CS 31: Homework 3

Thomas Monfre

January 31, 2019

Problem 3-1

Exercise 14.2-3: Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that we want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes ... \otimes x_m.a$, where $x_1, x_2, ..., x_m$ is the inorder listing of nodes in the subtree rooted at x. Show how to update f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the *size* attributes in order-statistic trees.

We can update f attributes in $O(1)$ time after a rotation by observing the following: in a red-black tree, the inorder list of nodes stays the same during rotation.

When describing rotations, let us use the naming conventions described in CLRS 13.2 (see page 313) where for a left rotation, elements x and y are incident to the rotated link, y is the right child of x prior to rotation and x is the left child of y after rotation. Similarly, for a right rotation, the same elements x and y are incident to the rotated link, x is a left child of y prior to rotation and y is the right child of x after rotation.

Since we are told $\otimes$ is associative, we know each individual computation using $\otimes$ can be performed in any order. Since the definition of the problem does not state that $\otimes$ is commutative, however, we must ensure the arrangement of elements to compute follows the arrangement in which they were provided. Therefore, by the definition of f provided in the problem, the binary operations must be performed on an inorder walk of the tree.

Then, since the arrangement of elements to compute follows an inorder walk of the tree, we know the following for the elements of the tree rooted at an element x:

$x.left.f = x_1.a \otimes x_2.a \otimes ... \otimes x_j.a$ for some j that denotes the split across x's children
$x.right.f = x_{j+1}.a \otimes x_{j+2}.a \otimes ... \otimes x_m.a$ for the same j as above.

When rotations are performed in the tree, the inorder walk of elements does not change (as described above), but the split j across x's children does. This indicates a child of x on a rotation will change, but their relative order will not.

Since the rotated link shuffles children but not order, for a left rotation we can calculate f for both x and the element y that is incident to x on the rotated link (as described above) with the following:

$y.f = x.f$
$x.f = x.left.f \otimes x.a \otimes x.right.f$

We can prove the correctness of this computation with the following. On a left rotation, the left child of y becomes the right child of x and x becomes y's left child. Therefore, by setting $y.f$ to be

the f value of x before the rotation, we handle the transfer of y's left child to x. Then, since the inorder walk of elements in the tree has not changed, we can recompute $x.f$ as above, knowing x's right child is now y. This handles the new split j across x's children (as shown above).

For a right rotation, the computation is similar. For the same x and y we instead have:

$x.f = y.f$
$y.f = y.left.f \otimes y.a \otimes y.right.f$

These operations take constant time because no propogation up or down the tree is required to update f. Since the inorder walk of elements has not changed, we've held to our lack of knowledge about the commutative property by maintaining the same arrangement of elements to be computed, but taken advantage of our knowledge that the associative property applies by changing the order in which computations are performed.

Since we have shown maintaining the f attribute requires two static computations that do not rely on propogation up or down the tree, we then have that f attributes can be updated in $O(1)$ time after a rotation.

In order to apply this argument to the $size$ attribute in an order-statistic tree, let us define the attribute a for each node as 1. Then, for every node n, $n.a = 1$. Next, let us make $\otimes$ be the addition function, in that $a \otimes b \otimes c = a + b + c$ for some arbitrary a, b, c.

Using the above argument with these modifications, we then have:

$x.left.f = \sum_{i=1}^{j} 1 = j$ for some j that denotes the split across x's children
$x.right.f = \sum_{i=j+1}^{m} 1 = m - j$ for the same j as above.

Using this and the above definition for $x.f$, for a left rotation we then have:

$$
\begin{aligned}
y.f &= x.f \\
\\
x.f &= x.left.f \otimes x.a \otimes x.right.f \\
&= x.left.f + x.a + x.right.f \\
&= j + 1 + m - j \\
&= m + 1
\end{aligned}
\tag{1}
$$

Since we are told $x_1, x_2, ..., x_m$ is the inorder listing of nodes in the subtree rooted at x, we know the subtree rooted at x contains m nodes. Since the size of a node x (as defined in CLRS Chapter 14) is the number of nodes in the subtree rooted at x plus 1 (accounting for x itself), we then know the size of x is $m + 1$. This relationship holds because each node in the subtree rooted at x contributes 1 to $x.size$. Since we have defined the attribute a for each node to be 1, we therefore have that $x.f = m + 1$. Since $x.size = m + 1$, we therefore have shown that the f attribute can be modified to calculate the $size$ attribute of a node x in an order-statistic tree during a left rotation. Since the same logic applies to a right rotation, we therefore can maintain the $size$ attribute using f.

Problem 3-2

Exercise 15.3-5: Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, ..., n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

The optimal-substructure property described in Section 15.1 states that after making an optimal cut for a rod of length j, the left and right pieces produced from the cut are optimal. Specifically, each sub-rod is optimal in terms of the revenue it will produce.

If we impose a limit on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, ..., n$, this optimal-substructure no longer holds because we have no way of guaranteeing that each sub-rod will be used in the final solution.

If we are taking the bottom-up approach, we compute the optimal cut for all possible length rods for $i = 0, 1, ..., n$. If we impose the limit l_i when computing the optimal cuts starting from 0, it is possible that we will reach a limit for some l_i at which point we must choose to make a different cut. After computing the optimal cuts for all possible length rods, when we actually determine what cuts will be made, it is possible that we will choose a cut in which a non-optimal choice was made because, when computing all cuts, we hit a limit. Since not all cuts are guaranteed to be used, however, we cannot guarantee that that limit will be met in our final solution. Therefore, it is possible that we could have made a cut that was less optimal in order to account for a limit that was not met. Since this is possible, we can no longer guarantee that each sub-rod is optimal.

If we are taking the top-down approach, the same logic applies. At some point, our stored solution for a given cut might be non-optimal in order to account for a limit. The final solution is not guaranteed to hit such a limit, indicating we cannot guarantee each sub-rod is optimal.

Furthermore, the optimal-substructure property no longer holds because our decisions made for how to cut each rod are no longer independent choices. The length of the right sub-rod now depends on the left sub-rod for each cut. Since we cannot make independent choices for each possibility, we cannot guarantee that each sub-rod is optimal.

Therefore, the optimal-substructure property described in Section 15.1 no longer holds if we impose a limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, ..., n$.

Problem 3-3

Exercise 15.3-6. Assume that you may not repeat a currency in a sequence of trades.

Let us define our currencies as $1, 2, ..., n$, where currency 1 is our starting currency and currency n is our ending currency. For each pair of currencies i and j, we are given an exchange rate r_{ij}. Let c_k be the commission that we are charged when we make k trades.

Part A: Show that if $c_k = 0$ for all $k = 1, 2, ..., n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure.

Let us first observe from CLRS 15.3 that a problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. We can identify such an optimal substructure by showing the solution consists of making a choice, supposing we are given the inital choice, determining what subproblems ensue from such a choice, then showing the solutions to those subproblems must be optimal (typically using a cut and paste style proof) in order to have an optimal final solution.

For the problem of exchanging currencies, let i, j represent currencies where i, j take on the values $1...n$. Suppose we are seeking to convert currency 1 to currency j. Then let us define our subproblem as the choice of the currency i to directly exchange with j. As an example of this, let us assume we have currencies $1...j$ and we are told currency i is the optimal choice of the currency to directly exchange with j. Then, we define the total value of our exchange from 1 to j as the value of the optimal exchange from currencies 1 to $i-1$ plus the value of the direct exchange from currency i to j. In this case, if we started with d units of currency i, this would be dr_{ij} units of currency j.

Let us therefore observe that the solution to this problem consists of making a choice for currency i. Given this choice, we then rely on an optimal solution to the subproblem of exchanging currency 1 with currency $i-1$ in 0 or more transactions. This optimal solution allows us to then directly exchange currency i with currency j. Since we do not know the optimal choice of currency i, however, the solution to this problem would involve trying all values of i for each possible value of j and storing the choice of i that led to the most optimal total solution for exchanging currencies 1 and j. This is not necessarily the currency i that creates the largest gain when directly exchanged with currency j. We therefore rely on an optimal substructure to ensure the value of exchanging currencies $1...i-1$ plus the value of the exchange from i with j is maximal.

We can prove the choice of the optimal currency to directly exchange with j yields an optimal solution for an exchange from currencies 1 to j with the classic cut-paste proof. As defined in our optimal substructure above, the optimal value computed for currencies $1...j$ for some $j \leq n$ is comprised of the optimal value of 0 or more transactions from currency 1 to currency $i-1$ where i represents the optimal choice of the currency to directly exchange with j, plus the value of the direct exchange between currencies i and j. Let us assume our choice of i yields an optimal solution to the total value of exchanging currencies 1 and j. Since we have assumed the solution for $1...j$ is optimal and our solution for $1...j$ relies on an optimal solution to $1...i-1$, if there exists a better (more optimal) solution for exchanging currencies $1...i-1$, then our solution for $1...j$ is not optimal. Hence we have arrived at a contradiction. Therefore, the solution to exchanging currencies $1...i-1$ must be optimal.

This process assumes $c_k = 0$ since no commissions were charged for the number of exchanges we make. We have therefore proven if $c_k = 0$ for all $k = 1, 2, ..., n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure because each subproblem produced by exchanging currencies 1 and j must produce an optimal solution in order for the solution to exchanging currencies 1 and j to be optimal. The final solution to this problem would be the optimal value when $j = n$. Note that the bottom-up solution to this problem would have us compute the optimal value for exchanging currencies $1...i-1$ prior to accessing it, so when determining the optimal exchange between currency 1 and currency j, our optimal value for all choices of i has already been computed.

Part B: Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does ***not necessarily*** exhibit optimal substructure.

If commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure because our choice for how many trades to make and what currencies to use depend on each other. We no longer have independent calculations. Instead, our choices for a solution to a given subproblem may be dependent on our choices for a solution to a different subproblem. Specifically, our choice of the optimal currency i to directly exchange with currency j where $1 \leq i \leq n$ and $1 \leq j \leq n$, is no longer guaranteed to be the choice that maximizes the sum of the value of exchanging currencies $1...i-1$ and the value of directly exchanging i with j.

Since we are penalized for a certain number of exchanges made, we are not guaranteed to know the optimal solution for exchanging currencies $1...i-1$. Because we are incentivized to exchange less, our computations for the best sequence of 0 or more transactions to make in order to exchange currency 1 with currency $i-1$ is no longer independent of the final value for j. Each subproblem does not necessarily have to be optimal in order to produce an optimal final result, and each optimal subproblem is not guaranteed to produce an optimal final solution.

Thus, in imposing this restriction, it is possible that in determining the best choice of currency i to directly exchange with currency j, we could choose a less-optimal sequence of exchanges from currency 1 to currency $i-1$ because the commission penalty is higher. This would create a potentially less optimal subproblem that could potentially end up being used to produce an optimal final result. Further, we could compute an optimal solution to a subproblem (without any respect to the commission penalties) that ends up producing a less optimal final solution because such a solution caused a high penalty to be paid. In this example, we do not have optimal substructure because (1) when computing a solution to each problem, we do not know if that solution will end up being used in the final solution and therefore may promote a less-optimal result, (2) we could promote an optimal subproblem that causes a penalty to be paid, creating a less optimal final solution, and (3) each subproblem solution is no longer considered independent.

Therefore, if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

Problem 3-4

Problem 15-4. For this problem, you must implement your solution in Python, Java, C, or C++. 30 of the 70 points are for your implementation. Submit code and the textual output produced for the input file paragraph.txt. Use a maximum line length of M=70 characters. To make reading input easier for you, each word in this input file appears on a separate line. If punctuation immediately precedes or follows a word, it appears before or after the word without an intervening space; consider the punctuation as part of the word. Analyze the running time and space requirements of your algorithm.

The Python script I wrote as well as the output text file my algorithm generated is included in the submission of this assignment.

Let us first argue this problem has optimal substructure. Let i, j represent input words where $1 \leq i \leq j \leq n$ for words $1...n$. Then, let us define our sub-problem as the choice of the first word in the last line for a given set of words $1...j$. As an example of this, let us assume we have words $1...j$ and we are told the word i that is the optimal choice for first word in the last line when arranging a neatly printed collection of words $1...j$. Then, we define the total cost of words $1...j$ is the cost (cube of the extra characters on each line) of the optimal solution to the sub-problem for the lines above i (i.e. $1...i-1$) plus the cost of the final line from words $i...j$. If $j = n$, then the cost of the final line is 0 since the problem definition states we seek to minimize the cube of the extra space characters at the end of every line but the last. Therefore, the optimal substructure to this problem is the collection of words in all lines but the last, reflected by our optimal choice of the first word in the last line. After choosing the optimal first word i for the last line in words $1...j$, all words that come before i create their own sub-problem of an identical form for words $1...i-1$.

We can prove each choice of the optimal first word i of the last line yields an optimal solution for words $1...j$ with the classic cut-paste proof. As defined in our optimal substructure above, the optimal cost computed for words $1...j$ for some $j \leq n$ is comprised of the optimal cost for some sequence of words $1...i$ where i represents the optimal choice of the first word in the last line of words $1...j$, plus the cost of the line $i...j$. Assume our choice of i yields an optimal solution to the cost of printing words $1...j$ neatly. Since we have assumed the solution for $1...j$ is optimal and our solution for $1...j$ relies on an optimal solution to $1...i-1$, if there exists a better (more optimal) solution for neatly printing words $1...i-1$, then our solution for $1...j$ is not optimal. Hence we have arrived at a contradiction. Therefore, the solution to neatly printing words $1...i-1$ must be optimal.

Using this, let us then define a recursive solution for this problem. We seek to minimize the cost of neatly printing words $1...j$ for $1 \leq j \leq n$. Let us define the choice of the first word in the last line for words $1...j$ as i. Our optimal cost can be defined as follows. The base case occurs when $j = 1$. In this case, the optimal solution to the words above/before word 1 is 0 since no words exist above/before word 1. Otherwise, our recursive solution, as defined above, takes hold. Then the optimal solution to neatly printing words $1...j$ is the optimal cost to neatly print words $1...i-1$ plus the cost to print the line $i...j$. We can define this relationship in the piece-wise function below, where *cost* is an array holding the optimal costs for neatly printing words $1...j$ and *line_cost* is a matrix holding the cost of printing words $i...j$ on a single line.

$$cost[j] = \begin{cases} 0 & \text{if } j = 1 \\ \min\limits_{1<i\leq j} cost[i-1] + line\_cost[i,j] & \text{otherwise} \end{cases}$$

Using this, let us define our bottom-up dynamic programming algorithm as follows. We first compute the number of extra space characters at the end of every possible line from word i to word j for $i, j = 1..n$. Using this, we define the cost of a given line based on the metrics provided in the problem definition. If a given line does not fit within the character constraint (i.e. the number of extra space characters at the end of the line is negative), we define its cost as infinite. This ensures we do not consider lines that contain too many characters. If a line includes the last word in the input file, we define its cost as 0 since the problem specifically states to minimize the cube of the number of extra characters on all lines, but the last. If neither of these conditions hold, we then set the cost to the cube of the number of extra space characters left on the line.

Once we've computed this, we know the cost for every possible line. We now must find the choice of lines that is optimal. To do this, we rely on our optimal substructure, proved above. For each possible choice of j in words $1...j$, we seek to determine the optimal cost for words $1...j$ based on the optimal substructure defined above. We will therefore find the optimal first word i in the last line of words $1...j$. Since we cannot assume we know the best value of i, we will iteratively try each possible choice of i in a bottom up fashion.

In order to ensure the optimal subproblem we rely upon has already been solved, we examine all possible choices of word j from 1 to n. For each choice, we try all possible values of i where $1 \leq i \leq j$. Since we are only considering words $1...j$ on a given iteration, i can take on values from $1...j$. For each choice of i and j, we know our optimal substructure dictates the cost of words $1...j$ is the cost of the words $1...i-1$ plus the cost of the line $i...j$. Since j starts at 1 and iterates through n, and i is at most j on a given iteration, the maximum sequence of words that could comprise our optimal substructure is words $1...j-1$. Since j takes on values starting at 1 and iterating to n, we then know the maximum sized subproblem on each iteration has been solved in a previous iteration of the loop. Therefore, we can compute the cost of words $1...j$ for a given choice of i as our stored optimal solution for words $1...i-1$ plus the cost of the line from $i...j$ (which was computed above).

For each choice of j, we then store the optimal cost for words $1...j$ in an array called $optimal\_costs$. We also store the choice of the first word in the last line i that yielded the optimal cost in an array called $indices\_of\_first\_word\_in\_line$. We store the choice of the first word in the last line since that is the optimal substructure we defined. Holding onto these indices provides the information necessary to construct each line.

Then, after computing these values, we determine the desired output by following the indices backwards until we hit the first line. The process operates as follows. We know we must optimally print words $1...n$. The choice of the first word in the last line of words $1...n$ was stored in $indices\_of\_first\_word\_in\_line[n]$. We therefore know our last line holds words $indices\_of\_first\_word\_in\_line[n]$ to n. Then, the second to last line ends with word $indices\_of\_first\_word\_in\_line[n]-1$. We then must determine the optimal choice for the first word in the last line of words $1...(indices\_of\_first\_word\_in\_line[n]-1)$. Since our optimal substructure holds the choice of the first word in the last line for all sequences of words starting with 1, we then know we've stored the optimal choice of the first word in the second to last line at

$indices\_of\_first\_word\_in\_line[indices\_of\_first\_word\_in\_line[n] - 1]$. We can continue finding the words on each line until we've hit the line in which the optimal word of the last line is the first word. Then, we will have constructed the optimal choice of lines, yielding an optimal solution for neatly printing words 1...n.

To review, we first find the number of extra characters at the end of every possible line, then compute the cost associated with each line. After this, we determine the optimal cost of printing words 1...j for each possible j from 1 to n. When determining each optimal cost, we store the optimal choice of the first word in the last line of words 1...j. Using the choice of the first word, we then construct the optimal way to print words 1...n by first finding the word in the last line, and using our optimal substructure to determine the words on the last line of the set of words preceeding it, which is itself a form of the given problem.

We can prove that the running time of this algorithm is $\Theta(n^2)$ with the following. On lines 26 and 27 as well as 39 and 40, we have sets of doubly-nested for loops, where each loop iterates from 1...n. These loops are used to compute the number of extra characters and the cost of every possible line. These loops give us an upper bound of $O(n^2$ and a lower bound of $\Omega(n^2)$. Additionally, on lines 61 and 66 we construct another set of for loops where the outer iterates from 1 to n, and the inner iterates from 1 to j (where j is the outer loop element). These loops are used to determine the optimal cost for words 1...j. This also give us an upper bound of $O(n^2)$ and a lower bound of $\Omega(n^2)$. Since we have $O(n^2)$ and $\Omega(n^2)$, we have $\Theta(n^2)$.

We can prove that the space requirements of this algorithm is $\Theta(n^2)$ with the following.
num_extra_spaces and $cost\_of\_line$ are both of size n by n. This gives us an upper bound of $O(n^2)$ and a lower bound of $\Omega(n^2)$. We fill the entire contents of each. A simpler version of this algorithm could only fill slightly more than half of each matrix. Using asymptotic notation, however, we would still arive at $\Theta(n^2)$. The only other collections kept in memory are $optimal\_costs$, $indices\_of\_first\_word\_in\_line$, and $output\_indices$, which are all of size n. We therefore have an upper bound of $O(n^2)$ and a lower bound of $\Omega(n^2)$, yielding a space requirement of $\Theta(n^2)$.

Problem 3-5

Problem 15-5.

When determining the best way (associated with lowest cost) to transform a word x to a word y, let us observe that we must convert 0 or more characters in x into a character $y[j]$ using the six transformation methods provided in the book. A combination of transformations then creates a solution. If we assume we are told the best (most optimal) way to transform 0 or more characters in x into $y[2]$. Then we can split y into two sections. The section to the left of $y[2]$ (inclusive) has been transformed and optimally solved. The section to the right remains unsolved. Unfortunately, without being told the most optimal transformations to make so as to achieve $y[2]$, we cannot know the best way to transform 0 or more characters in x into $y[2]$ without knowing the best way to transform characters $y[2..n]$ (i.e. the right section of the split). The brute-force method would be to try all possible solutions. Naturally, we can use dynamic programming to do much better.

Let us then argue this problem has optimal substructure with the following. Let us define $y[j]$ as the character in y we seek to transform using 0 or more characters in x. We define our sub-problem as the optimal solution to transforming characters $1...j-1$ in y. Then, when finding the best way to transform 0 or more characters in x to $y[j]$, we know all characters occuring before $y[j]$ in y have already been optimally transformed.

Using the example in the book where $x =$ "algorithm" and $y =$ "altruistic", our final solution to transforming x to y would then require finding the best way to transform 0 or more elements in x to "c", knowing an optimal way to transform "altruisti" has already been found. Since we know an optimal solution to "altruisti" has already been found, we can focus solely on finding an optimal transformation for "c". Our repeated sub-problems are therefore the repeated series of characters at the start of the string. In order to know the optimal solution to "altruisti" when solving for "c," we need to have previously solved it. Therefore, our optimal substructure will require us to work front to back in the target string y. When solving for the first character in y, we know the optimal solution to all characters before it has been found since no characters exist to the left of the first character. Therefore, using the example from the book, once we determine the best way to transform 0 or more characters in x to "a", we can determine the best way to transform 0 or more characters in x after our index in x to "al". Once we have "al", we can then determine the best way to achieve "alt", followed by "altr" and so on until we have computed a complete transformation from "algorithm" to "altruistic".

We can prove this substructure is optimal with the classic cut-paste proof. Let us assume our solution for transforming characters $y[1...j]$ is optimal. Then using the substructure defined above, the minimum cost computed for transforming 0 or more characters in x to $y[1...j]$ is comprised of the optimal cost for the achieving the sequence of characters $y[1...j-1]$ plus the cost of the most optimal way to achieve a transformation for $y[j]$. Since we rely on the optimal solution to $y[1...j-1]$ in computing this solution, if there is a better (more optimal) solution to $y[1...j-1]$, we have then arrived at a contradiction since our solution for transforming characters $y[1...j]$ is not optimal. Therefore, this problem has optimal substructure.

Using this, let us then define a recursive solution for this problem. Let us define i as our index into x and j as our index into y. We seek to minimize the cost of transforming 0 or more characters

in $x[i...n]$ to $y[1...j]$. Our optimal cost can be defined as follows. The base case occurs when $i = 1$ or $j = 1$. In this case, the minimum cost required to transforming 0 or more characters in $x[i...n]$ to $y[j]$ is the minimum cost of all possible transformations since no characters exist before x or y to transform. Otherwise, our recursive solution, as defined above, takes hold. Then the optimal solution to transforming 0 or more characters in $x[i...n]$ to $y[j]$ is the minimal cost each possible transformation at these indices plus that operation's corresponding optimal cost of the transformation of character $y[j-1]$ that would have put us in the position of i and j to make this transformation of $y[j]$. We are minimizing the sum of the cost of each transformation and its corresponding optimal sub-problem solution. Since multiple transformations exist and each increment i and j in different ways, our solution to the subproblem is the optimal cost of achieving a transformation for characters $y[1...j-1]$ based on what i and j values for the transformation of $y[j]$ that land us at the current i and j. We can define this relationship in the piece-wise function below, where *cost* is a matrix holding the optimal cost for transforming 0 or more characters in $x[1...i]$ to $y[1...j]$. Let us define op.*i_back* and op.*j_back* as the indices we must travel backwards to arrive at this transformation for $y[j]$. Let us define op.*cost* as the cost of performing this operation (copy, delete, etc.)

$$cost[i,j] = \begin{cases} 0 & \text{if } i = 1 \text{ or } j = 1 \\ \min\limits_{op} op.\text{cost} + cost[\text{i} - op.\text{i\_back}, \text{j} - op.\text{j\_back}] & \text{otherwise} \end{cases}$$

Let us then define when each operation is determined possible. Using the variable definitions from the textbook, let i denote the index into x we are currently visiting and let j denote the index into y we are currently visiting.

Using the transformation descriptions from the book, let us then define when we will use each transformation as follows:

- **Copy:** if $x[i] = y[j]$
- **Replace:** if $x[i]$ never appears in $y[j...m]$
- **Delete:** if $x[i]$ never appears in $y[j...m]$ and $x[i+1] = y[j]$
- **Insert:** if $x[i]$ appears in $y[j...m]$ and $x[i] \neq y[j]$
- **Twiddle:** if $x[i] = y[j+1]$ and $x[i+1] = y[j]$
- **Kill:** only after determining all solutions. This will be used when outputting our final optimal series of transformations.

Using this, we can define our bottom-up, dynamic programming algorithm as follows. We first create a matrix called optimal_costs that stores the lowest cost incurred when transforming 0 or more characters in $x[1...i]$ to $y[1...j]$. We will also store a matrix called tranformations that holds the associated last transformation used to acheive that lowest cost at i and j. For each possible choice of i and j, we determine all transformations that we can perform at the current indices. For each possible transformation, we determine the cost of the transformation based on where its previous transformations would have placed us. This relies on the optimal substructure that we have defined. Therefore, we compute the cost of each transformation knowing the cost of the previous transformations that would have gotten us to this point. Since we are iterating forwards, we know we

have already solved the optimal cost of these previous transformations. We then take the operation that achieves the lowest value for that operation's cost plus its associated cost and store it in our matrices optimal_costs and transformations. We repeat this until we have examined all characters in x and y. Note, the base case cost of previous transformations is 0. Since the first operation could be twiddle, we store 0 for the first 4 (2 by 2) possible costs achieved in optimal_costs.

Then, after determining all possible costs and transformations, we find the transformation that achieved the lowest cost for character $y[m]$ for all possible choices of i in x. The transformation at some character $x[i]$ that was chosen will then have been the most optimal final transformation to make since it yielded a full transformation of y. Therefore, in order to handle the **Kill** operation and start our backwards walk through the table in an optimal location, we will find the best location i in x to perform **Kill** by choosing the location with the lowest cost. This location occurs at indices $[i, m]$ in each table. Then, the cost at this location (optimal_costs$[i, m]$) will be our final **edit-distance**. We store this cost. Then, all we must do is find the optimal choices of transformations to make based on this optimal cost.

Now that we are located at indices $[i, j]$ where $j = m$ in each table, all we must do is store the operation at operations$[i, j]$ then decrement i and j depending on what operation we performed. Since the matrix transformations holds objects that each have attributes i_back and j_back, we can then move left and up in the table based on the operation we chose to perform. For **Copy** and **Replace**, the i_back and j_back values are both 1. For **Delete**, the i_back value is 1 and the j_back is 0. For **Insert**, the i_back value is 0 and the j_back is 1. Lastly, for **Twiddle**, the the i_back and j_back values are both 2.

We then subtract off these i_back and j_back values from i and j, then continue. When we reach the first character in x or y, we know we will have found all transformations required to convert x to y. We then add "KILL" to the end of transformations to denote the final transformation made, then we return the **edit-distance** computed earlier as well as our array of transformations.

The pseudocode for this algorithm is given on the following page:

```
Get − Edit − Distance(x, y)
 1  let optimal_costs be an n + 2 by m + 2 matrix
 2  let transformations be an n + 2 by m + 2 matrix
 3
 4  // fill in optimal_costs values of zero for first four entries to handle the base case
 5  // note that we add four values to handle the case where the first transformation is Twiddle
 6  for i = 1 to 2
 7      for j = 1 to 2
 8          optimal_costs[i, j] = 0
 9          transformations[i, j] = null
10
11  // note we loop from 3 to n + 2 because we just stored four base case values in each matrix
12  for i = 3 to (n + 2)
13      for j = 3 to (m + 2)
14          possible_operations = [ ]
15
16          // determine possible operations
17          // note each object (i.e. Copy()) has attributes attributes name, cost, i_back, and j_back
18          if x[i] = y[j]
19              possible_operations.add(new Copy())
20          elseif x[i] never appears in y[j...m] // note this also serves as a base case
21              possible_operations.add(new Replace())
22          elseif x[i] never appears in y[j...m] and x[i + 1] = y[j]
23              possible_operations.add(new Delete())
24          elseif x[i] appears in y[j...m] and x[i] ≠ y[j]
25              possible_operations.add(new Insert())
26          elseif x[i] = y[j + 1] and x[i + 1] = y[j]
27              possible_operations.add(new Twiddle())
28
29          min_cost = ∞
30          best_op = null
31
32          for op in possible_operations
33              cost = op.cost + cost[i - op.i_back, j - op.j_back]
34              if cost < min_cost
35                  min_cost = cost
36                  best_op = op
37
38      optimal_costs[i, j] = min_cost
39      transformations[i, j] = best_op
```

The following pseudocode follows from that above. The line numbers should begin with 40.

```
1  // handle kill by finding minimum cost in c[ : , m + 2]
2  min_cost = ∞
3  index_of_min = null
4  for i = 3 to m + 2
5        if optimal_costs[i, m] = min_cost
6              min_cost = optimal_costs[i, m]
7              index_of_min = i
8
9  // find best set of transformations
10 i = index_of_min
11 j = (m + 2)
12 edit_distance = optimal_costs[i, j]
13 operations = [ ]
14 while i ≠ 1 and j ≠ 1
15       op = transformations[i, j]
16       operations.insert(0, op.name)
17       i = i - op.i_back
18       j = j - op.j_back
19 operations.append("KILL")
20 return (edit_distance, operations)
```

We can prove this algorithm has a running-time of $\Theta(nm)$ with the following. On lines 12 and 13 we have a set of doubly-nested for loops. The outer loop iterates from 3 to $n + 2$ and the inner iterates from 3 to $m + 2$. Since each loop is guaranteed to perform each iteration, we then have m operations performed n times, hence an upper bound of $O(nm)$ and a lower bound of $\Omega(nm)$, yielding $\Theta(nm)$. Although these loops contain an inner for loop at line 32 to examine all possible transformations, this loop iterates a maximum five times. Since five is a constant, we do not add an order to this computation. The only other iterations performed in this algorithm are in handling **Kill** and walking backwards through the table. Since kill iterates from 3 to $m + 2$, we then have a $O(m)$. Since each possible transformation decrements i or j by at least 1, we then have a maximum number of operations of $\Omega(n)$ or $\Omega(m)$ depending on if $n > m$ or $m > n$. We therefore have $\Theta(nm)$.

We can prove this algorithm has a space requirement of $\Theta(nm)$ as well. The matrices optimal_costs and transformations are both of size $n + 2$ by $m + 2$. This give us $\Theta(nm)$. The only other data structure with a size of order magnitude is transformations which holds at most m elements. Therefore we have $\Theta(nm) + (m) = \Theta(nm)$.

Part B: We can cast the problem of finding an optimal alignment as an edit distance problem by using the following operations: copy, replace, and insert. Let us denote the input sequence as x, the target sequence as y, our index into x as i, and our index into y as j. For each sequence of characters, we will determine the optimal way to transform 0 or more characters in x in order to properly align with characters in y. We then try all possible combinations of indices i and j and determine the operations that apply.

For this problem, we will apply the following rules:

- **Copy:** if $x[i] = y[j]$
- **Replace:** if $x[i] \neq y[j]$ and $x[i+1] = y[j+1]$
- **Insert space in x:** if $x[i] = y[j+1]$
- **Insert space in y:** if $x[i+1] = y[j]$

Where the cost of copy is $+1$, the cost of replace is -1, and the cost to insert is -2.

Then, rather than minimizing "cost," we will maximize it since we are incentivized to copy and discouraged to insert spaces. The process then follows the same format as the **edit-distance** problem. We determine the optimal cost for each string of characters, taking advantage of the optimal substructure we previously defined. This optimal substructure uses the previously computed optimal cost of a subproblem to define the optimal cost for a new selection of i and j. Then after calculating all costs, we iteratively work backwards through our matrix of costs until we hit the beginning of each string. The code follows the same structure except we choose to maximize "cost" rather than minimize.

Problem 3-6

Problem 15-12.

Let us first argue this problem has optimal substructure. Let N be the number of positions, P be the players for each position, and X be the budget. Further, let n denote the currently observed position for $1 \leq n \leq N$. Let p denote the currently observed player for $1 \leq p \leq P$. Finally, let x denote the size of the current budget ceiling for $1 \leq x \leq X$. Then, let us define our sub-problem as the act of signing the best collection of players at the current budget ceiling x minus the cost of signing the current player p where $p.cost$ denotes player p's cost. Essentially the substructure defines that for each player p we observe, we know signing the player with the maximum value will create an optimal solution for a certain budget ceiling x if our choice of players at budget ceiling $x - p.cost$ is optimal.

We can prove this sub-problem is optimal with the classic cut-paste style proof. When observing a player p of position n whose cost is lower than the current budget ceiling x, there are two cases. Let us assume for this proof that either case yields an optimal choice of players that maximize value. The first case is that signing this player and the optimal choice of players at budget ceiling $x - p.cost$ will create a total value greater than the value associated with signing the optimal choice of players at positions $1...n$ and budget ceiling x. In this case, our optimal solution relies on an optimal solution to the players at budget ceiling $x - p.cost$. The second case is that signing this player and the optimal choice of players at budget ceiling $x - p.cost$ will create a total value less than the value associated with signing the optimal choice of players at positions $1...n$ and budget ceiling x. In this case, our optimal solution relies on an optimal solution to the players at positions $1...n$ and budget ceiling x. Since we have assumed both cases yield an optimal solution when the conditions to arrive at each case are satisfied, and both cases rely an an optimal solution to a subproblem, if the solution to either subproblem is not optimal, then we have contradicted our assumption that the current solution is optimal. Therefore, we know our optimal substructure holds.

Using this, let us then define a recursive solution for this problem. We seek to maximize the sum of each signed player's VORP while ensuring the cost of that collection of players is less than X and we sign 0 or 1 players per position. Let n denote the current position and x denote our current budget ceiling. Our optimal cost can be defined as follows. The base case occurs when $n = 1$. In this case, the optimal solution to signing the player at the previous position is 0 since no position exists before the first. Otherwise, our recursive solution, as defined above, takes hold. Then the optimal solution to signing players for positions $1...n$ takes two cases. The first case is that the current player's cost is less than or equal to our budget ceiling x and that signing this player and the optimal choice of players at budget ceiling $x - p.cost$ will create a total value greater than the value associated with signing the optimal choice of players at positions $1...n$ and budget ceiling x. In this case, our optimal total value is player p's VORP plus the optimal value achieved for signing players for positions $1...n - 1$ with the remaining budget ceiling $x - p.cost$. The second case is that player p's cost is greater than the current budget ceiling x or that signing this player and the optimal choice of players at budget ceiling $x - p.cost$ will create a total value less than the value associated with signing the optimal choice of players at positions $1...n$ and budget ceiling x. In this case, our optimal total value is the total value achieved by hiring players for positions $1...n - 1$ at this budget ceiling x since we either cannot afford to hire player p or signing them and the optimal

collection of other players whose total cost is less than or equal to x yields a lower total value at this current budget ceiling. We can define this relationship in the piece-wise function below, where *opt_values* is a matrix holding the optimal values for signing players for positions $1...n-1$ at the budget ceiling x.

Let case A occur when ($p.cost \leq x$) and ($p.vorp$ + opt_values$[n-1, x-p.cost]$ > opt_values$[n-1, x]$). This is defined so the following piece-wise function fits on the page.

$$opt\_values[n,x] = \begin{cases} 0 & \text{if } n = 1 \\ \text{p.vorp+}opt\_values\text{[n-1, x - p.cost]} & \text{if case } A \text{ (above) occurs} \\ \text{opt\_values}[n-1, x] & \text{otherwise} \end{cases}$$

Using this, let us define our bottom-up dynamic programming algorithm as follows. We first construct two N by X matrices. opt_values hold the maximum sum of VORP values achieved for positions $1...n$ with a budget of size $1...X$. best_player holds the best player to sign at each position $1...n$ for each budget size $1...X$. If the best option is to sign no player for position n and budget size x, best_player$[n, x] = 0$. We then initialize every value in opt_values and best_player to 0. Notice our arrays start at index 0. This handles the base case optimum value and choice of player defined above.

Using our optimal subproblem defined above, for each position, we know we can only sign at most one player. We also know we can spent at most X dollars on all players. Furthermore, it might not be the case that the optimal solution involves hiring the player with the greatest VORP at each position. Therefore, we compute a bottom-up solution to the problem. For each position, we must consider each possible budget ceiling. This allows us to determine the best possible choice of players for each sum of money that might be available. Since we do not know which player is the optimal player to hire for each position, we try all choices of players and hold onto the choice of player at each position that yielded the largest sum for each possible budget ceiling.

Using the knowledge that iterating from 1 to N, P, X ensures all previous subproblems have been solved, we then perform the following actions for each position n, each player p, and each budget ceiling x. We first determine if the cost associated with signing player p is less than the current budget ceiling. If it is, we then determine the solution to each possible case outlined above. We perform whatever choice yields a greater total result and store the player for this position and budget ceiling that yielded an optimal result. If the cost associated with signing player p is greater than the current budget ceiling, we then do not sign this player and pass along the optimal value achieved for hiring players for positions $1...n-1$. We continue iterating until we have examined the optimal values for each possible position, player, and budget ceiling.

After we finish iterating, we then will have determined the optimal player to hire for each position for each possible budget ceiling. Since we know our actual budget ceiling is X, we then know the best VORP value for all the possible players is the value in opt_values at position (n, x). This tells us the largest VORP value we acheived while abiding by the budget X. We then must determine the choice of players associated with this optimal value. To do this, we simply walk back through our matrix of optimal values and determine at what budget ceiling each choice was performed at. Since the matrix best_player holds the player hired at each position and each possible budget ceiling, when we determine the budget ceiling that each decision was made at we know the player hired

for that position is at the identical location in best_player. In order to determine at what budget ceiling we made our choice of each position, we simply walk up the budget column and determine at which point we find a larger total VORP value. The first location in which we see a change in VORP value in the column denotes the budget ceiling that we hired a player. This tells us where in best_player this person resides. As we pick players, we also sum up their costs. Then on line 42, we return the total VORP value, the total cost incurred, and an array of players to hire.

The pseudocode for this algorithm is provided below:

```
1  opt_values = new N by X matrix
2  best_player = new N by X matrix
3
4  for n = 0 to N
5      for x = 0 to X
6          opt_values[n, x] = 0
7          best_player[n, x] = 0
8
9  for n = 1 to N
10     for p = 1 to P
11         for x = 1 to X
12             if p.cost ≤ x
13                 new_val = p.vorp + opt_values[n − 1, x − p.cost]
14                 if new_val > opt_values[n − 1, x]
15                     opt_values[n, x] = new_val
16                 else
17                     opt_values[n, x] = opt_values[n − 1, x]
18                     best_player[n, x] = best_player[n − 1, x]
19             else
20                 opt_values[n, x] = opt_values[n − 1, x]
21                 best_player[n, x] = best_player[n − 1, x]
22
23 // we will now walk back through the array to find the optimal choice of players
24 players = [ ]
25 total_cost = 0
26 total_vorp = opt_values[N, X]
27 n = N
28 x = X
29
30 while opt_values[n, x] > 0
31     // if we did not choose the player at position n for this sized budget, skip them
32     if opt_values[n, x] == opt_values[n, x − 1]
33         n = n − 1
34         // if we choose a player at position n for this sized budget, sign the player, incur their cost,
35         // then move to the next position
36     elseif opt_values[n, x] > opt_values[n, x − 1]
37         players.insert(0, best_player[n, x])
38         total_cost = total_cost + best_player[n, x].cost
39         x = x− best_player[n, x].cost
40         n = n − 1
41
42 return (total_vorp, total_cost, players)
```

We can prove this algorithm runs in $\Theta(NPX)$ time with the following. At lines 9, 10, and 11 we have a set of triply nested for loops. The outer loop iterates from 1 to N, the middle iterates from 1 to P and the inner iterates from 1 to X. Since each loop iterates to its full value, we have an upper bound of $O(NPX)$ and a lower bound of $\Omega(NPX)$ yielding $\Theta(NPX)$. At lines 4 and 5 we have a set of doubly nested for loops where the outer iterates from 1 to N and the inner iterates from 1 to X. This give us $\Theta(NX)$. The only other loop in this algorithm is the while loop at line 30. This loop iterates until opt_values$[n, x] > 0$. Since opt_values[0,0] $= 0$, we initialize n to N at line 24, we decrement n on each iteration, and we cannot have a negative budget, we then know this loop iterates at most n times. This gives us $O(n)$. We therefore have $\Theta(NPX) + \Theta(NX) + O(n) = \Theta(NPX)$.

We can prove this algorithm has a space requirement of $\Theta(NX)$ with the following. On lines 1 and 2 we create two N by X matrices and at lines 4 and 5 we initialize every value in each to 0. This gives us an upper bound of $O(NX)$ and a lower bound of $\Omega(NX$ yielding $\Theta(NX)$. The only other data structure used in this algorithm is the arrays players that is created at line 21. Since we can hire at most N players and at least 0 players, we have an upper bound of $O(N)$ and a lower bound of $\Omega(N)$ yielding $\Theta(N)$. We therefore have $\Theta(NX) + \Theta(N) = \Theta(NX)$.

Who I Worked With

Problem 3-1: I worked with Emma Rafkin on determining how to handle the two indices that are incident to the rotated vertex.
Problem 3-4: I worked with Kunal Rathi and Emma Rafkin on determining the optimal subproblem and producing the output.
Problem 3-5: I worked with Kunal Rathi on determining which direction to iterate through the strings.