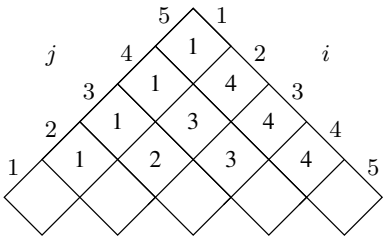
Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input `character`, your algorithm should return `carac`. What is the running time of your algorithm?

[Note: There are several other interesting problems on pages 407–415 [3rd ed. pp. 404–412].]

NIT1: The tables generated by a certain run of the matrix chain multiplication algorithm from class are shown below. Find the optimal parenthesization for this instance.



m table



s table

Page 453, Exercise 16.1-1 [3rd ed. Page 456, Ex 17.1-1]: If the set of stack operations includes a MULTIPUSH operation, which pushes k items onto the stack, does the $O(1)$ bound on the amortized cost of stack operations continue to hold?

Page 453, Exercise 16.1-2 [3rd ed. Page 456, Ex 17.1-2]: Show that if a DECREMENT operation is included in the k-bit counter example, n operations can cost as much as $\Theta(nk)$ time.

NIT2, OPTIONAL: Modify the k-bit binary counter data structure so that it supports both INCREMENT and DECREMENT operations in $O(1)$ amortized time, as well as a READOUT operation in $O(k)$ amortized time, assuming the initial value of the counter is zero. (*Hint:* Allow each "bit" b_i to have one of four possible values: -1, 0, 1, or 2. The value of the counter is still $\sum_{i=0}^{k-1} b_i 2^i$ as before. The READOUT operation prints the counter's value in binary (using 0's and 1's only). Show how to implement the three operations, then analyze their amortized running time using the potential method.

Pages 455–456, Exercise 16.2-1 [3rd ed. Page 458 Ex 17.2-1]: You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Page 456, Exercise 16.2-3 [3rd ed. Page 459, Ex 17.2-3]: You wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(n)$ time on an initially zero counter. (*Hint:* Keep a pointer to the high-order 1.)

Page 459, Exercise 16.3-3 [3rd ed. Page 462, Ex 17.3-3]: Consider an ordinary binary min-heap data structure supporting the instructions INSERT and EXTRACT-MIN that, when there are n items in the heap, implements each operation in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that your potential function yields these amortized time bounds. Note that in the analysis, n is the number of items currently in the heap, and you do not know a bound on the maximum number of items that can ever by stored in the heap.

Page 460, Exercise 16.3-5 [3rd ed. Page 462, Ex 17.3-6]: Show how to implement a queue with two ordinary stacks (Exercise 10.1-7) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$. [Write pseudocode for DEQUEUE and ENQUEUE operations, using the PUSH and POP operations of the two stacks, then use the potential method to analyze the amortized run time.]

NIT3: Consider the dynamic table data structure from the textbook and the lecture, but modified so that, when the table is reallocated, the size increases by 10, rather than doubling. (That is, replace the "$\times 2$" with a "$+10$.") Find the amortized run time of each operation, using the same $\Phi(T) = 2T.num - T.size$ potential function as we used in class. Explain why this variant is a terrible idea.

NIT4: Consider the dynamic table data structure from the textbook and the lecture. Find the amortized run time of each operation, using a potential function that evaluates to the number of elements in the table.