Homework 4

Due by 4:30pm on October 7, 2020

Viewings and Readings

[Links to the slides, recordings, and quiz are on the course web site.]
Review the week 4 lecture slides.
View the week 4 lecture recordings and the discussion session recording.
Complete the week 4 quiz.
Read sections 2.5-8 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 **1-5** via D2L as a Word or PDF file or as scans/photos of legible handwritten notes. Submit your solution to problem **6** via Kattis.

1. Problem 14(a) page 51 in your textbook.
Suppose you are given two sets of n points, one set {p1 , p2 , . . . , pn } on the line y = 0 and the other set {q1 , q2 , . . . , qn } on the line y = 1. Create a set of n line segments by connect each point p_i to the corresponding point q_i . Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in O(n log n) time. [Hint: See the previous problem.

```
Two lines will intercect in two scenarios
Either p[i] > p[j] AND q[i] < q[j]
OR
p[i] < p[j] AND q[i] > q[j]

We can solve this by with two merge sorts with some extra functionality

We do the classic mergesort on set p.
Recurseively dividing p in half and using the left and right halfs as input in recusrion
Base case is when there is only one element left in the array
Merge the left and right together by going through left and right and adding them to new array sorted
Any sorting done When the left index > right index we need to make sure the same index changes on set Q

This Recurseively runs until p is completely sorted and q also has the same changes

We can then run the merge sort again on q with one difference
Whenever left > right we add one to a counter
Since p is sorted, any sorting that happens in q means theres an intercect based on the scenarios we layed out earlier

We return that count at the end of the sort and thats the number of intersections
```

2. Problems 24(a)(b)(c) page 57 in your textbook.
Consider the following classical recursive algorithm for computing the factorial n! of a non-negative integer n:

```
Factorial(n):
  if n = 0
    return 1
  else
    return n * Factorial(n - 1)
```

(a) How many multiplications does this algorithm perform?

n multiplications are performed

(b) How many bits are required to write n! in binary? Express your answer in the form Θ(f (n)), for some familiar function f (n). [Hint: $(n/2)^{n/2} < n! < n^n$.]

(c) Your answer to (b) should convince you that the number of multiplications is not a good estimate of the actual running time of Factorial. We can multiply any k-digit number and any l-digit number in O(k * l) time using either the lattice algorithm or duplation and mediation. What is the running time of Factorial if we use this multiplication algorithm as a subroutine?

O(n)

3. Problem 2(a) page 94 in your textbook. Do this by appropriately modifying the pseudo-code on page 84 of the textbook.
Describe recursive 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".
(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.

```
{{Is the suffix A[i .. n] Splittable?}}
  // set global var sum to keep track of all valid answers
  sum = 0

  Splittable(i):
    if i > n
      return true
    for j = i to n
```

```
      if IsWord(i, j)
        if Splittable(j + 1)
          sum = sum + 1
    return False
```

4. Problem 4(a) page 94 in your textbook. You may answer this question with a recursive formula or an algorithm described using pseudo-code. Hint: compare `A[1]` and `B[1]`, consider the two cases `A[1]==B[1]` and `A[1] != B[1]`, and write/compute the `LCS(A[1..n], B[1..n])` in terms of `LCS(A[2..n],B[2..n])`, `LCS(A[1..n],B[2..n])`, and `LCS(A[2..n], B[1..n])`.

(a) Let A[1 .. m] and B[1 .. n] be two arbitrary arrays. A common subsequence of A and B is both a subsequence of A and a subsequence of B. Give a simple recursive definition for the function lcs(A, B), which gives the length of the longest common subsequence of A and B.

```
Basically recursively check if the first index of each matches
If they do move on to the next character for both
If not remove the first index of the longer one and recurse until one array is empty

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])
    }
  }
```

[One, but not both, of problems 5. and 6. is optional]

5. Problem 6(a) page 96.

This problem asks you to design backtracking algorithms to find the cost of an optimal binary search tree that satisfies additional balance constraints. Your input consists of a sorted array A[1 .. n] of search keys and an array f [1 .. n] of frequency counts, where f [i] is the number of searches for A[i]. This is exactly the same cost function as described in Section 2.8. But now your task is to compute an optimal tree that satisfies some additional constraints.

(a) AVL trees were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most one. Describe a recursive backtracking algorithm to construct an optimal AVL tree for a given set of search keys and frequencies.

```
We can use the formula from lecture that calculates total cost recursively for each specific node chosen to be the root
Cost(T, f [1..n]) =SUM(i=1 -> n for f[i]) + Cost(LEFT SUBTREE(T), f [1..r - 1]) + Cost(RIGHT SUBTREE(T), f [r + 1])

To do a normal cost optimization of a tree we
We pass in as input the tree, height of current node, and A
For each A[i] we insert A[i] into the tree
We recursively cut A in half and run and sum the cost functions on the left and right half of the array so the new A[] will be either the left or
We save this sum in a minValue var and change whats saved if any other node being the root creates a more optimal cost
After the sums return we can also check here whether the tree is AVL, if the height of the left subtree - height of right subtree is > 1 or < -1 t
We keep cutting the array in half until the base case on one element in the array in which case the forumla above would equal Cost(T, f [1..n]) =S
```

6. Week 4 problem *boggle* on Kattis. Submit your solution via Kattis.