Recurrence Relation

Recurrence Relation

- A recurrence relation for the sequence, a0, a1,...an, is an equation that relates an to certain of its predecessors a0, a1, ... , an-1.

- Initial conditions for the sequence a0, a1, ... are explicitly given values for a finite number of the terms of the sequence.

Definition

- A recurrence relation, $T(n)$, is a recursive function of integer variable n
- Like all recursive functions, it has both recursive case and base case.
- Example:

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

- The portion of the definition that does not contain T is called the base case of the recurrence relation; the portion that contains T is called the recurrent or recursive case.
- Recurrence relations are useful for expressing the running times (i.e., the number of basic operations executed) of recursive algorithms

Example: Fibonacci Sequence

- The Fibonacci sequence is defined by recurrence relation

- $f_n = f_{n-1} + f_{n-2}$, n>=3 and initial conditions

- $f_1 = 1$, $f_2 = 1$.

- Fibonacci Sequence: 1, 1, 2, 3, 5, 8, 13, 21,…….

Forming Recurrence Relation

- For a given recursive method, the base case and the recursive case of its recurrence relation correspond directly to the base case and the recursive case of the method.
- Example 1: Write the recurrence relation for the following method.

```
void f(int n)
    {  if (n >
    0) {
         cout<<n;
         f(n-1);
    } }
```

- The base case is reached when n == 0. The method performs one comparison. Thus, the number of operations when n == 0, T(0), is some constant a.
- When n > 0, the method performs two basic operations and then calls itself, using ONE recursive call, with a parameter n – 1.
- Therefore the recurrence relation is:

$T(0) = a$ where a is constant
$T(n) = b + T(n-1)$ where b is constant, $n>0$

Forming Recurrence Relation

- Example 2: Write the recurrence relation for the following method.

```
 int g(int n) {
   if (n == 1)
      return 2;
   else
      return 3 * g(n / 2) + g( n / 2) + 5;
}
```

- The base case is reached when n $==1$. The method performs one comparison and one return statement. Therefore, T(1), is constant c.
- When n>1, the method performs TWO recursive calls, each with the parameter n/2, and some constant # of basic operations.
- Hence, the recurrence relation is:

$$T(1) = c \qquad \text{for some constant c}$$
$$T(n) = b + 2T(n / 2) \qquad \text{for a constant b}$$

Solving Recurrence Relation

- To solve a recurrence relation T(n) we need to derive a form of T(n) that is not a recurrence relation. Such a form is called a “closed form” of the recurrence relation.
- There are four methods to solve recurrence relations that represent the running time of recursive methods:
  - Iteration method (unrolling and summing)
  - Recursion Tree method
  - Substitution method
  - Master method

Solving Recurrence Relations - Iteration method

Steps:

- Expand the recurrence
- Express the expansion as a summation by plugging the recurrence back into itself until you see a pattern.
- Evaluate the summation

Solving Recurrence Relations - Iteration method

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

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

- Geometric Series:

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

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

Special Cases of Geometric Series:

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

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

Solving Recurrence Relations - Iteration method

- Harmonic Series

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

- Others:

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

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

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

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

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

Analysis Of Recursive Factorial 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);
}
```

- $T(0) = c$

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

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

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

  …

  $= kb + T(n - k)$

  When $k = n$, we have:

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

  $= bn + T(0)$

  $= bn + c.$

  Therefore method factorial is $O(n)$.

Analysis Of Recursive Binary Search

```
public int binarySearch (int target, int[] array,
                                   int low, int high) {
    if (low >
        high)
        return -1;
    else {
        int middle = (low + high)/2;
        if (array[middle] == target)
            return middle;
        else if(array[middle] < target)
            return binarySearch(target, array, middle + 1,
        high);  else
            return binarySearch(target, array, low, middle - 1);
      }
}
```

The recurrence relation for the running time of the method is:

$T(1) = a$ if $n = 1$ (one element array)

$T(n) = T(n / 2) + b$ if $n > 1$

Analysis Of Recursive Binary Search

Expanding:

$$T(n) = T(n / 2) + b$$
$$= [T(n / 4) + b] + b = T(n / 2^2) + 2b$$
$$= [T(n / 8) + b] + 2b = T(n / 2^3) + 3b$$
$$= \ldots\ldots\ldots$$
$$= T(n / 2^k) + kb$$

When $n / 2^k = 1$ $n = 2^k$ $k = \log_2 n$, we have:

$$T(n) = T(1) + b \log_2 n$$

$$= a + b \log_2 n$$

Therefore, Recursive Binary Search is **O(log n)**

Recursion-tree method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion tree method is good for generating guesses for the substitution method.
- The recursion-tree method can be unreliable.
- The recursion-tree method promotes intuition, however.
- In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the $\Theta(\cdot)$ notation.

Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

Root

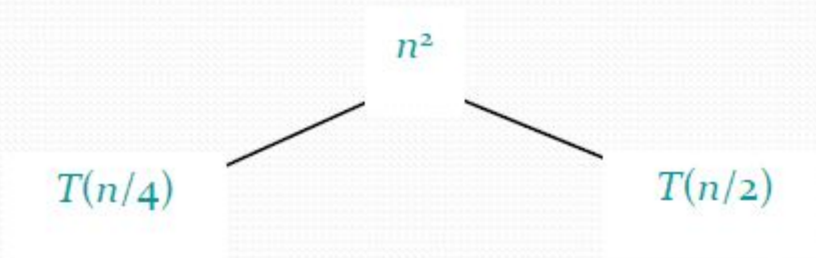
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:

$$T(n)$$

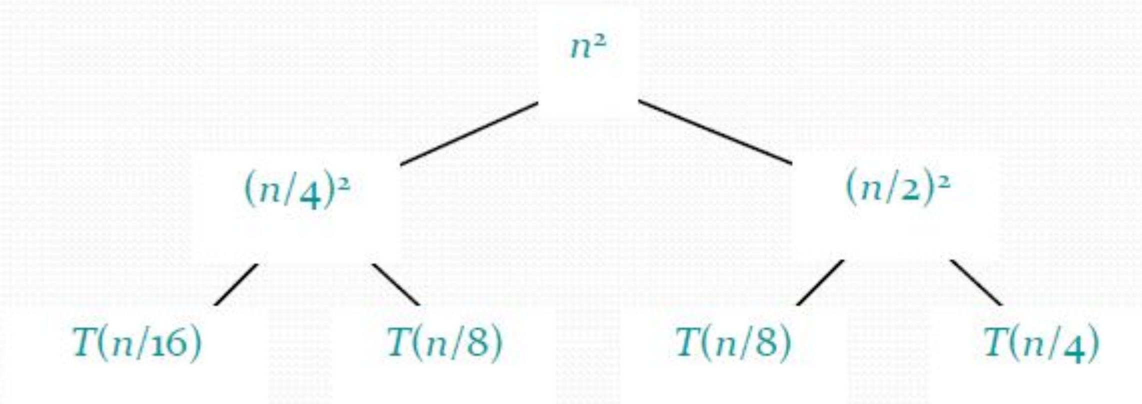
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



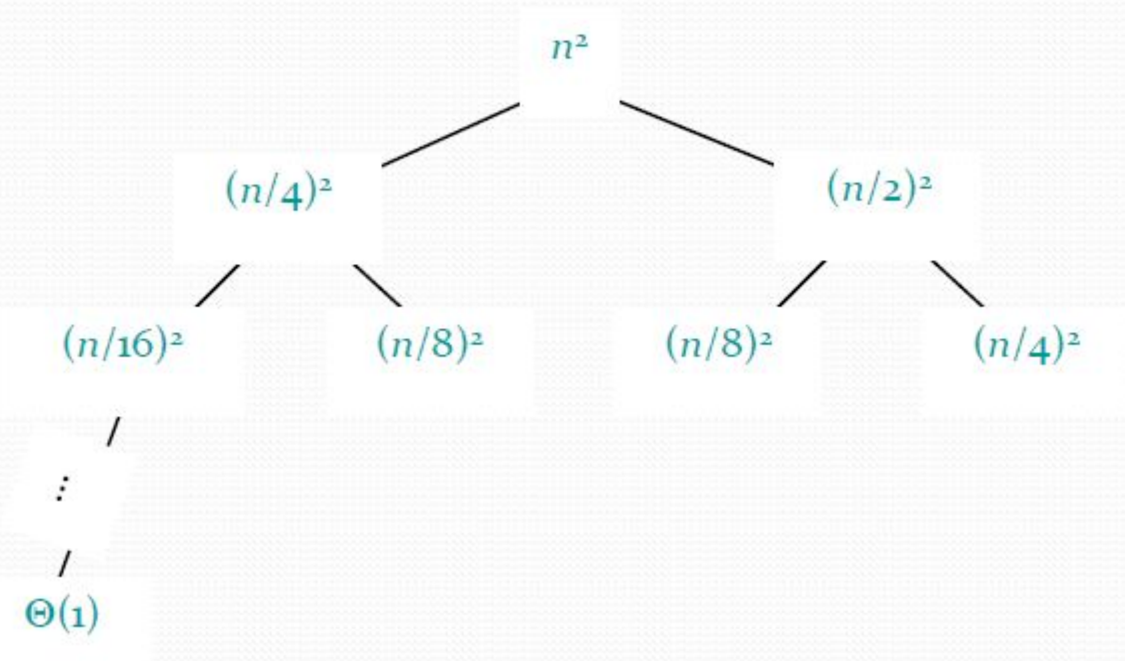
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



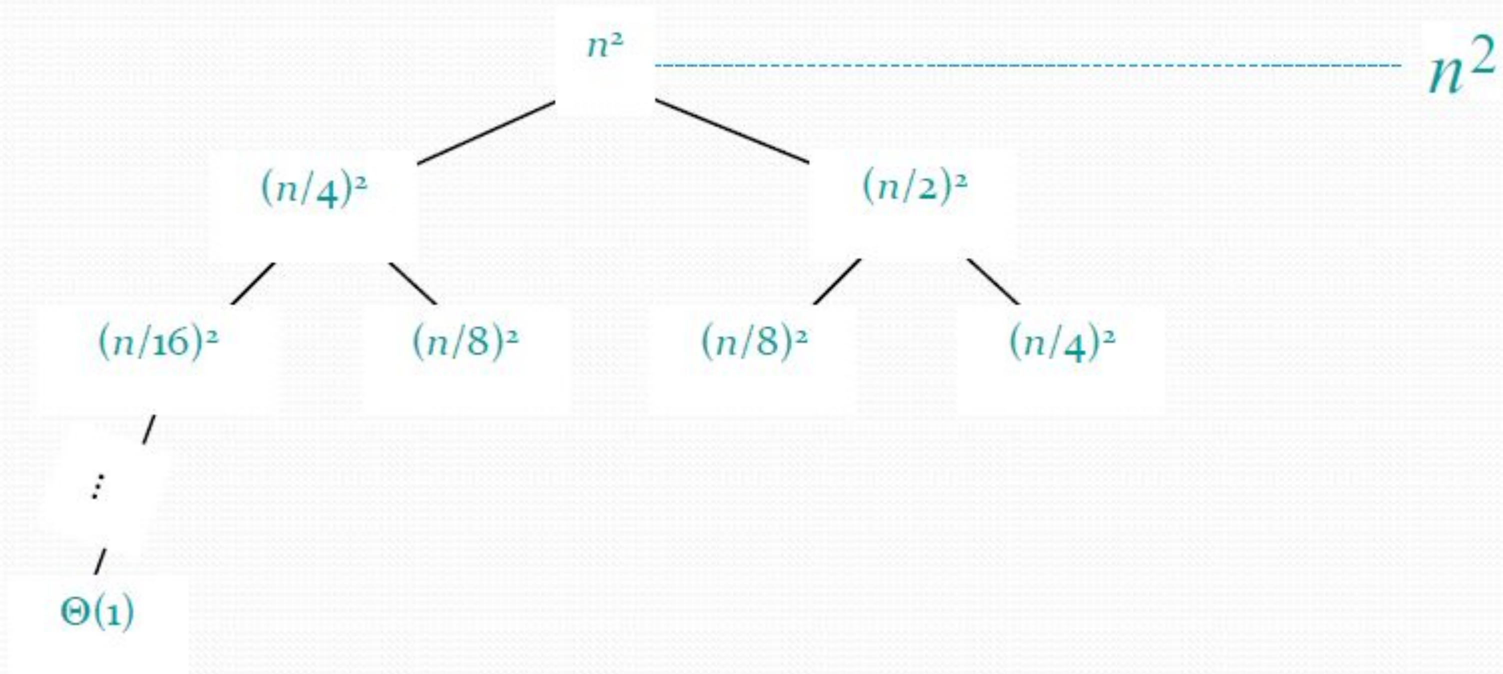
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



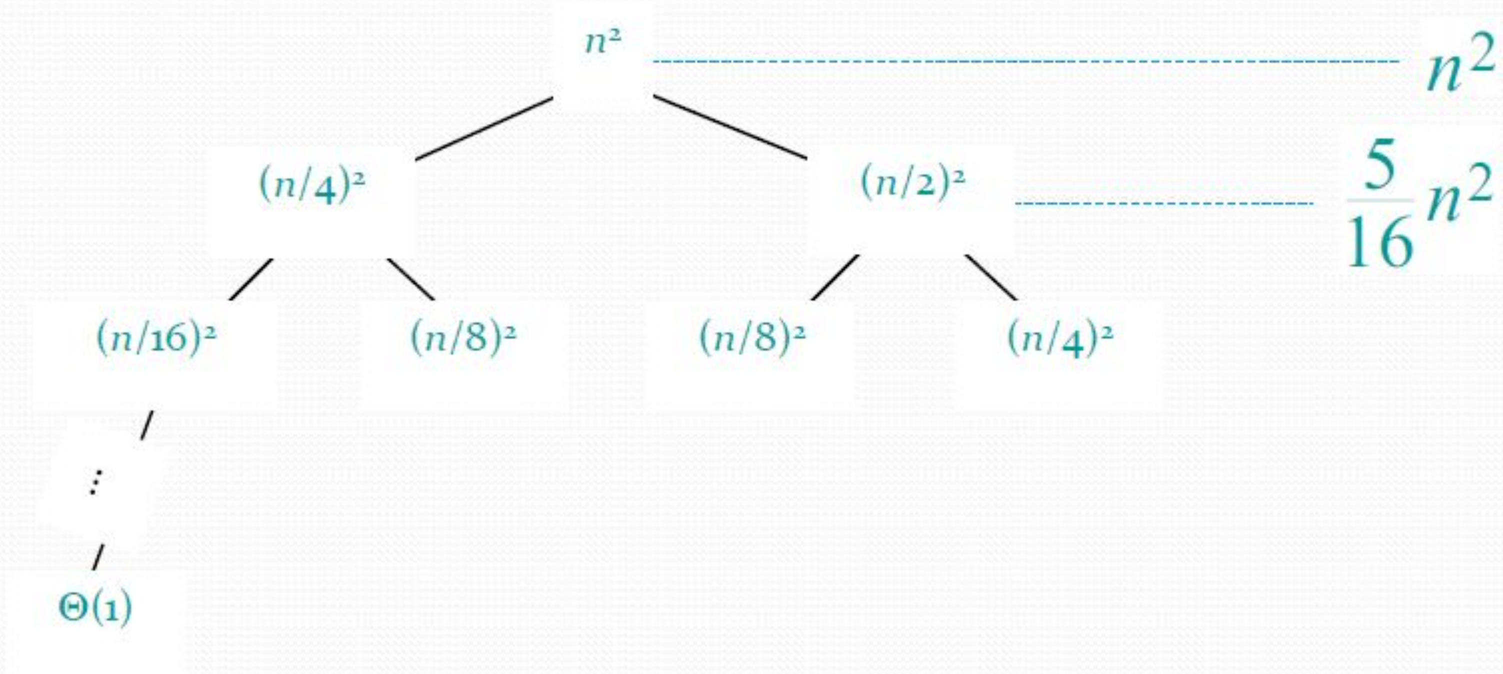
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



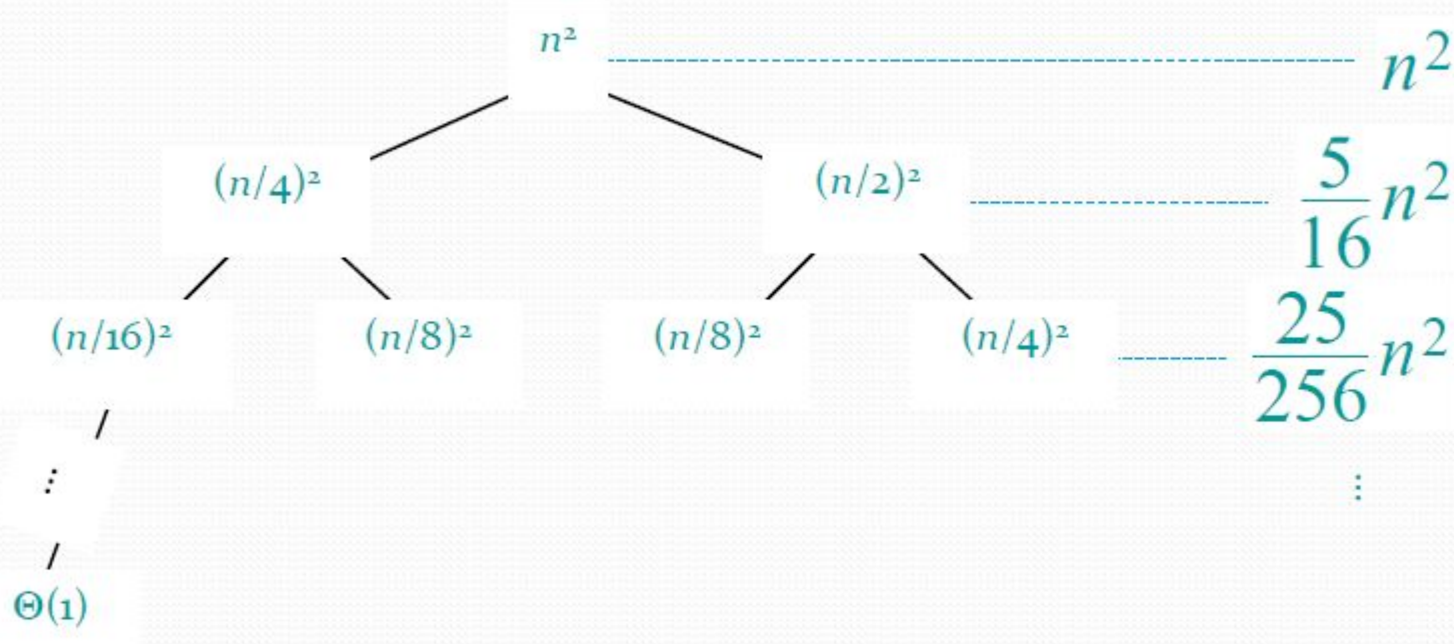
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



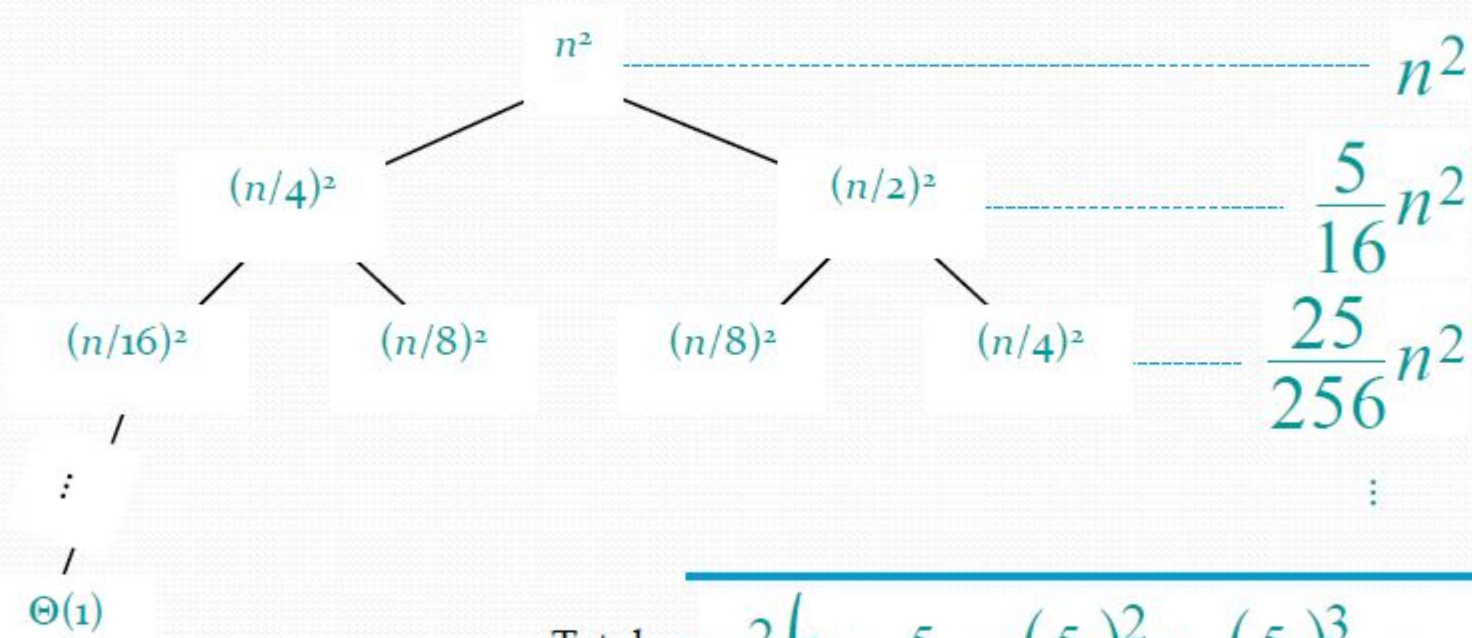
Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



Example of recursion tree

Solve $T(n) = T(n/4) + T(n/2) + n^2$:



$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

$= \Theta(n^2)$ *geometric series*

Solving Recurrence Relations-Substitution Method

- The substitution method
  - A.k.a. the “making a good guess method”
  - Guess the form of the answer, then use induction to find the constants and show that solution works
  - Run an example: merge sort
    - $T(n) = 2T(n/2) + cn$
    - We guess that the answer is $O(n \lg n)$
    - Prove it by induction
  - Can similarly show $T(n) = \Omega(n \lg n)$, thus $\Theta(n \lg n)$ 24

Solving Recurrence Relations-Substitution Method

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

- Guess $T(n) \leq cn \log n$ for some constant c (that is, $T(n) = O(n \log n)$)
- $T(n) = 2T(n/2) + n \leq 2(c\, n/2 \log n/2\ ) + n$
  - $= cn \log n/2 + n$
  - $= cn \log n - cn \log 2 + n$
  - $= cn \log n - cn + n$

  if $c \geq 1$, condition is satisfied.

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

Solving Recurrence Relations- The Master Theorem

- Given: a divide and conquer algorithm
  - An algorithm that divides the problem of size n into a subproblems, each of size n/b
  - Let the cost of each stage (i.e., the work to divide the problem + combine solved subproblems) be described by the function f(n).
- Then, the Master Theorem gives us a method for the algorithm's running time:

The Master Theorem

- if T(n) = aT(n/b) + f(n) then (where a>=1, b>1)

$$T(n) = \left\{ \begin{array}{ll} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{array} \right\} \begin{array}{l} \varepsilon > 0 \\ c < 1 \end{array}$$

The Master Theorem

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

- $a=9$, $b=3$, $f(n) = n$
- $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$
- Since $f(n) = n$, $f(n) < n^{\log_b a}$
- **Case 1 applies:**

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

$$T(n) = \Theta\left(n^{\log_b a}\right) \text{when } f(n) = O\left(n^{\log_b a - \varepsilon}\right)$$

- Thus the solution is $T(n) = \Theta(n^2)$

The Master Theorem

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

- $a = 4$, $b = 2$, $f(n)=n^2$
- $n^{\log_b a} = n^2$
- Since, $f(n)=n^2$
- Thus, $f(n)= n^{\log_b a}$

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta\left(f(n)\right) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{ AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

- **Case 2 applies:**

$$f(n) = \Theta(n^2 \log n)$$

- Thus the solution is $T(n) = \Theta(n^2 \log n)$.

The Master Theorem

$$T(n) = \begin{cases} \Theta\left(n^{\log_b a}\right) & f(n) = O\left(n^{\log_b a - \varepsilon}\right) \\ \Theta\left(n^{\log_b a} \log n\right) & f(n) = \Theta\left(n^{\log_b a}\right) \\ \Theta(f(n)) & f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right) \text{AND} \\ & af(n/b) < cf(n) \text{ for large } n \end{cases} \quad \begin{matrix} \varepsilon > 0 \\ c < 1 \end{matrix}$$

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

- $a = 4$, $b = 2$, $f(n)=n^3$
- $n^{\log_b a} = n^2$; $f(n) = n^3$.
- *Since,* $f(n)=n^3$
- *Thus,* $f(n)> n^{\log_b a}$
- **Case 3 applies**:

$$f(n) = \Omega(n^3)$$

- ***and*** $4(n/2)^3 \leq cn^3$ (regulatory condition) for $c = 1/2$.

Some Common Recurrence Relation

Recurrence Relation	Complexity	Problem
$T(n) = T(n/2) + c$	$O(\log n)$	Binary Search
$T(n) = 2T(n-1) + c$	$O(2^n)$	Tower of Hanoi
$T(n) = T(n-1) + c$	$O(n)$	Linear Search
$T(n) = 2T(n/2) + n$	$O(n\log n)$	Merge Sort
$T(n) = T(n-1) + n$	$O(n^2)$	Selection Sort, Insertion Sort
$T(n) = T(n-1)+T(n-2) + c$	$O(2^n)$	Fibonacci Series

Summary

- Recurrence Relation
  - Iteration method (unrolling and summing)
  - Recursion Tree method
  - Substitution method
  - Master method