**†Dynamic Programming Homework**

For each problem in your comments you must:

* State the general subproblem form (including semantics)
* Give the recursive definition (including the base case).

We will provide something you can use for problem 1. However, you are welcome to use an alternative formulation if you’d prefer **After stating all problems, some additional guidance/hints are provided for those who would like that.**

1. You are traveling by a canoe down a river and there are *n* trading posts along the way. Before starting your journey, you are given for each 0 *i* < *j*  *n-1*, the fee *fi, j* for renting a canoe from post *i* to post *j*. These fees are arbitrary. For example it is possible that *f1, 3* = 10 and *f1, 4*  = 5. You begin at trading post 0 and must end at trading post *n-1* (using rented canoes). Your input will be provided as a triangular matrix. **You can assume the input is in the proper form and is not empty!**  For example:

input = [[3, 5, 7, 8], # costs from post 0 -> 1, 2, 3, 4

[4, 7, 6], # costs from post 1 -> 2, 3, 4

[5, 1], # costs from post 2 -> 3, 4

[10]] # costs from post 3 -> 4

Your output is the cost of an optimal solution For this input the optimal solution has cost 6. You go from post 0 to 2 with a cost of 5 followed by going from post 2 to 4 with a cost of 1. We recommend you use this general subproblem form and recursive definition:

# opt[i] is the cheapest trip from post i and ending at post n-1.

# opt[n-2] = cost of getting from post n-2 to post n-1

# for i = n-3, …, 0

# cost is min over all first stops j (possibly none) of

# the (cost from post i to j) + opt[j]

# The solution to the problem is opt[0]

1. Your input for this problem is strings X, Y and Z. We say Z is an interleaving of X and Y if it can be obtained by interleaving the characters in X and Y. In a way that maintains the left-to-right order of X and Y and includes all characters from both X and Y.. For example, if X = babd and Y = adc, then Z = badabcd is an interleaving of X and Y , whereas Z = baaddcb is not. You are to return True if Z is an interleaving of X and Y, and False otherwise.
2. Consider the problem of optimally placing line breaks when formatting text. The input is a list W of strings and an integer M which is the maximum number of characters that can be placed on a line. A space must be placed between two words when they are in the same line. You can assume that all words have length at most M. The output should be the cost of an optimal formatting defined as minimizing the sum, over all lines except the last line, of the square of the number of extra spaces at the end of the line where the extra spaces are the number of characters required to pad out the line to M total characters.. As an example W = [“This”, “is”, “an”, “example”, “input.”] and M = 8. The optimal layout (of cost 8) is below where “^” shows the extra spaces:

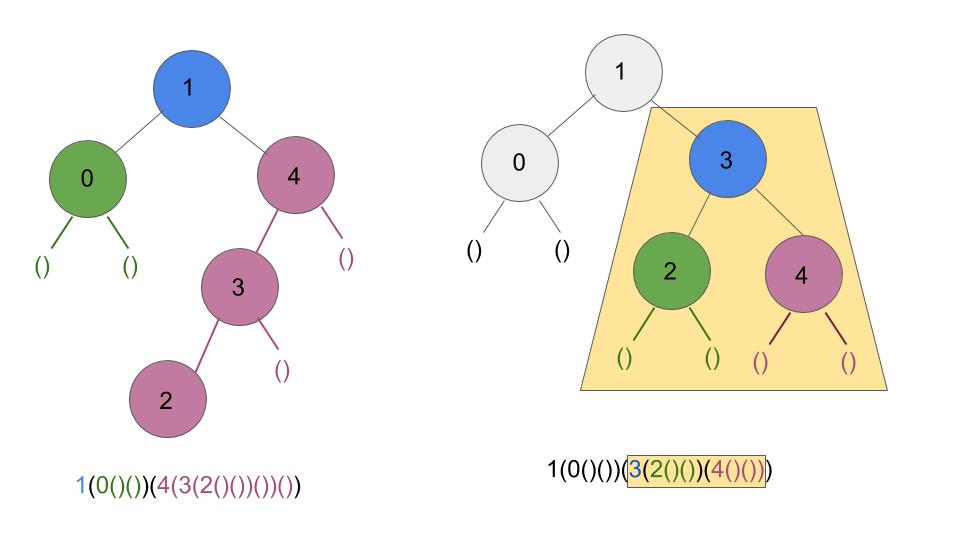
This^^^^ line cost of 4\*4 = 16

is an^^^ line cost of 3\*3 = 9

example^ line cost of 1\*1 = 1

input. line cost of 0 (since last line)

1. Consider the problem of constructing an optimal binary search tree where there are *n* keys (that we will assume are the integers 0, …, n-1). The input to this problem is a list *freq* of length *n* where we are given that key i will be searched for with probability *freq[i].* You can assume that all elements in *freq* are between 0 and 1 and that they sum to 1. The search cost, cost(*T*) = (*pi · di(T)*) where *di(T)* is the depth of key *i* in *T* where we define the root as depth 1, the second level is depth 2, and so on. The output from your code must be a tree with minimum cost returned as a string obtained via “root(left\_subtree)(right\_subtree)”. The figure below shows two trees with “()” used for null. Below each tree is the string form for it.



As an example, for input *freq* = [0.25, 0.20, 0.05, 0.20, 0.30], an optimal solution is 1(0()())(4(3(2()())())()) with cost 2.1. The tree on the right while being more balanced has a higher cost of 2.15. Confirm this for yourself. For the input: *freq* = [0.25, 0.20, 0.1, 0.20, 0.25] the optimal tree has cost 2.15. Note there could be more than one optimal subtree

***If you’d like us to help you write the recursive definition, let us know. We don’t want you spending more than 30 mins trying this on your own before getting help.***

**Dynamic Programming Homework Hints/Guidance**

For problems 3 and 4 you might find the following “standard trick” good to know. Suppose you have an array A of n elements and there is a need to compute A[i] + …. + A[j] for all 0 <= i <= j < n. If you compute this sum by scratch each time you’ll spend O(n3) time since there are O(n2) such sums that take O(n) time each. The pseudo-code below shows how to use linear time and space for preprocessing that will allow you to then compute each sum in O(1) time.

# Returns an array where element i is the sum of A[0] +... + A[i]

running\_sum(A)

n = len(A)

Create array all\_sums of n elements

sum = 0

for i = 0, …, n-1

sum = sum + A[i]

all\_sums[i] = sum

return all\_sums

# Returns A[i] + … + A[j] for 0 <= i < j < n-1 from running\_sum(A)

all\_sums = running\_sum(A)

sum(all\_sums, i, j):

ans = all\_sums[j]

if i > 0

ans = ans - all\_sums[i-1]

return ans

Why does this work? Note that all\_sums[j] = A[0]+ … A[i-1] + A[i] + …. + A[j]. If i = 0 then this is the desired output. Otherwise by subtracting sums[i-1] = A[0] + … + A[i-1] from sum[j] we are left with A[i] + … + A[j] as desired.

**Problem 1:** Your solution should have O(n2) time complexity.

**Problem 2:** Review the minimum edit distance example done in class. The form of the general subproblem is of a similar flavor. There will be some “easy” cases where you have no choice to make and the harder case where you must try both options which is why you need dynamic programming for this problem. You will have O(nm) subproblems each that takes constant time to solve for an O(nm) solution where X has n characters and Y has m characters.

**Problem 3:** Since the last line is special (with no cost for extra spaces), you might want to think about how you structure your subproblems so that the base case corresponds to all words in the subproblem fitting on the last line. To keep your solution as efficient as possible use the running sum “trick” to compute how many characters will be used by words i to j. You can do this within the code or with the pre-processing. Don’t forget to add the spaces between the words.

**Problem 4:** This is the hardest problem but also illustrates a common situation when dynamic programming is applied when you are building a tree or creating a parenthesization. We hope the output illustrates building a tree and creating a parenthesization are just two ways of viewing the same process. We recommend you first write the code to find the cost of the optimal solution. Once you’ve done that you can easily add in the secondary array and then create a separate recursive function to use this secondary array to build the solution. If you are having trouble with this, come see someone. Don’t spend a lot of time on it without asking for help. You just need to write a handful of lines of code to do this. That’s the beauty of it.

The hardest part of this problem is going through the math in computing the recursive definition which ends up having a really nice form. We help walk you through that below if you’d like the help.

Let’s go through the process of designing a dynamic programming solution. Think about each of these questions and how you’d answer it before reading on. The first thing to ask yourself when applying dynamic programming to a problem is:

**If I had an oracle to ask one question that would allow me to use the answer to create a recursive solution, what would that question be?**

If only, I knew which key k was the root of an optimal solution then I could recursively find an optimal tree for nodes 0,...., k -1 that would form the left subtree TL and also recursively find an optimal tree for the nodes k+1,...., n-1 that would form the right subtree TR. Though it takes a bit of thought to be convinced, the tree T with root k, left subtree TL and right subtree TR is an optimal tree.

**Is it possible to efficiently consider all possible roots?**

There are n possible roots, so we can just try them all. Note that if the root in the optimal tree is 0 then left subtree is empty. Likewise, if the root of the tree is n-1 then the right subtree is empty. Be sure to test these cases.

**What is the general subproblem form that results from recursively picking the root?**

If you recursively repeat this then the general subproblem form that we get is to find an optimal solution for keys i,....j (which need not sum to 1). Lest cost[i][j] be the cost of the optimal solution for the input with keys i,...j. Observe that cost[0][n-1] is the solution for the original problem.

**What is the recursive definition?**

This portion is a little tricky. For the base case if i==j then cost[i][j] = freq[i] and for convenience if i > j then cost[i][j] = 0. For the general case, the one complication to think through is that the depth of each node in both the left and the right subtree will increase by 1. Let’s just go through the math given that k is the root in an optimal solution. This means the left subtree includes i,...k-1 (possible empty) and the right subtree includes k+1,..., j (possible empty). Let cost\_L be the cost of the optimal solution for the left subtree and cost\_R be the cost of the optimal solution for the right subtree. Observe that

cost[i][j] = freq[k] +

(cost\_L + freq[i] + … + freq[k-1]) +

(cost\_R + freq[k+1] + … + freq[j])

The first line is the cost for the root. The second line is the cost for the keys in the left subtree. Observe that every node in the left subtree will be deeper by 1 than when it was as its own subproblem, and thus we must add the frequency of each key in the left subtree to the cost when the input was only keys i,...k-1. Likewise, the 3rd line is the cost for keys in the right subtree.

Just simplifying a bit we get:

cost[i][j] = cost\_L + cost\_R + freq[i] + …. + freq[j]

Now apply the running sum trick to efficiently compute the sum freq[i] + … + freq[j].

**What is the bottom up order?**

Problems of 1 key, problems of 2 keys, continuing until the full subproblem with all n keys. We saw an example like this in the classwork. You might find it useful to review that.

**How can I now build the optimal solution?**

The approach you will always use to build the solution when you’ve used dynamic programming is to have a secondary matrix with exactly the same structure as your cost matrix that stores the choice made. In this problem, that secondary array roots[i][j] will store the root for the optimal subproblem with keys i, … j. Once you have this array then a simple recursive function can be used to create the solution. Come see us if you need help on this.