Basic Algorithms — Fall 2020 — Problem Set 5
Due: Wed, Nov 18, 11am

1. **Huffman coding mechanics.** You are to encode the character string "`she␣sells␣sea␣shells`" using Huffman encoding.

   (a) Compute the frequency count for each character (including the blank character "␣") that appears in the above string.

   (b) Using these frequency counts as weights, run the Huffman encoding algorithm. Show the intermediate trees built by the algorithm, along with the final tree, and also the encoding of each individual character as a bit string.

   (c) Show the encoding of the above character string as a bit string using this encoding of the individual characters.

2. **Better load balancing.** Consider again the load balancing problem from class. In this problem, you have m machines and n jobs, where job j has running time $t_j > 0$ for $j = 1, \ldots, n$. The goal is to assign jobs to machines so that the max load is minimized. Here, the load of any one machine is the sum of the running times of the jobs assigned to it, and the max load is the maximum of the loads of the machines.

   Now consider the greedy algorithm from class, but suppose that we process the jobs in order of *decreasing* running time. Your goal is to *prove* that the max load T computed by this modified greedy algorithm is at most $\frac{3}{2}T^*$, where T^* is the optimal max load.

   (a) Suppose the jobs are ordered so that $t_1 \geq t_2 \geq \cdots \geq t_n$, and assume that $n > m$. Show that $T^* \geq 2t_{m+1}$.

   (b) Using part (a), show that $T \leq \frac{3}{2}T^*$. To do this, mimic the proof from class: consider a machine k whose load T_k is equal to T, and let ℓ be the last job scheduled on machine k. You should use the inequality

   $$T_k \leq t_\ell + \frac{1}{m}\sum_j t_j,$$

   which we derived in that proof, and combine this with Lemmas 1 and 2 from that proof as well as part (a) of this exercise.

   Hint. Consider two case separately: (1) only one job is scheduled on machine k, (2) more than one job is scheduled on machine k.

   Note. The bound $T \leq \frac{3}{2}T^*$ is not tight. It is possible to prove that $T \leq \frac{4}{3}T^*$, which *is* tight.

3. **Road trip.** You are going on a long trip. You start on the road at mile post a_0. Along the way there are n hotels, at mile posts $a_1, \ldots, a_n$, where $a_0 < a_1 < a_2 < \cdots < a_n$. 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 mile post 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.

   To do this, you are to use Dynamic Programming.

   (a) To begin with, design a recursive algorithm that computes the minimum penalty by filling in the details of the following algorithm outline.

```
Opt(i) :
// Compute minimum penalty for a trip that starts at a_0 and ends at a_i,
//    where i ∈ [0..n]
    if i = 0 then
        result ← 0
    else
        result ← ∞
        for k in [0..i) do
            penaltyForLastDay ← [                    ]
            penaltyForPreviousDays ← [               ]
            result ← min(result, penaltyForLastDay + penaltyForPreviousDays)
    return result
```

(b) The distinct subproblems for this algorithm are $Opt(i)$ for $i \in [0\,..\,n]$. Show how to modify your algorithm from part (a) using "memoization" to get an efficient algorithm. Estimate the running time of your algorithm. Your algorithm should be a recursive algorithm that fills in a table $T[0\,..\,n]$, where each $T[i]$ is initialzed to $\perp$ (meaning undefined), and eventually gets set to the value $Opt(i)$.

(c) Next, show how to turn the algorithm from part (b) into an iterative algorithm, and estimate the running time of your algorithm.

(d) Finally, give an algorithm that actually computes an itinerary that yields the minimum penalty, and estimate the running time of your algorithm. To do this, assume you have already pre-computed the table $T[0\,..\,n]$ as in parts (b) or (c).

4. **Another tiling variation.** Consider the following variation of the tiling problem discussed in class. We are given "target" string t and several strings ("tiles") $s_1, \ldots, s_k$. You are to design an algorithm that determines whether or not the target t can be obtained by concatenating several tiles in any order, possibly using some tiles more than once. That is, your algorithm should determine whether or not there exist indices $i_1, i_2, \ldots, i_\ell$, *not necessarily distinct* and in *any order*, such that $t = s_{i_1} \,||\, s_{i_2} \,||\, \cdots \,||\, s_{i_\ell}$. Here, $||$ denotes string concatenation. Your algorithm does *not* have to compute the indices themselves (if they exist).

   Your algorithm should run in time $O\big(|t| \cdot (\sum_j |s_j|)\big)$. Here, $|\cdot|$ denotes the length of a string. You should argue (briefly) why your algorithm runs in the stated time.

   For example, if $t = \texttt{abracadabra}$, $s_1 = \texttt{ra}$, $s_2 = \texttt{ab}$, $s_3 = \texttt{ca}$, $s_4 = \texttt{d}$, $s_5 = \texttt{cada}$, then $t = s_2 \,||\, s_1 \,||\, s_3 \,||\, s_4 \,||\, s_2 \,||\, s_1$, and so your algorithm would output *true* on this input.

   Hint: Use Dynamic Programming. Approach the problem as in Problem 3: first design a recursive algorithm, identify the subproblems, and then memoize. However, you don't need to give an iterative algorithm, and you don't even have to give the details of memoization.

5. **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 $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$. Consider two genes (strings) x and y. An *alignment* of x and y is a way of matching up these two strings by lining them up in columns, possibly with gaps.

   For example, if $x = \texttt{ATGCC}$ and $y = \texttt{TACGCA}$, here are three possible alignments of x and y:

$$\begin{array}{|c|c|c|c|c|c|c|} \hline \texttt{-} & \texttt{A} & \texttt{T} & \texttt{-} & \texttt{G} & \texttt{C} & \texttt{C} \\ \hline \texttt{T} & \texttt{A} & \texttt{-} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{A} \\ \hline \end{array} \qquad \begin{array}{|c|c|c|c|c|c|c|c|} \hline \texttt{A} & \texttt{T} & \texttt{-} & \texttt{-} & \texttt{G} & \texttt{C} & \texttt{C} & \texttt{-} \\ \hline \texttt{T} & \texttt{-} & \texttt{A} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{-} & \texttt{A} \\ \hline \end{array} \qquad \begin{array}{|c|c|c|c|c|c|c|c|} \hline \texttt{A} & \texttt{-} & \texttt{T} & \texttt{-} & \texttt{G} & \texttt{-} & \texttt{C} & \texttt{C} \\ \hline \texttt{-} & \texttt{T} & \texttt{A} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{A} & \texttt{-} \\ \hline \end{array}$$

   Here, the "-" character indicates a "gap". The characters of each string must appear in order, and no column can contain two gaps. For instance, the following are not valid alignments of x and y:

$$\begin{array}{|c|c|c|c|c|c|c|} \hline \texttt{-} & \texttt{T} & \texttt{A} & \texttt{-} & \texttt{G} & \texttt{C} & \texttt{C} \\ \hline \texttt{T} & \texttt{A} & \texttt{-} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{A} \\ \hline \end{array} \qquad \begin{array}{|c|c|c|c|c|c|c|c|} \hline \texttt{A} & \texttt{T} & \texttt{-} & \texttt{-} & \texttt{-} & \texttt{G} & \texttt{C} & \texttt{C} & \texttt{-} \\ \hline \texttt{T} & \texttt{-} & \texttt{-} & \texttt{A} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{-} & \texttt{A} \\ \hline \end{array}$$

   The first example is invalid because the letters of x are out of order. The second is invalid because the third column contains only gaps.

   Any given alignment has a *cost*. The cost of an alignment is defined to be the sum of the costs of its columns. A column with two matching letters has a cost of 0. A column with a gap in one position has a cost of 1. A column with two mismatched letters has a cost of 2. For instance, the alignment

$$\begin{array}{|c|c|c|c|c|c|c|} \hline \texttt{-} & \texttt{A} & \texttt{T} & \texttt{-} & \texttt{G} & \texttt{C} & \texttt{C} \\ \hline \texttt{T} & \texttt{A} & \texttt{-} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{A} \\ \hline \end{array}$$

   has a cost of $1 + 0 + 1 + 1 + 0 + 0 + 2 = 5$. Note that a *lower* cost represents a *better* alignment. An *optimal alignment* is an alignment of least cost.

   Give an algorithm that takes as input two strings $x[1\,..\,m]$ and $y[1\,..\,n]$, and returns an optimal alignment for $x$and y. The running time should be $O(mn)$. You should argue (briefly) why your algorithm runs in the stated time.

   Hint: Use Dynamic Programming.

- Approach the problem as in Problem 3: first design a recursive algorithm that just finds the cost of the optimal alignment, but not the alignment itself. To design your recursive algorithm, think about what the last column in any alignment may look like.
- Identify the subproblems and memoize. You don't need to give an iterative algorithm, and you don't even have to give the details of memoization.
- Finally, give an efficient algorithm that computes an optimal alignment, given a table of optimal cost values for all (relevant) subproblems. You may represent an optimal alignment as a sequence of pairs, where each pair represents the two characters in a given column.

Note: The problem of finding optimal alignments also arises in many areas, including spell checking, speech recognition, plagiarism detection, file revisioning, and computational linguistics.

6. **Load balancing on 3 machines.** Consider again the load balancing problem from class. In this problem, you have m machines and n jobs, where job j has running time $t_j > 0$ for $j = 1, \ldots, n$. The goal is to assign jobs to machines so that the max load is minimized. Here, the load of any one machine is the sum of the running times of the jobs assigned to it, and the max load is the maximum of the loads of the machines.

   Your task is to design an algorithm to compute the optimal max load for the case where $m = 3$. You may also assume that each t_j is a positive integer. Your algorithm only needs to compute the optimal max load, and not an assignment.

   Your algorithm should run in time $O(nt^2)$, where $t := \sum_{j=1}^{n} t_j$. You should argue (briefly) why your algorithm runs in the stated time.

   Hint: Use Dynamic Programming. Approach the problem as in Problem 3: first design a recursive algorithm, identify the subproblems, and then memoize. However, you don't need to give an iterative algorithm, and you don't even have to give the details of memoization.

   Hint: Unlike the 2-machine problem discussed in class, you should not attempt to reduce this to subset sum.

7. **Road trip (II)** ***[Food for thought (will not be graded)].*** Consider again Problem 3. Suppose at each mile post $a_0, a_1, \ldots, a_n$ there is a gas station. You start out with u units of gas in your gas tank at mile post a_0 and you want to get to mile post a_n. Your car's gas tank has a capacity of c units of gas and your car gets R miles per unit of gas. You can stop at any mile post along the way and buy as much gas as you like. However, the following constraint must always be satisfied:

   The amount of gas in your tank should never go below 0 *or above* c. $(*)$

   (a) Suppose the goal is to plan your trip so that you make as few stops as possible along the way to buy gas. Consider the following simple greedy strategy:

   while you are not at the destination do
   fill up your tank at the current location
   drive to the farthest gas station you can reach on a full tank of gas

   Prove that this strategy results in an itinerary that minimizes the number of stops.

   Hint: prove by induction on n.

   (b) Now assume that the gas stations sell gas at different prices, and that the goal is to plan the trip to minimize the total amount of money you spend on gas. To this end, assume that gas costs $p_i > 0$ dollars per unit of gas at mile post a_i, for $i = 0, \ldots, n-1$, and set $p_n := 0$.

   Consider the following greedy strategy:

   while you are not the destination do
   (1) if you could reach a gas station with a cheaper price on a full tank of gas, then:
   buy the minimal amount of gas (possibly none) needed to the reach the first such station, and drive to it;
   (2) otherwise, fill up your tank and drive to the next gas station

   Prove that this greedy strategy is optimal.

   Hint: prove by induction on n. To do this, you might proceed as follows. Consider an input $I = (u; a_0, \ldots, a_n; p_0, \ldots, p_n; c, R)$, where $n > 0$, and a strategy X on input I. The goal is to show that X is no better than the greedy strategy. To do so, it is helpful to first transform X into a strategy that is

somewhat similar to the greedy stategy. Specifically, in the first loop iteration of the greedy strategy, either case (1) or case (2) applies. For case (1), transform X into a strategy Y_1 for input I that is at least as good as X, and which buys the same amount of gas at a_0 as greedy, and does not buy any gas at the subsequent stations up to but not including the cheaper one. For case (2), transform X into a strategy Y_2 for input I that is at least as good as X, and which buys the same amount of gas at a_0 as greedy. After you do this, it should be straightforward to apply the induction hypothesis to a smaller input $\widetilde{I}$, and conclude that X is no better than the greedy strategy on input I. See the "general correctness proof strategy" from the class notes on greedy algorithms.

(c) Suppose that the greedy algorithm in part (b) is modified so that case (1) reads as follows:

> (1) if you could reach a gas station with a cheaper price on a full tank of gas, then:
> buy the minimal amount of gas (possibly none) needed to the reach the *cheapest* such station, and drive to it;

Show that this strategy is not optimal. You should do this by providing an explicit counter-example.