

Introduction to Algorithm

Recurrences

Recurrences

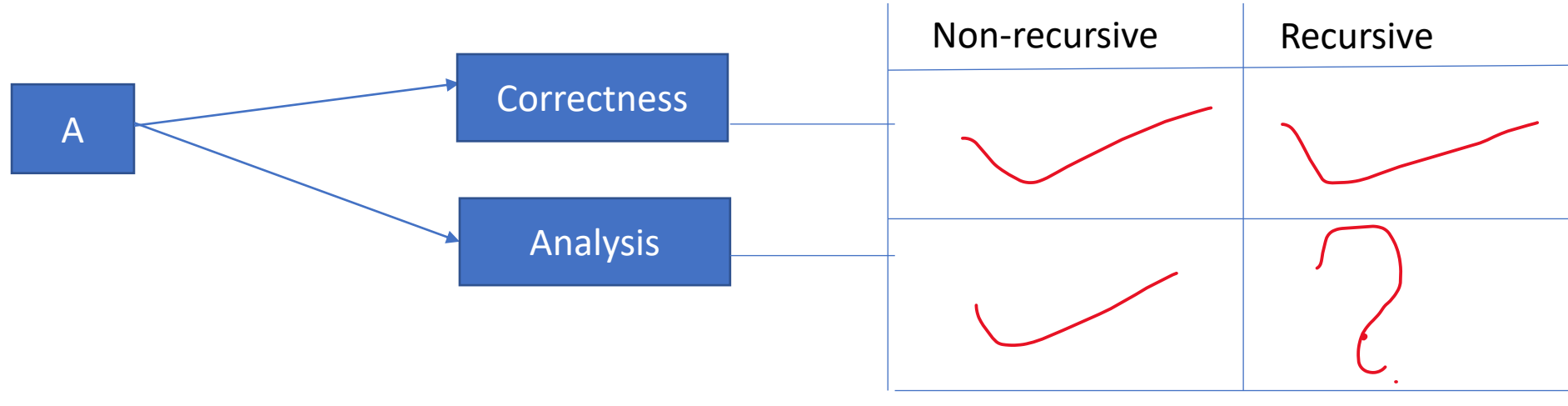
- Any problem can be solved either by writing recursive algorithm or by writing non-recursive algorithm.
- A recursive algorithm is one which makes a recursive call to itself with smaller inputs.
- We often use a recurrence to describe the running time of a recursive algorithm.
- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs or as a function of preceding (or lower) terms.
- In an Analysis of Algorithm, recurrence relations are used to analyze the running time of a recursive function.

The running time of a recursive function is denoted by $T(n)$ where n is the size of the input. In recurrence relation, the running time of a recursive function of input size n is expressed in terms of the running time of the lower value of n . For example

$$T(n)=T(n-1)+C$$

Here, the running time for size n is equal to the running time for size $n-1$ plus a constant time.

Recurrences



Recursive sum:

`rsum(a[1.....n],n)`

{

if (n==1) then

return(a[1])

else

return(rsum(a[1....n-1],n-1)+a[n])

}

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

$$T(n) = \begin{cases} 2 & \text{if } n == 1 \\ T(n-1) + 2 & \text{if } n > 1 \end{cases}$$

Recursive : function call itself

Recurrence

Recurrences arise when an algorithm contains recursive calls to itself

Recurrences

$T(n)$ is expressed as a conditional statement having the following two conditions:

Base Case: The base case is the running time of an algorithm when the value of n is small such that the problem can be solved trivially.

Recursive case: The recursive running time is given by a recurrence relation for a large value of n .

Using base and recursive case, the running time of a recursive function would look like the following.

$$T(n) = \begin{cases} O(1) & \text{if } n \leq 2 \\ T(n-1) + O(1) & \text{otherwise} \end{cases}$$

when the value of n is less than or equal to 2, we can find the solution trivially in constant time. If the value of n is greater than 2, we can use the recurrence relation to find the running time.

Recurrences

- The process of translating a code into a recurrence relation
 1. The first thing to look in the code is the base condition and note down the running time of the base condition. Remember: every recursive function must have a base condition.
 2. For each recursive call, notice the size of the input passed as a parameter.
 3. Calculate the running time of operations that are done after the recursion calls.
 4. Finally, write the recurrence relation.

Example: starting with the factorial function. calculate the factorial of a number n .

```
int fact(int n) {  
    if (n == 0) {  
        return 1;  
    }  
  
    return n * fact(n - 1);  
}
```

The base condition happens when the value of n is 0. We can find the value of $0!$ in constant time (c) trivially i.e. for base condition

$$T(0)=C$$

we look for the recursive call. There is only one recursive call with input size $n-1$ and after the recursive function returns ,the total running time is

$$T(n)=T(n-1)+C$$

Recurrences

Combining this with the base condition, we get

$$T(n) = \begin{cases} \underline{C} & \text{if } \underline{n=0} \\ T(n-1)+C & \text{otherwise} \end{cases}$$

calculating n^{th} Fibonacci number

```
int fib(int n) {  
    if (n <= 1)  
    {  
        return n;  
    }  
    return fib(n - 1) + fib(n - 2);  
}
```

When the value of n is 0 or 1, we can trivially find the corresponding Fibonacci number in constant time C i.e.

There are two recurrence relations - one takes input $n-1$ and other takes $n-2$. Once we get the result of these two recursive calls, we add them together in constant time i.e.

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

Combining with the base case, we get

$$T(n) = \begin{cases} C & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + C & \text{otherwise} \end{cases}$$

Recurrences

To solve a recurrence relationship $T(n)$ that represents the running time we are using four methods.

There are four methods for solving Recurrence

1. Substitution Method

2. Iteration Method

3. Recursion Tree Method

4. Master Method

1. Iteration Method

Convert the recurrence into a summation and try to bound it using known series

- Iterate the recurrence until the initial condition is reached.
- Use back-substitution to express the recurrence in terms of n and the initial (boundary) condition.

Iteration Method

- In evaluating the summation one or more of the following summation formulae may be used:
- Arithmetic series:

$$\sum_{k=1}^n k = 1 + 2 + \dots + n = \frac{n(n+1)}{2}$$

- Geometric Series:

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

$$\sum_{k=0}^{n-1} x^k = \frac{x^n - 1}{x - 1} (x \neq 1)$$

- Special Cases of Geometric Series:

$$\sum_{k=0}^{n-1} 2^k = 2^n - 1$$

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1 - x} \quad \text{if } x < 1$$

Iteration Method

- Harmonic Series:

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \dots + \frac{1}{n} \approx \ln n$$

- Others:

$$\sum_{k=1}^n \lg k \approx n \lg n$$

$$\sum_{k=0}^{n-1} c = cn.$$

$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=0}^n k(k+1) = \frac{n(n+1)(n+2)}{3}$$

Iteration Method

Recursive sum:

rsum(a[1.....n],n)

{

if (n==1) then

return(a[1])

else

return(rsum(a[1....n-1],n-1)+a[n])

}

$$T(n)=T(n-1)+ C$$

where C=constant

$$T(n)=\underline{T(n-1)}+C$$

$$= \{T(n-2)+C\}+C$$

$$= T(n-3)+C +C+C$$

$$=T(n-4)+C+C+C$$

.....

$$=T(1)+\{C+C+.....(n-1)\text{-times}\}$$

$$=T(1)+C . (n-1)$$

$$= \Theta(1)+C . n - C)=\Theta(n)$$

If we knew $T(n - 1)$, we could solve $T(n)$.

$$T(n) = T(n - 1) + c$$

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

$$= T(n - 2) + c + c = T(n - 2) + 2c$$

$$T(n - 2) = T(n - 3) + c = T(n - 3) + c + 2c = T(n - 3) + 3c$$

$$T(n - 3) = T(n - 4) + c = T(n - 4) + 4c = \dots = T(n - k) + kc$$

$$T(n) = T(n - k) + k * c \text{ for all } k$$

If we set $k = n$, we have:

$$T(n) = T(n - n) + nc$$

$$= T(0) + nc = c1 + nc$$

Iteration Method

- Example 1: Form and solve the recurrence relation for the running time of factorial method and hence determine its big-O complexity:

```
long factorial (int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial (n - 1);  
}
```

$$\begin{aligned}T(o) &= c \\T(n) &= b + T(n-1) \\&= b + b + T(n-2) \\&= b + b + b + T(n-3) \\&\dots \\&= kb + T(n-k)\end{aligned}$$

When $k = n$, we have:

$$\begin{aligned}T(n) &= nb + T(n-n) \\&= bn + T(o) \\&= bn + c.\end{aligned}$$

Therefore method factorial is $O(n)$.

$$T(n) = \begin{cases} T(n-1) + b & \text{if } n > 0 \\ c & \text{if } n == 0 \end{cases}$$

Iteration Method

Few Question :

1. $T(n) = T(n-1) + n$

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

$$= T(n-3) + n-2 + n-1 + n$$

.....

$$= T(1) + 2 + 3 + 4 + \dots + (n-1) + n$$

$$= T(1) + \{1 + 2 + \dots + n-1 + n\} - 1$$

$$= (c-1) + \{n(n+1)/2\} = \Theta(n^2)$$

2. $T(n) = T(n-1) + 1/n$

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

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

$$= \dots\dots\dots$$

$$= T(1) + 1/2 + 1/3 + 1/4 + \dots + 1/n$$

$$= T(1) - 1 + \{1 + 1/2 + \dots + 1/n\}$$

$$= C + \log n = \Theta(\log n)$$

3. Try to solve

$$T(n) = T(n-1) + \log n$$

Hint: $\log(1.2.3.4\dots n) = \log(n!)$

$$\log_c(ab) = \log_c a + \log_c b$$

$\log(n!) \leq n \log n$ as $n! \leq n^n$ (Stirling's Approximation)

Substitution method

1. Guess a solution
2. Use induction to prove that the solution works

We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

Example:

$$T(n) = \begin{cases} T(1) & \text{for } n \leq 1 \\ T(n/2) + c & \text{for } n > 1 \end{cases}$$

We will substitute the formula in each step to get the result –

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

By substituting $T(n/2)$ we can write,

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

$$\Rightarrow T(n) = (T(n/4) + c) + c$$

$$T(n) = T(n/4) + 2c$$

$$T(n) = T(n/4) + 2c$$

$$T(n) = T(n/8) + 3c$$

$$T(n) = T(n/8) + 3c$$

$$T(n) = T(n/2^k) + kc \Rightarrow T(n) = T(1) + kc = C + \log_2 n = \Theta(\log_2 n)$$

Assume $n = 2^k \Rightarrow \log n = k \log 2$

$$\Rightarrow k = \log n / \log 2$$

$$[\log_b a = \log_c a / \log_c b]$$

$$\Rightarrow k = \log_2 n$$

Substitution method

```
long power(long x, long n)
```

```
    if (n==0) return 1;
```

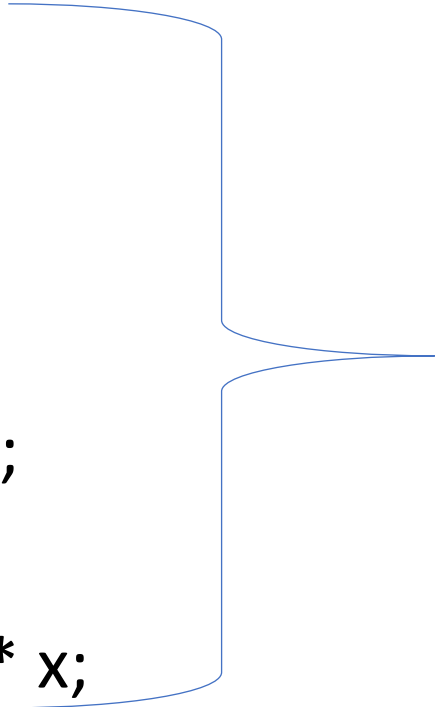
```
    if (n==1) return x;
```

```
    if ((n % 2) == 0)
```

```
        return power(x*x, n/2);
```

```
    else
```

```
        return power(x*x, n/2) * x;
```


$$T(0) = c_1$$

$$T(1) = c_2$$

$$T(n) = T(n/2) + c_3 = \Theta(\log_2 n)$$

Substitution method

$$T(n) = \begin{cases} T(1) & \text{for } n=1 \\ 2T(n/2) + cn & \text{for } n>1 \end{cases}$$

We will substitute the formula in each step to get the result –

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

By substituting $T(n/2)$ we can write,

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

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

$$= 4T(n/4) + 2cn \Rightarrow T(n) = 4T(n/4) + 2cn$$

$$= 8T(n/8) + 3cn \Rightarrow T(n) = 8T(n/8) + 3cn$$

$$\{ \underline{T(n/2) = 2T(n/4) + c*n/2} \}$$

$$\{ \underline{T(n/4) = 2T(n/8) + c*n/4} \}$$

$$\Rightarrow T(n) = 2^k T\left(\frac{n}{2^k}\right) + kcn \Rightarrow T(n) = 2^k T\left(\frac{n}{2^k}\right) + kcn \Rightarrow T(n) = nT(1) + kcn$$

$$\text{Assume } n = 2^k \Rightarrow k = \log_2 n$$

$$\Rightarrow T(n) = c1 \cdot n + k \cdot c \cdot n \Rightarrow T(n) = c1 \cdot n + c \cdot n \log_2 n = \Theta(n \log_2 n)$$

Substitution method

Recurrence relation using change of variable technique

Problem: 1 $T(n) = T(\sqrt{n}) + c$ and $T(2) = k$

Introduce a change of variable by taking $n = 2^m \Rightarrow m = \log n$

$$\Rightarrow T(2^m) = T(\sqrt{2^m}) + c \Rightarrow T(2^m) = T(2^{m/2}) + c$$

$$= T(2^{m/4}) + c + c = T(2^{m/8}) + c + c + c + \dots$$

$$= T(2^{m/m}) + c * p = k + c \log m = k + c \log(\log n) = \Theta(\log \log_2 n)$$

Assume $m = 2^p \Rightarrow p = \log m$

Question: $T(n) = 2 T(\sqrt{n}) + n$ and $T(1) = 1$

Master Method

The Master method is applicable for solving recurrences of the form

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function. Depending upon the value of a, b and function $f(n)$, the master method has three cases.

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 $f(n) = \Theta(n^{\log_b a} \log^k n)$ where $k > 0$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
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 $T(n) = \Theta(f(n))$

Basically in master method, we check if $f(n)$ is upper, tight or lower bound on $n^{\log_b a}$. If it is upper bound, then we use case 1. If it is a tight bound, we use case 2 and if it is a lower bound, we use case 3.

Now we will use The Master method to solve some of the recurrences.

Master Method

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

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$

$n^{\log_b a} = n^{\log_3 9} = (n^2)$. 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 $T(n) = \Theta(n^2)$.

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

For this recurrence $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$.

Case 2 applies, since $f(n) = (n^{\log_b a} * \log^0 n) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(1 * \log^{0+1} n) = \Theta(\log n)$

Master Method

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

we have $a = 2$, $b = 2$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_2 2} = n$.

Since $f(n) = \Theta(n^{\log_2 2 + 1})$, where $\epsilon = 1$,

Case 3 applies hence $T(n) = \Theta(f(n)) = \Theta(n^2)$

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 4T(n/2) + n$.

b. $T(n) = 4T(n/2) + n^2$.

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

Master Method

Example 1: Consider a recurrence,

$$T(n)=2T(n/4)+1$$

The recurrence relation is in the form given by (1), so we can use the master method. Comparing it with (1), we get

$$a=2, b=4 \text{ and } f(n)=1$$

Next we calculate $n^{\log_b a} = n^{\log_4 2} = n^{0.5}$

Since $f(n)=O(n^{0.5-0.1})$ where $\epsilon=0.1$,

we can apply the case 1 of the master method. Therefore,

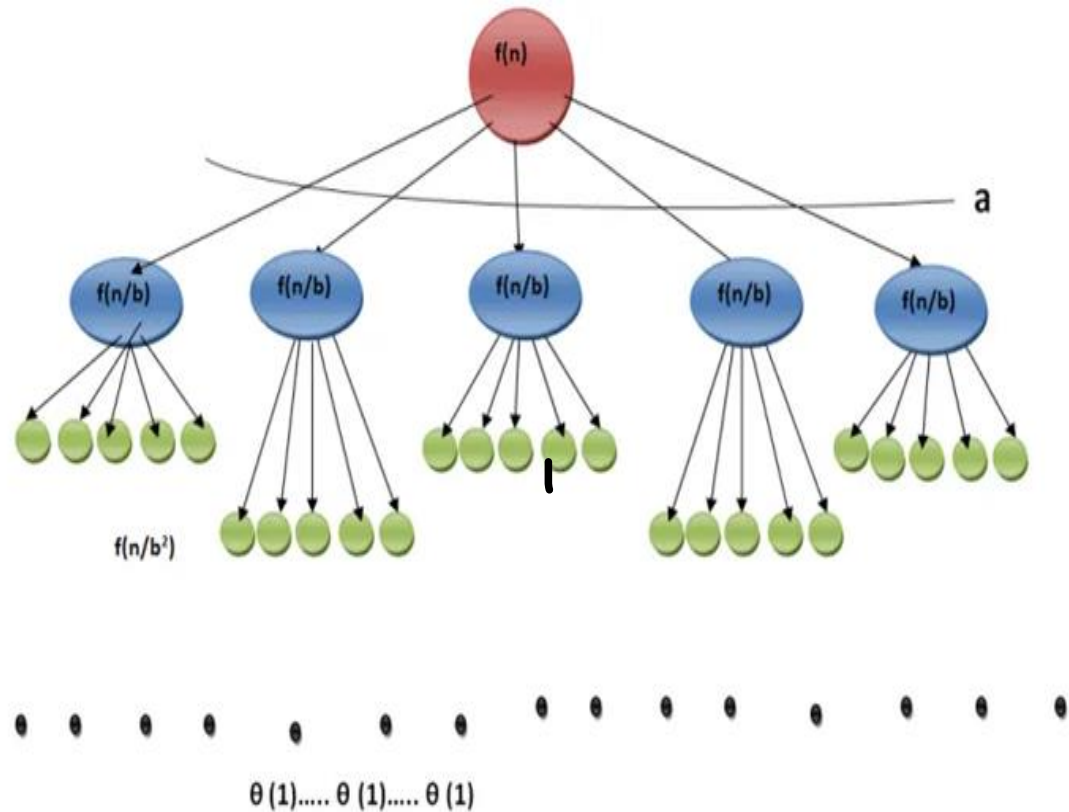
$$T(n)=\Theta(n^{\log_b a})=\Theta(n^{0.5})$$

Example 2: $T(n)=4T(n/2)+cn$

Here $a=4$, $b=2$, $f(n)=cn$

$n^{\log_2 4} = n^2$ hence case 1 of masters theorem $T(n)=\Theta(n^2)$

Recursion Tree Method



Remember that n is the size of the original problem you can imagine the recursion continuing until only a single element remains. This would happen at $T(1)$.

This means the subproblem size (where n is the original problem size) is n/b^i at the i th level.

At the boundary, the subproblem size is 1

$$n/b^i = 1$$

$$\log(n/b^i) = \log(1)$$

$$\log(n) - \log(b^i) = 0$$

$$\log(n) = \log(b^i)$$

$$\log_b(n) = i$$

Since the height of the tree is the level where the boundary condition is met, the tree has height $\log(n)$.

H is the height of the tree, then **H** = **$\log_b n$** .

The problem size reduces by a factor **b** at each level. So it would take $\log_b n$ levels to reduce it to a problem of size 1 and it cannot be divided further.

Recursion Tree Method

In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically a arithmetic or geometric series.

- 1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.
- 2. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.
- 3. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.
- 4. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

Recursion Tree Method

Steps: 1 . solving recurrences expanding the recurrence into a tree

Step:2. summing the cost at each level

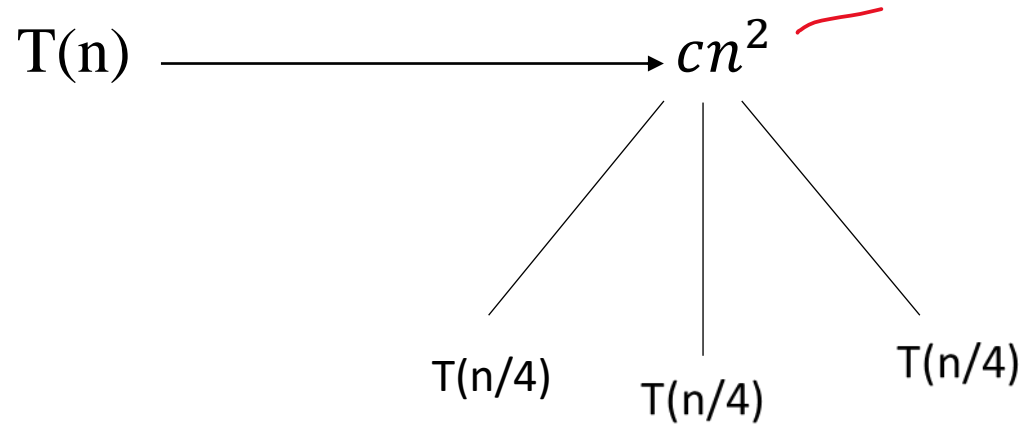
Step:3. applying the substitution method

Example:

Consider the recurrence relation $T(n) = 3T(n/4) + cn^2$ for some constant c

We assume that n is an exact power of 4.

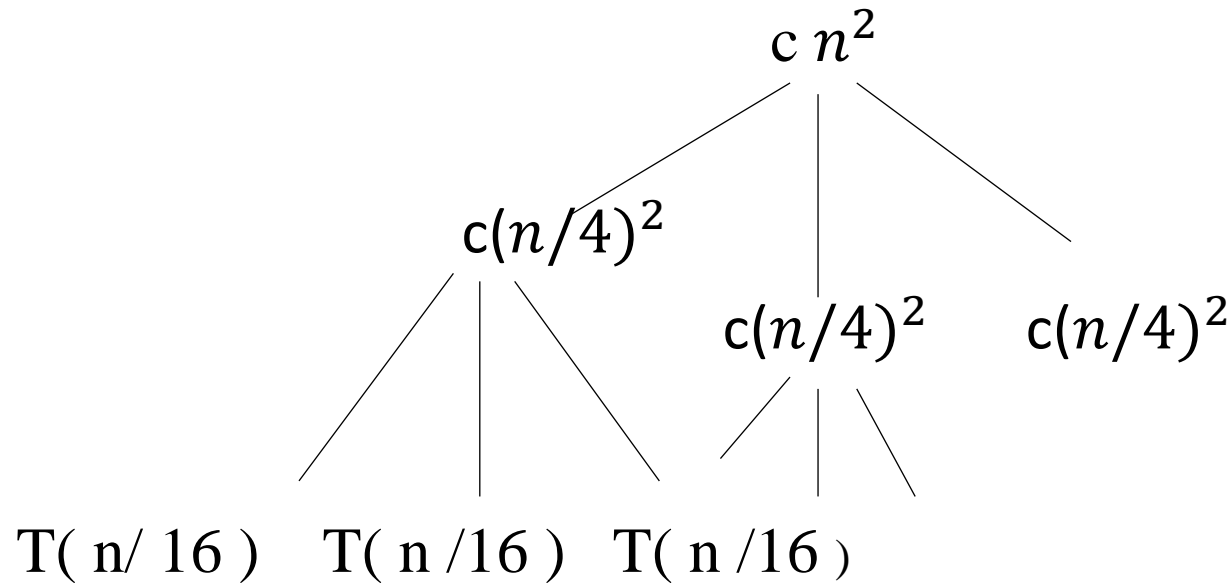
In the recursion-tree method we expand $T(n)$ into a tree:



Recursion Tree Method

- we expand $T(n/4)$

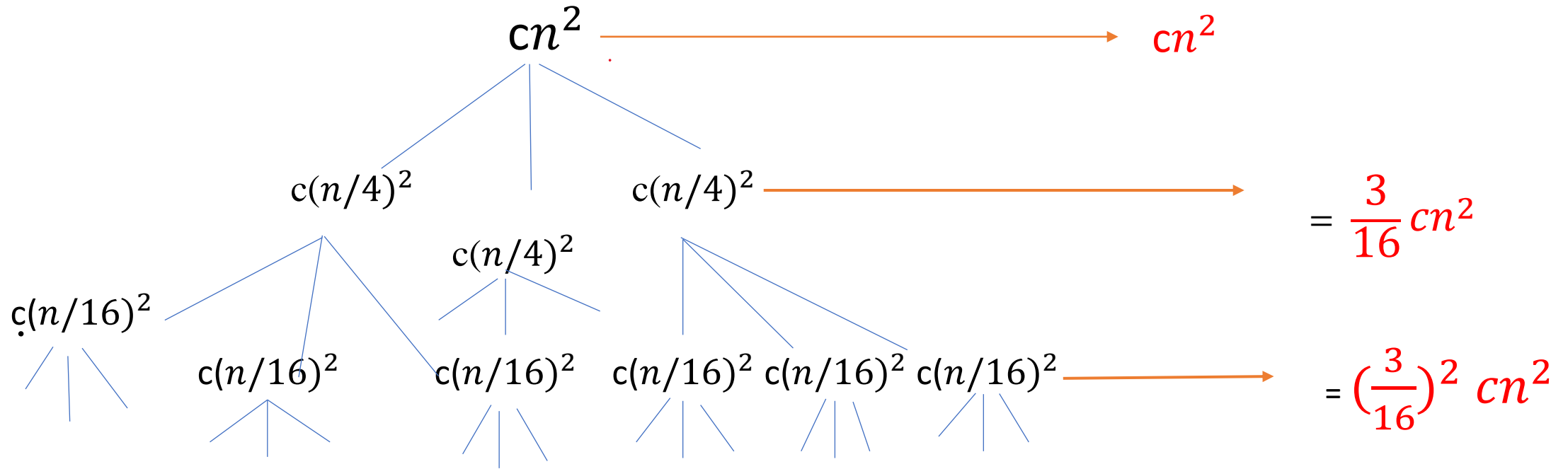
Applying $T(n) = 3T(n/4) + c n^2$ to $T(n/4)$ leads to $T(n/4) = 3T(n/16) + c(n/4)^2$, expanding the leaves:



we expand $T(n/16)$

Applying $T(n) = 3T(n/4) + c n^2$ to $T(n/16)$ leads to $T(n/16) = 3T(n/64) + c(n/16)^2$, expanding the leaves:

Recursion Tree Method



2.summing the cost at each level

We sum the cost at each level of the tree:

$$T(n) = cn^2 + (3/16)cn^2 + (3/16)^2 cn^2 + \dots = cn^2(1 + 3/16 + (3/16)^2 + \dots)$$

if $n = 16$, or the tree has depth at least 2 if $n \geq 16 = 4^2$.

Recursion Tree Method

- For $n = 4^k$, $k = \log_4(n)$,
- we have: $T(n) = cn^2 \sum_{i=0}^{\log n} (3/16)^i$