CS160 Coding Project
Brett Silverberg
11/18/2021

Structure of the Code

- The project is coded into one .py file named “Largest_contig_subsequence.py”. At the top of the file, I declare the 4 lists that I used to demonstrate the capability of the project. In comments, I describe the 4 cases that the 4 lists represent.
- Below the list definitions, I define the function that can find the largest contiguous subsequence with a simple, linear time array traversal using a for loop.
- Below that I define the dynamic programming solution that recursively solves the problem. It recursively calls itself, setting the current sum to be the maximum between the ith index value, and the running sum from the last index where the sum would have been negative, all the way up to the ith index variable. It works this way because a largest contiguous subsequence will only be negative when every array value is negative. By comparing each index to the running sum, this case is taken care of by letting the function grab the maximum negative value as the largest sum. This function reaches its base case when it reaches its maximum recursive depth with an array of size one (the subset is just the last index of the list). From there it returns the maximum value between that last index’s value, and the running sum from the last negative sum up to that last index.
- Next in the file, I included two functions that just standardize the command line output so that it looks nicer. One function prints a python list on a single line, and the other prints the outputs from the 2 solution functions along with additional text for metadata and formatting.
- Finally, we call the functions manually on each list that we defined at the top of the file.

How to Run the Code

To run the code, simply use the command line to navigate to the folder containing the file “Largest_contig_subsequence.py”, and enter the line
“python Largest_contig_subsequence.py“ into the command line.
As long as you have python installed on the machine that you are accessing the file from, the file will be able to run. This project contains no imported modules; only native python functions.

How I Tested the Code

The main way that I tested the code was by testing as many edge cases as I could in the list definitions at the top of the file. I figured that the best way to test this code would be to try to break it. The test cases I used specifically are all mentioned in the comments above the list definitions in the python file

What I Learned From This Project / Conclusions

- Dynamic Programming has been my favorite topic that we’ve covered in the class so far. Using intuition to find elegant solutions for complex problems is (at least to me) what algorithms are all about. Being able to implement the concepts that we've learned was very fun, and was so far one of the more fulfilling parts of this course for me. By implementing both the linear array walk-through and the dynamic programming solution I feel like I got a much more holistic understanding of how to approach different problems. While they both take linear time, I feel that the D.P. solution is much more elegant, as it works efficiently for all subsequences. The linear array-crawl just iterates and evaluates several 'if' conditions with many constant-time checks (I can't give an exact number of constant-time operations as I don't know how Python calls its methods, but there are 3 'if' checks and 5 total possible constant-time calls). The Dynamic Programing solution, however, is only 3 total lines, with one being an 'if' check for the base case, and the other two being return calls whose only embedded operations are 2 max(a,b) calls (that return the maximum value between variable a and variable b), and a few very basic math operations (mostly incrementing the index).

Dynamic Programming Solution Steps

1. Define Subproblem - The subproblem here was finding the largest contiguous sum of any subsequence of an array
2. Base Case - When an array has only one element (i), the largest sum will be that one element. For every element i+1, the largest subsequence will be the maximum value between i and i + i+1.
3. Recursion - For every additional ith element, the largest subsequence will be the maximum value between i and i + sum at i-1.
4. Explain Recursion - In the case where i is bigger than i + sum(i-1), the sum(i-1) must be negative. A maximum sum over an entire subsequence will only be negative if every element in the subsequence is also negative, in which case this recursion will only store the maximum value in the negative array anyways. In all other cases, the maximum sum will not include all of the elements in the sum(i-1) when it is negative, so we restart the sum as being equal to i.
5. Time and Space Complexity - The time and space complexity will both be $\Theta(n)$. The way I implemented it, we always maintain the entire array with n elements, as well as ~3 variables (max sum, current sum, and in the recursive call index) which are all constant space. Additionally the D.P. solution gets called recursively on all subsequences in an array from size 1 to size n, meaning the constant time recursion is called n times, taking $\Theta(n)$ time.