Problem Set 3

Solution Key

Problem 1

In order to demonstrate the numerous possible approaches to problem 1, we present 2 different solutions. The first uses a window size of 5, while the second uses a window size of 4.

a. Consider as input a sequence of cars $[c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9, c_{10}]$ where every car is compatible with every other car. If you have decided to place c_7 in parking spot number 5 then what cars can be parked before slot 5 and what cars can be parked after slot 5?

   Cars that can go before slot 5: $c_1, c_2, c_3, c_4, c_5, c_6$
   Cars that can go after slot 5: $c_4, c_5, c_6, c_8, c_9, c_{10}$

b. Let a *window* be a set of 4 consecutive parking spots $i-3$, $i-2$, $i-1$, i, where $4 \leq i \leq n$. Assume you know which cars are parked in the window (i.e. slots $i-3$, $i-2$, $i-1$,i). Show that, for all cars you know whether they are parked before slot $i-3$ or after slot i.

   By the constraints of this problem, a car can move at most 2 parking spots from where it is initially located. There are four cases for the initial location of a car.
   Case 1: The car was initially located before the window.
   The car must remain before the window since crossing the window would involve moving at least 5 spots.
   Case 2: The car was initially located after the window.
   By the same logic as above, the car must remain after the window, since crossing the window would involve moving at least 5 spots.
   Case 3: The car was initially located in the first or second spot of the window
   The car will be located before the window, since crossing to the other side would involve moving 3 or 4 spots.
   Case 4: The car was initially located in the first or second spot of the window.
   By the same logic as Case 3, the car will be located after the window since crossing to the other side would involve moving 3 or 4 spots.
   Thus, in all cases the location of all non-window cars can be determined to be either before or after the window.

c. Design a dynamic programming algorithm for the Car Parking problem which takes as input the initial sequence of cars $[c_1, \cdots, c_n]$ and the compatibility matrix P and outputs *true* if a feasible sequence of cars exists, and otherwise outputs $false$ in $O(n)$ time.
   For this algorithm *I will assume that length(cars) > 0.* For the algorithm (and the recurrence relation), I will use the following helper procedures:

```
TAKE($list, n$)
    return list with first n elements
```

```
DROP($list, n$)
    return list with first n elements removed

VALID-SET($prev, curr, n$)
    ▷ we will be comparing all but the last element of current
    $compare \leftarrow$ TAKE(curr, n-1)
    ▷ returns true if ∃ list in prev where the last 4 characters are the first four characters of curr
    for each list in prev
    ▷ compare last $n-1$ elements of prev to $compare$
        do if $compare$ equals DROP($list, 6-n$)
            then return True

    ▷ otherwise
    return False

POSSIBLE-PERMS($cars, P, prev, n$)
    perms ← Generate all permutations of n elements from cars
    List $output$
    for each elt in perm
        do if ∀ neighbor cars, $a, b$ in elts, P[a,b] does not equal 0 and
           ∀ cars, car $a$ is no more than two spaces from initial location and
           VALID-SET($prev, elt, n$)
              then output.add(elt)

    return output
```

Recurrence Relation:
where S is the set of cars, P is the compatibility matrix, and R is the output list, where

$$T(S,P,R) = \begin{cases} True \text{ if } length(S) < 5 \text{ and } R \text{ does not equal } \emptyset \\ False \text{ if } length(S) < 5 \text{ and } R \text{ equals } \emptyset \\ T(\text{DROP}(S,1), P, \text{POSSIBLE-PERMS}(\text{DROP(S,1), P, R})) \quad \text{otherwise} \end{cases}$$

A 1-D array will be used for DP. The possible 5-entry permutations for permutations starting at index i will be stored at index i in the array. The array will be filled from index 0 to length - 1. To improve for space-efficiency, we could simply keep the last result.

```
COMPATIBLE-CARS($cars, P$)
    ArrayList validCars
    validCars[0] = possiblePerms(cars, P, 0)
    ▷ visit every 5 element window
    for index equals 1 to (length(cars) - 5)
        do validCars[index] = POSSIBLE-PERMS($cars, P, index, validCars[index-1]$)

    if validCars[index - 1] equals ∅
       then return False
       else return True
```

d. Prove the correctness of your algorithm

Proof. To prove correctness, we will show three things:

1 All valid sequences are covered

2 For any car list and corresponding compatibility matrix if any valid sequence exists, the algorithm will always return true, and

3 For any car list and corresponding compatibility matrix if no valid sequence exists, the algorithm will always return false.

1 All valid sequences are covered (We will prove by induction)
(note that, by assumption, $n = 0$ initial case is not possible)
Base Case: $0 < length(S) \leq 5$
By POSSIBLE-PERMS, all valid permutations will be generated and thus all valid sequences will be covered.
Now suppose that the possible permutations for the first n elements have been generated,
Inductive Case: $length(S) > 5$ At each step of the algorithm, we find all permutations of 5 elements such that they are compatible with direct neighbors, contain only elements within 2 units from their initial location, and the first $n-1$ elements align with the last $n-1$ elements of one permutation from the nth call. As shown in part (b), given a window of 4 (or 5) it is always possible to know whether or not a car can occur before or after the window. Thus, for each window there are a limited (constant) number of cars that can occur in the window and all others must occur before (handled in a $0 \cdots n$ case) or after (handled in a future case). So, any possible 5-element window starting at index $n+1$ will be considered and the recursive procedure will result in all valid sequences.

2 For any car list and corresponding compatibility matrix if any valid sequence exists, the algorithm will always return true (a false-negative is not possible)
By the proof above, we know all valid sequences will be discovered. Thus, the first case of our recurrence relation is all that needs to be considered. By its definition, when $length(S) = 0$, if the last element of the results array is non-empty, the algorithm returns True. Since all valid sequences for the last elements 5 will have been generated in the last element of this array, we will only return true if a valid sequence exists.

3 For any car list and corresponding compatibility matrix if no valid sequence exists, the algorithm will always return false (a false-positive is not possible).
At each step of the algorithm, only permutations where each car is within 2 spaces from initial location and where the neighbors are compatible are kept. Additionally, any permutation that could not be appended to a permutation from the previous list of valid permutations is removed. Thus, a false positive is not possible since the last element of R will be empty when $length(S) = 0$ if no such sequence is possible.

□

e. Prove that the running time of your algorithm is $O(n)$.

Proof. COMPATIBLE-CARS traverses an array, *validCars*, of length n. For each index in the array it calls POSSIBLE-PERMS which generates the permutations and iterates through them, calling VALID-SET. VALID-SET iterates through the 5-element (constant-length) permutation and does a constant-time check (using $O(1)$ procedures Take and Drop) at each step, thus it is $O(1)$ For, Possible-Perms since the window is of constant length (5), there will always be a constant number of permutations and traversing through these permutations retains constant time. Thus, since Valid-Set and Possible-Perms can run in $O(1)$ time, Compatible-Cars can run in $O(n)$ time. □

Solution 2:

a. If c_7 is in slot 5 then c_1 through c_6 can be before slot 5 and c_4 through c_{10} (minus c_7 because it is at 5) can be after slot 5.

b. Every car must have originally lay to the left of the window, in the window, or to the right of the window. If a car originally lay to the left of the window, it can either continue to be to the left of the window, or it can fall in the window. It cannot be to the right of the window because that would require a jump of at least 4 spots, so that cannot happen.

   The cars that originally fell to the right of the window can either stay to the right or jump to the window. They cannot go to the left of the window because that would require a jump of greater than 2 slots.

   If a car originally fell in the middle of the window, it can either stay in the middle of the window, or it can go to one of the sides. If the car fell in either $i-4$ or $i-3$ then it must either stay in the window or go to the left because a jump to $i+1$ would require moving at least 3 spots, which is too much. The same logic applies to cars in $i-1$ and i. They cannot move to the left of the window because it would require a jump of at least 3. They can either remain in the window or jump to the right of the window.

   So, if something fell in spot $i-2$ or further to the left, it is either in the window or to the left. If something fell in spot $i-1$ or further to the right, it is either in the window or to the right. And if we know what is in the window then we know where the rest fall.

c. The idea behind this algorithm is that you divide the array into windows of size 4, each overlapping with its neighbors on either end by 2, and compute every valid arrangement of cars within that window in isolation. Then, we take the valid windows and we want to determine whether or not there exists some sequence of overlapping windows such that the overlaps "agree." By this, I mean that the two windows agree on the placement of spots 3 and 4 in the given six-spot unioned window.

   The first thing we need to do is come up with a list of lists in which we store all the permutations of cars within every window of size 4 starting at every even index in the array.I will not go into the details about this function because they are relatively uninteresting. The point is that for an input of 6 cars in the initial array, you get an output that looks something like this:

window 0	$\{c_1, c_2, c_3, c_4\}$	$\{c_2, c_1, c_3, c_4\}$	$\{c_3, c_1, c_2, c_4\}$	...
window 2	$\{c_3, c_4, c_5, c_6\}$	$\{c_4, c_3, c_5, c_6\}$	$\{c_5, c_3, c_4, c_6\}$	...
...	...	...	...	...
window i	$\{c_{2i}, c_{2i+1}, c_{2i+2}, c_{2i+3}, c_{2i+4}\}$			...
...	...	...	...	...

The width of the row for window i is of size 1 on purpose. There can be varying lengths of the lists of permutations, as the compatability matrix may be more restrictive for certain cars than for others.

Note that if the array of cars is of an odd length we will have trouble when we get to the end of our array. However, we can add a special check case that creates one last window of size 3 if the length of the grid is odd and our DP algorithm would work just as well as if it were even.

Now we have to come up with a sequence of windows that agree throughout the entire length of the array of cars. If we cannot find a sequence then we return false. If we can, we return true.

We do this by taking our list of valid permutations of isolated windows and iterating through them window by window (starting at window 0) and removing the permutations for a given window k for which there does not exist a single permutation of window $k-1$ such that k and $k-1$ "agree." So we iterate from top left to bottom right of our grid and a given spot $G[i,j]$ (after it has been processed) will represent the existence a valid sequence of windows ending in the window at $G[i,j]$

Our recursive function V takes three parameters: *rest*, *curr*, and *prev*. The parameter *rest* is the list of isolated valid permutations for every window after a given window k. The parameter *curr* is the list of valid permuatations of window k taken in isolation, and *prev* is the list of permutations of window $k-1$ that are both valid in isolation and valid ends of sequences of windows 0 through $k-1$. We will assume the existence of a function *rem* that takes two lists of permutations, *curr* and *prev*, and removes the permutations of *curr* that agree with nothing in *prev*. We will assume that we have a built a special case into this function that will simply return *curr* if *prev* is empty, and that is to ensure our first window permutations are all valid.

Here is what our recursive function looks like:

V(*rest*, *curr*, *prev*) =

$$\begin{cases} False & : \text{if } rem(curr, prev) \text{ is empty} \\ True & : \text{if } rest \text{ empty} \\ V(rest[1::], rest[0], rem(curr, prev)) & : \text{otherwise} \end{cases}$$

We will now write this in pseudocode:

Input: Grid G of size $P \times \frac{n}{2}$ where P is the maximum number of valid permutations of of a given window of size 4 and n is the length of the array of cars, so $\frac{n}{2}$ is the number of windows in the array. This contains the valid permutations of all the windows in isolation at the beginning of the program

Output: True or false value representing whether or not there is a valid sequence of windows that "agree" with each other.

```
(1)HasValidSequence($G$):
(2)    for $i = 1$ to $length(G)$
(3)        $curr = G[i]$
(4)        $prev = G[i-1]$
(5)        for $j = 0$ to $length(curr)$
(6)            if $curr[j]$ does not agree with $prev[k]$ for any $k \in length(prev)$:
(7)                remove $curr[j]$
(8)    return False if the last row of G is empty, True otherwise
```

d. Our graph G contains every possible valid premutation of every window of size 4, separated by 2. We do not need to prove that this is correct because our windows are of a constant size and they can be brute-forced in constant time. The only thing that we need to prove the truth of is our DP step.

We will prove this by induction. Our base case is the top row of G. We know that every permutation in the top row of G is valid so every sequence of windows ending in one of those permutations must be valid, since these windows are the first and only windows in the series and they are valid in isolation by construction.

For our inductive step, we are going to compute the validity of a given permutation $G[i, j]$, assuming that the validity has been correctly computed for all rows from $G[0]$ to $G[i-1]$. There are two options for the row $G[i-1]$. Either it contains at least one permutation or it does not. If it contains no permutations then our algorithm will remove the permutation at spot $G[i, j]$ because line 6 and 7 in the pseudocode remove all permutations in row i that agree with nothing in row $i-1$. And if there is nothing in $i-1$ then every permutation in row i will agree with nothing in $i-1$ so they will all get deleted and our function will return false.

If there was at least one permutation in row $i-1$ then there was a valid sequence ending in at least one permutation of window $i-1$. Let's say that valid permutation was at spot k. There are two options at this point. Either $G[i, j]$ does not "agree" with $G[i-1, k]$ or it does.

If they do not "agree" then $G[i, j]$ and $G[i-1, k]$ cannot be adjacent in the series because they have assigned different cars to the same spots. So if you were to continue with this sequence you would have placed two cars in one spot, which is bad, so line 6 checks for this and removes permutations that force this situation.

If they do agree then we know that this is a valid sequence ending in window i. To prove this, let's assume that it is not. Let's assume that somewhere between 0 and i there is a duplicated car (a skipped car cannot happen without a duplicate taking the spot that the skipped car would have occupied, so just checking for duplicates is sufficient). This mistake has to be just between somewhere in the 6 spots in the window $k+i$ because we assumed that 0 through k

was valid by the inductive hypothesis.

That must mean that the duplicate is somewhere in spots 1 through 6 of the window $k + i$. Every duplicate consists of two elements. So let's figure out the necessary placements of these two elements. No element of the duplicate can sit in spots 3 or 4 because if they did then the other element would have to lie in the same window (either i or k), and that cannot be the case because each window was valid within itself (contained no duplicates). So one element must be in either 1 or 2 and the other must be in 5 or 6. But that cannot be the case because any car that can go in spot 2 cannot jump all the way to spot 5 because that is a jump of 3. So there must be no duplicates in $k + i$. So our assumption must be false and this is a valid sequence ending in window i. And because we proved the base case and the inductive step, we know that this algorithm produces true if a sequence exists and false if one does not exist.

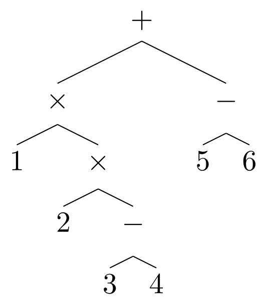
e. There are two steps in this DP algorithm. The first is building the grid G. This takes time $P \times n$ where P is a large constant. We are just getting all valid permutations, which is a (large) constant-time (P is the constant) operation, for every even location in the grid of size n. So this function takes time $P \times \frac{n}{2}$, and the $\frac{P}{2}$ is a constant so this becomes $O(n)$.

   Now we think about the DP part of the algorithm. Our grid is a maximum of size $P \times \frac{n}{2}$. For every spot after the first row we do a constant amount of work per spot. That constant is, once again, P because for every location in G we look at every element in the row above it. So for every spot in G we do a maximum of P operations (it could be less because we're deleting as we go), which is just a constant. And the size of our grid G is $P \times n$ so we do $P \times P \times n$ operations throughout this part of our algorithm. And since P is a constant, P^2 is a constant, so this runs in $O(n)$

   So the whole thing runs in $O(n + n)$ or $O(n)$

Problem 2

a. $w_1 = 1 - 2 + 3 \times 4 \times 6 + 7$ will not be evaluated correctly on the embedded system.
   $w_2 = 1 \times 2 \times 3 - 4 + 5 - 6$ will be evaluated correctly on the embedded system, with the following parse tree:



b. $$T(i,j) = \begin{cases} \emptyset, & \text{if } j < i \\ \{A\}, & \text{if } i = j \\ \displaystyle\bigcup_{k=i}^{j-1} f(\mathrm{T}[i][k], \mathrm{S}_{k,k+1}, \mathrm{T}[k+1][j]), & \text{otherwise, with } f \text{ described below} \end{cases}$$

We create a 2-D array, where both the rows and columns of the array are indexed by the number of integers in the input expression w. An entry $T(i,j)$ in the array represents the set of possible nonterminals that can generate the $(i\text{-}j)^{th}$ substring of w. That is, it is the set of nonterminals to which we can apply a sequence of rules from AAE such that the i^{th} through j^{th} integers of w (inclusive), and all symbols between them, are generated. To determine whether w will be correctly evaluated by the embedded system, we have to check to see if the nonterminal A (which the start symbol generates) is contained in the upper right corner of the array, since that means A can generate the $(0\text{-}(|w|-1))^{th}$ substring of w, which is simply the entire string.

We will fill the array along successive diagonals, from the top left to the bottom right, starting with the main diagonal. At each position, if $i = j$, then the nonterminal A can generate that substring (since it is just a single integer). Else, we must examine all possible pairs of sub-substrings between i and j, splitting on each possible symbol between them. For each pair of sub-substrings, the function f maps two sets of nonterminals and a symbol to a set of nonterminals, as follows. We scan all rules of the form $X \rightarrow YoZ$, where X, Y, Z are nonterminals and o is an operation (there are three of these rules in the AAE grammar). For each rule, we check first to see if the operator is the same as the operator on which we split (check if $o = S_{k,k+1}$, the symbol between the k^{th} and $(k+1)^{st}$ integers of w). If it is, we then check to see if Y can generate the sub-substring to the left of o, and Z can generate the sub-substring to the right. If they can, then we know we can generate the entire substring, YoZ, with the nonterminal X. f returns the set of nonterminals X which can generate the substring, splitting on $S_{k,k+1}$, and we union all sets returned by f $\forall k : i \leq k \leq j-1$ (assuming union and set-add excludes repeats). The algorithm is as follows (note we assume valid input of the form "N o N o N"..., where N is a number and o is a symbol):

```
ARITH-EVAL($w$)
 1  int $n = |w|$                    ▷ $|w|$ is the number of *integers* in input expression $w$.
 2  T[$n$][$n$]                      ▷ Table used for Dynamic Programming (0-based).

 3  for $i \leftarrow 0$ to $n-1$
 4        do for $j \leftarrow 0$ to $n-1$
 5              do T[$i$][$j$] = EMPTY-SET
                                     ▷ Initialize each cell to an empty set. $O(n^2)$ time.
 6  for diff $\leftarrow 0$ to $n-1$        ▷ diff is the difference between $i$ and $j$.
 7      do for $i \leftarrow 0$ to $n-1$
                                     ▷ $O(n^2)$ possible substrings.
 8        do if $i$ + diff $< n$
 9            then $j = i$ + diff
10                if $i == j$
11                  then T[$i$][$j$] = $\{A\}$
12                  else  for $k \leftarrow i$ to $j-1$
                                     ▷ $O(n)$ work for each substring.
13                          do for each rule $R = X \rightarrow YoZ \in AAE$
14                              do if $o = S_{k,k+1}$
                                 and SET-CONTAINS(T[$i$][$k$], $Y$)
                                 and SET-CONTAINS(T[$k+1$][$j$], $Z$)
                                   then SET-ADD(T[$i$][$j$], $X$)
                                     ▷ Where $S_{k,k+1}$ is the symbol between the
                                     ▷ $k^{th}$ and $(k+1)^{st}$ integers in $w$. Basic
                                     ▷ set functions assumed; contains is linear in length
                                     ▷ of set (at most 2 in $AAE$), add is constant.
15  return SET-CONTAINS(T[0][$n-1$], $A$)
                                     ▷ If $A$ (start symbol) is contained in the set of
                                     ▷ nonterminal that can generate the entire string,
                                     ▷ then return true. Else, return false.
```

c. *Proof of Correctness.* We will prove the correctness of our algorithm by proving two things. First we show that our recurrence relation correctly finds all nonterminals that can generate an $(i\text{-}j)^{th}$ substring for arbitrary (i, j), and second that our algorithm both terminates and examines the correct final (i, j). We show the first by induction on i and j.

First, we can assert that there are no nonterminals that generate substrings where $j < i$, since it does not make sense to have a substring where the end character is before the start character. Next, when $i = j$, we know that the nonterminal A (and only that nonterminal) can generate the single integer which is the i^{th} integer of w. Thus, our recurrence relation is correct in the base case.

Assume that our recurrence relation correctly finds nonterminals generating all $(i\text{-}j)^{th}$ substrings, $\forall(i, j) : i = i'$ and $j < j'$ **or** $i > i'$ and $j = j'$, for some $i' > 1, j' > 1$ (in terms of the algorithm, we have correctly filled all cells to the left and below the $(i'\text{-}j')^{th}$ cell). Assume now that our recurrence relation does **not** find all nonterminals that can generate the $(i', j')^{th}$ substring. Call the nonterminal not found X. If X can generate the $(i', j')^{th}$ substring, there must be a rule $X \rightarrow YoZ$, for some nonterminals Y and Z, such that o is a symbol somewhere between the $(i')^{th}$ and $(j')^{th}$ integers. However, our algorithm splits on **every** symbol between the $(i')^{th}$ and $(j')^{th}$ integers. Thus, at some point, our algorithm will split the substring into two sub-substrings on o. Furthermore, if $X \to YoZ$, then Y must generate the sub-substring to the left of o (namely the $(i'\text{-}k)^{th}$ substring, for $k < j'$), and Z must generate the sub-substring to the right of o (namely the $((k+1)\text{-}j')^{th}$ substring, for $k+1 > i'$). However, by inductive hypothesis, we have found **all** nonterminals that generate those sub-substrings, so we know Y can generate the left and Z can generate the right. Thus, at (i', j'), we will necessarily add X to our set of nonterminals, since we examine every rule, and thus will find the rule $X \to YoZ$ when splitting on o. This is a contradiction, and thus by induction we have shown that our algorithm correctly finds all nonterminals for arbitrary (i, j).

Proving the second claim is trivial; our algorithm only loops through the finite dimensional array twice, and since there are a constant number (and thus finite number) of rules, our algorithm will definitely terminate. If T[0][$|w|-1$] contains A, then we know A can generate our entire input w, and thus w will be correctly evaluated by the embedded system. Since our algorithm directly implements our recurrence relation, we have proved our recurrence relation is correct, and that we correctly examine the final (i, j) of our recurrence, our algorithm determines whether w will be correctly evaluated. □

d. *Proof of Runtime.* We will prove the runtime of our algorithm directly. Assume $n = |w|$. Initializing the array used for dynamic programming takes $O(n^2)$ time, because we must initialize each cell in an nxn array. Next, the loops that examine each substring run in $O(n^2)$ time, because they examine every cell of the upper right triangle of the array. We can also reason that there are order n^2 possible substrings for a given input w (each character has order n other characters that can be added to it to make a unique substring), and we must examine each of these substrings. There are $O(n)$ possible pairs of sub-substrings for a given $(i\text{-}j)^{th}$ substring, and we examine each of those pairs. At each examination of a pair, we perform a constant amount of work, since the sets can contain a maximum of two nonterminals (A and B), and there are only three rules which we must examine (of the form $X \to YoZ$). Thus, at each $(i\text{-}j)^{th}$ substring, we do a linear amount of work, to check each pair of sub-substrings that may generate the $(i\text{-}j)^{th}$ substring. The final examination of the last set takes constant time, since the size of the set is at most two. Since we do $O(n^2)$ work to initialize, there are $O(n^2)$ substrings, and we do $O(n)$ work for each substring, the algorithm runs in $O(n^2 + n^2 * n) = O(n^3)$ time. □

e. In the more general case of an arbitrary grammar, we must first convert our grammar into a form similar to AAE. Specifically, we must convert the input rules, AAE', into Chomsky Normal Form (call the new grammar in CNF G'). Rules will then be one of four forms: $S \to S_0$, $S \to \epsilon$, $X \to YZ$, and $T \to t$, where S is the start symbol of G', S_0 was the start symbol for AAE', $T, X, Y, Z \in N_{G'}$ (where N is G''s nonterminal set), and $t \in T_{G'}$ (G''s set of terminals). After this conversion, we can essentially perform the same algorithm as the one above. We no longer need to deal with splitting on arithmetic symbols. Since we have converted our general grammar into a form where each nonterminal can only be replaced by two nonterminals, we only need to check **pairs** of sub-substrings. That is, for every $(i\text{-}j)^{th}$ substring, we only need to split once between each pair of consecutive characters in the substring. If we had not converted our grammar, we may have needed to split our $(i\text{-}j)^{th}$ substring into many different size sub-substrings, to account for the fact that nonterminals could be replaced by any number of characters, which would increase the runtime of our algorithm.

Instead, our rule set will now have an arbitrary number of rules. The runtime of our algorithm will be the runtime of converting AAE' to G' plus the runtime of the algorithm on a grammar in CNF. Converting our grammar to CNF is bound by $O(|AAE'|^2)$, where $|AAE'|$ is the number of rules in the original grammar, for the following two reasons. First, we may generate a linear number of new rules for each of the old rules. Second, fixing rules of the form $M \to N$ takes at most quadratic time, because we need to check to make sure we aren't re-adding rules. Furthermore, the new grammar will contain at most $O(|AAE'|^2)$ rules, by the same reasoning. Call the size of this new grammar $|G'|$. The set contains operations are $O(|G'|)$, since the number of possible nonterminals is bound by the number of rules. Iterating through rules of the form $X \to YZ$ is also $O(|G'|)$. There are still order n sub-substrings for a given $(i\text{-}j)^{th}$ substring, and at each $(i\text{-}j)^{th}$ substring we have to do $O(|G'|)$ work (from above). Thus, the work required to find all nonterminals for a given $(i\text{-}j)^{th}$ substring is bound by $O(n*|G'|)$. There are still $O(n^2)$ possible substrings, and thus the runtime of the algorithm is now bound by $O(n^3 * |G'|)$. Therefore, the overall runtime of the new, general algorithm is bound by $O(|AAE'|^2 + n^3 * |G'|) = O(|AAE'|^2 + n^3 * |AAE'|^2) = O(n^3 * |AAE'|^2)$. The algorithm is broken up into converting our grammar, and then parsing the input. The procedures are as follows:

```
CNF($G$)
1  COPY-GRAMMAR($G_c$, G) ▷ Copy grammar to $G_c$ to reference later. $O(|AAE'|)$.
2  ADD-RULE($G$, $S \to S_0$)  ▷ Add to $G$ the new start rule.
                               ▷ $S_0$ is now considered to be a nonterminal.
     ▷ First, factor our terminals by making rules of the form $T \to t$.
3  for each rule $R \in G : R = A \to \alpha t_i \beta$      ▷ $t_i$ terminal, $\alpha, \beta$ any strings.
4       do ADD-RULE($G$, $T_i \to t_i$)
                               ▷ Add new rule to $G$, where $T_i$ is a new nonterminal.
5          CHANGE-RULE($R$, $A \to \alpha T_i \beta$)   ▷ Replace $t_i$ with $T_i$ in rule $R$.

     ▷ Now were are left with rules of two forms: $T \to t$ ($t$ may be $\epsilon$),
     ▷ and $C \to N_1 N_2 \ldots N_k$ (not including start rule). To fix rules of the second form
     ▷ (generates at most a linear number of new rules for each old rule):
6  for each rule $R \in G : R = C \to N_1 N_2 \ldots N_k, k \geq 3$
7       do ADD-RULES($G$, $\{C \to N_1 C_1,\ C_1 \to N_2 C_2,\ \ldots,\ C_{k-1} \to N_{k-1} N_k\}$)
              ▷ Add set of new rules, where $C_1 \to C_{k-1}$ are new nonterminals.
8          REMOVE-RULE($G$, $R$)      ▷ Remove rule $R$ from $G$.

     ▷ Rules are now in one of four forms: $T \to t$, $A \to B$, $C \to N_1 N_2$, or $X \to \epsilon$.
     ▷ Fix $\epsilon$ rules: (note this will generate the rule $S \to \epsilon \iff \epsilon \in L_G$,
     ▷ where $L_G$ is the language recognized by $G$)
9  for each rule $R \in G : R = X \to \epsilon$
     ▷ Examine all other rules that contain at least one $X$ on the right-hand side.
10      do for each rule $R' \in G : R' = Y \to \alpha X \gamma$
                                    ▷ Note $\alpha, \gamma$ may contain $X$'s.
11             do ADD-RULES($G$, COMBINATIONS($R'$, $X$))
     ▷ Add to the grammar **all** rules that can be generated by replacing
     ▷ any combination of the $X$'s in the RHS of $R'$ with an empty string
     ▷ (except the rule which is the same as $R'$). There will be
     ▷ $2^r - 1$ of these rules, where $r :=$ the number of $X's \in R'$.
     ▷ Since we fixed chaining first, $r$ is at most 2 (constant).
12         REMOVE-RULE($G$, $R$)

     ▷ Rules are now in one of three forms: $T \to t$, $A \to B$, $C \to N_1 N_2$.
     ▷ Consider rules of the second type.
13 for each rule $R \in G : R = A \to B$
14      do for each rule $R' \in G : R' = B \to \omega$
15             do if !SINGLE-NONTERMINAL($\omega$) or !CONTAINS-RULE($G_c$, $A \to \omega$)
16                   then ADD-RULE($G$, $A \to \omega$)
     ▷ To get around re-adding rules of the form $A \to Z$, if $\omega$ is a single terminal,
     ▷ make sure new rule $A \to \omega$ isn't in original grammar.
17         REMOVE-RULE($G$, $R$)
     ▷ We are now done. Rules are of one of the following forms: $S \to \epsilon$, $S \to S_0$,
     ▷ $A \to BC$, $T \to t$, for $S$ start symbol, $N, A, B, C, T$ nonterminals, $t$ terminal.
```

```
GEN-ARITH-EVAL($AAE', w$)
 1  CNF($AAE'$)                    ▷ Convert $AAE'$ to Chomsky.
 2  int $n = |w|$                  ▷ $|w|$ is the number of characters in input expression $w$.
 3  T[$n$][$n$]                    ▷ Table used for Dynamic Programming (0-based).

 4  for $i \leftarrow 0$ to $n-1$
 5      do for $j \leftarrow 0$ to $n-1$
 6             do T[$i$][$j$] = EMPTY-SET
                                   ▷ Initialize each cell to an empty set. $O(n^2)$ time.
 7  for diff $\leftarrow 0$ to $n-1$   ▷ diff is the difference between $i$ and $j$.
 8     do for $i \leftarrow 0$ to $n-1$
                                   ▷ $O(n^2)$ possible substrings.
 9      do if $i$ + diff $< n$
10           then $j = i$ + diff
11                if $i == j$
12                  then T[$i$][$j$] = $\{A\}$
13                  else  for $k \leftarrow i$ to $j-1$
                                   ▷ $O(n * |AAE'|^2)$ work for each substring.
14                          do for each rule $R = X \rightarrow YZ \in AAE'$
15                               do if SET-CONTAINS(T[$i$][$k$], $Y$)
                                 and SET-CONTAINS(T[$k+1$][$j$], $Z$)
                                   then SET-ADD(T[$i$][$j$], $X$)
16  return SET-CONTAINS(T[0][$n-1$], $A$)
                                   ▷ If $A$ (start symbol) is contained in the set of
                                   ▷ nonterminal that can generate the entire string,
                                   ▷ then return true. Else, return false.
```

Problem 3

a. *Give examples shown that the following three greedy algorithms are not correct.*

1. *Ordered by increasing duration*

   Solution. Consider the following jobs:

i	duration $t(i)$	deadline $d(i)$	profit $p(i)$
1	1	1	1
2	2	2	2
3	3	3	3

   In the above case, only one job can be run (the others will all be past their deadlines by the time they finish), so the optimal strategy is to run job 3 first. However, if we order by increasing duration, job 1 is run first.

2. *Ordered by increasing deadline*

   Solution. Using exactly the same example as in the previous section, we see that if we order by increasing deadline we only run job 1, so jobs 2 and 3 cannot run, meaning we get a profit of 1 when we could have a profit of 3.

c. *Ordered by decreasing profit*

Solution. Consider the following jobs:

i	duration $t(i)$	deadline $d(i)$	profit $p(i)$
1	1	1	2
2	1	2	2
3	2	2	3

In the above case, this algorithm will run job 3 first, giving a total profit of 3 (since jobs 1 and 2 will already be past their deadlines). However, if we ran jobs 1 and 2, we would be unable to run job 3, but we would have a total profit of 4.

b. *Show that, there always exists an optimal solution where the included jobs are scheduled by order of increasing deadline.*

Proof. Let $i_1, \dots, i_n$ be an optimal scheduling for a set of jobs I. If $d(i_1) \le d(i_2) \le \cdots \le d(i_n)$, then we are done, so assume there is some $1 \le \ell < n$ such that $d(i_\ell) > d(i_{\ell+1})$. Let

$$T_k = \sum_{j=1}^{k} t(i_j)$$

denote the time at which job k ends (with $T_0 = 0$ by definition). By the definition of the problem, we know that $T_k \le d(i_k)$ for all k. That is to say, all jobs in the solution finish at or before their deadline. In particular, this means

$$d(i_{\ell+1}) \ge T_{\ell+1} = T_{\ell-1} + t(i_\ell) + t(i_{\ell+1}).$$

Thus, if we run job $i_{\ell+1}$ before job i_ℓ, then it will finish at time $T_{\ell-1} + t(i_{\ell+1}) \le d(i_{\ell+1})$, so it will finish before its deadline. If we run job i_ℓ immediately afterward, it will finish at time $T_{\ell-1} + t(i_{\ell+1}) + t(i_\ell) \le d(i_{\ell+1}) < d(i_\ell)$, and thus it will finish before its deadline as well. Thus we have swapped the jobs that's deadlines were out of order and we still ran all of the same jobs, thus netting the same profit, meaning the solution is still optimal.

Because we can sort simply by swapping adjacent pairs that are out of order (bubble sort), this proves that, given any optimal job scheduling, if we sort the included jobs in order of increasing deadline, all included jobs will still finish before their deadlines and we will have an optimal solution. □

c. *Design a dynamic programming algorithm for the Job Scheduling problem which takes as input the jobs $1, \dots, n$ and outputs the maximum profit in $O(n \times \mathit{MaxDeadline})$ time, where $\mathit{MaxDeadline} = \max_{i=1}^{n} d(i)$. Assume that the duration, deadline, and profit functions are defined for each job.*

Solution. First sort the input jobs in order of increasing deadline. We will now index based on the new list. Let T be a two-dimensional $(n+1) \times (\mathit{MaxDeadline}+1)$ table. The entry $T[i,j]$ represents the maximum profit possible using only the first i jobs (earliest i deadlines) and finishing by time j. Since the table is 0-indexed, we have that $T[n, \mathit{MaxDeadline}]$ represents the maximum profit possible with all of the jobs. We fill T using the recurrence relation

$$T[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ T[i, d(i)] & d(i) < j, \\ \max\{T[i-1,j], \\ \qquad T[i-1, j-t(i)] + p(i)\} & t(i) \le j \le d(i), \\ T[i-1,j] & t(i) > j. \end{cases}$$

We thus fill in the table by looping from $j = 0$ to *MaxDeadline* and for each j, for $i = 0$ to n.

This gives us the following pseudocode:

```
JobScheduling($I = \{1, \ldots, n\}$)
 1  if $n = 0$ return 0
 2  Let $I \leftarrow \text{sort}(I)$

 3  for $i \leftarrow 0$ to $n$
 4        do $T[i, 0] = 0$
 5  for $j \leftarrow 0$ to $MaxDeadline$
 6        do $T[0, j] = 0$

 7  for $j \leftarrow 1$ to $MaxDeadline$
 8        do for $i \leftarrow 1$ to $n$
 9              do if $d(i) < j$
10                    then $T[i, j] = T[i, d(i)]$
11                    else  if $t(i) \leq j$
12                             then $T[i, j] = \max\{T[i-1, j], T[i-1, j-t(i)] + p(i)\}$
13                             else  $T[i, j] = T[i-1, j]$

14  return $T[n, MaxDeadline]$
```

d. *Prove the correctness of your algorithm.*

Proof. We claim that the algorithm in part (c) returns the profit for the optimal scheduling where all included jobs run in order of increasing deadline. Hence by part (b), it must be the profit for the overall optimal solution. We prove the claim by proving that the recurrence relation is correct.

First we note that any entry $T[i, 0]$ indicates that all jobs must be finished by time 0. Since it is never possible to complete a job in 0 time, there can never be any profit, so $T[i, 0] = 0$. Similarly, $T[0, j]$ indicates the maximum amount of profit we can achieve in j time while using the first 0 jobs (ordered by increasing deadline). Since there are no jobs we can use, we cannot have any profit, so $T[0, j] = 0$. This proves the base case.

Now consider the cases where $i, j > 0$. If $d(i) < j$, we know that job i is the job with the latest deadline, so no job we are considering can end after time $d(i)$. Thus the maximum profit we can achieve with jobs 1 through i by time j is the maximum profit we can achieve with those jobs by time $d(i)$. Hence $T[i, j] = T[i, d(i)]$ is correct.

Now assume $t(i) \leq j \leq d(i)$. In this case, there are two possible options: either we run job i or we do not.

Run Job i: Note that job i is the job with the latest deadline. So, by part (b), we can run it last and still get an optimal solution (if one exists that runs job i). Moreover, since $d(i) \geq j$, if we run it such that it finishes at exactly time j, it will finish by its deadline and leave the most possible time for other jobs. Thus, we take the profit $p(i)$ from job i and add it to the maximum profit we get by running some subset of jobs $1, \ldots, i-1$, ensuring that they must end by time $j - t(i)$, since job i (which is run last) must be finished by time j. This means the maximum possible profit gained if we run job i is $T[i-1, j-t(i)] + p(i)$.

Don't Run Job i: In this case, our maximum profit is the maximum profit we can get by running some subset of jobs $1, \ldots, i-1$, having them all finish by time j. In other words, our maximum profit is $T[i-1, j]$.

Therefore the maximum possible profit at $T[i, j]$ is the max of those two, which is precisely the term in the recurrence relation for $i, j > 0$ and $t(i) \leq j \leq d(i)$.

Finally, consider what happens when $t(i) > j$. In this case it is impossible to finish job i before time j, even if we start it absolutely first. Thus we cannot run job i at all, so the maximum profit is the maximum profit we get from running some subset of jobs $1, \ldots, i-1$ before time j. In other words, $T[i, j] = T[i-1, j]$.

Hence we have proven by induction that the recurrence relation is correct, so $T[i, j]$ is the maximum profit obtainable by running some subset of the i jobs with the earliest deadlines in the first j timesteps. Since all jobs must be finished by *MaxDeadline* and there are n jobs, we see that $T[n, \mathit{MaxDeadline}]$ must be the maximum possible profit. This completes the proof that the algorithm is correct. □

e. *Prove that the running time of your algorithm is $O(n \times \mathit{MaxDeadline})$.*

Proof. The first step of the algorithm is sorting by deadline. If we use a radix sort, we can sort the list in $O(n \log(\mathit{MaxDeadline}))$ time. Next we note that the size of T is $(n+1) \times (\mathit{MaxDeadline} + 1)$, so the total number of elements in T is $O(n \times \mathit{MaxDeadline})$. For each element of T, the most work we can do is checking two previously-filled elements of T, doing one addition, and taking a max. Hence we do constant work for each cell of T. Thus we have, for some constant C,

$$\mathit{Runtime} \in O\left(n \log(\mathit{MaxDeadline})\right) + C\left(n \times \mathit{MaxDeadline}\right) = O(n \times \mathit{MaxDeadline}).$$

□

Problem 4

a. The sequence of cuts to minimize the cost of cutting a piece of wood of length 10 at positions $\{a_1, a_2, a_3, a_4\} = \{1, 4, 5, 8\}$ is to cut it at a_3 (cost = 10), a_1 or a_2 (cost = 5) and a_4 (cost = 5), and then a_2 or a_1 (whichever was not already cut) (cost = 4), for a total cost of 24.

b. Input: a sequence of cuts $A = a_1, a_2, ..., a_n$, where a_i is the distance from the left edge of the piece of wood, and L, the length of the piece of wood

For the recurrence relation, redefine $A = \{a_1, a_2, ..., a_n, a_{n+1}\} = \{a_1, a_2, ..., a_n, L\}$ such that in the recurrence relation, $a_{n+1} = L$. The recurrence relation for this problem can be described as

$$C[i,j] = \begin{cases} 0, & \text{if } i = j \text{ or } i = j-1 \\ a_j - a_i + \min_{i<k<j}\{C[i,k] + C[k,j]\}, & \text{if } i \neq j \text{ and } i \neq j-1 \end{cases},$$

where $C[i, j]$ is the minimum cost of cutting the wood between cuts a_i and a_j and $i \leftarrow 0$ **to** n and $j \leftarrow 1$ **to** $n+1$.

$C[i, j]$ is a table in which only the upper half is filled (we only need $i \leq j$ since we are looking at the cost of cutting between positions a_i and a_j, so all the other entries would be redundant). Each entry $C[i, j]$ holds the minimum cost of cutting the wood between positions

a_i and a_j, and the table is filled diagonally, from the center diagonal to the upper-right corner.

Algorithm:

```
MIN-COST($\{a_1, a_2, ..., a_n\}, L$)
1  $A \leftarrow \{a_1, a_2, ..., a_n, L\}$
2  for $i \leftarrow 1$ to $n$
3       do
4          for $j \leftarrow 0$ to $n+1-i$
5              do
6                 if $i = 1$
7                    then $C[j, j+i] \leftarrow 0$
8                    else $C[j, j+i] \leftarrow a_{j+i} - a_j + \min_{j<k<j+i}\{C[j,k] + C[k, j+i]\}$
9  return $C[0, n+1]$
```

c. We can prove that the algorithm is correct by showing that the recurrence relation works and each $C[i,j]$ is the minimum cost of cutting between positions a_i and a_j, since the algorithm follows the recurrence relation. The recurrence relation is

$$C[i,j] = \begin{cases} 0, & \text{if } i = j \text{ or } i = j-1 \\ a_j - a_i + \min_{i<k<j}\{C[i,k] + C[k,j]\}, & \text{if } i \neq j \text{ and } i \neq j-1 \end{cases},$$

where $C[i,j]$ is the minimum cost of cutting the wood between cuts a_i and a_j.

First, the base cases are true. In the base case where $i = j$, it is obvious that there is no cost because there is no cut between a position and itself. In the base case where $i = j-1$, there are no cuts between positions a_i and a_j, hence the cost is again 0.
Now take all other cases in which there is at least one cut between positions a_i and a_j. Suppose the minimum cost $C[i,j]$ for making all cuts between a_i and a_j is obtained by making the next cut at some position $a_{k'}$, $i < k' < j$. Then $C[i,j] = a_j - a_i + C[i,k'] + C[k',j]$. Because $a_j - a_i$ is a constant for any given i, j (the length of a piece of wood does not change regardless of how it will be cut), $C[i,k'] + C[k',j]$ is the minimum of all $C[i,k] + C[k,j]$ for all $i < k < j$, where a_k is the next cut, and the recurrence relation (and hence the algorithm) holds.

d. The recurrence relation fills in the upper triangle of a table with dimensions n by n. There are thus $n^2/2$ or $O(n^2)$ entries in the table. For every entry $C[i,j]$, we perform on the order of $O(n)$ operations by comparing the sum of the entries $C[i,k] + C[k,j]$ for all $i < k < j$. This is a linear number of operations on the order of n since in the worst case, we have $0 < k < n+1$ and have to sum the entries $C[0,1] + C[1, n+1]$ through $C[0,n] + C[n, n+1]$. The total runtime is thus $O(n^2) * O(n) = O(n^3)$.