Homework #4

Megan Ku
Data Structures and Algorithms

February 21, 2020

1 Theory

1.1 Happy Meals

We can represent the happiness scores for each of the food items as a list, where the index corresponds to the order in which the food is laid out. We can then split the list in half around its midpoint. In order to find the maximum "happy meal", we need to consider the three locations where the solution may lie:

1. The sub-list is contained entirely in the left side of the list.
2. The sub-list is contained entirely in the right side of the list.
3. The sub-list crosses the midpoint of the list.

We'll define the function `happiest_meal` to contain our algorithm for finding the happiest score. `happiest_meal` will return the starting and ending indices of the sub-list, as well as the maximum score. These values are likely best stored in a tuple, since they won't be altered. The base case for this recursive algorithm will be when the list has a length of 1. In this case, `happiest_meal` simply returns the index of the sub-list as the start and end indices, as well as the value of the element in that list. If the list has a length greater than 1, then the list gets divided in half and then `happiest_meal` gets called on both halves. For each sub-list, we need to compare the maximum value for each of the above stated cases.

To check the sum that crosses the midpoint of the list, we can create a new function that finds the largest sum that crosses the midpoint of the list. We'll call this `happiest_crossed_meal`. `happiest_crossed_meal` starts at the last index of the left half of the list and adds the numbers before it, keeping track of the highest sum and the index where the highest sum was achieved. This process will be repeated on the right half of the list, except the starting point will be the first index on the right half. Because any sum that crosses the midpoint will have to include the rightmost value of the left half and the leftmost value of the right half (quite the mouthful!), then the maximum crossing sum will be the combination of the sums generated on each side of the list. Like with `happiest_meal`, `happiest_crossing_meal` will return the sum as well as the starting and ending indices of the sub-list.

This algorithm is effective because it covers all solution cases listed above. There is no solution that lies outside those three cases. It also covers edge cases, like if the maximum sub-list is a single element or if the maximum sub-list is the entire list.

Runtime Analysis
Let n be the number of elements in the list. We know that `happiest_crossing_meal` operates in O(n) because it has to traverse the whole list, and we know that just returning the value for the max at the base case is constant time, or O(1). We also know that we keep dividing the list in half until we get to the base case. The recurrence relation for the above algorithm is as follows:

$$T(n) = O(n) + 2 * T(n/2) \tag{1}$$

$$T(1) = O(1) \tag{2}$$

Referring to the master theorem of recurrence relations, the runtime is O(nlog(n)) using asymptotic analysis.

Pseudocode

```
def happiest_meal(happy_meal, start, end):
    ‘‘‘
    Uses divide-and-conquer method to find largest sub-list sum.

    happy_meal: list of happiness values
    start: starting index of happy_meal
    end: ending index of happy_meal

    Returns (sum, start, end) where
    sum: maximum sum
    start: starting index of max sub-list
    end: ending index of max sub-list

    ’’’

    if len(happy_meal == 1):
        return (happy_meal, start, end)

    else:
        mid = len(happy_meal)//2

        happiest_left = happiest_meal(happy_meal[start:mid], start, mid)
        happiest_right = happiest_meal(happy_meal[mid+1, end], mid+1, end)
        happiest_cross = happiest_crossing_meal(happy_meal, start, mid, end)

        return max(happiest_left, happiest_right, happiest_cross)
```

```
def happiest_crossing_meal(happy_meal, start, mid, end):
    ‘‘‘
    Finds largest crossing sum.

    happy_meal: list of happiness values
    start: starting index of happy_meal
    end: ending index of happy_meal

    Returns (sum, start, end) where
    sum: maximum sum
    start: starting index of max sub-list
    end: ending index of max sub-list

    ’’’

    for i in range(start, mid, -1):
        if at mid:
            l_sum = happy_meal[mid]
            l_max_sum = l_sum
            l_index = mid
        else:
            l_sum += happy_meal[i]
            if l_sum > l_max_sum:
                l_max_sum = l_sum
                l_index = i

    for i in range(mid, end, 1):
        if at mid:
            r_sum = happy_meal[mid]
            r_max_sum = r_sum
            r_index = mid
        else:
            r_sum += happy_meal[i]
            if r_sum > r_max_sum:
                r_max_sum = r_sum
                r_index = i

    return (l_max_sum + r_max_sum, l_index, r_index)
```

1.2 Curing the Olin Academitis Plague

We can cure the Olin population in a similar way to how we found whether a path existed in the maze problem. We can use a queue to keep track of who we need to search, and have a list to store all the potentially sick students. We start by `enqueue`ing patient zero. Now, while the queue is not empty, we can `dequeue` one student from the queue and add that student to our final list. We then `enqueue` any classmates of the current student that aren't already in the final list. This process repeats until the queue is empty, which signifies that every potential student has been added to the final list. The final list can be represented as a python list, stack, DLL, queue, or any other structure that is mutable and can hold multiple values as order doesn't matter and we are only adding values to these structures.

By using a queue to keep track of students not yet checked for their classmates, we are making sure that no students are checked more than once. We are also making sure that all potential students are checked by `enqueue`ing them all.

2 Practice

2.1 Randomly Generated Food Lists

The average largest sum for the randomly generated list is around 50, while the average length of that sub-list is around 26. Basically, on average, the length of the sub-list is about 1/4 the length of the original list. This outcome seems about right given the distribution.

2.2 Probabilistic Distributed Food Lists

With the probabilistic distribution, the maximum sums and lengths are sub-lists that are almost as big as the original lists (the average length is around 90 elements). The sums are also large; the average max sum is around 230. This is because if there are a lot of positive values in the list, and if they are present at the ends of the list, the longest sub-list will include most of those values. The negative values are also close in magnitude to the positive values, which "cancel out" when next to a positive value. Overall, the results make sense.