

# 11-711: Algorithms for NLP

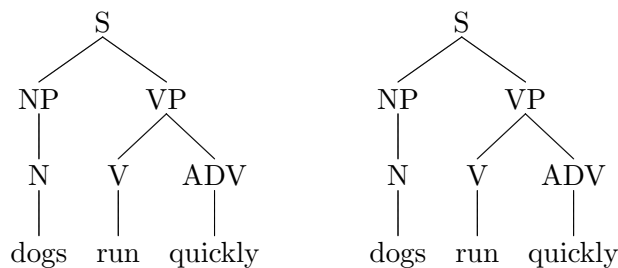
## Extra Exercises: Analysis of Algorithms and Search

Out: October 8, 2009  
Not Assigned

### Problem 1

A parse tree, as we have seen in class, is a tree with intermediate nodes labeled by non-terminals and leaf nodes labeled by terminals.

1. Often it is desirable to compare two parse trees to determine if they are exactly the same. (Thus we could determine if a grammar is ambiguous by finding out if there exists more than one distinct parse tree for a given sentence.) Design an algorithm `CompareTrees` using the divide and conquer approach. To be more specific, `CompareTrees` should return `true` if and only if the two trees compared are exactly the same, *including the ordering of the sibling nodes*. For example, the following two trees are the same:



Note: In designing your algorithm, do *not* give a specific implementation in a particular programming language — all you need to do is to come up with the pseudocode. (Look at the lecture slides on Analysis of Algorithms for examples.) However, you *do* need to be explicit about the input and the output of your algorithm.

2. What is the time complexity of your algorithm? Show all of the derivation. Hint: One possible way is to write down the recurrence formula for the time complexity first, guess a growth function as the time complexity, and prove it is correct by induction, but you are free to derive it using other means, as long as you can convince me.

## Solution

1. [10 points] The algorithm `CompareTrees(t1, t2)` recursively calls `RecursiveCompareTrees(n1, n2)` on different nodes of the input trees. For a node `n`, we assume that `n.label` gives its node label (V, NP, etc.), that `n.degree` gives the number of its children, and that `n.child[i]` returns the node's  $i$ th child (indexing from left to right).

`RecursiveCompareTrees(n1, n2)`

Input: `n1` and `n2` are two nodes

Output: TRUE iff the subtrees rooted at `n1` and `n2` are identical

```
if n1.label != n2.label return FALSE
if n1.degree != n2.degree return FALSE
for i = 1 to n1.degree
    if RecursiveCompareTrees(n1.child[i], n2.child[i]) is false
        return FALSE
return TRUE
```

`CompareTrees(t1, t2)`

Input: `t1` and `t2` are two (ordered) trees

Output: TRUE iff `t1` and `t2` are identical

```
if t1 and t2 are not empty
    return RecursiveCompareTrees(t1.root, t2.root)
else return TRUE
```

2. [10 points] Since `CompareTrees()` only takes constant time, we only have to derive the time complexity of `RecursiveCompareTrees()`. For two trees  $t_1$  and  $t_2$ , the algorithm stops after comparing  $\min(|t_1|, |t_2|)$  nodes if we select  $t_1$  to be the smaller of the two trees. For a general “order-of” analysis, we can compute the running time  $T(n)$ , where  $n$  is the number of nodes in either tree.

Here is a somewhat informal argument based on induction. Again,  $n$  is the number of nodes in the (smaller) of the two trees being compared.

In the base case, let  $n = 1$ . The `for` loop in `RecursiveCompareTrees()` is executed exactly once, and before we call the function again from the body of the loop we have executed three statements. Then `RecursiveCompareTrees()` is called on the unique child of `n1`; since  $n = 1$ , we know that his child node has no children. In the inner call to `RecursiveCompareTrees()`, therefore, we will execute only a total of four statements before returning either TRUE or FALSE.

Thus, for  $n = 1$ , we execute a total of exactly eight statements overall. We guess  $T(n) = O(n) = cn + c'$ , so we can conclude (as one possibility) that  $c = 8$  and  $c' = 0$ . Thus  $T(n) = O(n)$  for  $n = 1$ .

For the induction step, we can write the recurrence formula for `RecursiveCompareTrees()`

as

$$T(n) = O(1) + \sum_{i=1}^d T(n_i)$$

where  $d$  is the degree of the root node and  $n_i$  is the size of the subtree rooted at the  $i$ th child of the current root node. We know that  $n_i \leq n \forall i$  and that  $\sum_{i=1}^d n_i = n - 1 < n$  since we're dealing with trees. Therefore,  $n_i < n \forall i$ .

By the induction hypothesis, we assume  $T(n_i) = O(n_i)$  as the complexity of the algorithm. In the recurrence formula, let  $O(1) = c_0$  and let  $O(n_i) = c_i n_i + c'_i$ . Thus,

$$\begin{aligned} T(n) &= c_0 + \sum_{i=1}^d (c_i n_i + c'_i) \\ &= c_0 + c_1 n_1 + c'_1 + c_2 n_2 + c'_2 + \dots + c_d n_d + c'_d \\ &< c_0 + c'_1 + c'_2 + \dots + c'_d + n(c_1 + c_2 + \dots + c_d) \\ &\equiv c'_0 + c' n \\ &\equiv O(n) \end{aligned}$$

## Problem 2

Let  $p(n) = \sum_{i=0}^d a_i n^i$ , where  $a_d > 0$ , be a degree- $d$  polynomial in  $n$ , and let  $k$  be a constant. Prove the following properties:

1. If  $k \geq d$ , then  $p(n) = O(n^k)$ .
2. If  $k \leq d$ , then  $p(n) = \Omega(n^k)$ .
3. If  $k = d$ , then  $p(n) = \Theta(n^k)$ .

## Solution

1. [**7 points**] By definition, we need to show that  $0 \leq p(n) \leq cn^k$  for all  $n \geq n_0$ , so we find  $c$  and  $n_0$  that satisfy this. Let  $a = \max(\{|a_i|\})$ , so that we know  $a \geq a_i$  and  $-a \leq a_i$  for  $i = \{0 \dots d\}$ . Now

$$p(n) = \sum_{i=0}^d a_i n^i \leq \sum_{i=0}^d a n^i \leq \sum_{i=0}^d a n^k = a(d+1) \cdot n^k$$

So, if we pick  $c = a(d+1)$ , the inequality will hold for  $n \geq 1$ .

Now we need to pick  $n_0$  such that  $p(n) \geq 0$  for all  $n \geq n_0$ .

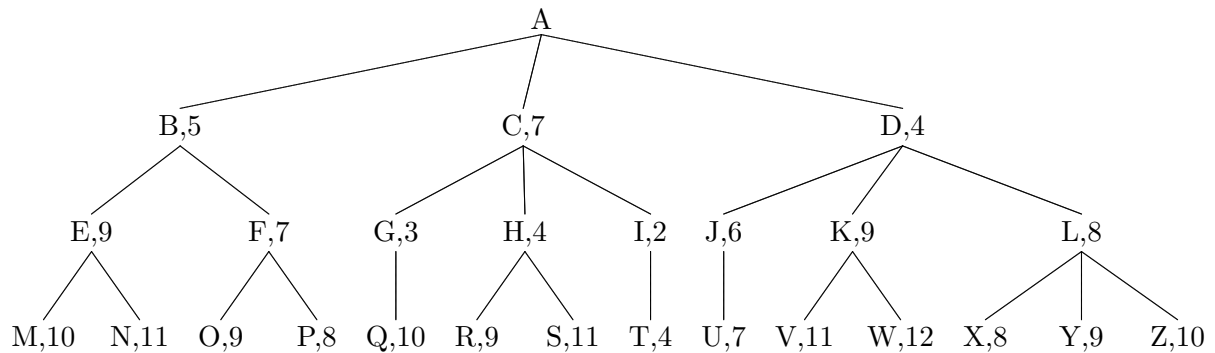
$$p(n) = \sum_{i=0}^d a_i n^i \geq a_d n^d + \sum_{i=0}^{d-1} -a n^i \geq a_d n^d + \sum_{i=0}^{d-1} -a n^{d-1} = a_d n^d - a d n^{d-1} = n^{d-1} (a_d n - a d)$$

The function  $n^{d-1}(a_d n - a d)$  is non-negative when  $n \geq \frac{ad}{a_d}$ , so we pick  $n_0 = \frac{ad}{a_d}$ . To make sure both inequalities are satisfied ( $p(n) \geq 0$  and  $p(n) \leq a(d+1)n^k$ ), we must check that  $n_0 \geq 1$ . It is, because  $a \geq a_d$ . Therefore we have satisfied the definition that  $p(n) = O(n^k)$  for  $k \geq d$ .

2. [7 points] By definition, we need to show that  $0 \leq cn^k \leq p(n)$  for all  $n \geq n_0$ . Let's consider a new polynomial  $q(n) = p(n) - cn^k$ , with  $0 < c < a_d$ , that is (like  $p(n)$ ) a polynomial of degree  $d$  with a positive coefficient for the leading term since  $k \leq d$ . Since  $q(n)$  has the same form as  $p(n)$ , we know from Part 1 that we can find a positive  $n_0$  such that  $q(n) \geq 0$  for  $n \geq n_0$ . If  $q(n)$  is non-negative after this point, then it must also be the case that  $cn^k \leq p(n)$  for  $n \geq n_0$  as well. Because of the bounds we defined on  $c$ ,  $cn^k > 0$  as well, so the proof that  $p(n) = \Omega(n^k)$  for  $k \leq d$  is complete.
3. [6 points] When  $k = d$ , we know from Parts 1 and 2 that  $p(n) = O(n^k)$  and  $p(n) = \Omega(n^k)$ . Therefore, by definition,  $p(n) = \Theta(n^k)$ .

### Problem 3

Here is a search tree. Each node is labeled with a letter indicating its name and with a number indicating the estimated value of the node ( $f'$ ). A larger  $f'$  indicates a better node.



For each of the following search techniques, list the nodes that are searched in the order that they are searched.

1. Breadth-first
2. Depth-first
3. Iterative deepening
4. Hill climbing
5. Best-first

### Solution

For the first three search techniques, assume that the children of a node are explored in alphabetical order. For the last two techniques, assume that ties are broken by alphabetical order.

1. [4 points] A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
2. [4 points] A B E M N F O P C G Q H R S I T D J U K V W L X Y Z

3. [4 points]

Iteration 0: A

Iteration 1: A B C D

Iteration 2: A B E F C G H I D J K L

Iteration 3: A B E M N F O P C G Q H R S I T D J U K V W L X Y Z

4. [4 points] A C

Pure steepest-ascent hill climbing would not explore any of C's children because they all look worse than C, and also would not backtrack to explore any of A's other children.

5. [4 points] A C B E N M F O P D K W V L Z Y X J U H S R G Q I T