CS170 — Fall 2017— Homework 6 Solutions

Jonathan Sun, SID 25020651

0. Who Did You Work With?

Collaborators: Kevin Vo, Aleem Zaki, Jeremy Ou

1. Longest Palindrome Subsequence

Main Idea:
The main idea is that we will use dynamic programming to break up the problem into subsequences. How I approached the problem is to go through the array from both ends (i starting at the 0 index and j starting at the last index) and have them work towards each other until they meet. If there are character matches, then I will recursively call the function on the part of the array between i and j and adding 2 to a count since a character match adds 2 to the length of the subsequence. My other base case will take care of if the array is odd lengthed. If the values at the i index and j index are not the same, then I will do two recursive calls on decreasing the array from both sides each and taking the maximum value of those two recursive calls. Doing so will essentially give me a tree of recursive calls.

Pseudocode:

```
LongestPalindromicSubsequence(array, i, j):
  if i == j:
    return 1
  if array[i] == array[j] and i + 1 == j:
    return 2
  if array[i] == array[j]:
    return LongestPalindromicSubsequence(array, i + 1, j - 1) + 2
  else:
    return max(LongestPalindromicSubsequence(array, i, j - 1),
       LongestPalindromicSubsequence(array, i + 1, j))
```

Run Time:
$O(n^2)$.

Justification:
The runtime is $O(n^2)$ because the recursion tree will be a tree where each layer has two times as many nodes as the layer above it. Normally, this would result in $O(2^n)$ runtime. However, each layer has nodes being solved twice, which means many subproblems are being called again. By using the memoization technique to optimize the calculations, I can avoid solving the same subproblems again. This results in each layer having one more node than the layer above it. This means that the number of nodes can be represented as $1+2+3+...+n = \sum_{i=1}^{n} i = \frac{n\,(n+1)}{2}$. This gives $O(n^2)$.

2. Bridge Hop

Main Idea:
The main idea is like that of number one where each subproblem will be looking at the remainder of the array in each iteration. For this problem, we know that the hop size can remain the same, increase by one, or decrease by one compared to the iteration before it. This means that each iteration of our i (which can vary between $-n$ and n when they only increase or only decrease in each iteration, which means of size $2n$) and iteration of our h (which can vary between 0 and n, since hop lengths cannot be negative) will make our algorithm split into 3 subsections (when hop size remains the same, increases by one, and decreases by one). Therefore, each subpart will look at the last $i-h$ part of the array with each subpart having differing sizes of h. Lastly, if the algorithm reaches the end of the bridge then it will return true and I will also have to ensure that there is not a negative hop size.

Pseudocode:

```
CanHop(V[n], i, h):
  if i <= 0:
    return True
  if h < 0:
    return False
  else:
    for i = 0; i < 2n; i++:
      for h = 0; h < n; h++:
        return V[i] and (CanHop(i - h, h - 1) or CanHop(i - h, h) or
            CanHop(i - h, h + 1))
```

Run Time:
$O(n^2)$.

Justification:
The runtime is $O(n^2)$ because the two for loops will result in $O(2n \times n) = O(2n^2) \approx O(n^2)$ runtime. Although there are 3 recursive calls in each inner iteration, the memoization will result in each layer having 2 more new calculations compared to the layer above it. Doing so will also have $O(n^2)$ runtime so the entire runtime will be $O(n^2)$.

3. A Sisyphean Task

(a) **Main Idea:**
The main idea of the algorithm is a modification of the Knapsack Problem except instead of allowing the weight to be less than or equal to k, my algorithm will need it to be exactly k. So, the idea is that I will end up recursively splitting the problem into subproblems where I will check if continuing to removing boulders of weight w_i from k will give me a weight of 0 or not. Therefore, my modification will be $Sisyphean(k, i) = Sisyphean(k, i-1) or Sisyphean(k - w_i, i-1)$. If the sum of the weights of the boulders that I have removed and their weights subtracted from k that gave 0 have a total weight of k, this would mean that there exists a set of boulders that together weigh exactly k pounds.

Pseudocode:

```
SisypheanTask(n, k):
  for i = 0; i < n; i++:
    for weight = 0; weight < k; weight++:
      if weight == 0:
        T[weight, i] = True
      if i == 0 and weight > 0:
        T[weight, i] = False
      if T[weight - w_weight, i - 1] or T[weight, i - 1]:
        T[weight, i] = True
      else:
        T[weight, i] = False
```

Run Time:
$O(kn)$.

Justification:
The runtime is $O(kn)$ because the nested for-loops give a runtime of $O(k \times n)$ and each conditional takes constant time. This means that the runtime is $O(k \times n + n) = O(kn)$.

(b) No, the algorithm polynomial is not the size of the input. This is because the algorithm polynomial is in the value of the input and not the size of the input and there is a difference between input size and input value.

4. Non-Prefix Code

Four-Part Solution:

Main Idea:

The main idea is basically that of a tree of all possible combinations of ways to represent the string. In this tree, once you traverse to the bottom node, then you know that it is one possible way to interpret s. Once you hit the bottom node, you return to the parent node and see if there are any other children and if there is, then you can traverse down until you reach the end of that traversal. Therefore, the number of ways one can interpret s is the number of times you reach an end node. To implement this idea, my algorithm checks if each binary for the items in the dictionary matches the same beginning binary encoding in the string. If it does, then the algorithm will recursively call on the rest of the string.

Pseudocode:

```
NPC_global(stringArray[], dictionary):
  global_count = 0
  NPC(stringArray[], dictionary):
    if len(stringArray) == 0:
      global_count += 1
    else:
      for d in dictionary:
        if stringArray[0 : len(dictionary[d])] == dictionary[d]:
          return NPC(stringArray[len(dictionary[d]) : ])
  return global_count
```

Proof of Correctness:

My algorithm rests on the concept that I need to find the number of distinct ways to represent the string. To do so, I need to create a tree with distinct path traversals. So, I will have to traverse through the dictionary and check if each of the key's binary values matches those at the front of the string's binary values. If they match, then it is a prefix and the recursive call will be done on the rest of the string without that prefix. Since we iterate through the dictionary every recursive call, the algorithm will ensure that every possible distinct combination is reached. $\square$

Run Time:

$O(nml)$.

Justification:

The runtime is $O(nml)$ because each time we make a recursive call, we end up iterating through the dictionary. The worst case scenario for us is if each d in dictionary is of binary length 1. This would mean that there are l bit strings and this would give us m symbols in total. Since this worst case scenario would mean that each symbol is used, this would mean that each input bit string is also matched to those m symbols. This means that there are $m \times l$ combinations for each bit. Since there are n bits in the input bit string, the total combinations would be $n \times m \times l$. Furthermore, the time needed to traverse through arrays is constant time which means our runtime is about $O(nml + n) \approx O(nml)$.

5. Lexicographically Smallest Subsequence

Four-Part Solution:

Main Idea:
The main idea for this problem is that the first character of the subsequence should be the minimum valued character in the string and the characters following it being the minimum valued characters in the string. If the minimum valued character cannot be the first character of the subsequence, then it needs to be the second character in the subsequence. If it cannot be the second character of the subsequence, then it needs to be the third character and so on. If this is the case, then the characters before it should also follow the same rule to minimize the first character as much as possible, then minimize the second character as much as possible, and so on.

Pseudocode:

```
answer = ""
LSS(array[], k, len):
  if k == 0:
    return answer
  index = 0
  min = infinity
  for i = 0; i < len; i++:
    if min > array[i]:
      index = i
  if index < len - index:
    return answer[index] + LSS(array[index + 1 : end], k - 1, len(
       array[index + 1 : end]))
  else:
    return LSS(array[0 : index], k - (len - index), len(array[0 :
       index])) + answer[index : ]
```

Proof of Correctness:
My algorithm rests on the concept that I should minimize the first character as much as possible, then minimize the second character as much as possible, and so on. So, there are two major types of subproblems. The first type of subproblem is if the smallest valued character in the string is part of the k last characters in the string, then that means the characters after that smallest value will be part of the subsequence. This means that I will need to search for the minimum valued character in the string in fron the the kth character. The other type of subproblem is if the smallest valued character in the string is not part of the k last characters in the string. This means that I will need to find the minimum valued characters after it. □

Run Time:
$O(nk)$.

Justification:
The runtime is $O(nk)$ because in each subproblem, the algorithm will find the minimum character in the array that is passed into that subproblem. Doing so will take $O(n)$ time since we need to look through the elements of that array. We will also be doing k recursive calls because each recursive

call will decrease k by 1 until it reaches 0. This means that for each recursive call (there are k of recursive calls), the algorithm will find the new minimum, which takes n time. Therefore, the total runtime will be $O(n \times k) = O(nk)$.