Homework 5

Due by 4:30pm on October 14, 2020

Viewings and Readings

[Links to the slides, recordings, and quiz are on the course web site.]
Review the week 5 lecture slides.
View the week 5 lecture recordings and the discussion session recording.
Complete the week 5 quiz.
Read sections 3.1-6 of the textbook.

Problems

Please write precise and concise answers. Except where explicitly indicated, your algorithm descriptions should use either clear, concise, and precise plain English or clear, concise, and precise pseudo-code that uses a style similar to the pseudo-code in your textbook. Submit your solutions to problems **1-4(a)** via D2L as a Word or PDF file or as scans/photos of legible handwritten notes. Submit your solutions to problems **4(b)** and **5** via Kattis.

1. Problem 4(b) page 95 in your textbook.

Let A[1 .. m] and B[1 .. n] be two arbitrary arrays. A common supersequence of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function scs(A, B), which gives the length of the shortest common supersequence of A and B.

```
The shortest supersequence would be the same as the longest common subsequence plus the other letters that arent in the LCS.
The formula of the shortest supersequence of A and B would be len(A) + len(B) - len(LCS of A and b)

So in our function we can just find the length of A and B and then use recusion to find the LCS and subtract the length from the sum of A and B leng

LCS recursions is something we've defined before, it would look like

lcs(A[i .. n], B[j .. n]):
  if length A == 0 OR length B == 0
    return 0
  if A[1] == B[1] {
    return lcs(A[i + 1 .. n], B[j + 1 .. n]) + 1
  } else {
    if length A > length B {
      return lcs(A[i + 1 .. n], B[j .. n])
    } else {
      return lcs(A[i .. n], B[j + 1 .. n])
    }
  }

Our main function would then be

shortestSupersequence:
  lenA = length(A)
  lenB = length(B)
  lenLCS = lcs(A, B)

  return lenA + lenB - lenLCS
```

2. Problem 2(a) page 124 in your textbook. Do this by going through the steps 1(a), 1(b), 2(a), ... covered during in lecture 5, that one needs to do in order to develop a dynamic programming algorithm. Steps 1(a) and 1(b) are already done for you since they are solutions to problem 3 in homework 4. Therefore focus on steps 2(a), 2(b), 2(c), 2(d), 2(e) and 2(f).

Describe efficient algorithms for the following variants of the text segmentation problem. Assume that you have a subroutine IsWord that takes an array of characters as input and returns True if and only if that string is a "word". Analyze your algorithms by bounding the number of calls to IsWord.

(a) Given an array A[1 .. n] of characters, compute the number of partitions of A into words. For example, given the string ARTISTOIL, your algorithm should return 2, for the partitions ARTIST-OIL and ART-IS-TOIL.

```
Steps 1a and 1b were solved for this in a previous HW with backtracking and the algorithm

Splittable(i):
    if i > n
      return 1
    result = 0
    for j = i to n
      if IsWord(i, j)
      result += Splittable(j + 1)
    return result

2a Each subproblem is Splittable(i) between i and n + 1
2b Our memozation structure should be a array with each index i being the number of times that index was used in a spitable solution
2c This means i depends on solutions to splitable greater than i
2d So we have to build our solution up from right to left on the array
2e We can slightly modify the function in lecture to solve this

FastSplittable(A[1..n]):
  SplitTable[n + 1] <- 1
  for i <- n down to 1
    SplitTable[i] <- 0
    for j <- i to n
      if IsWord(i, j) and SplitTable[j + 1] > 0
```

```
        SplitTable[i] += 1
  return SplitTable[1]


SplitTable[1] will be the number of times the whole array A has a different valid splittable solution
```

2f This run in $O(n^2)$

3. There are n trading posts along a river numbered 1, 2, 3, ..., n. At any of the posts you can rent a canoe to be returned at any other post downstream. (It is impossible to paddle against the river, since it is too fast...) For each possible departure point i and each possible arrival point j (> i), the cost of a rental from i to j is known: it is C[i,j]. However, it can happen that the cost of renting from i to j is higher than the total costs of a series of shorter rentals. In this case you can return the first canoe at some post k between i and j and continue your journey in a second (and, maybe, third, fourth ...) canoe. There is no extra charge for changing canoes in this way.

Give a $O(n^2)$ time dynamic programming algorithm to determine the minimum cost of a trip from trading post 1 to trading post n.

To develop your algorithm, go through the steps 1(a), 1(b), 2(a), ... covered during in lecture 5, that one needs to do in order to develop a dynamic programming algorithm. In particular, you will need to describe 1(a) the problem that needs to be solved recursively and then 1(b) give a recursive solution to that problem. To find the problem that needs to be solved recursively, start with the initial problem of computing the minimum cost of a trip from trading post 1 to trading post n and find a way to describe a solution for it in terms of solutions to sub-problems of that problem. Then proceed with steps 2(a), 2(b). etc.

```
1a Recursively backtrack through all possible stops from i-n and return the minimum sum

n // total posts
C[] // array of costs from i->j
Cost(i, j):
  if j+1 == j:
    return C[i,j]
  min = C[i,j]
  for stopPoint <- i to j:
    tempCost = Cost(i, stopPoint) + Cost(stopPoint, j)
    if(tempCost < min) {
      min = tempCost
    }
  return min

2a Each subproblem is Cost(i, j)for each i and j where j > i
2b Our memozation structure could be an array where index i contains the minimum cost from i to n
2c i then depends on solutions to Cost > i
2d We can build our memozation structure up down starting from the solution (n-1, n)
2e

FastCost(C[1...n])
  costTable[n-1] = C[n-1, n]
  for i <- n - 2 down to 1:
    min = C[i, n]
    for j <- i to n:
      // check all stops from i to n + the existing calulations of the min of the rest of the trip
      tempMin = C[i, j] + costTable[j + 1]
      if(tempMin < min) {
        min = tempMin
      }
    costTable[i] = min
  return costTable[1]
```

2f Two for loops -> n makes this $O(n^2)$

4. Week 5 problem *commercials* on Kattis.

(a) Describe the recursive solution algorithm for the commercials problem. (Hint: note that the problem reduces to the maximum sum sub-array problem we did in Homework 2. Then, if X[1..n] is the input subarray, note that the maximum sum subarray of X[1..j] is either the maximum sum subvector in X[1..j-1] or it is a subarray that ends in position j. The maximum sum subarray that ends in position j is either empty or includes the maximum sum subarray that ends in position j-. Use these two insights to express the sum S[1..j] of the maximum sum subarray of X[1..j].

```
Insight #1 implies that we can loop through the array X and at each
index the max sum has either been found before or its a subarray that ends on the current index j.
So this implies we only really need to loop through X once which would give us O(n) running time

Insight #2 implies that a subarray that ends at position j either adds on the
the sub array of j-1 or ignores what came before in j-1.
So we can use this to check the max of index j or index j + sub array sum ending at j-1

With these two insights we can loop through each index of X.
At index 1 the max subarray is just X[1], but at two it's the max of X[1] + X[2] or just X[2]
We save each max in an array and keep comparing until we reach the end of X
Then whatever the max value in the array is will be the max subset sum
```

(b) Implement your solution using your preferred language and submit your implementation via Kattis.

5. [Optional] Week 5 problem *uxuhulvoting* on Kattis. Submit your solution via Kattis.