Homework 4: Recursive Algorithms

Elias Gabriel

Theory

1. (9 points) Olin's dining hall is considering implementing a new dining format to improve efficiency. Instead of having stations, all the food options will be available in a single line. Further, students must choose a continuous interval of foods to take. That is, if the available options are [french fries, brussel sprouts, chicken sandwiches, tomato soup, fruit salad], then a student could take french fries, brussel sprouts, and a chicken sandwich, but not just the fries and sandwich.

   Luckily, you can use your DSA skills to find the optimal food interval for you. Suppose that there are n foods available, and let h_i be the happiness value for item i (possibly negative if you dislike a food). Give a divide-and-conquer or recursive algorithm to find the interval $[i, i+1, \ldots, j-1, j]$ that maximizes the sum of happiness scores $\sum_{k=i}^{j} h_i$. In a few sentences, convincingly argue why your algorithm finds an optimal solution, and analyze the runtime of your algorithm (it might help to write pseudocode to do the latter.) The description of your algorithm should be clear and precise enough that I could write code for it just given your solution.

Solution: A potential recursive divide-and-conquer algorithm for finding a globally maximum consecutive subsequence sum involves finding decreasingly larger subsequences (until reaching the base case of a single element), and then recombining them until an optimal solution is found. The algorithm follows a simple set of steps:

(a) Base Case: If the current range to be checked [*start*, *end*] has length 1, return the single element

(b) Get the center index for the current range and set it to a variable *middle*

(c) Get the best optimal left-handed range, starting from the middle and slowly working towards the start index. If in that loop the current index is i:

   i. Add the element at index i to a temporary variable s.

   ii. If the current sum is greater than the previous best sum l_{best}, set the. new best sum and keep track of the new best left-handed range in a variable l_{best}.

(d) Do exactly the same as the step above, except check the right-handed range (*middle*, *end*]. The middle index is omitted in the right-handed side because we don't want to double-count its value when rerecombine the two sides in the next step.

(e) Combine the ranges l_{best} and r_{best}, and their sums l_{best} and r_{best} to find the optimal range that crosses the middle index.

(f) Find the most optimal range fully on the left side of the middle index (omitting the middle) by recursively performing this entire algorithm with the new range [*start*, *mid*).

(g) Find the most optimal range fully of the right side of the middle index (omitting the middle) by recursively performing this entire algorithm with the new range (*mid*, *end*].

(h) With the three ranges found above, return the range with the maximum sum.

The recursive algorithm above works becuase at any given step, the best range that can be found exists in one of three possible places in the array:

(a) The left side of the middle element

(b) The right side of the middle element

(c) Spanning across the middle element (including it)

The base case for the algorithm is an array range containing a single element, so we know that the best possible subsequence for that single element range is the entire range. By recursively building up from that base case, getting the maximum subsequence between the three possible subsequence locations at each increasing range, we know that at any given step the returned subsequence will be the maximum for that level. Therefore we can assert that because each subsequence is the maximum at its respective level, and that the global subsequence is a combination of all of those subsequences, the global subsequence will definitely be the globally best subsequence.

At each level in the recursive tree of the algorithm, the list must be traversed $2 * \frac{n}{2}$ times, where n is the number of elements in the list at that step. At the top level of the recurrence tree, the algorithm still needs to compute the left and right sums, yielding a top-level performance of $O(n)$. Using the Master Theorem, we can therefore expresses the time complexity as:

$$T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log_2(n)) \tag{1}$$

2. (6 points) The students at Olin are suffering a new version of the freshman plague called academitis. Fortunately, this unique bug can only be passed between students during class time and there's a known cure – leweekend. Suppose that you want to distribute the cure to all potentially infected students and you've identified patient zero that started the outbreak. In other words, patient zero is the first student who actually got the disease and they could infect all their classmates who could then infect all their classmates etc. In the worst case, this could mean the whole school might be infected.

Design an your algorithm to find the list of all students who *could* have this disease. Be sure to give a clear description and mention any supporting data structures you use. Then, in a few sentences, argue the correctness of the output. You do *not* have to analyze the runtime, and you may assume that you can find a list of a student's classmates.

Solution: A list of all the possible infected students can be obtained using a depth-first traversing approach. Representing each student as a node structure that contains some boolean flag indicating whether or not that student has been seen/visited already:

(a) If the student has not been seen, add them to the potentially infected list and mark them as visited. Then recursively call the algorithm on each of that students classmates.

(b) If the student has been seen, simply do nothing since we've already processed them and their classmates.

At every level of the recursion tree, starting with patient zero, the algorithm above will check the current "primary" student and all their classmates. Because we call the algorithm recursively on each of the classmates at every depth, we know that the students at any level of the recursion tree can be traced back to patient zero either through a single classmate or multiple levels of classmates in different classes. Therefore, we know that by the end of the recursive call, every student on the final list must be connected to patient zero in some way, making them potentially infected. Without the use of a set, we ensure that our final list doesn't contain duplicate students because we check whether or not we have seen them already. Any student that we've already seen doesn't need to get added to the list again because they were added and processed at some prior level.

Practice

Analyzing the output of an algorithm on different instances can yield insight into the behavior of the algorithm. For this assignment, you will generate random inputs using the `numpy.random` package in python and compare the results. Documentation: `https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.random.html`.

3. (3 points) Implement your algorithm from problem 1 and a corresponding test function.

4. (3 points) Generate 100 random instances of happiness value lists of length $n = 100$ where each value h_i is drawn uniformly between -10 and 10. This represents a situation in which you have a wide and evenly distributed set of preferences. Record the average length and value of the max interval returned by your function. Comment on the results.

Solution: In the uniform distribution of happiness values, the average subsequence size and average score seem to both be around 50 ± 3. This makes sense, because a uniform distribution of scores would mean that every food item has an equal chance of terminating the current longest subsequence. Because half the elements in the distribution are negative, theres a 50% chance at every step of terminating. In effect, that makes the total subsequence length $\frac{100}{2} = 50$.

5. (3 points) Now, repeat the same experiments but this time each value h_i with probability 0.7 is a normal random variable with mean 6 and standard deviation 1 and with probability 0.3 is a normal random variable with mean -7 and standard deviation 0.5. In this case, you are sometimes very picky but often pretty happy. Again, comment on the results and compare to your results from above.

Solution: When using weighted distributions, the outcome is much different. Rather than a 50-50 split, the average length of the weighted subsequence hovers closer to 85 $\pm$ 5 while the average score hovers closer to 420 $\pm$ 30. That makes sense given the weights since most of the time the food choices are favorable and result in happy values. More happy values translate to longer subsequences with higher happiness ratings. The pickiness does have an impact, however, because a perfect score would be 600 given our high mean and standard deviation. 30% is enough to drop the average happiness score to around 420, which is significant but not nearly as siginifiacnt as a uniform distribution of foods.