Divide-and-Conquer

Divide-and-Conquer

A well-known algorithm design strategy:

1. Divide instance of problem into two or more smaller instances
2. Solve smaller instances recursively
3. Obtain solution to original (larger) instance by combining these solutions

Divide and Conquer

Divide: Break the problem into smaller sub-problems

Conquer: Solve the sub-problems. Generally, this involves waiting for the problem to be small enough that it is trivial to solve (i.e. 1 or 2 items)

Combine: Given the results of the solved sub-problems, combine them to generate a solution for the complete problem

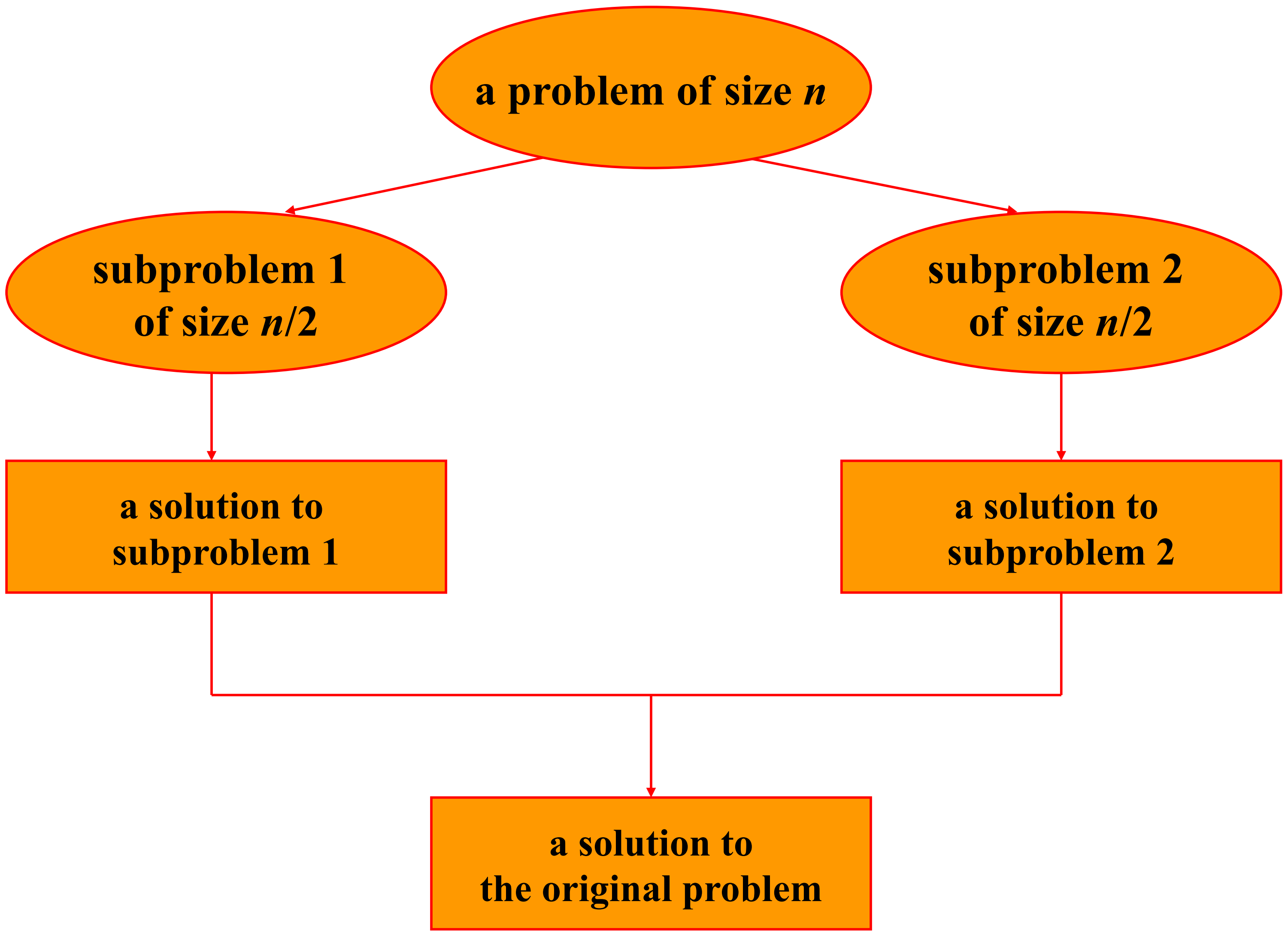
Divide and Conquer: Sorting

How should we split the data?

What are the sub-problems we need to solve?

How do we combine the results from these sub-problems?

Divide-and-Conquer Technique (cont.)



a problem of size *n*
subproblem 1 of size *n*/2
subproblem 2 of size *n*/2
a solution to subproblem 1
a solution to subproblem 2
a solution to the original problem

Divide-and-Conquer Technique (cont.)

As an example, let us consider the problem of computing the sum of n numbers $a_0, \ldots, a_{n-1}$. If $n > 1$, we can divide the problem into two instances of the same problem: to compute the sum of the first $\lfloor n/2 \rfloor$ numbers and to compute the sum of the remaining $\lceil n/2 \rceil$ numbers. (Of course, if $n = 1$, we simply return a_0 as the answer.) Once each of these two sums is computed by applying the same method recursively, we can add their values to get the sum in question:

$$a_0 + \cdots + a_{n-1} = (a_0 + \cdots + a_{\lfloor n/2 \rfloor - 1}) + (a_{\lfloor n/2 \rfloor} + \cdots + a_{n-1}).$$

Divide-and-Conquer Examples

- Classic examples of such algorithms:
  - Sorting: mergesort and quicksort
  - Finding maximum subarray
  - Binary tree traversals
  - Matrix multiplication: Strassen’s algorithm (for square Matrices)
  - Multiplication of large integers
- Though we consider only sequential algorithms here, it is worth keeping in mind that the divide-and-conquer technique is ideally suited for parallel computations.

General Divide-and-Conquer Recurrence

- In the most typical case of divide-and-conquer a problem's instance of size *n* is divided into two instances of size *n/2*.
  - *More generally,* an instance of size *n can be divided into b instances of size n/b, with a of them* needing to be solved. (Here, *a and b are constants;* $a \geq 1$ *and* $b > 1$*.)*

- Assuming that size *n* is a power of *b* to simplify our analysis, we get the following recurrence for the running time *T (n)*:

  $T(n) = aT(n/b) + f(n)$, *(general divide-and-conquer recurrence)*

  where $f(n)$ is a function that accounts for the time spent on dividing an instance of size *n* into instances of size *n*/*b* and combining their solutions.

Mergesort

- Split array A[0..*n*-1] in two about equal halves and make copies of each half in arrays B and C
- Sort arrays B and C recursively
- Merge sorted arrays B and C into array A as follows:
  - Repeat the following until no elements remain in one array:
    - compare the first elements in the remaining unprocessed portions of the arrays
    - copy the smaller of the two into A, while incrementing the index indicating the unprocessed portion of that array
  - Once all elements in one of the arrays are processed, copy the remaining unprocessed elements from the other into A.

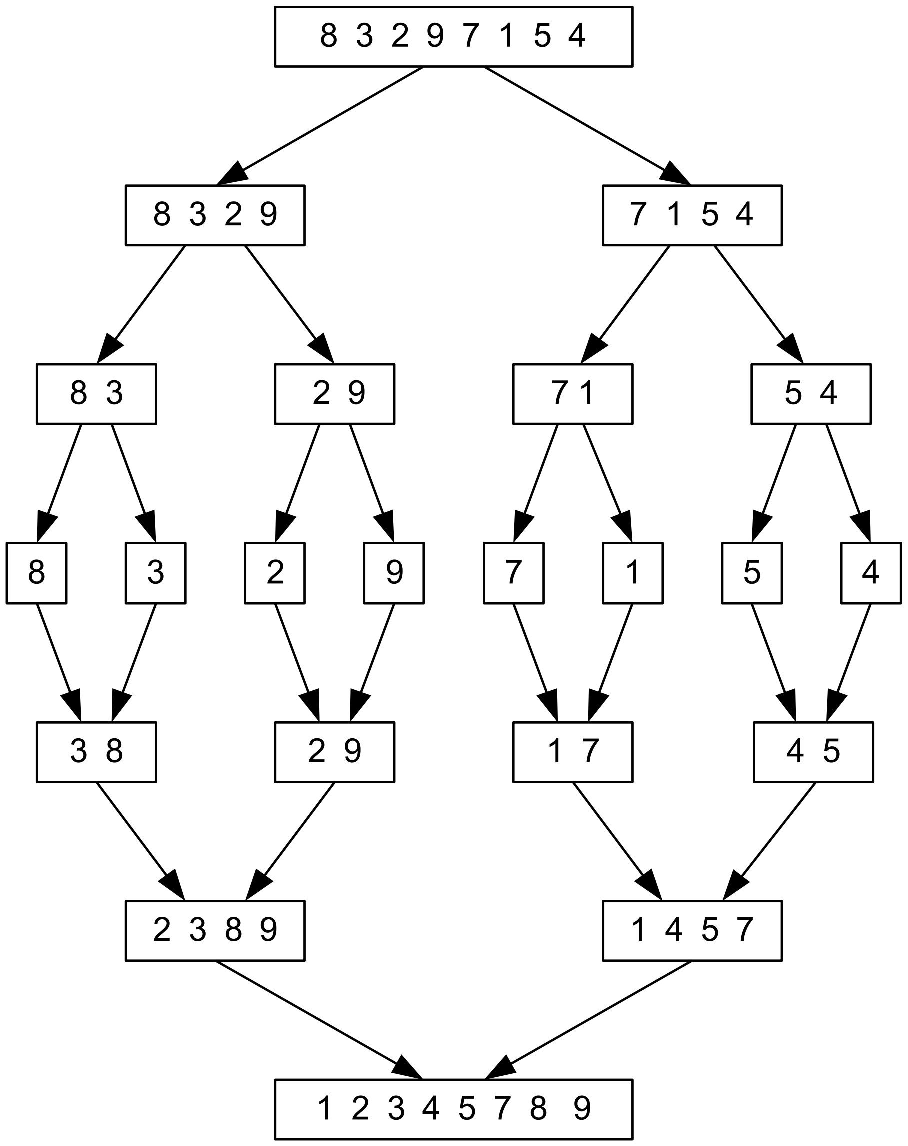
Pseudocode of Mergesort

```
ALGORITHM    $Mergesort(A[0..n-1])$
    //Sorts array $A[0..n-1]$ by recursive mergesort
    //Input: An array $A[0..n-1]$ of orderable elements
    //Output: Array $A[0..n-1]$ sorted in nondecreasing order
    if $n > 1$
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        $Mergesort(B[0..\lfloor n/2 \rfloor - 1])$
        $Mergesort(C[0..\lceil n/2 \rceil - 1])$
        $Merge(B, C, A)$    //see below
```

Pseudocode of Merge

```
ALGORITHM  $Merge(B[0..p-1], C[0..q-1], A[0..p+q-1])$
    //Merges two sorted arrays into one sorted array
    //Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
    //Output: Sorted array $A[0..p+q-1]$ of the elements of $B$ and $C$
    $i \leftarrow 0$;  $j \leftarrow 0$;  $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if $B[i] \leq C[j]$
            $A[k] \leftarrow B[i]$;  $i \leftarrow i + 1$
        else $A[k] \leftarrow C[j]$;  $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    else copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Mergesort Example



Analysis of Mergesort

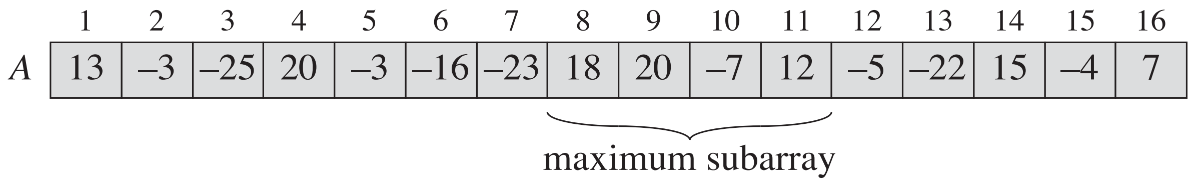
Assuming for simplicity that *n is a power of 2, the* recurrence relation for the number of key comparisons *C(n) is*

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1, \quad C(1) = 0.$$

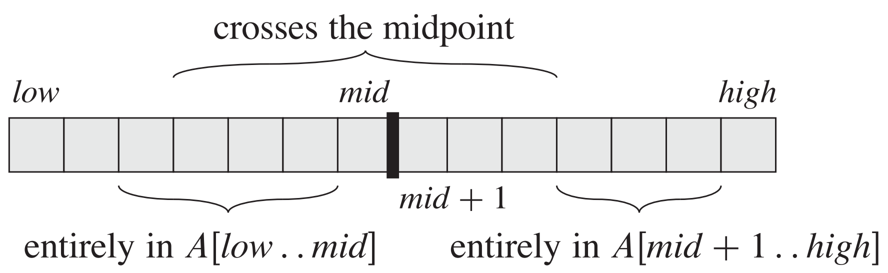
Let us analyze $C_{merge}(n)$, the number of key comparisons performed during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in the two arrays still needing to be processed is reduced by 1. In the worst case, neither of the two arrays becomes empty before the other one contains just one element (e.g., smaller elements may come from the alternating arrays). Therefore, for the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$$C_{worst}(n) = 2C_{worst}(n/2) + n - 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 0.$$

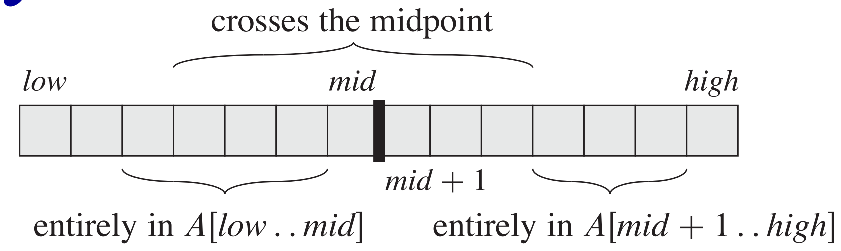
Maximum Subarray Problem



- Find a maximum subarray of array A[*low* … *high*] of positive and negative values
- Find the midpoint, say *mid*, of the array, and consider the subarrays A[*low* … *mid*] and A[*mid+1* … *high*] .
- Any contiguous subarray A[i…j] must be in exactly one of
  - entirely in the subarray $A[low \,..\, mid]$, so that $low \le i \le j \le mid$,
  - entirely in the subarray $A[mid+1 \,..\, high]$, so that $mid < i \le j \le high$, or
  - crossing the midpoint, so that $low \le i \le mid < j \le high$.



Maximum Subarray Problem



- Divide and Conquer
  - Find max subarray A[low … mid]
  - Find max subarray A[mid+1 … high]
  - Find max subarray crossing mid.
  - The maximum among the above three is the max subarray.
- Pseudo Code

```
FIND-MAXIMUM-SUBARRAY($A, low, high$)
 If (low=high) return A;
 Else
        mid= (low+high)/2;
        L = FIND-MAXIMUM-SUBARRAY($A, low, mid$)
        R = FIND-MAXIMUM-SUBARRAY($A, mid + 1, high$)
        X= FIND-MAX-CROSSING-SUBARRAY($A, low, mid, high$)
        return max (L, R, X);
```

Mid Crossing Max Subarray

- Can be found in linear time
  - Find max subarrays A[i … mid] and A[mid+1…j] and combine them

```
FIND-MAX-CROSSING-SUBARRAY$(A, low, mid, high)$
$left\text{-}sum = -\infty$
$sum = 0$
for $i = mid$ downto $low$
    $sum = sum + A[i]$
    if $sum > left\text{-}sum$
        $left\text{-}sum = sum$
        $max\text{-}left = i$
$right\text{-}sum = -\infty$
$sum = 0$
for $j = mid + 1$ to $high$
    $sum = sum + A[j]$
    if $sum > right\text{-}sum$
        $right\text{-}sum = sum$
        $max\text{-}right = j$
return $(max\text{-}left, max\text{-}right, left\text{-}sum + right\text{-}sum)$
```

Time Complexity of Max Subarray

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \,, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \,. \end{cases}$$

Recurrence

A function that is defined with respect to itself on smaller inputs

$$T(n) = 2T(n/2) + n$$

$$T(n) = 16T(n/4) + n$$

$$T(n) = 2T(n-1) + n^2$$

Why are we interested in recurrences?

Computational cost of divide and conquer algorithms

$$T(n) = aT(n/b) + D(n) + C(n)$$

- a subproblems of size n/b
- $D(n)$ the cost of dividing the data
- $C(n)$ the cost of recombining the subproblem solutions

In general, the runtimes of most recursive algorithms can be expressed as recurrences.

A Simpler Example

- Finding Factorial of n

$$n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n \quad \text{for } n \geq 1$$

and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.

```
ALGORITHM   $F(n)$
    //Computes $n!$ recursively
    //Input: A nonnegative integer $n$
    //Output: The value of $n!$
    if $n = 0$ return 1
    else return $F(n-1) * n$
```

Analysis of Recursive F(n)

- The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n). Since* the function *F(n) is computed according to the formula*

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$$

- The number of multiplications *M(n) needed to compute it must satisfy the equality*

$$M(n) = \underbrace{M(n-1)}_{\substack{\text{to compute}\\ F(n-1)}} + \underbrace{1}_{\substack{\text{to multiply}\\ F(n-1) \text{ by } n}} \qquad \text{for } n > 0.$$

- *M(n − 1) multiplications are spent to compute F(n − 1), and one more* multiplication is needed to multiply the result by *n*.

Analysis of Recursive F(n)

- Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications *M(n):*

$$\begin{aligned} M(n) &= M(n-1) + 1 \quad \text{for } n > 0, \\ M(0) &= 0. \end{aligned}$$

Analysis of Recursive F(n)

- Backward substitution

$$M(n) = M(n-1) + 1 \qquad \text{substitute } M(n-1) = M(n-2) + 1$$
$$= [M(n-2) + 1] + 1 = M(n-2) + 2 \qquad \text{substitute } M(n-2) = M(n-3) + 1$$
$$= [M(n-3) + 1] + 2 = M(n-3) + 3.$$

- After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern : $M(n) = M(n - i) + i$.

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

The challenge

Recurrences are often easy to define because they mimic the structure of the program

But… they do not directly express the computational cost, i.e. n, n^2, …

We want to remove self-recurrence and find a more understandable form for the function

Three approaches

Substitution method: when you have a good guess of the solution, prove that it's correct

Recursion-tree method: If you don't have a good guess, the recursion tree can help. Then solve with substitution method.

Master method: Provides solutions for recurrences of the form:

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

Substitution method

The ***substitution method*** for solving recurrences comprises two steps:

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

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Unfortunately, there is no general way to guess the correct solutions to recurrences.

Substitution method

- For both Mergesort and Maximum Subarray, we derived time complexity as the following recursive equation

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

- Making a good guess,

$T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n \,, \end{aligned}$$

True for c>=1

Substitution method

- Now prove this for base case, say when n=2

- Now complete the inductive step.

T(n) = O(n lg n) Proved.

Changing variables

Guesses? $T(n) = 2T(\sqrt{n}) + \log n$

We can do a variable change: let $m = \log_2 n$ (or $n = 2^m$)

$$T(2^m) = 2T(2^{m/2}) + m$$

Now, let $S(m)=T(2^m)$

$$S(m) = 2S(m/2) + m$$

Changing variables

$$S(m) = 2S(m/2) + m$$

Guess? $S(m) = O(m \log m)$

$$T(n) = T(2^m) = S(m) = O(m \log m)$$

substituting m=log n

$$T(n) = O(\log n \log \log n)$$

Recursion Tree

Guessing the answer can be difficult

$$T(n) = 3T(n/4) + n^2$$

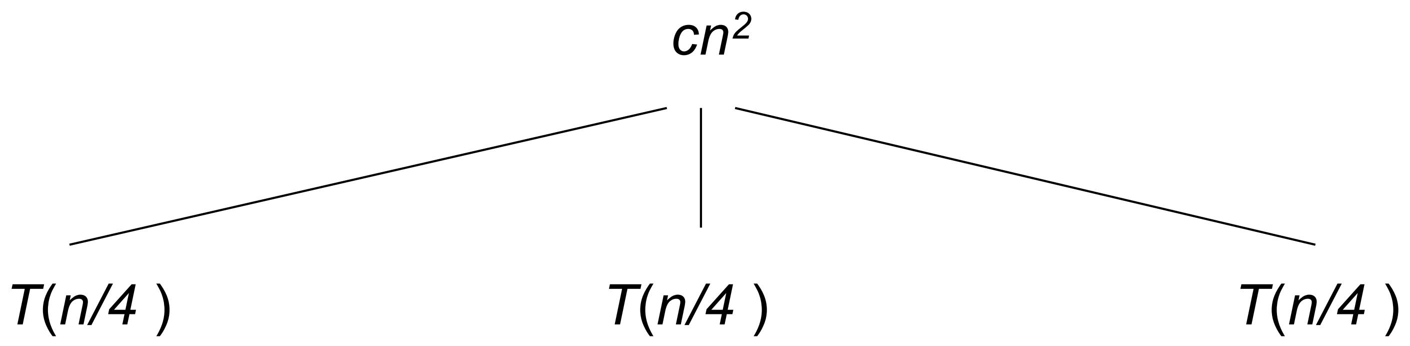
$$T(n) = T(n/3) + 2T(2n/3) + cn$$

The recursion tree approach

- Draw out the cost of the tree at each level of recursion
- Sum up the cost of the levels of the tree
  - Find the cost of each level with respect to the depth
  - Figure out the depth of the tree
  - Figure out (or bound) the number of leaves
- Verify your answer using the substitution method

$$T(n) = 3T(n/4) + cn^2$$

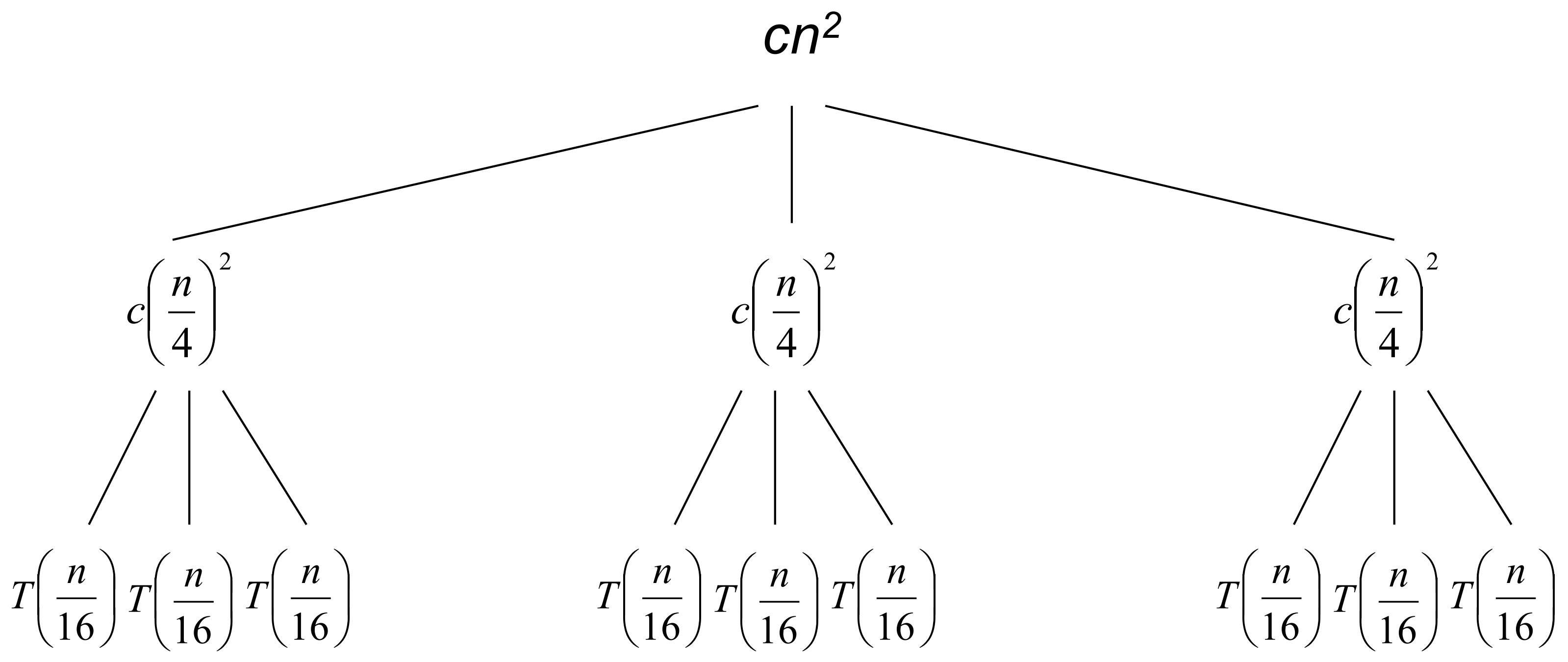
cost



cn^2

$$T(n) = 3T(n/4) + cn^2$$

cost



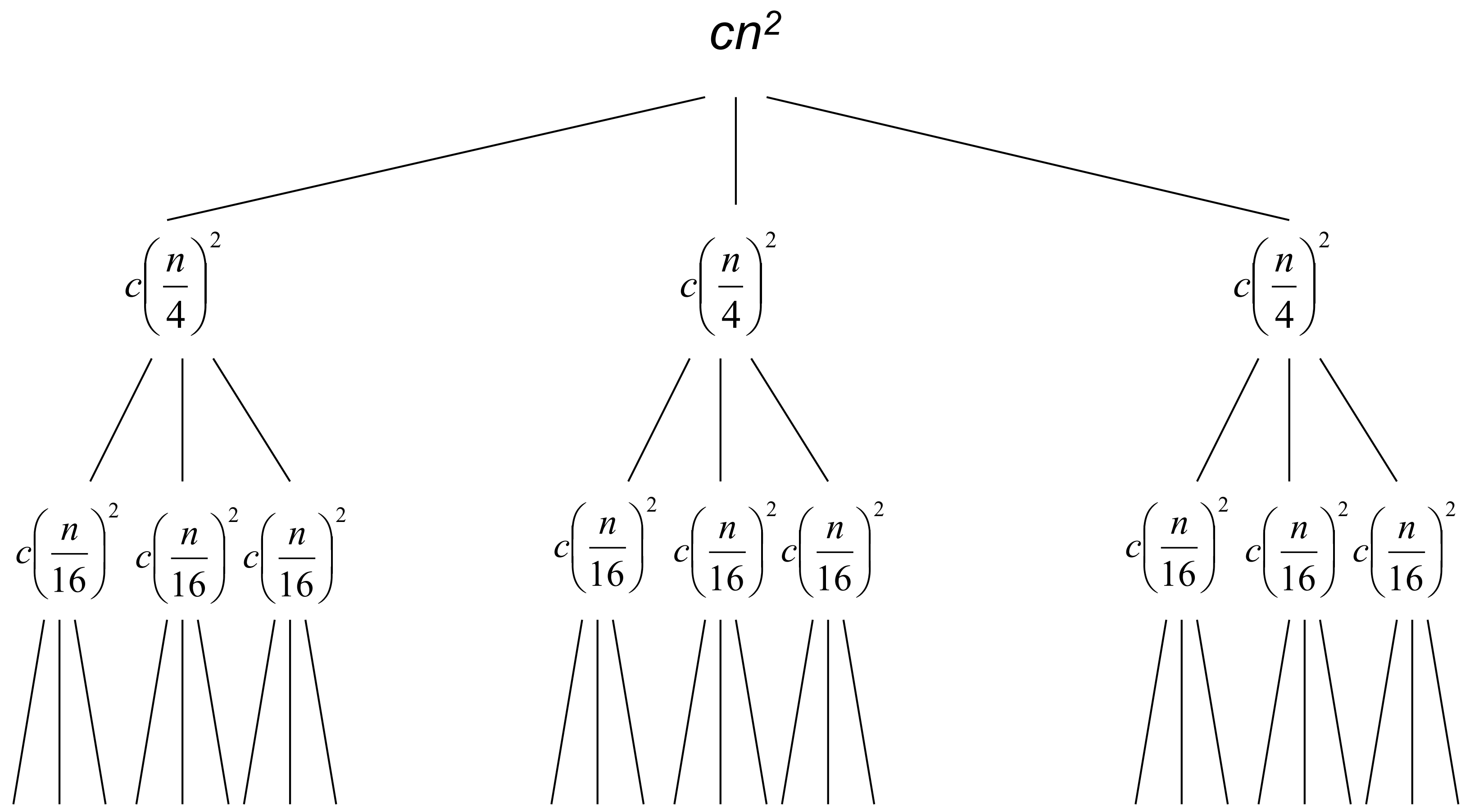
cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$
$T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$ $T\left(\frac{n}{16}\right)$

cn^2

$3/16cn^2$

$$T(n) = 3T(n/4) + cn^2$$

cost


cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$

cn^2

$3/16cn^2$

$(3/16)^2cn^2$

What is the cost at each level?

$$\left(\frac{3}{16}\right)^d cn^2$$

What is the depth of the tree?

At each level, the size of the data is divided by 4

$$\frac{n}{4^d} = 1$$

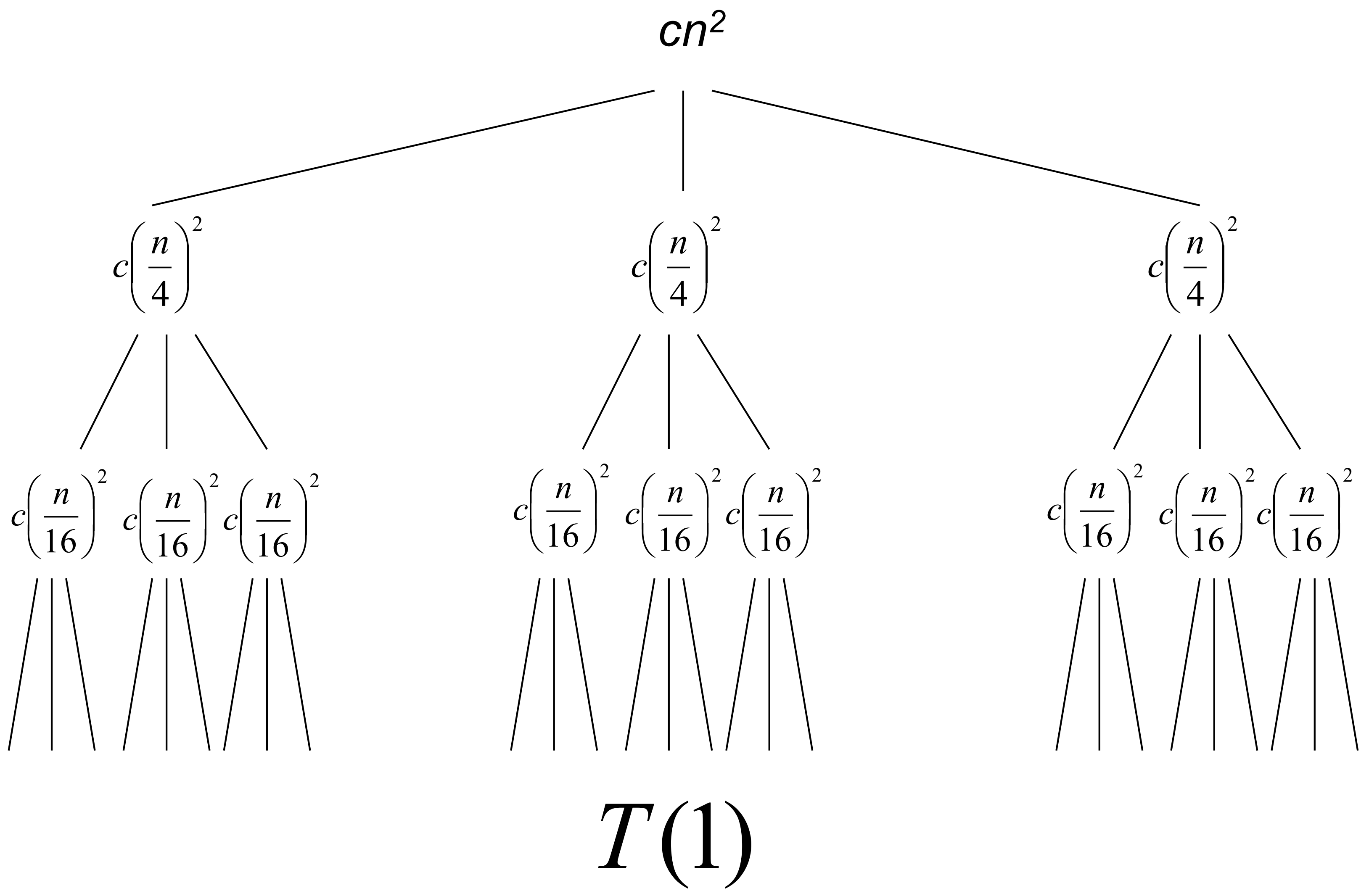
$$\log\left(\frac{n}{4^d}\right) = 0$$

$$\log n - \log 4^d = 0$$

$$d \log 4 = \log n$$

$$d = \log_4 n$$

$$T(n) = 3T(n/4) + n^2$$



cn^2
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{4}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$c\left(\frac{n}{16}\right)^2$
$T(1)$

How many leaves are there?

How many leaves?

How many leaves are there in a complete ternary tree of depth *d*?

$$3^d = 3^{\log_4 n}$$

Total cost

$$T(n) = cn^2 + \frac{3}{16}cn^2 + \left(\frac{3}{16}\right)^2 cn^2 + \ldots + \left(\frac{3}{16}\right)^{d-1} cn^2 + \Theta(3^{\log_4 n})$$

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

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

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

$$= \frac{16}{13} cn^2 + \Theta(3^{\log_4 n})$$

?

$$\boxed{\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}}$$

let x = 3/16

Total cost

$$3^{\log_4 n} = n^{\log_4 3}$$

Proof?

$$T(n) = \frac{16}{13}cn^2 + \Theta(3^{\log_4 n})$$

$$T(n) = \frac{16}{13}cn^2 + \Theta(n^{\log_4 3})$$

$$T(n) = O(n^2)$$

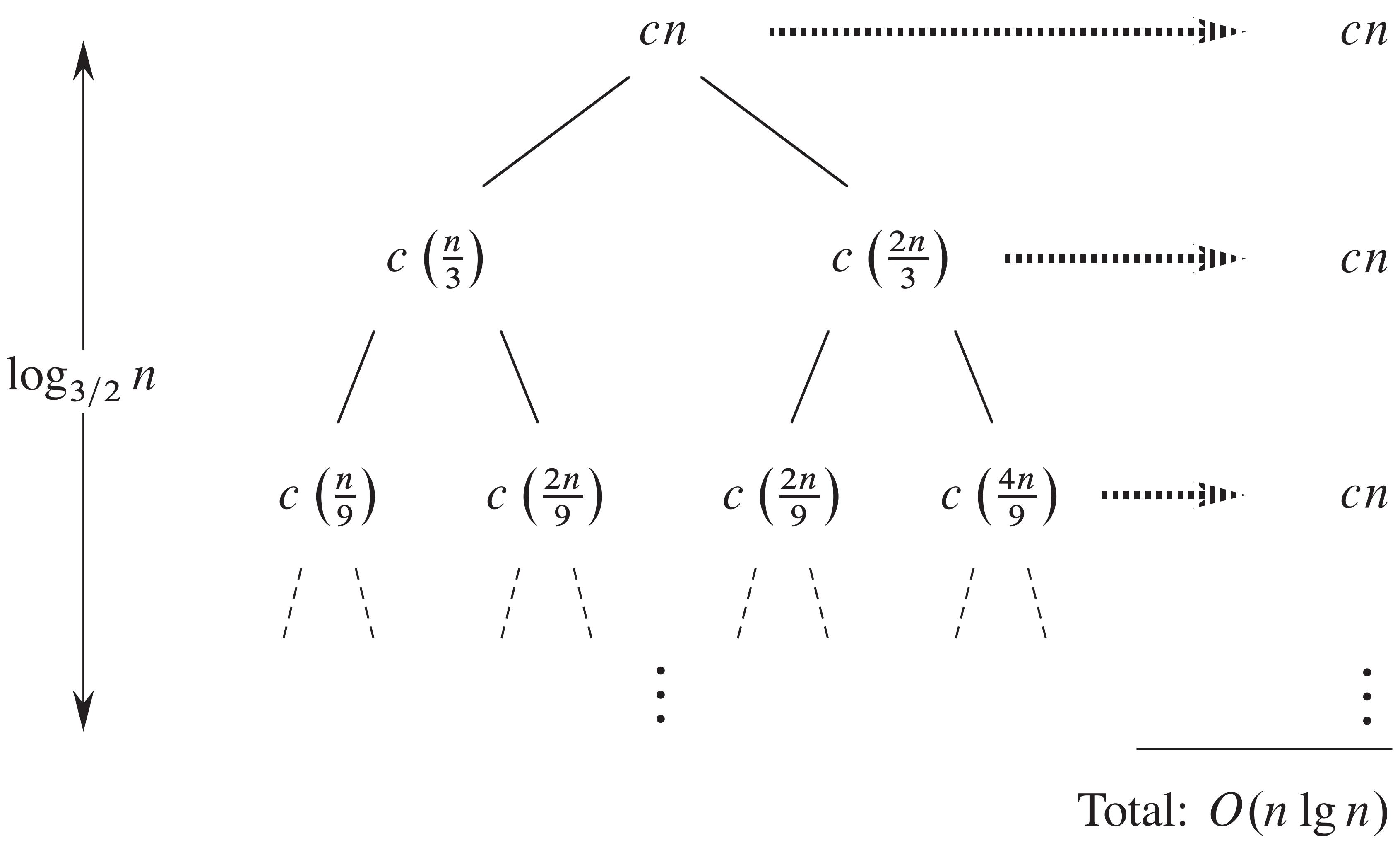
Verify solution using substitution

Now we can use the substitution method to verify that our guess was correct, that is, $T(n) = O(n^2)$ is an upper bound for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$. We want to show that $T(n) \le dn^2$ for some constant $d > 0$. Using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 , \end{aligned}$$

where the last step holds as long as $d \ge (16/13)c$.

Rec. Tree for T(n)=T(n/3)+T(2n/3)+cn


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

Master Method

Master Theorem:

Let $a \geq 1$ 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 $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master Method

- Example 1:

$$T(n) = 9T(n/3) + n \ .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9-\epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

- Example 2:

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

Master Method

- Example 3:

$$T(n) = 3T(n/4) + n \lg n \ ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3 + \epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

- Example 4: Can you solve using Master Theorem?

$$T(n) = 2T(n/2) + n \lg n$$

Master Method

- Example 3:

$T(n) = 3T(n/4) + n \lg n$,

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, we have that $af(n/b) = 3(n/4)\lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

- Example 4: Master Theorem cannot be applied.

$$T(n) = 2T(n/2) + n \lg n$$

$f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio $f(n)/n^{\log_b a} = (n \lg n)/n = \lg n$ is asymptotically less than n^ϵ for any positive constant ϵ.