1 Introduction

The substitution method is a condensed way of proving an asymptotic bound on a recurrence by induction. In the substitution method, instead of trying to find an exact closed-form solution, we only try to find a closed-form bound on the recurrence. This is often much easier than finding a full closed-form solution, as there is much greater leeway in dealing with constants.

The substitution method is a powerful approach that is able to prove upper bounds for almost all recurrences. However, its power is not always needed; for certain types of recurrences, the master method (see below) can be used to derive a tight bound with less work. In those cases, it is better to simply use the master method, and to save the substitution method for recurrences that actually need its full power.

Note 1: The substitution method still requires the use of induction. The induction will always be of the same basic form, but it is still important to state the property you are trying to prove, split into one or more base cases and the inductive case, and note when the inductive hypothesis is being used.

Note 2: We would usually use a recursion tree to generate possible guesses for the runtime, and then use the substitution method to prove them. However, if you are very careful when drawing out a recursion tree and summing the costs, you can actually use a recursion tree as a direct proof of a solution to a recurrence.

2 Solving recurrences

A recurrence relation, such as $T(n) = 2T(\lfloor n/2 \rfloor) + n$ typically reflects the runtime of recursive algorithms. For example, the recurrence above would correspond to an algorithm that made two recursive calls on subproblems of size $\lfloor n/2 \rfloor$, and then did n units of additional work.

Today we will be learning about how to solve these recurrences to get bounds on the runtime (like $T(n) = O(n \log n)$).

2.1 Substitution method

A lot of things in this class reduce to induction. In the substitution method for solving recurrences:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

2.1.1 Example

Recurrence: $T(1) = 1$ *and* $T(n) = 2T(\lfloor n/2 \rfloor) + n$ *for* $n > 1$.

We guess that the solution is $T(n) = O(nlogn)$. So we must prove that $T(n) \leq cnlogn$ for some constant c. (We will get to n_0 later, but for now let's try to prove the statement for all $n \geq 1$.)

As our inductive hypothesis, we assume $T(n) \leq cn \log n$ for all positive numbers less than n. Therefore, $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor log(\lfloor n/2 \rfloor)$, and

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor log(\lfloor n/2 \rfloor)) + n \\ &\leq cn\, log(n/2) + n \\ &= cnlogn - cnlog2 + n \\ &= cnlogn - cn + n \\ &\leq cnlogn\ (for\ c \geq 1) \end{aligned}$$

Now we need to show the base case. This is tricky, because if $T(n) \leq cn \log n$, then $T(1) \leq 0$, which is not a thing. So we revise our induction so that we only prove the statement for $n \geq 2$, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are $n = 2$ and $n = 3$. (We are allowed to do this because asymptotic notation only requires us to prove our statement for $n \geq n_0$, and we can set $n_0 = 2$.)

We choose $n = 2$ and $n = 3$ for our base cases because when we expand the recurrence formula, we will always go through either $n = 2$ or $n = 3$ before we hit the case where $n = 1$.

So proving the inductive step as above, plus proving the bound works for $n = 2$ and n = 3, suffices for our proof that the bound works for all n > 1.

Plugging the numbers into the recurrence formula, we get $T(2) = 2T(1) + 2 = 4$ and $T(3) = 2T(1) + 3 = 5$. So now we just need to choose a c that satisfies those constraints on $T(2)$ and $T(3)$. We can choose $c = 2$, because $4 \leq 2 \cdot 2log2$ and $5 \leq 2 \cdot 3log3$.

Therefore, we have shown that $T(n) \leq 2nlogn$ for all $n \geq 2$, so $T(n) = O(nlogn)$.

2.1.2 Warnings

Warning: Using the substitution method, it is easy to prove a weaker bound than the one you're supposed to prove. For instance, if the runtime is $O(n)$, you might still be able to substitute cn^2 into the recurrence and prove that the bound is $O(n^2)$. Which is technically true, but don't let it mislead you into thinking it's the best bound on the runtime. People often get burned by this on exams!

Warning: You must prove the exact form of the induction hypothesis. For example, in the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$, we could falsely "prove" $T(n) = O(n)$ by guessing $T(n) \leq cn$ and then arguing $T(n) \leq 2(c\lfloor n/2 \rfloor) + n \leq cn + n = O(n)$. Here we needed to prove $T(n) \leq cn$, not $T(n) \leq (c + 1)n$. Accumulated over many recursive calls, those "plus ones" add up.

2.2 Recursion tree

A recursion tree is a tree where each node represents the cost of a certain recursive sub- problem. Then you can sum up the numbers in each node to get the cost of the entire algorithm.

Note: We would usually use a recursion tree to generate possible guesses for the runtime, and then use the substitution method to prove them. However, if you are very careful when drawing out a recursion tree and summing the costs, you can actually use a recursion tree as a direct proof of a solution to a recurrence.

If we are only using recursion trees to generate guesses and not prove anything, we can tolerate a certain amount of "sloppiness" in our analysis. For example, we can ignore floors and ceilings when solving our recurrences, as they usually do not affect the final guess.

2.2.1 Example

Recurrence: $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

We drop the floors and write a recursion tree for $T(n) = 3T(n/4) + cn^2$.

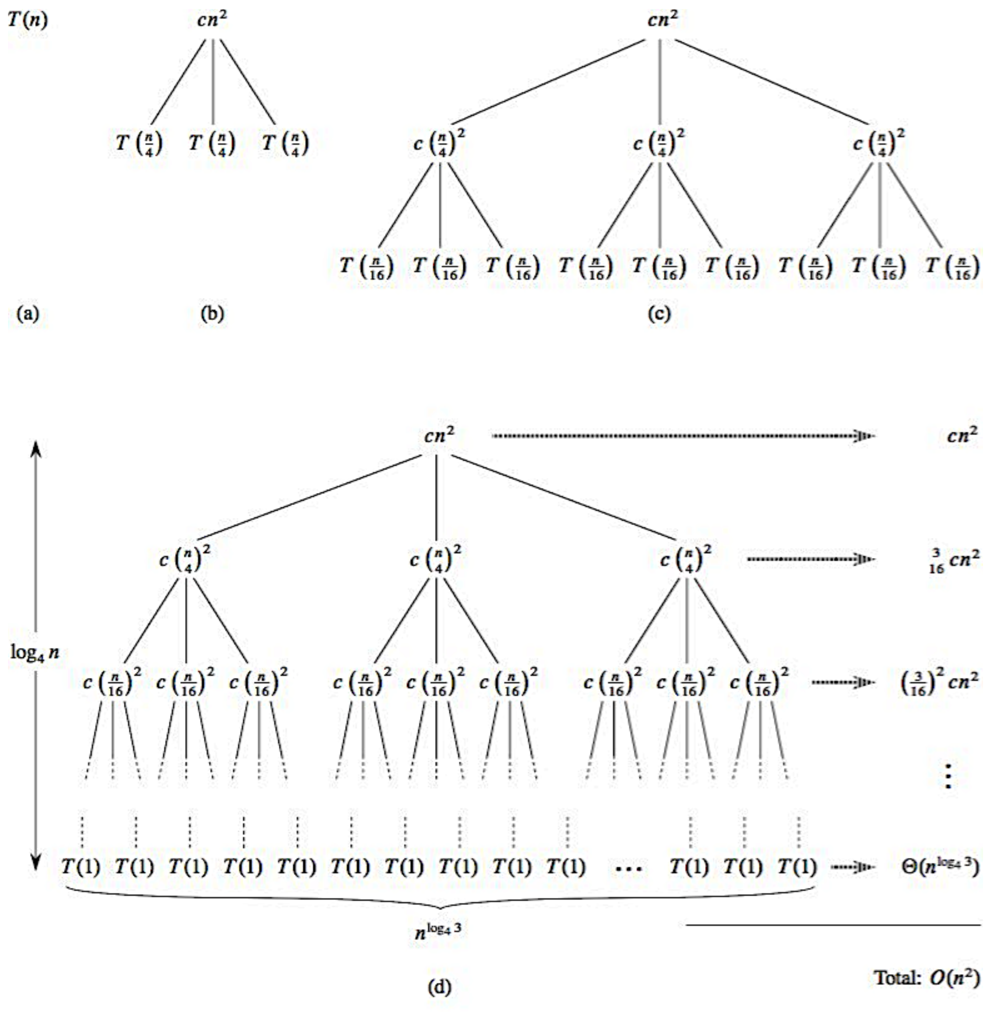

$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$T(1)$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
(d)
Total: $O(n^2)$

Figure 4.5 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part **(a)** shows $T(n)$, which progressively expands in **(b)–(d)** to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

The top node has cost cn^2, because the first call to the function does cn^2 units of work, aside from the work done inside the recursive sub calls. The nodes on the second layer all have cost $c(n/4)^2$, because the functions are now being called on problems of size $n/4$, and the functions are doing $c(n/4)^2$ units of work, aside from the work done inside their recursive sub calls, etc. The bottom layer (base case) is special because each of them contribute $T(1)$ to the cost.

Analysis: First we find the height of the recursion tree. Observe that a node at depth i reflects a subproblem of size $n/4^i$. The subproblem size hits $n = 1$ when $n/4^i = 1$, or $i = log_4 n$. So the tree has $log_4 n + 1$ levels.

Now we determine the cost of each level of the tree. The number of nodes at depth i is 3^i.

Each node at depth $i = 0,1,\ldots, log_4 n - 1$ has a cost of $c(n/4^i)^2$, so the total cost of level i is $3^i c(n/4^i)^2 = (3/16)^i cn^2$. However, the bottom level is special. Each of the bottom nodes contribute cost $T\ (1)$, and there are $3^{log_4^n} = n^{log_4^3}$ of them.

So the total cost of the entire tree is

$$\begin{aligned} T(n) &= cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \cdots + \left(\frac{3}{16}\right)^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\ &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \end{aligned}$$

The left term is just the sum of a geometric series. So $T\ (n)$ evaluates to

$$\frac{(3/16)^{\log_4 n} - 1}{(3/16) - 1} cn^2 + \Theta(n^{\log_4 3})$$

This looks complicated but we can bound it (from above) by the sum of the infinite series

$$\sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) = \frac{1}{1 - (3/16)} cn^2 + \Theta(n^{\log_4 3})$$

Since functions in $\Theta(n^{log_4^3})$ are also in $O(n^2)$, this whole expression is $O(n^2)$. Therefore, we can guess that $T\ (n) = O(n^2)$.

2.2.2 Back to the substitution method

Now we can check our guess using the substitution method. Recall that the original recurrence was $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We want to show that $T(n) \leq dn^2$ for some constant $d > 0$. By the induction hypothesis, we have that $T(\lfloor n/4 \rfloor) \leq d\lfloor n/4 \rfloor^2$. So using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \left(\frac{3}{16}\right) dn^2 + cn^2 \\ &\le dn^2 \left(when\ c \le \left(\frac{13}{16}\right) d, \qquad i.e.\ d \ge (16/13)c\right) \end{aligned}$$

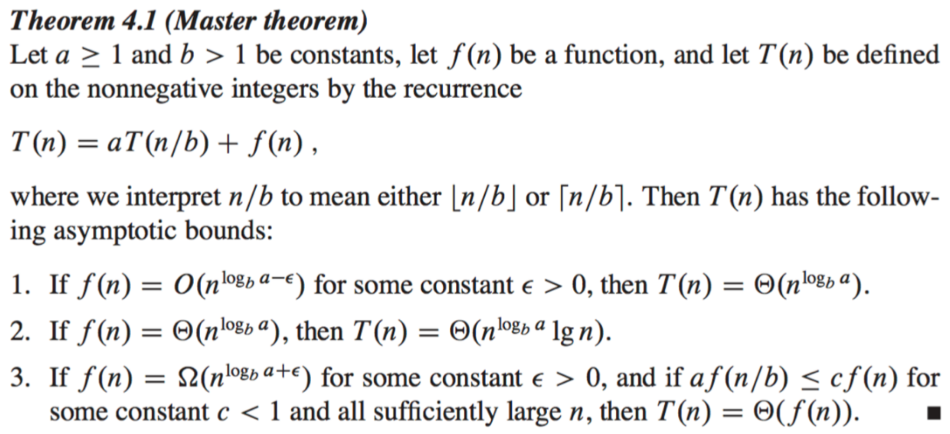
Note that we would also have to identify a suitable base case and prove the recurrence is true for the base case, and we don't have time to talk about this in lecture, but you should do that in your homework.

2.3 Master theorem

The master theorem is a formula for solving recurrences of the form $T(n) = aT(n/b) + f(n)$, where $a \ge 1$ and $b > 1$ and f(n) is asymptotically positive. (Asymptotically positive means that the function is positive for all sufficiently large n.)

This recurrence describes an algorithm that divides a problem of size n into a subproblems, each of size n/b, and solves them recursively. (Note that n/b might not be an integer, but in section 4.6 of the book, they prove that replacing $T(n/b)$ $with$ $T(\lfloor n/b \rfloor)$ or $T(\lceil n/b \rceil)$ does not affect the asymptotic behavior of the recurrence. So we will just ignore floors and ceilings here.)

The theorem is as follows:



Theorem 4.1 (Master theorem)
Let $a \ge 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \le cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

The master theorem compares the function $n^{logb\ a}$ to the function $f(n)$. Intuitively, if $n^{logb\ a}$ is larger (by a polynomial factor), then the solution is $T(n) = \Theta(n^{logb\ a})$. If f(n) is larger (by a polynomial factor), then the solution is $T(n) = \Theta(f(n))$. If they are the same size, then we multiply by a logarithmic factor.

Be warned that these cases are not exhaustive – for example, it is possible for $f(n)$ to be asymptotically larger than $n^{log_b a}$, but not larger by a polynomial factor (no matter how small the exponent in the polynomial is). For example, this is true when $f(n) = n^{log_b a} \log n$. In this situation, the master theorem would not apply, and you would have to use another method to solve the recurrence.

2.3.1 Examples:

To use the master theorem, we simply plug the numbers into the formula.

Example 1: T(n)=9T(n/3)+n.

Here a=9,b=3,f(n)=n,and$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\varepsilon})$ for ε = 1, case 1 of the master theorem applies, and the solution is $T(n) = \Theta(n^2)$.

Example 2: T(n)=T(2n/3)+1.

Here a=1,b=3/2,f(n)=1, and $n^{\log_b a} = n^0 = 1$. Since $f(n) = \Theta(n^{\log_b a})$, case 2 of the master theorem applies, so the solution is T(n) = Θ(logn).

Example 3: T(n) = 3T(n/4) + nlogn.

Here $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. For ε = 0.2, we have $f(n) = \Omega(n^{\log_4 3+\varepsilon})$. So case 3 applies if we can show that af(n/b) ≤ cf(n) for some c < 1 and all sufficiently large n. This would mean $3^n{}_4 \log {}^n{}_4 \leq cn \log n$. Setting c = 3/4 would cause this condition to be satisfied.

Example 4: T (n) = 2T (n/2) + n log n.

Here the master method does not apply. $n^{\log_b a} = n$, and f(n) = nlogn. Case 3 does not apply because even though nlogn is asymptotically larger than n, it is not polynomially larger. That is, the ratio $f(n)/n^{\log_b a} = \log n$ is asymp- totically less than n^{ε} for all positive constants ε.

3 Maximum subarray problem

Now we will present another divide and conquer algorithm.
Suppose you are given the price of a stock on each day, and you have to decide when to buy and when to sell to maximize your profit. Note that you cannot sell before you buy (so you can't just sell on the highest day and buy on the lowest day).

Naive strategy: Try all pairs of (buy, sell) dates, where the buy date must be before the sell date. This takes $\Theta(n^2)$ time.

```
bestProfit = -MAX_INT
bestBuyDate = None
bestSellDate = None
for i = 1 to n:
    for j = i + 1 to n:
        if price[j] - price[i] > bestProfit:
            bestBuyDate = i
            bestSellDate = j
            bestProfit = price[j] - price[i]
return (bestBuyDate, bestSellDate)
```

3.1 Divide and conquer strategy

Instead of the daily price, consider the daily change in price, which (on each day) can be either a positive or negative number. Let array A store these changes. Now we have to find the subarray of A that maximizes the sum of the numbers in that subarray.

Now divide the array into two. Any maximum subarray must either be entirely in the first half, entirely in the second half, or it must span the border between the first and the second half. If the maximum subarray is entirely in the first half (or the second half), we can find it using a recursive call on a subproblem half as large.

If the maximum subarray spans the border, then the sum of that array is the sum of two parts: the part between the buy date and the border, and the part between the border and the sell date. To maximize the sum of the array, we must maximize the sum of each part.

We can do this by simply (1) iterating over all possible buy dates to maximize the first part (2) iterating over all possible sell dates to maximize the second part. Note that this takes linear time instead of quadratic time, because we no longer have to iterate over buy and sell dates simultaneously.

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
left-sum = −∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
right-sum = −∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
return (max-left, max-right, left-sum + right-sum)
```

3.1.1 Runtime analysis

Note that we are omitting the correctness proof, because the main point is to give an example of the divide and conquer strategy. In the homework, you would normally need to provide a correctness proof, unless we say otherwise.

First we analyze the runtime of *FindMaxCrossingSubarray*. Since each iteration of each of the two for loops takes $\Theta(1)$ time, we just need to count up how many iterations there are altogether. The for loop of lines 3-7 makes $mid - low + 1$ iterations, and the for loop of lines 10-14 makes $high - mid$ iterations, so the total number of iterations is $high - low + 1 = n$. Therefore, the helper function takes $\Theta(n)$ time.

Now we proceed to analyze the runtime of the main function.

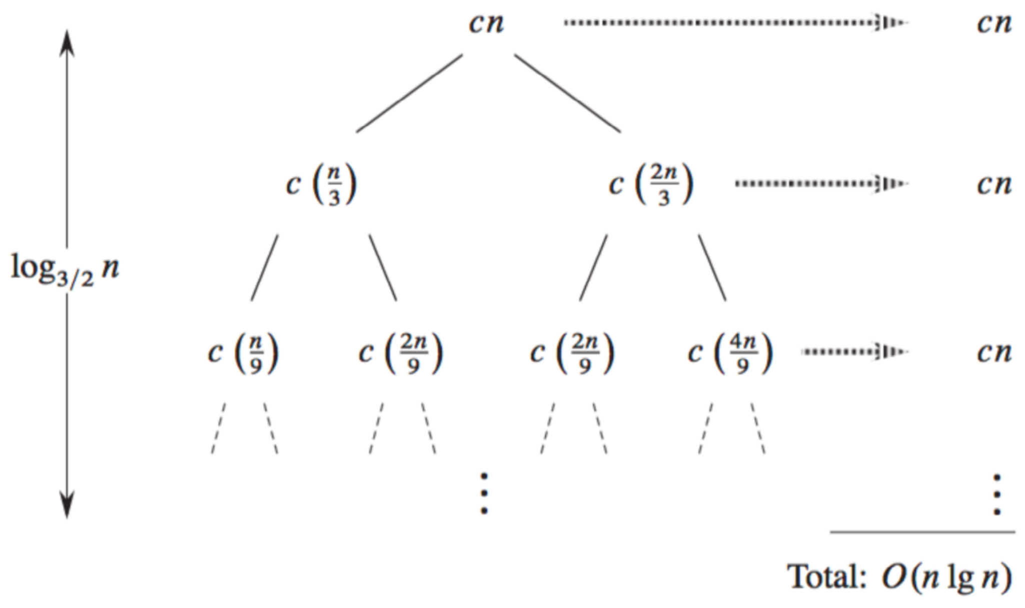
For the base case, $T(1) = \Theta(1)$, since line 2 takes constant time.

For the recursive case, lines 1 and 3 take constant time. Lines 4 and 5 take $T(\lfloor n/2 \rfloor)$ and $T(\lceil n/2 \rceil)$ time, since each of the subproblems has that many elements. The FindMax- CrossingSubarray procedure takes $\Theta(n)$ time, and the rest of the code takes $\Theta(1)$ time. So $T(n) = \Theta(1) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) + \Theta(1) = 2T(n/2) + \Theta(n)$ (ignoring the floors and ceilings).

By case 2 of the master theorem, this recurrence has the solution $T(n) = \Theta(n \log n)$.

Extra recursion tree problem

Consider the recurrence $T\ (n)\ =\ T\ (n/3)\ +\ T\ (2n/3)\ +\ O(n)$. Let c represent the constant factor in the $O(n)$ term.


cn
$c\left(\frac{n}{3}\right)$
$c\left(\frac{2n}{3}\right)$
$c\left(\frac{n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{2n}{9}\right)$
$c\left(\frac{4n}{9}\right)$
cn
cn
cn
$\log_{3/2} n$
Total: $O(n \lg n)$

The longest simple path from the root to a leaf is $n\ \rightarrow\ (2/3)n\ \rightarrow\ (2/3)^2 n\ \rightarrow \cdots \rightarrow\ 1$. Since $(2/3)^k n\ =\ 1$ when $k\ =\ log^{n}_{3/2}$, the height of the tree is $log^{n}_{3/2}$.

We get that each level costs at most cn, but as we go down from the root, more and more internal nodes are absent, so the costs become smaller. Fortunately we only care about an upper bound.

Based on this we can guess $O(n\ log\ n)$ as an upper bound, and verify this by the substitution method.

$$\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n\,,
\end{aligned}$$

when d ≥ c/(log 3 − (2/3)).

Note that we should be proving that the claim holds for the base case as well. You should do this in your homework.