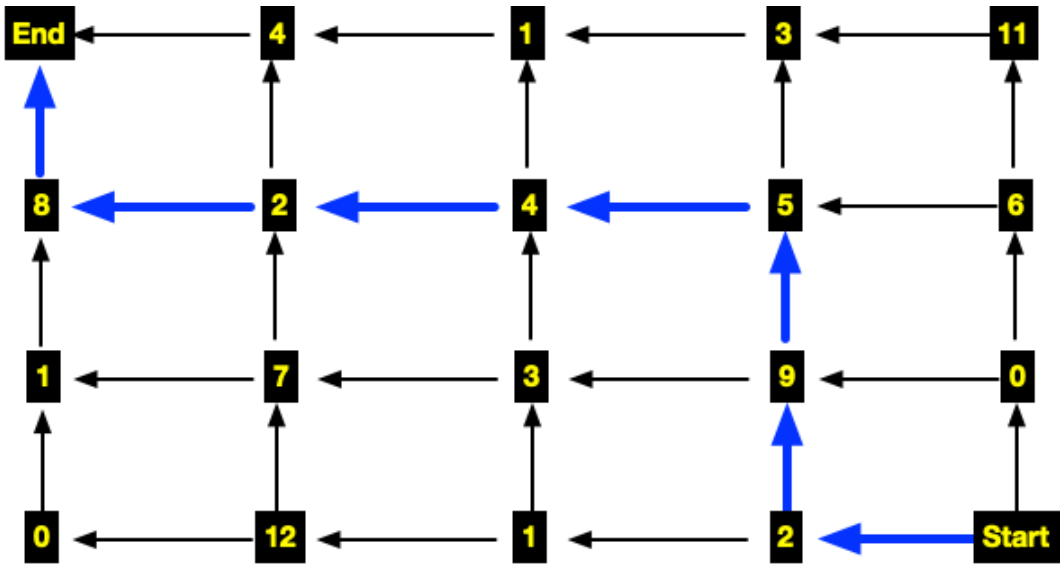
Due: Nov 21, 2022

For this homework, you will not have any proofs, just implementation problems. However, you will graph the performance of each of your problems and submit a pdf with those graphs. For each problem, solve it first without dynamic programming, measure the performance, then add dynamic programming (this should not require a major code rewrite—if you followed my advice, it is adding/removing `@cache`), then measure the performance with dynamic programming. We leave it up to you to pick reasonable values of N/data sizes/input files to perform this experimentation on.

1. For your first problem, there is a dragon in a magical treasure vault that is laid out like a grid. The dragon enters the vault at the south eastern corner and can only move North or West from any given room. The dragon may never go South nor East (because of the magic of the vault). Once the dragon leave the vault, she may not return. Each room in the vault contains an integer number of gold coins. The dragon is aware of the number of gold coins in each room before she starts her journey. For example, the vault might look like this:



Here, if the dragon takes the blue path, she collects 2+1+12+7+2+8=32 coins. The dragon wishes to maximize the number of coins she obtains so she has sought out your programming services.

Your task is to write a Python program `vault.py` which solves this problem. Your program should take one command line argument, the name of a file describing the vault, and print out three lines of output. The first line should be the path the dragon should take. This path is a series of Ns (for go North) and Ws (for go West). The second line should be one integer which is the total number of coins she obtained. The third is the time in nanoseconds to compute the solution (from `time.perf_counter_ns()`) On our example above, if the blue path were the maximum, your program would print

```
WWWNNWN
32
34827
```

Note that we do not expect the time to match the number above, nor even to be close—we are running on different hardware which

can have large impacts on the performance. Also note that you should *only* time the part where you compute the solution—do not time the part of your code where you read the input, nor print the results.

The input file format is one line per "row" in the vault, with comma separated integers (which are the number of coins in each room). Note that for simplicity, we include a number of coins in the start and exit rooms, so that each row has the same number of entries. There are some vault example files enclosed.

2. For this problem, you are going to select the order in which you perform matrix multiplications to reduce the total number of operations (in `mat.py`). Recall that multiplying an NxM matrix by an MxR matrix results in an NxR matrix. Doing this multiplication takes $O(NMR)$ work—which we will just call N*M*R operations. As matrix multiplication is associative, you can choose the order in which you multiply the matrices. For example if A is a 7x3 matrix, B is a 3x5 matrix, and C is a 5x4 matrix, you could compute the product A*B*C in two way: (A*B)*C or A*(B*C). The first way uses 105 operations to produce an intermediate result of size 7x5, then multiplies that by C taking 150 operations for a total work of 255. However, if you do A*(B*C) you do 60 work to perform that multiplication, producing a 3x4 matrix, which you multiply with A. That multiplication costs 84, for a total work of only 144—just over half as much work.

   In particular, your program will take one command line argument, which names a file. That file will list one matrix per line, with the format name,rows,columns. The example above would be:

```
A,7,3
B,3,5
C,5,4
```

Your program should print out three lines of output. The first is a fully parenthesized description of how to perform the multiplication with the fewest operations. The second is the number of operations that your solution produced. The third is the time in nanoseconds for the computation (from `time.perf_counter_ns()`)For the above example, your program should print

```
('A', ('B', 'C'))
144
13327
```

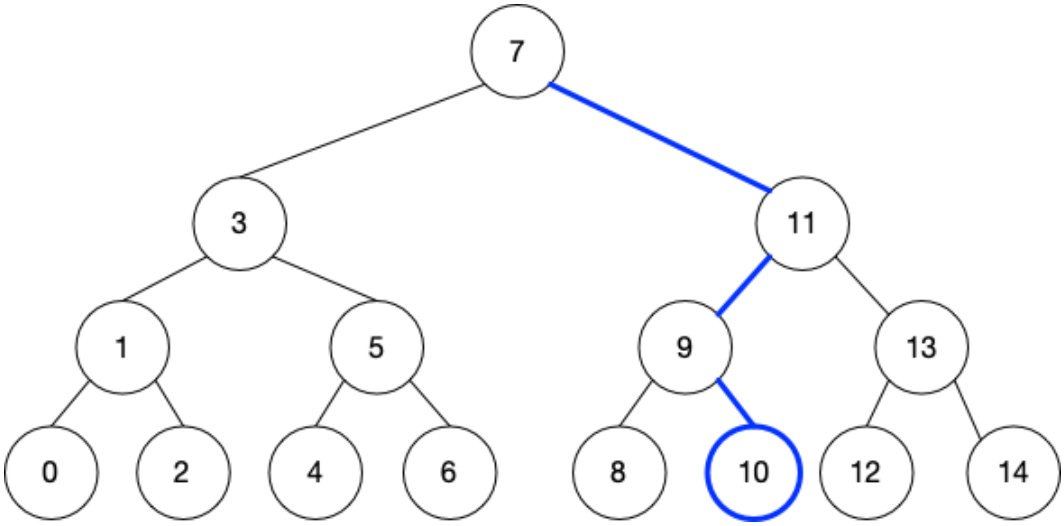
Note that the format of `('A', ('B', 'C'))` is what you get in Python if you build up a tuple and print it. That is

```
t1= ('B', 'C')
t2= ('A', t1)
print(t2)
```

will print the output in the format above (so we picked that output format to try to make things simpler for you).

3. This problem will work with binary search trees (in `tree.py`). If you aren't familiar with binary search tress, you will learn a LOT more about them in 551 soon. However, for what we are going to do here, you only need to know a few things. A binary search tree is a way to store and retrieve data. It is based on the idea that "everything larger goes right and everything smaller goes left". This property means that when searching for

a particular item, the algorithm can choose the right direction, ideally discarding half the tree at once. As an example, consider the following BST:

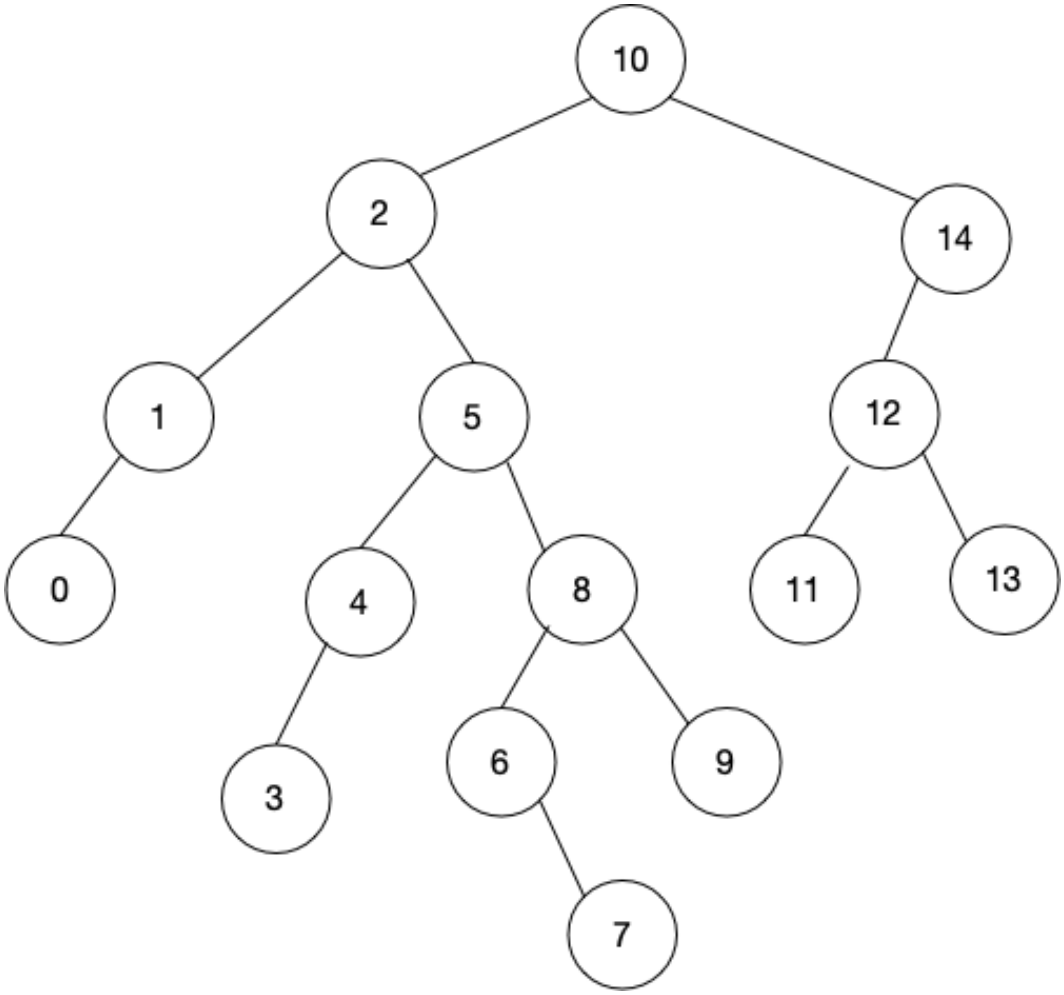


This tree has 15 elements, and is perfectly balanced. To search for 10, we only need to look at 4 nodes: 7, 11, 9, and 10. The path that our search algorithm takes is shown in blue. In the general case (which is what we will cover in 551), we want our tree to be nicely balanced like the one shown, as that minimizes the average search time for an arbitrary element. However, if we knew *a priori* that certain elements would be searched with much higher probability than others, we would want a different arrangement. For example, if we searched for 2, 10, and 14 very often, and searched for 7, 3, and 11 quite rarely, this arrangement of the tree would be inefficient. For example, suppose that we have the following probabilities for looking up a given node:

```
10: 25%
2, 14: 20% (each)
1,  5, 12: 5% (each)
0, 4, 6, 9, 13: 3% (each)
```

```
8: 2%
3, 7, 11: 1% (each)
```

Then we would prefer this tree arrangement



Even though our tree is less balanced, the nodes that we search for often require very few steps (10 is 25% and takes 1 step, 2 and 14 are 20% each and take 2 steps) and the nodes that take many steps are looked up much less often (7 takes 5 steps, but is only searched for 1% of the time). This tree yields an *expected* lookup time of 1*0.25 + 2 * 0.20*2 + 3* 0.05*3 + 4 * (0.03 *3 + 0.02 + 0.01) + 5 * (0.01+0.03*2) + 0.01 * 6 = 2.39 nodes visited per lookup. Note that this is much better than we would get on

our original balanced tree, which would be 0.01 * 1 + 2* 2* 0.01 + 3 * (2*0.05+2*0.03) + 4*(0.25 + 2*0.2 + 0.05 + 3*0.03 + 0.02) = 3.77 nodes visited per lookup (almost 40% better!)

You program will read in a file which contains the keys (integers, guaranteed to be in order, but may not be consecutive), and the relative frequency of each access, as an integer. We give the relative frequency rather than the probability so that we can use integers. Note that the main difference is that the relative frequencies may not sum to 100.

The above example would have an input file like this:

```
0:3
1:5
2:20
3:1
4:3
5:5
6:3
7:1
8:2
9:3
10:25
11:1
12:5
13:3
14:20
```

Your program will then build the optimal tree (minimum expected cost given the relative frequencies) and print it out using the `str()` function on the tree (which in turn uses the `__str__` we provided). On this input, your program should print:

```
(10 (2 (1 0 ()) (5 (4 3 ()) (8 (6 () 7) 9))) (14 (12 11 13) ()))
239
4090611
```

The first line is the an pre-order traversal of the tree (you'll learn more about that in 551: but it shows the tree's structure). The second line is the total expected cost. Note that we get 239 instead of 2.39 as we gave weights of *e.g.*, 20 instead of probabilities of 0.2. The third line is the time in nanosecond (from `time.perf_counter_ns()`), which you will use for experimenting below. Note that your first two lines should match ours. Your third (time) will not match ours, but should be generally close.

4. **Experimentation + graphing** As we noted at the start, we want you to perform a quantitative experiment on each of your programs to demonstrate the benefits of dynamic programming. For each of the problems above, you should determine sizes of inputs to run, make graphs, and write up any analysis you find relevant