1. **MST Mechanics.** Consider the following weighted, undirected graph:

(a) Assume we run Prim’s MST algorithm starting at vertex A. List the edges that get added to the tree in the order in which the algorithm adds them.

(b) Now do the same thing for Kuskal’s algorithm.

2. **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.

$$\text{Penalty}((a_0, a_1, \ldots, a_n)) :$$

if $n = 0$

\[ \text{result} \leftarrow 0 \]

else

\[ \text{result} \leftarrow \infty \]

for $i$ in $[0..n]$ do

\[ \text{penaltyForLastDay} \leftarrow \] blank

\[ \text{penaltyForPreviousDays} \leftarrow \] blank

\[ \text{result} \leftarrow \min(\text{result}, \text{penaltyForLastDay} + \text{penaltyForPreviousDays}) \]

return result

(b) Next, identify the subproblems and describe the subproblem graph. In particular, estimate the total number of subproblems, as a function of $n$.

(c) Next, show how to modify your algorithm from part (a) using “memoization” to get an efficient algorithm. Estimate the running time of your algorithm.

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

(e) Finally, show how to modify your algorithm from part (d) to actually compute an itinerary that yields the minimum penalty, and estimate the running time of your algorithm.
3. **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 “itwasthebestoftimes...”). 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 \( w \), \( \text{dict}(w) = \text{true} \) if \( w \) is a valid word, and \( \text{dict}(w) = \text{false} \) 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.

*Hint:* Use Dynamic Programming. Approach the problem as in Problem 2: first design a recursive algorithm, identify the subproblems, and then memoize (you don’t need to give an iterative algorithm).

4. **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 \( r_1 r_2 \cdots r_n \). 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 \( (1 - r_i)^{m_i} \), 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.

*Hint:* Use Dynamic Programming. Approach the problem as in Problem 2: first design a recursive algorithm, identify the subproblems, and then memoize (you don’t need to give an iterative algorithm).

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 \( \Sigma = \{A,C,G,T\} \). 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:

\[
\begin{align*}
&\text{ATGCC} \\
&\text{TACGCA}
\end{align*}
\]

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 \( O(mn) \).

*Hint:* Use Dynamic Programming. Approach the problem as in Problem 2: first design a recursive algorithm, identify the subproblems, and then memoize (you don’t need to give an iterative algorithm).

6. **Interval scheduling with weights.** Consider the interval scheduling problem discussed in class. In that problem, we wanted to scheduling as many non-overlapping jobs as possible, and we saw that a greedy algorithm based on the “earliest finish time first” strategy worked.

Now suppose that each job \( j \) has a non-negative weight \( w_j \), and the goal is to schedule a set \( A \) of non-overlapping jobs such that the sum of weights \( \sum_{j \in A} w_j \) is maximized.

(a) Show that the greedy algorithm based on “earliest finish time first” does not work in this setting. You should give a counter-example for which the greedy algorithm fails to find the best solution.

(b) Show how to efficiently solve this problem using Dynamic Programming. As in Problem 2, first design a recursive algorithm, identify the subproblems, and then memoize.

7. **Huffman code mechanics.** Consider the following frequencies on an alphabet of 27 characters (a-z plus a “blank”).
Show the Huffman encoding tree arising from this data.

8. **Morse Code.** A *Morse code* is like a Huffman code, except that we drop the prefix-freeness requirement. To use a Morse code in practice, one needs to insert a “blank” between each encoding.

Suppose we have $n$ letters and for $i = 1,\ldots,n$, letter $i$ occurs with probability $p_i$. Since we do not require prefix-freeness, we may as well use the $n$ shortest possible bit-strings to encode our $n$ letters:

$$0, 1, 00, 01, 10, 11, 000, \ldots$$

To simplify notation, let $\ell_i$ be the length of the $i$th bit string in this ordering. Note that the $\ell_i$'s form a non-decreasing sequence.

We want to assign letters to bit strings. Mathematically, such an assignment is a permutation on the index set $\{1,\ldots,n\}$, where we assign letter $\pi(i)$ to the $i$th bit string. For such an assignment $\pi$, the average cost of transmitting a series of letters with the given probability distribution is

$$\text{Cost}_\pi := \sum_{i=1}^{n} \ell_i p_{\pi(i)}.$$ 

We want to prove that the “greedy strategy” of assigning the most likely letters to the shortest encoding minimizes $\text{Cost}_\pi$. That is, we want to show that an optimal assignment is one where $p_{\pi(i)} \geq p_{\pi(i+1)}$ for all $i$.

(a) Prove the following “swapping property”: if $\pi$ is an assignment such that $p_{\pi(i)} \leq p_{\pi(i+1)}$ and $\pi'$ is an assignment that is the same as $\pi$ except $\pi'(i) := \pi(i+1)$ and $\pi'(i+1) := \pi(i)$, then $\text{Cost}_{\pi'} \leq \text{Cost}_{\pi}$.

(b) Use part (a) to show that the greedy strategy is optimal.