CS/ECE 374 A ✧ Fall 2023

Homework 7

Due Tuesday, October 17, 2023 at 9pm Central Time

---

1. The City Council of Sham-Poobanana needs to partition Purple Street into voting districts. A total of n people live on Purple Street, at consecutive addresses $1, 2, \ldots, n$. Each voting district must be a contiguous interval of addresses $i, i+1, \ldots, j$ for some $1 \le i < j \le n$. By law, each Purple Street address must lie in exactly one district, and the number of addresses in each district must be between k and $2k$, where k is a positive integer parameter.

   Every election in Sham-Poobanana is between two rival factions: Oceania and Eurasia. A majority of the current City Council are from Oceania, so they consider a district to be *good* if more than half the residents of that district voted for Oceania in the previous election. Naturally, the City Council has complete voting records for all n residents.

   For example, the figure below shows a legal partition of 22 addresses (of which 9 are good and 13 are bad) into 4 good districts and 3 bad districts, where $k = 2$ (so each district contains either 2, 3, or 4 addresses). Each **O** indicates a vote for Oceania, and each **X** indicates a vote for Eurasia.

   Describe an algorithm to find the largest possible number of *good* districts in a legal partition. Your input consists of the integer k and a boolean array GoodVote$[1..n]$ indicating which residents previously voted for Oceania (True) or Eurasia (False). You can assume that a legal partition exists. Analyze the running time of your algorithm in terms of the parameters n and k. (In particular, do **not** assume that k is a constant.)

2. The StupidScript language includes a binary operator `@` that computes the *average* of its two arguments. For example, the StupidScript code `print(3 @ 6)` would print `4.5`, because $(3+6)/2 = 4.5$.

   Expressions like 3 @ 7 @ 4 that use the @ operator more than once yield different results when they are evaluated in different orders:

$$(3 \mathbin{@} 7) \mathbin{@} 4 = 5 \mathbin{@} 4 = 4.5 \qquad \text{but} \qquad 3 \mathbin{@} (7 \mathbin{@} 4) = 3 \mathbin{@} 5.5 = 4.25$$

   Here is a larger example:

$$\begin{aligned}((((8 \mathbin{@} 6) \mathbin{@} 7) \mathbin{@} 5) \mathbin{@} 3) \mathbin{@} (0 \mathbin{@} 9) &= 4.5\\ ((8 \mathbin{@} 6) \mathbin{@} (7 \mathbin{@} 5)) \mathbin{@} ((3 \mathbin{@} 0) \mathbin{@} 9) &= 5.875\\ (8 \mathbin{@} (6 \mathbin{@} (7 \mathbin{@} (5 \mathbin{@} (3 \mathbin{@} 0))))) \mathbin{@} 9 &= 7.890625\end{aligned}$$

   Your goal for this problem is to describe and analyze an algorithm to compute, given a sequence of integers separated by @ signs, the ***largest possible*** value the expression can take by adding parentheses. Your input is an array $A[1..n]$ listing the sequence of integers.

   For example, if your input sequence is $[3, 7, 4]$, your algorithm should return 4.5, and if your input sequence is $[8, 6, 7, 5, 3, 0, 9]$, your algorithm should return 7.890625. Assume all arithmetic operations (including @) can be performed exactly in $O(1)$ time.

   (a) Tommy Tutone suggests the following natural greedy algorithm: Merge the adjacent pair of numbers with the smallest average (breaking ties arbitrarily), replace them with their average, and recurse. For example:

$$\begin{aligned}&8 \mathbin{@} 6 \mathbin{@} 7 \mathbin{@} 5 \mathbin{@} \underline{3 \mathbin{@} 0} \mathbin{@} 9\\ &8 \mathbin{@} 6 \mathbin{@} 7 \mathbin{@} \underline{5 \mathbin{@} 1.5} \mathbin{@} 9\\ &8 \mathbin{@} 6 \mathbin{@} \underline{7 \mathbin{@} 3.25} \mathbin{@} 9\\ &8 \mathbin{@} \underline{6 \mathbin{@} 5.125} \mathbin{@} 9\\ &\underline{8 \mathbin{@} 5.5625} \mathbin{@} 9\\ &\underline{6.78125 \mathbin{@} 9}\\ &7.890625\end{aligned}$$

   Tommy reasons that with an efficient priority queue, this algorithm will run in $O(n \log n)$ time, which is *way* faster than any dynamic programming algorithm.

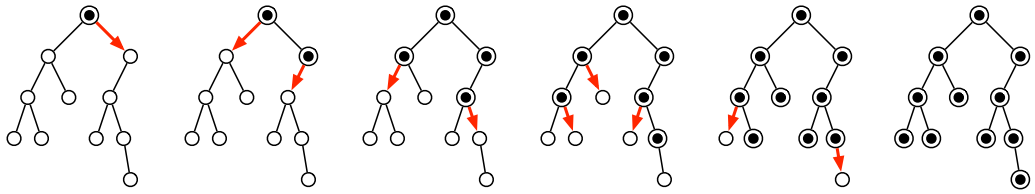
   Prove that Tommy's algorithm is incorrect, by describing a specific input array and proving that his algorithm does not yield the largest possible value for that array.

   (b) Describe and analyze a correct algorithm for this problem. Poor, poor Tommy.

3. ***Practice only. Do not submit solutions.***

Suppose we need to broadcast a message to all the nodes in a rooted binary tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. See the figure below for an example.

Design an algorithm to compute the minimum number of rounds required to broadcast the message to every node.



A message being distributed through a binary tree in five rounds.

Solved problems

3. 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 $w = \texttt{([()][]())[()()]()}$ is balanced, because $w = xy$, where

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

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:

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

Here $\boldsymbol{A[i] \sim A[k]}$ indicates that $A[i]$ is a left delimiter and $A[k]$ is the corresponding right delimiter: Either $A[i] =$ `(` and $A[k] =$ `)`, or $A[i] =$ `[` and $A[k] =$ `]`.

We can memoize this function into a two-dimensional array $LBS[1..n, 1..n]$. Because each entry $LBS[i,k]$ depends only on entries in later rows or earlier columns (or both), we can fill this array row-by-row from bottom up (decreasing i) in the outer loop, scanning each row from left to right (increasing k) in the inner loop.

We can compute each entry $LBS[i,k]$ in $O(n)$ time, so the resulting algorithm runs in $\boldsymbol{O(n^3)}$ ***time***. ■

Solution (pseudocode): The following 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] = ( and A[k] = )) or (A[i] = [ and 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]
```

Here $LBS[i, k[$ stores the length of the longest balanced subsequence of the substring $A[i..k]$. ■

Rubric: 10 points, standard dynamic programming rubric. Yes, each of these solutions is independently worth full credit.

4. 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 $\sum \varnothing = 0$.[a]

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.

The resulting algorithm spends $O(1)$ time at each node of T, and therefore runs in ***O(n) time***. ■

[a] A naïve recursive implementation of these recurrences 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 occurs when T is a single path.

Solution (two functions, pseudocode): The following algorithm runs in ***O(n) time***.

```
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}
```

We are storing two pieces of information in each node v of the tree:

- $v.yes$ is the maximum total "fun" of a legal party among the descendants of v, assuming v is invited.
- $v.no$ is the maximum total "fun" of a legal party among the descendants of v, assuming v is not invited.

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

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.

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

Solution (one function, pseudocode):

```
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}
```

Here $v.maxFun$ stores the maximum total "fun" of a legal party among the descendants of v, where v may or may not be invited.

Each value $v.maxFun$ is read at most three times during the algorithm's execution: Once in ComputeMaxFun($v.parent$), and once in ComputeMaxFun($v.parent.parent$), and at most once in the non-recursive part of BestParty. Thus, the entire algorithm runs in $O(n)$ *time*. ■

Rubric: 10 points: standard dynamic programming rubric. These are not the only correct solutions. Yes, each of these solutions is independently worth full credit.