Basic Algorithms — Spring 2016 — Problem Set 4
Due: March 9

1. **Road trip.** You are going on a long trip. You start on the road at mile post 0. Along the way there are \(n\) hotels, at mile posts \(a_1 < a_2 < \cdots < a_n\), where each \(a_i\) is measured from the starting point. The only places you are allowed to stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance \(a_n\)), which is your destination. You’d ideally like to travel 200 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel \(x\) miles during a day, the penalty for that day is \((200-x)^2\). You want to plan your trip so as to minimize the total penalty—that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

2. **Corrupted text.** You are given a string of \(n\) characters \(s[1..n]\), which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like “itwas the best of times…”). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function \(\text{dict}(\cdot)\): for any string \(w\), \(\text{dict}(w) = true\) if \(w\) is a valid word, and \(\text{dict}(w) = false\) otherwise.

   (a) Give an algorithm that determines whether the string \(s\) can be reconstituted as a sequence of valid words. The running time should be at most \(O(n^2)\), assuming calls to \(\text{dict}\) take unit time.

   (b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

3. **Coping with failure.** A mission-critical production system has \(n\) stages that have to be performed sequentially; stage \(i\) is performed by machine \(M_i\). Each machine \(M_i\) has a probability \(r_i\) of functioning reliably and a probability \(1-r_i\) of failing (and the failures are independent). Therefore, if we implement each stage with a single machine, the probability that the whole system works is \(r_1r_2\cdots r_n\). To improve this probability we add redundancy, by having \(m_i\) copies of the machine \(M_i\) that performs stage \(i\). The probability that all \(m_i\) copies fail simultaneously is only \(1-(1-r_i)^{m_i}\), so the probability that stage \(i\) is completed correctly is \(1-(1-r_i)^{m_i}\) and the probability that the whole system works is \(\prod_{i=1}^{n}(1-(1-r_i)^{m_i})\). Each machine \(M_i\) has a cost \(c_i\), and there is a total budget \(B\) to buy machines. (Assume that \(B\) and \(c_i\) are positive integers.) Given the probabilities \(r_1, \ldots, r_n\), the costs \(c_1, \ldots, c_n\), and the budget \(B\), find the redundancies \(m_1, \ldots, m_n\) that are within the available budget and that maximize the probability that the system works correctly.

4. **Cutting strings.** A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes \(n\) units of time for a string of length \(n\), regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of 20 + 17 = 37, while doing position 10 first has a better cost of 20 + 10 = 30.

   Give an efficient algorithm that, given the locations of \(m\) cuts in a string of length \(n\), finds the minimum cost of breaking the string into \(m+1\) pieces.

5. **A card game.** Consider the following game. A “dealer” produces a sequence \(s_1, \ldots, s_n\) of “cards,” face up, where each card \(s_i\) has a value \(v_i\). Then two players take turns picking a card from the sequence, but can only pick the first or the last card of the (remaining) sequence. The goal is to collect cards of largest total value. (For example, you can think of the cards as bills of different denominations.) Assume \(n\) is even.

   (a) Show a sequence of cards such that it is not optimal for the first player to start by picking up the available card of larger value. That is, the natural “greedy” strategy is suboptimal.

   (b) Give an \(O(n^2)\)-time algorithm to compute an optimal strategy for the first player. Given the initial sequence, your algorithm should precompute in \(O(n^2)\) time some information, and then the first
player should be able to make each move optimally in $O(1)$ time by looking up the precomputed information.

6. **Sequence alignment.** When a new gene is discovered, a standard approach to understanding its function is to look through a database of known genes and find close matches. The closeness of two genes is measured by the extent to which they are aligned. To formalize this, think of a gene as being a long string over an alphabet $\Sigma = \{A, C, G, T\}$. Consider two genes (strings) $x = ATGCC$ and $y = TACGCA$. An alignment of $x$ and $y$ is a way of matching up these two strings by writing them in columns, for instance:

\[
\begin{array}{c}
- & A & T & G & C & C \\
T & A & C & G & C & A \\
\end{array}
\]

Here the “$-$” indicates a “gap.” The characters of each string must appear in order, and each column must contain a character from at least one of the strings. The score of an alignment is specified by a scoring matrix $\delta$ of size $(|\Sigma| + 1) \times (|\Sigma| + 1)$, where the extra row and column are to accommodate gaps. For instance the preceding alignment has the following score:

$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A)$.

Give an algorithm that takes as input two strings $x[1..n]$ and $y[1..m]$ and a scoring matrix $\delta$, and returns the highest-scoring alignment. The running time should be $O(mn)$.

7. **Min interval cover.** You are given a list of closed intervals, $J_i := [a_i, b_i]$ for $i = 1, \ldots, n$. Each interval also has a nonnegative weight $w_i$. You are also given a “target” interval $T = [x, y]$. A nonempty index set $S \subseteq \{1, \ldots, n\}$ is called a cover if

$T \subseteq \bigcup_{i \in S} J_i$.

The weight of an index set $S$ is defined to be $\sum_{i \in S} w_i$. Your task is to find a cover of minimal weight (or report that no there is no cover). Give an algorithm that solves this problem in time $O(n^2)$.

For example, given

$T = [0, 5], \ J_1 = [1, 6], \ w_1 = 5, \ J_2 = [0, 2], \ w_2 = 1, \ J_3 = [2, 4], \ w_3 = 2, \ J_4 = [3, 7], \ w_4 = 1$,

the index set $\{1, 2\}$ is a cover of weight 6, while $\{2, 3, 4\}$ is a cover of weight 4. Your algorithm would output $\{2, 3, 4\}$ on this input.