CS/ECE 374 A ✧ Fall 2019

Homework 6

Due Tuesday, October 22, 2019 at 8pm

---

1. A non-empty sequence $S[1..\ell]$ of positive integers is called a ***perfect ruler sequence*** if it satisfies the following conditions:

   - The length of S is one less than a power of 2; that is, $\ell = 2^k - 1$ for some integer k.
   - Let $m = \lceil \ell/2 \rceil = 2^{k-1}$. Then $S[m]$ is the unique maximum element of S.
   - If $\ell > 1$, then the prefix $S[1..m-1]$ is a perfect ruler sequence.
   - If $\ell > 1$, then the suffix $S[m+1..\ell]$ is a perfect ruler sequence.

   For example, the following sequence is a perfect ruler sequence:

   $$\langle 2, 7, 6, 9, 5, 8, 5, 12, 1, 9, 4, 10, 7, 8, 3 \rangle$$

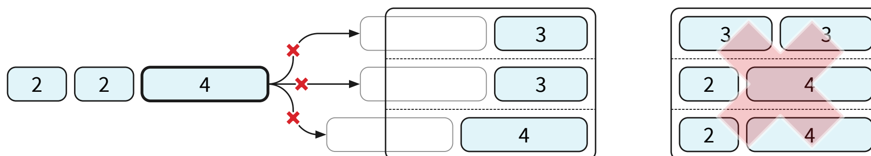
   Describe and analyze an efficient algorithm to compute the longest perfect ruler subsequence of a given array $A[1..n]$ of integers.

2. Suppose you are running a ferry across Lake Michigan.[1] The vehicle hold in your ferry is L meters long and three lanes wide. As each vehicle drives up to your ferry, you direct it to one of the three lanes; the vehicle then parks as far forward in that lane as possible. Vehicles must enter the ferry in the order they arrived; if the vehicle at the front of the queue doesn't fit into any of the lanes, then no more vehicles are allowed to board.

   Because your uncle runs the concession stand at the ferry terminal, you want to load as *few* vehicles onto your ferry as possible for each trip. But you don't want to be *obvious* about it, if the vehicle at the front of the queue fits anywhere, you must assign it to a lane where it fits. You can see the lengths of all vehicles in the queue on your security camera.

   Describe and analyze an algorithm to load the ferry. The input to your algorithm is the integer L and an array $len[1..n]$ containing the (integer) lengths of all vehicles in the queue. (You can assume that $1 \le len[i] \le L$ for all i.) Your output should be the *smallest* integer k such that you can put vehicles 1 through k onto the ferry, in such a way that vehicle $k+1$ does not fit. Express the running time of your algorithm as a function of both n (the number of vehicles) and L (the length of the ferry).

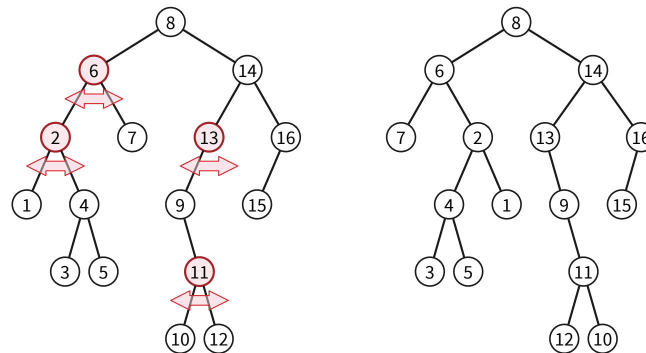
   For example, suppose $L = 6$, and the first six vehicles in the queue have lengths 3, 3, 4, 4, 2, and 2. Your algorithm should return the integer 3, because if you assign the first three vehicles to three different lanes, the fourth vehicle won't fit. (A different lane assignment gets all six vehicles on board, but that would rob your uncle of three customers.)



[1] Welcome aboard the Recursion Ferry! How we get across the water is none of your business!

3. CS 125 students Chef Gallon and Fade Waygone wrote some inorder traversal code for their MP on binary search trees. To keep things simple, they wisely chose the integers 1 through n as their search keys. Unfortunately, their code contained a subtle bug (which was nearly impossible to track down, thanks to version inconsistencies between Fade's laptop, the submission/grading server, and Oracle's ridiculous licencing terms) that would sporadically swap left and right child pointers in some binary tree nodes. As a result, their traversal code rarely returned the search keys in sorted order.

   For example, given the binary search tree below, if the four marked nodes had their left and right pointers swapped, Chef and Fade's traversal code would return the garbled "inorder" sequence $7, 6, 3, 4, 5, 2, 1, 8, 13, 9, 12, 11, 10, 14, 15, 16$.



   Chef and Fade submitted the output of several garbled traversals, but before they could submit the actual traversal code, Fade's laptop was infested with bees. After receiving a grade of 0 on their MP, Chef and Fade argued with their instructor that they should get *some* partial credit, because the sequences their code produced were at least consistent with correct binary search trees, and anyway the bees weren't their fault.

   Design and analyze an efficient algorithm to verify or refute Chef and Fade's claim (about the binary search trees, not the bees). The input to your algorithm is an array $A[1\,..\,n]$ containing a permutation of the integers 1 through n. Your algorithm should output TRUE if this array is the inorder traversal of an actual binary search tree with keys 1 through n, possibly with some left and right child pointers swapped, and FALSE otherwise. For example, if the input array contains $[5, 2, 3, 4, 1]$, your algorithm should return TRUE, and if the input array contains $[2, 5, 3, 1, 4]$, your algorithm should return FALSE.

Solved Problems

4. A string w of parentheses (and) and brackets [and] is ***balanced*** if and only if w is generated by the following context-free grammar:

$$S \to \varepsilon \mid (S) \mid [S] \mid SS$$

For example, the string $w = ([()][]())[()()]()$ is balanced, because $w = xy$, where

$$x = ([()][]()) \quad \text{and} \quad y = [()()]().$$

Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets. Your input is an array $A[1..n]$, where $A[i] \in \{(,),[,]\}$ for every index i.

Solution: Suppose $A[1..n]$ is the input string. For all indices i and k, let $\boldsymbol{LBS(i,k)}$ denote the length of the longest balanced subsequence of the substring $A[i..k]$. We need to compute $LBS(1,n)$. This function obeys the following recurrence:

$$LBS(i,j) = \begin{cases} 0 & \text{if } i \geq k \\ \max \left\{ \begin{array}{c} 2 + LBS(i+1,k-1) \\ \max\limits_{j=1}^{k-1} \big(LBS(i,j) + LBS(j+1,k)\big) \end{array} \right\} & \text{if } A[i] \sim A[k] \\ \max\limits_{j=1}^{k-1} \big(LBS(i,j) + LBS(j+1,k)\big) & \text{otherwise} \end{cases}$$

Here $\boldsymbol{A[i] \sim A[k]}$ indicates that $A[i]$ and $A[k]$ are matching delimiters: Either $A[i] = ($ and $A[k] =)$ or $A[i] = [$ and $A[k] =]$.

We can memoize this function into a two-dimensional array $LBS[1..n, 1..n]$. Since every entry $LBS[i,j]$ depends only on entries in later rows or earlier columns (or both), we can evaluate this array row-by-row from bottom up in the outer loop, scanning each row from left to right in the inner loop. The resulting algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

```
LongestBalancedSubsequence(A[1..n]):
  for i ← n down to 1
      LBS[i,i] ← 0
      for k ← i+1 to n
          if A[i] ~ A[k]
              LBS[i,k] ← LBS[i+1,k−1] + 2
          else
              LBS[i,k] ← 0
          for j ← i to k−1
              LBS[i,k] ← max{LBS[i,k], LBS[i,j] + LBS[j+1,k]}
  return LBS[1,n]
```

■

Rubric: 10 points, standard dynamic programming rubric

5. Oh, no! You've just been appointed as the new organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: An employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all.

   Describe an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests. The input to your algorithm is a rooted tree T describing the company hierarchy, where each node v has a field $v.fun$ storing the "fun" rating of the corresponding employee.

Solution (two functions): We define two functions over the nodes of T.

- *MaxFunYes*(v) is the maximum total "fun" of a legal party among the descendants of v, where v is definitely invited.
- *MaxFunNo*(v) is the maximum total "fun" of a legal party among the descendants of v, where v is definitely not invited.

We need to compute *MaxFunYes*(*root*). These two functions obey the following mutual recurrences:

$$MaxFunYes(v) = v.fun + \sum_{\text{children } w \text{ of } v} MaxFunNo(w)$$

$$MaxFunNo(v) = \sum_{\text{children } w \text{ of } v} \max\{MaxFunYes(w), MaxFunNo(w)\}$$

(These recurrences do not require separate base cases, because $\sum \varnothing = 0$.) We can memoize these functions by adding two additional fields $v.yes$ and $v.no$ to each node v in the tree. The values at each node depend only on the vales at its children, so we can compute all $2n$ values using a postorder traversal of T.

```
BestParty(T):
  ComputeMaxFun(T.root)
  return T.root.yes
```

```
ComputeMaxFun(v):
  v.yes ← v.fun
  v.no ← 0
  for all children w of v
      ComputeMaxFun(w)
      v.yes ← v.yes + w.no
      v.no ← v.no + max{w.yes, w.no}
```

(Yes, this is still dynamic programming; we're only traversing the tree recursively because that's the most natural way to traverse trees![a]) The algorithm spends $O(1)$ time at each node, and therefore runs in $\boldsymbol{O(n)}$ ***time*** altogether. ■

[a] A naïve recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio. The worst-case tree is a path—every non-leaf node has exactly one child.

Solution (one function): For each node v in the input tree T, let $MaxFun(v)$ denote the maximum total "fun" of a legal party among the descendants of v, where v may or may not be invited.

The president of the company must be invited, so none of the president's "children" in T can be invited. Thus, the value we need to compute is

$$root.fun + \sum_{\text{grandchildren } w \text{ of } root} MaxFun(w).$$

The function $MaxFun$ obeys the following recurrence:

$$MaxFun(v) = \max \left\{ \begin{array}{c} v.fun + \displaystyle\sum_{\text{grandchildren } x \text{ of } v} MaxFun(x) \\ \displaystyle\sum_{\text{children } w \text{ of } v} MaxFun(w) \end{array} \right\}$$

(This recurrence does not require a separate base case, because $\sum \varnothing = 0$.) We can memoize this function by adding an additional field $v.maxFun$ to each node v in the tree. The value at each node depends only on the values at its children and grandchildren, so we can compute all values using a postorder traversal of T.

```
BestParty(T):
  ComputeMaxFun(T.root)
  party ← T.root.fun
  for all children w of T.root
      for all children x of w
          party ← party + x.maxFun
  return party
```

```
ComputeMaxFun(v):
  yes ← v.fun
  no ← 0
  for all children w of v
      ComputeMaxFun(w)
      no ← no + w.maxFun
      for all children x of w
          yes ← yes + x.maxFun
  v.maxFun ← max{yes, no}
```

(Yes, this is still dynamic programming; we're only traversing the tree recursively because that's the most natural way to traverse trees![a])

The algorithm spends $O(1)$ time at each node (because each node has exactly one parent and one grandparent) and therefore runs in ***$O(n)$ time*** altogether. ■

[a]Like the previous solution, a direct recursive implementation would run in $O(\phi^n)$ time in the worst case, where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio.

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions.