CS/ECE 374 A ✧ Spring 2018

Homework 6

Due Tuesday, March 13, 2018 at 8pm

---

1. Suppose you are given an array $A[1..n]$ of positive integers, each of which is colored either red or blue. An *increasing back-and-forth subsequence* is an sequence of indices $I[1..\ell]$ with the following properties:

   - $1 \le I[j] \le n$ for all j.
   - $A[I[j]] < A[I[j+1]]$ for all $j < \ell$.
   - If $A[I[j]]$ is red, then $I[j+1] > I[j]$.
   - If $A[I[j]]$ is blue, then $I[j+1] < I[j]$.

   Less formally, suppose we start with a token on some integer $A[j]$, and then repeatedly move the token Left (if it's on a bLue square) or Right (if it's on a Red square), always moving from a smaller number to a larger number. Then the sequence of token positions is an increasing back-and-forth subsequence.

   Describe and analyze an efficient algorithm to compute the length of the longest increasing back-and-forth subsequence of a given array of n red and blue integers. For example, given the input array

1	1	0	2	5	9	6	6	4	5	8	9	7	7	3	2	3	8	4	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

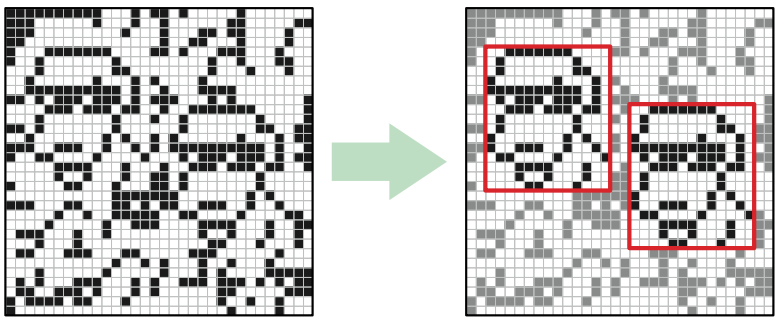
   your algorithm should return the integer 9, which is the length of the following increasing back-and-forth subsequence:

0	1	2	3	4	6	7	8	9
20	1	16	17	9	8	13	11	12

   (The small numbers are indices into the input array.)

2. Describe and analyze an algorithm that finds the largest rectangular pattern that appears more than once in a given bitmap. Your input is a two-dimensional array $M[1..n, 1..n]$ of bits; your output is the area of the repeated pattern. (The two copies of the pattern might overlap, but must not actually coincide.)

   For example, given the bitmap shown on the left in the figure below, your algorithm should return $15 \times 13 = 195$, because the same 15×13 doggo appears twice, as shown on the right, and this is the largest such pattern.



3. ***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 1.

   Describe and analyze an efficient algorithm to construct an optimal AVL tree for a given set of keys and frequencies. 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 $A[i]$. Your task is to construct an AVL tree for the given keys such that the total cost of all searches is as small as possible. This is exactly the same cost function that we considered in Thursday's class; the only difference is that the output tree must satisfy the AVL balance constraint.

   *[Hint: You do **not** need to know or use the insertion and deletion algorithms that keep the AVL tree balanced.]*

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

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

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:

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

(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.