CISC 380 – Homework #5 Solutions

Anthony Shishkin and Andy Phan

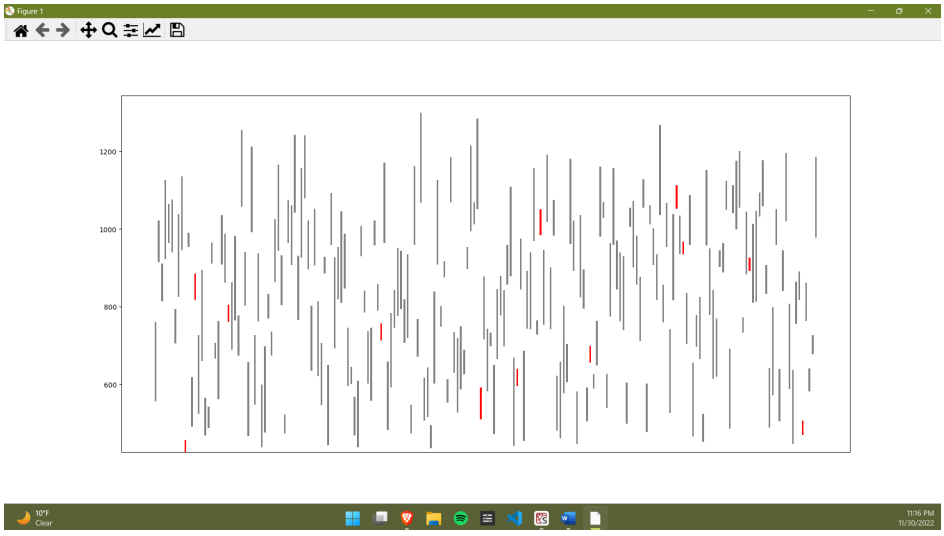
November 30th, 2022

Problem 1: Greedy Algorithms:

In this first problem we completed the methods `firstfinish()`, `shortestduration()`, and `leastconflicts()`. For the first one we sorted the `augmented` list by the finish time using `itemgetter`. Next, we created a list to store our answers. The first event will always be added to the list. Then we iterate through the sorted list and any upcoming events that have a start time which does not interfere with the finish time of the very last event in our answer, then we will include that next event in our answer. The result is a list of events prioritized by their finish time and do not conflict with each other. Below is the code:

```
def first_finish(events):
    augmented = [(i,events[i][0],events[i][1]) for i in range(len(events))]
    sortedList = sorted(augmented, key = itemgetter(2))
    ans = [sortedList[0]]
    ffIndices = [sortedList[0][0]]
    i = 1
    while i < NUM_EVENTS:
        if(sortedList[i][1] > ans[len(ans)-1][2]):
            ans.append(sortedList[i])
            ffIndices.append(sortedList[i][0])
        i += 1
    return ffIndices
```

Below is the generated events and the events in red are the ones that will be scheduled using this algorithm:



The next method, `shortestduration()`, we took a similar approach. First, we included another tuple value in the `augmented` list, and that is the duration. We can calculate this by simply getting the difference between the start and finish times and include that value in the tuple. Then, we sort the list by the shortest duration. Next, we include the event with the shortest duration in our answer. We iterate through every event in our sorted list and if that event does not conflict with any event in the answer, then we will include that event in the answer. The result is a list of events which do not conflict and are prioritized by their duration. Below is our implementation:

```
def shortest_duration(events):
    augmented = [(i,events[i][0],events[i][1], events[i][1]-events[i][0]) for i in range(len(events))]
    sortedList = sorted(augmented, key = itemgetter(3))
    ans = [sortedList[0]]
    sdIndices = [sortedList[0][0]]
    i = 1
    while i < NUM_EVENTS:
        noConflicts = True
        j = 0
        while j < len(ans) and noConflicts:
            if( not (sortedList[i][2] < ans[j][1] or ans[j][2] < sortedList[i][1])):
                noConflicts = False
            j += 1
        if(noConflicts):
            ans.append(sortedList[i])
            sdIndices.append(sortedList[i][0])
        i+=1
    return sdIndices
```

The last method is `leastconflicts()` and it uses a helper method called `getNumOfConflicts()` which takes in an event and caluclates the number of events that overlap it. We will include the number of conflicts for each event as the `augmented` list is generated. The rest of the implementation is much like what we did for `shorestduration()`. We sort by the least number of conflicts and include the event with the least as part of our final answer. Then we go through the sorted list and if the events do not interfere with any in the final answer, then we include it in the final answer. Below is our implementation:

```
def least_conflicts(events):
    augmented = [(i,events[i][0],events[i][1], getNumOfConflicts(i, events)) for i in range(len(events))
    sortedList = sorted(augmented, key = itemgetter(3))
    ans = [sortedList[0]]
    lcIndices = [sortedList[0][0]]
    i = 1
    while i < NUM_EVENTS:
        noConflicts = True
        j = 0
        while j < len(ans) and noConflicts:
            if( not (sortedList[i][2] < ans[j][1] or ans[j][2] < sortedList[i][1])):
                noConflicts = False
            j += 1
        if(noConflicts):
            ans.append(sortedList[i])
            lcIndices.append(sortedList[i][0])
        i += 1
    return lcIndices

def getNumOfConflicts(idx, events):
```

```
conflicts = 0
for i in range(len(events)):
    if( not (events[i][1]) < events[idx][0] or events[idx][1] < events[i][0]):
        conflicts += 1
return conflicts
```

The overall running time of `firstfinish()` is $O(nlogn)$ where n is the number of events. The first thing we did was create an augmented list containing the indices of each event. That runs in $O(n)$ time. Next, we used the python `sorted()` function and that runs in $O(nlogn)$ time. Next we assign values to a couple variables and that runs in constant time. Currently the dominating time complexity is $O(nlogn)$ time. Next, we use another loop to iterate through the events again to check for conflicts. This loop runs in O(n) time. Within this loop, the only operations we are doing is comparisons and appending. Since the 2 loops are not nested, we get the the overall time complexity from the python `sorted()` function and that gives us $O(nlogn)$.

Problem 2: Dynamic Programming:

In the second problem, we were tasked to create two functions that take in a string and a list containing a dictionary. From there we were told to find the maximum amount of words that could be generated from the string if it was cut perfectly. If it could not be cut perfectly, the output of the function should be "-infinity"

To start off we will first recursively solve this problem. In our solution, we used a recursive helper method to check for words rather than just one recursive function.

```
# TODO: this function should be recursive and
# ultimately too slow to run on large strings

counter = []
max_word_split_helper(s, d, counter)
return len(counter)
```

We start off by creating an empty list called `counter`. We then move call our helper function. Once that function is completed, we return the length of the counter and that should give use the final solution.

Looking at our helper function:

```
def max_word_split_helper(s, d, c):
    #Base case
    if (len(s) == 0):
        return 0
    #If the substring is in the dictionary
    elif (s in d):
        #check to see if the length of the string is greater than the current count -1
        #and see if the length of the string is less than 13
```

```
        if (len(s) >= len(c)-1 and len(s) < 13):
            #add 1 to the counter
            c.insert(0, 0)
            return 1
        else:
            return 0
```

We have 2 base cases: The first base case catches the recursive call at length 0. If it is, we return 0 when winding back up. The second case checks to see if the string is in the dictionary we passed. If it is, we check to see if the length of the string is greater than the current count - 1 and smaller than 13 in order to check the words and make sure they are correct.

After those checks, if it passes, we add one to the counter and return 1. If the string doesn't pass the check, we return 0.

```
    else:
        #for loop to iterate through the whole string
        for i in range(1, len(s)-1):
            #make the string into a substring
            anotherWord = s[:i]
            #recursivly check to see if the substring is a word
            result = max_word_split_helper(anotherWord, d, c)
            #if the substring is a word
            if (result == 1):
                #recursivly call on the final part of the string
                return max_word_split_helper(s[i:len(s)+1], d, c)
    return max_word_split_helper(s[1:len(s)], d, c)
```

The general case of the recursive function is to begin by iterating through the string at spot `i`. (i being the current letter you are on). We then create a variable to store the value s[:i] and recursively call the function on the other word.

If the resulting current variable was another word in the dictionary, we recursively call the function again, starting from s[i:len(s)+1]. This allows us to check for sub-strings. Lastly, we do one last recursive call in order starting from s[1:len(s)], allowing us to traverse the string further down.

Although we were not able to create a solution that does NOT need a helper function, this approach still gave us the correct answers.

The Recurrence Relation of our Recursive Solution: Using the information we have, we can use the

Master Theorem to find the recurrence relation. We have sub-problems of size n/b. We know that b = 2 due to the recursive calls. Our non recursive methods runs all have a maximum time complexity of O(n) however we use insert which is also a O(n) operation making our max time complexity $O(n^2)$, so we know that d = 2. $[O(n)^2 = O(n^2)]$. Lastly, our method solves one recursive sub-problem at a time, meaning a = 1. Now that we have all the variables, we can plug it all into the Master Theorem and get the Recurrence Relation

$$T(n) = (1)T(n/2) + O(n^2)$$

Solving this relation, we can use case 3 of the master theorem which makes our total time complexity of the function fall into the category O(n)

Our second method was to generate a function that could solve this problem using dynamic programming. This is how we did it:

```
#set up table
    ans = [0]*(len(s)+1)
    ans[0] = 0

    #For every letter in the string one at a time
    for k in range(1, len(s)+1):
```

We start off by creating a table with the length of the string plus one. We wanted to have a base case at spot 0. If the user were to send in an empty string, he shouldn't be able to cut the string into any amounts of words.

We then start off a loop starting at the first letter of the string and ending at the last. We used K to iterate through the problem.

```
            ans[k] = float('-inf')
                #To check for each possible substrings of k with a max length of 12
                for j in range(13):
```

We then automatically assumed that ans[k] is not a possible solution. This was used to help with calculating our sub problem. We then nest a loop with a range of 13 since we assumed (for simplicity sake) that the maximum word length is 12 characters in the dictionary.

```
                    #We check to see if the length of the substring is at least the
                    #length of the current letter we are at
                    if (k-j >= 0):
                        if (s[(k-j):k] in d):
                            #ans at spot k is max of ans[k] or ans[k-j] + 1
                            ans[k] = max(ans[k], ans[k-j] + 1)
    print(ans)
    return ans[len(s)]
```

Lastly, we check to see if the length of the substring wasn't larger than the actual iteration of the string we were on. If that check passes, we check to see if the substring is a word in our dictionary. If it was, then we finally update our table at spot k. At the very end of all the loops, we print the table and return the final entry in the table.

Our subproblem:

```
ans[k] = The maximum number of words in dictionary d that can be split perfectly from
         the input string, s.

ans[k] = max{ans[k], ans[k-j] + 1} with j being the length of all possible substrings
         in k that have a maximum length of 12
```

Time Complexity of our Dynamic Solution:

```
Seing as we have 2 nested loops, as well as a search through a dictionary,
one might assume that the time complexity of our solution is O(n^3). However,
the second loop is a constant, since it only goes up to 13 every iteration of k.

Our search through our dictionary will always be O(n) time since the dictionary
you use can be size n. The string you also input can also have a size n. Knowing
this, we see that there is a linear time search that is done 13 times per iteration
of k, which runs in O(n) time.
```

With all that in mind, The time complexity of our Dynamic Programing solution will run in

$$O(n^2)$$