Joshua Sloan
University of Wyoming
jsloan3@uwyo.edu

Homework 3
COSC 5110 - Analysis of Algorithms

1 Exercise 6.4

You are given a string of n characters $s[1...n]$, which you believe to be a corrupted text document in which all punctuation had vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function dict(): for any string w,
dict(w) = true, if w is a valid word
false, otherwise
(a)
Give a dynamic programming algorithm that determines whether the string s[] can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to dict() take unit time.
validSeq(S[1...n]**)**

```
0: valArr[length(S)+1] // create array of size n+1
0: valArr[0] = TRUE
  for i=1 to n+1 do
    if i == 1 then
      if dict(S[i-1]) == TRUE then
        valArr[i] = TRUE
      end if
    else
      j=i
      string = S[j-1]
      // loop until either valid sequence is found or we have scanned back to the beginning of the array
      while TRUE do
        string = concatenate(S[j-2],string) // we can confirm that this will never be an issue given i ≥ 2
        if dict(string) == TRUE then
          if valArr[previous] == TRUE then
            valArr[i] = TRUE
            BREAK
          end if
        end if
        if valArr[previous] == valArr[0] then
          valArr[i] = FALSE
          BREAK
        end if
        j = j-1
      end while
    end if
  end for
  return valArr[length(S)]
```

In the worst case, for every i, where i=1 to n+1 (n), we must traverse all the way back through the length of valArr to determine that there is no instance in which a substring (the entire string in this case) is a valid dictionary word, which is clearly quadratic in time (plus a constant amount of additional work). However, in most cases we are able to reduce this time by breaking the while loop early, rather than waiting until we arrive at the first element in valArr, given we can utilize previously learned information to determine that the previous substring has a valid sequence based on the prior label of TRUE at a point.

(b)
In the event that the string is valid , make the your algorithm output the corresponding sequence of words.
NOTE: Everything until the return statement is the same from (a)
validSeq(S[1...n]**)**

```
0: valArr[length(S)+1] // create array of size n+1
0: valArr[0] = TRUE
  for i=1 to n+1 do
    if i == 1 then
      if dict(S[i-1]) == TRUE then
        valArr[i] = TRUE
      end if
    else
      j=i
      string = S[j-1]
      // loop until either valid sequence is found or we have scanned back to the beginning of the array
      while TRUE do
        string = concatenate(S[j-2],string) // we can confirm that this will never be an issue given i ≥ 2
        if dict(string) == TRUE then
          if valArr[previous] == TRUE then
            valArr[i] = TRUE
            BREAK
          end if
        end if
        if valArr[previous] == valArr[0] then
          valArr[i] = FALSE
          BREAK
        end if
        j = j-1
      end while
    end if
  end for
  // THIS IS WHERE ADDITIONS ARE MADE TO THE PREVIOUS ALGORITHM
  if valArr[length(S)] == TRUE then
    stk = new stack //create an empty stack data structure
    while length(valArr) != 1 do
      iterate backwards through valArr until next instance of TRUE (at location p)
      if dict(concatenate(S[p],...,S[length(valArr)-1])) == TRUE then
        stk.push(concatenate(S[p],...,S[length(valArr)-1]))
        valArr = valArr[0,p] // remove corresponding characters from valArr
      end if
    end while
    sentence = "" //empty string
    while stk != empty do
        sentence = concatenate(sentence,stk.pop) if stk != empty then
        sentence = concatenate(sentence," ")
        add a space
      end if
    end while
    return sentence
```

Because we are building off of the previous problem, we only need to ensure that the extra work done is no greater than the bound of the previous implementation (given that is all contained within a separate external loop). This makes the analysis easy, given that we can clearly see that in the worst case we are simply just iterating backward through valArr for the length of the array, with an additional constant amount of work. In the worst case, our string contains exclusively single character words (i.e. "$aaaaaiiiiaaa$"), such that the total work to push all words onto and off the stack would be $O(n) + O(n) = O(n)$. Consequently, the total bound for the improved algorithm is $O(n^2) + O(n)$ which is still $O(n^2)$.

2 Exercise 6.8

Given two strings $x = x_1x_2...x_n$ and $y = y_1y_2...y_m$, we wish to find the length of their longest common substring, that is, the largest k for which there are indices i and j with $x_ix_{i+1}...x_{i+k-1} = y_jy_{j+1}...y_{j+k-1}$. Show how to do this in time $O(mn)$.

LCS(x=$[x_1x_2...x_n]$, y=$[y_1y_2...y_m]$ **)** //input is strings x and y of length n and m

0: k = 0 //initialize largest k to be 0
0: lcsMatrix[length(x)][length(y)] //create (n x m) matrix
for i=0 to n **do**
  for j=0 to m **do**
    if x[i] == y[j] **then**
      if i == 0 or j == 0 **then**
        lcsMatrix[i][j] = 1 //account for not having an i-1 or j-1 element in matrix
      else
        lcsMatrix[i][j] = lcsMatrix[i-1][j-1] + 1
      end if
    else
      lcsMatrix[i][j] = 0
    end if
    k = max(k,lcsMatrix[i][j])
  end for
end for
return k

3 Excercise 6.22

Give an $O(nt)$ algorithm for the following task.
$Input$: A list of n positive integers $a_1, a_2, ..., a_n$; a positive integer t.
$Question$: Does some subset of the a_i's add up to t? (You can use each a_i at most once.)

subsetSum(A[1...n],t**)** //input is a list of positive integers A and a positive integer t

0: ssMatrix[length(A)][t] //create (n x t) matrix
for i=0 to length(A)-1 **do**
  for j=1 to t **do**
    if i == 0 **then**
      if A[i] == j **then**
        ssMatrix[i][j-1] = TRUE
      else
        ssMatrix[i][j-1] = FALSE
      end if
    else
      if A[i] > j **then**
        ssMatrix[i][j-1] = ssMatrix[i-1][j-1] //take the value from the cell above
      else if A[i] == j **then**
        ssMatrix[i][j-1] = TRUE
      else if A[i] < j **then**
        //return true if either cell above is true || A[i] back from cell above is true,
        //otherwise false
        ssMatrix[i][j-1] = (ssMatrix[i-1][j-1] or ssMatrix[i-1][j-1-A[i]])
      end if
    end if
  end for
end for
return ssMatrix[length(A)-1][t-1] //return bottom right corner cell value of matrix

4

In the Minimum Array Partition problem we are given an array A[1..n] of n integers. The goal is to partition A into contiguous subarrays such that the maximum subarray sum is minimized. For a small example, suppose
A = [24 0 88 -59 -54 13 20 -11 22].
An optimal partition is
[24 0 88 -59 -54] [13] [20] [-11 22], with a score of 20.

(a)

Design an efficient algorithm for the Minimum Array Partition problem, argue its correctness, and analyze its run time.

SEE NEXT PAGE

```python
# get the minimum array partition from the input array: O(n^3) run-time
def minArrayPartition(arr):
    n = len(arr)
    k = n
    sumArr = [0]*(n+1)

    # O(n)
    for i in range(1,n+1):
        sumArr[i] = sumArr[i-1] + arr[i-1]

    # initialize the matrix O(n^2)
    mapMatrix = [[0 for i in range(n + 1)]
            for j in range(k + 1)]

    # O(n)
    for i in range(1, n+1):
        mapMatrix [1][i] = sumArr[i]

    # O(n)
    for i in range(1, k + 1):
        mapMatrix [i][1] = arr[0]

    # O(nk) where k = n S.T. O(n^2) with inner O(n) loop = O(n^3)
    for i in range(2, k + 1):
        for j in range(2, n + 1):

            best = math.inf

            #O(n) internal loop
            for p in range(1, j + 1):
                best = min(best, max(mapMatrix [i-1][p],
                    sumArr[j]-sumArr[p]))

            mapMatrix [i][j] = best #O(1)

    return mapMatrix[-1][-1]
```

Figure 1: Minimum Array Partition Algorithm

We can assure the correctness of the algorithm's score attributed to the maximum partition subset, given that it is a very similar instance to the case proposed in the painters partition problem. The distinct difference is that we are not limited by a choice of k, such that it is possible for us to utilize up to n contiguous subset partitions if that is the optimal value for k in our instance (however, this is extremely unlikely given that would entail only positive, or only negative instance values). As a result, the algorithm will only change values for the optimal number of partitions needed to minimize the maximum contiguous subset, where subsequent row values will be pulled from the row above (seen in the internal O(n) loop of the $O(n^3)$ nested set. Therefore, while we were not able to traverse the mapMatrix to obtain the optimal partitioned split values, we can guarantee the correctness of the associated score.
Received help/inspiration from: https://www.geeksforgeeks.org/painters-partition-problem/

(b)

Implement your algorithm and solve the instances in the files test500, test1000, and test2000. The first line of each file is a number n and the second line contains n integers separated by spaces. Your algorithm should print an optimal partition and its score. Submit your code and its output on these instances on WyoCourses.
I worked with Mason and Danny on this assignment.