

🌀 Homework 7 🌀

Due Tuesday, October 20, 2026 at 9pm Central Time

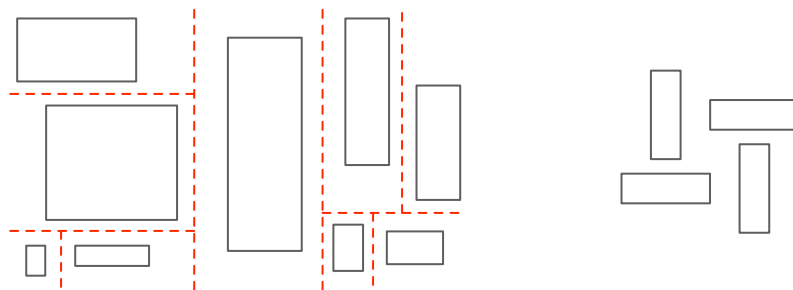
Please make sure that you read and understand the [standard dynamic programming rubric](#).

Dynamic programming solutions that do not include an English description of the recursive function being evaluated (or the *underlying* recursive function if the solution uses iterative pseudocode) will not receive any credit.

- Given two sequences A and B , another sequence X is **common subsequence** of A and B if X is a subsequence of both A and B .
 - Describe an algorithm that given two arrays $A[1..m]$ and $B[1..n]$ of *integers* computes the length of the longest common subsequence of A and B that is *decreasing*.
 - Describe an algorithm that given two arrays $A[1..m]$ and $B[1..n]$ of *characters* computes the length of the longest common subsequence of A and B that is a *palindrome*.
- A rectangle in 2D is called **axis-parallel** if two sides are horizontal and two sides are vertical. Any such rectangle can be specified by two points (a, b) and (c, d) where (a, b) are the coordinates of the rectangle's lower-left corner and (c, d) are the coordinates of the rectangle's upper-right corner.

A set S of axis-parallel rectangles in 2D are called **orthogonally separable** if and only if the set has at least one of the following recursively defined properties:

- $|S| \leq 1$.
- There exists a vertical line ℓ and a partition of S into two non-empty orthogonally separable subsets S_1 and S_2 such that all rectangles in S_1 lie left of ℓ and all rectangles in S_2 lie right of ℓ .
- There exists a horizontal line ℓ and a partition of S into two non-empty orthogonally separable subsets S_1 and S_2 such that all rectangles in S_1 lie below ℓ and all rectangles in S_2 lie above ℓ .



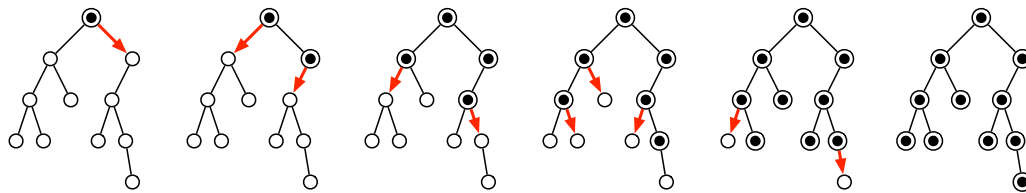
Left: An orthogonally separable set of rectangles. Right: A set of rectangles that is *not* orthogonally separable.

Describe an algorithm that given a set S of n (possibly overlapping) axis-aligned rectangles in 2D computes the maximum size of any orthogonally separable subset of S . You may assume the input is given as four arrays $A[1..n]$, $B[1..n]$, $C[1..n]$, and $D[1..n]$ such that the i th rectangle in S has a lower-left corner at $(A[i], B[i])$ and an upper-right corner at $(C[i], D[i])$. You may also assume all numbers given in the input are distinct.

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

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:

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

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 $LBS(i, k)$ 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:

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

Here $A[i] \sim A[k]$ 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 $O(n^3)$ time. ■

Solution (pseudocode): The following algorithm runs in $O(n^3)$ time:

```

LONGESTBALANCEDSUBSEQUENCE( $A[1..n]$ ):
  for  $i \leftarrow n$  down to 1
     $LBS[i, i] \leftarrow 0$ 
    for  $k \leftarrow i + 1$  to  $n$ 
      if ( $A[i] = ($  and  $A[k] = )$ ) or ( $A[i] = [$  and  $A[k] = ]$ )
         $LBS[i, k] \leftarrow LBS[i + 1, k - 1] + 2$ 
      else
         $LBS[i, k] \leftarrow 0$ 
      for  $j \leftarrow i$  to  $k - 1$ 
         $LBS[i, k] \leftarrow \max \{ LBS[i, k], LBS[i, j] + LBS[j + 1, k] \}$ 
  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.

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 $\sum \emptyset = 0$.^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 values 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. ■

^aA 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( $v$ ):
     $v.yes \leftarrow v.fun$ 
     $v.no \leftarrow 0$ 
    for all children  $w$  of  $v$ 
        COMPUTEMAXFUN( $w$ )
     $v.yes \leftarrow v.yes + w.no$ 
     $v.no \leftarrow v.no + \max\{w.yes, w.no\}$ 
```

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

- $v.\text{yes}$ is the maximum total “fun” of a legal party among the descendants of v , assuming v is invited.
- $v.\text{no}$ 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 $\text{MaxFun}(v)$ 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

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

The function MaxFun obeys the following recurrence:

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

(This recurrence does not require a separate base case, because $\sum \emptyset = 0$.) We can memoize this function by adding an additional field $v.\text{maxFun}$ 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 **$O(n)$ time** altogether. ■

Solution (one function, pseudocode):

```

BESTPARTY( $T$ ):
    COMPUTEMAXFUN( $T.\text{root}$ )
    party  $\leftarrow T.\text{root.fun}$ 
    for all children  $w$  of  $T.\text{root}$ 
        for all children  $x$  of  $w$ 
            party  $\leftarrow$  party +  $x.\text{maxFun}$ 
    return party

```

```

COMPUTEMAXFUN( $v$ ):
    yes  $\leftarrow v.\text{fun}$ 
    no  $\leftarrow 0$ 
    for all children  $w$  of  $v$ 
        COMPUTEMAXFUN( $w$ )
        no  $\leftarrow$  no +  $w.\text{maxFun}$ 
        for all children  $x$  of  $w$ 
            yes  $\leftarrow$  yes +  $x.\text{maxFun}$ 
     $v.\text{maxFun} \leftarrow \max\{\text{yes}, \text{no}\}$ 

```

Here $v.\text{maxFun}$ 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.