بسم الله الرحمن الرحيم

Design and Analysis of Algorithm

Tanveer Ahmed Siddiqui

Department of Computer Science
COMSATS Institute of Information Technology, Islamabad

Recap

Lecture No 10

How to derive a recurrence relation?

To derive a recurrence relation for the running time of an algorithm:

1) Figure out what "n", the ***problem size***, is.
2) See what value of n is used as the ***base of the recursion***. It will usually be a single value (e.g. $n = 1$), but may be multiple values. Suppose it is n_0.
3) Figure out what $T(n_0)$ is. You can usually use "**some constant c**", but sometimes a specific number will be needed.
4) The general $T(n)$ is usually a ***sum of various choices of $T(m)$ (for the recursive calls), plus the sum of the other work done***. Usually the recursive calls will be solving a sub problems of the same size $f(n)$, giving a term "$a*T(f(n))$" in the recurrence relation.

ALGORITHM $BinRec(n)$

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

if $n = 1$ **return** 1

else return $BinRec(\lfloor n/2 \rfloor) + 1$

T(1) = *d*

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

Lecture No 13

How to solve Recurrence Relation

Learning Outcomes

After completing this lecture you will be able

- To solve various types of recurrence relation using:
  - Iteration Method
  - Recursion Tree Method
  - Substitution Method
  - Master Theorem

Recap

Lecture No 9

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One:**
  - $T(n) = T(n-1) + f(n)$
- **Decrease-by-a-Constant-Factor:**
  - $T(n) = T(n/b) + f(n)$,
- **Divide-and-Conquer**
  - $T(n) = aT(n/b) + f(n)$

Formulae

- *The sum of finite arithmetic series can be found by using following formula*

$$S_n = \frac{n}{2}(a_1 + a_n)$$

- *The sum of finite Geometric series can be found by using following formula*

$$S_n = \frac{a_1(r^n - 1)}{r - 1}$$

- *The sum of infinite Geometric series can be found by using following formula*

$$S = \frac{a_1}{1 - r}$$

- *Special Cases of Geometric Series and Arithmetic series:*

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

- *Geometric Series:*

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

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

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

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

Formulae

- *Harmonic Series:* $$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \ln n$$

- *Others:*

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

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

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

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

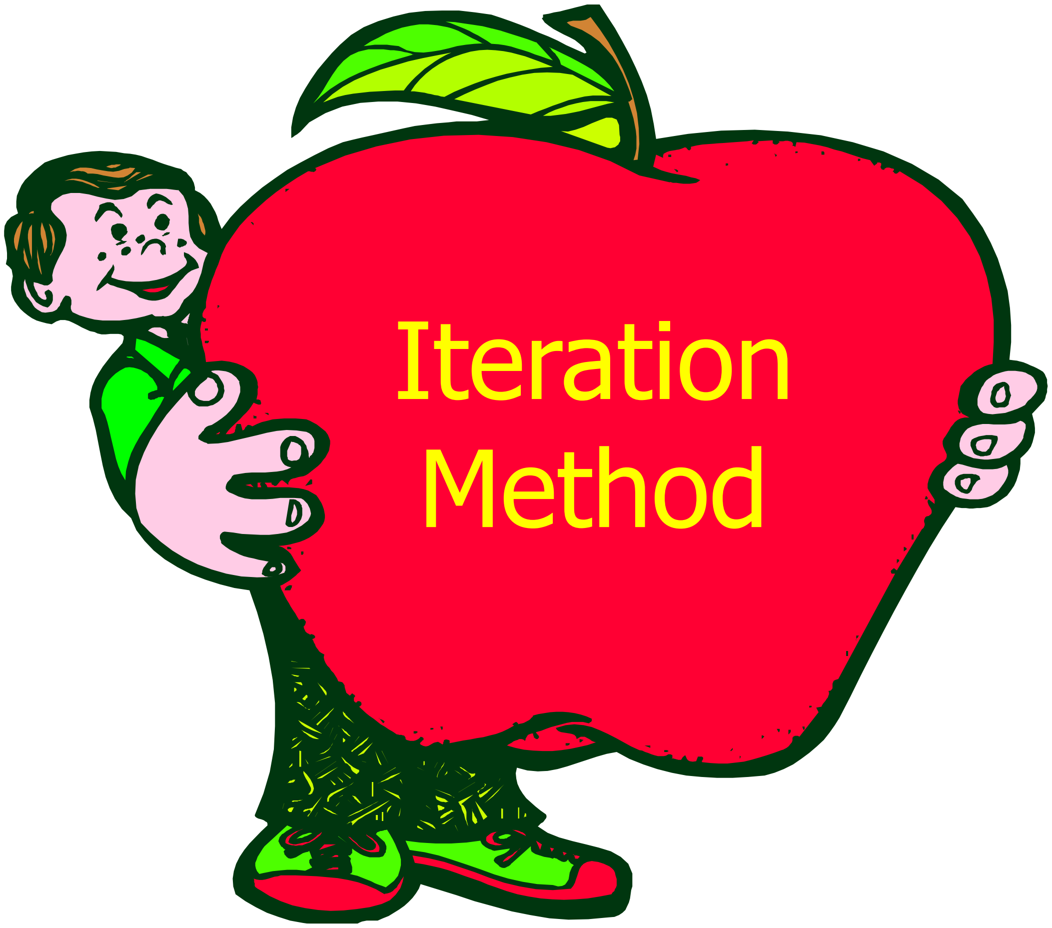
$$\sum_{k=0}^{n-1} \frac{1}{2^k} = 2 - \frac{1}{2^{n-1}}$$

SOLVING RECURRENCE RELATION

Solving Recurrence Relation

- There are four methods for solving a recurrence relation
  - **Iteration Method**
  - **Recursion Tree**
  - **Substitution method for recurrence relations**
  - **The Master Theorem**

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.

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

Solve the following

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ T(n-1) + d & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(n) &= T(n-1) + d \\ T(n-1) &= T(n-2) + d \\ T(n-2) &= T(n-3) + d \\ &\vdots \\ T(2) &= T(1) + d \\ T(1) &= c \end{aligned}$$

Repeated Substitution

$$\begin{aligned} T(n) &= T(n-1) + d \\ &= (T(n-2) + d) + d \\ &= T(n-2) + 2d \\ &= (T(n-3) + d) + 2d \\ &= T(n-3) + 3d \end{aligned}$$

There is a pattern developing. It looks like after i substitutions.

$$T(n) = T(n-i) + id.$$

Now choose $i = n - 1$. Then

$$\begin{aligned} T(n) &= T(1) + d(n-1) \\ &= dn + c - d. \end{aligned}$$

Example -2

$n! = n*(n-1)!$

$0! = 1$

Recurrence relation:

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

$T(1) = 1$

Telescoping:

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

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

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

...

$T(2) \quad = T(1\ ) + 1$

Add the equations and cross equal terms on opposite sides:

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

  $= n$

T (n) = T (n – 1) + f (n)

$$
\begin{aligned}
T(n) &= T(n-1) + f(n) \\
&= T(n-2) + f(n-1) + f(n) \\
&= \cdots \\
&= T(0) + \sum_{j=1}^{n} f(j).
\end{aligned}
$$

For a specific function $f(x)$, the sum $\sum_{j=1}^{n} f(j)$ can usually be either computed exactly or its order of growth ascertained. For example, if $f(n) = 1$, $\sum_{j=1}^{n} f(j) = n$; if $f(n) = \log n$, $\sum_{j=1}^{n} f(j) \in \Theta(n \log n)$; if $f(n) = n^k$, $\sum_{j=1}^{n} f(j) \in \Theta(n^{k+1})$.

Example

ALGORITHM $F(n)$

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

- We are dealing here with two recursively defined functions. The first is the factorial function F(n) itself; it is defined by the recurrence

n! = n*(n-1)!

0! = 1

- The second is the number of multiplications *M(n)* needed to compute *F(n)* by the recursive algorithm whose pseudocode is given

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

Recurrence relation:

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

T(1) = 1

Up Shot

Given a recurrence relation $T(n)$.

- Substitute a few times until you see a pattern
- Write a formula in terms of n and the number of substitutions i.
- Choose i so that all references to $T()$ become references to the base case.
- Solve the resulting summation

This will not always work, but works most of the time in practice.

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

$$\boldsymbol{T(n) = c + T(n/2)}$$

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

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

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

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

$$T(n/4) = c + T(n/8)$$

$$Assume\ n = 2^k$$

$$T(n) = \underbrace{c + c + \ldots + c}_{k\ times} + T(1)$$

$$= clgn + T(1)$$

$$= \Theta(lgn)$$

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

$$\begin{aligned} T(b^k) &= T(b^{k-1}) + f(b^k) \\ &= T(b^{k-2}) + f(b^{k-1}) + f(b^k) \\ &= \cdots \\ &= T(1) + \sum_{j=1}^{k} f(b^j). \end{aligned}$$

For a specific function $f(x)$, the sum $\sum_{j=1}^{k} f(b^j)$ can usually be either computed exactly or its order of growth ascertained. For example, if $f(n) = 1$,

$$\sum_{j=1}^{k} f(b^j) = k = \log_b n.$$

If $f(n) = n$, to give another example,

$$\sum_{j=1}^{k} f(b^j) = \sum_{j=1}^{k} b^j = b\frac{b^k - 1}{b - 1} = b\frac{n - 1}{b - 1}.$$

Example

- **Decrease-by-a-Constant-Factor T (n) = T (n/b) + f (n),**

ALGORITHM Binarysearch(A, *l*, *r*, key)
if(*l* > *r*) **then**
return -1
m (*l* + *r*)/2
if (key = A[*m*]) **then**
return m
if (key < A[*m*]) **then**
return Binarysearch (A , *l*, *m*-1 ,key)
else
return Binarysearch (A, *m*+1,*r*,key)

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

Upshot

- 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.

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

$$\mathbf{T(n) = n + 2T(n/2)}$$

Assume: $n = 2^k$

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

$$\begin{aligned}
T(n) &= n + 2T(n/2) \\
&= n + 2(n/2 + 2T(n/4)) \\
&= n + n + 4T(n/4) \\
&= n + n + 4(n/4 + 2T(n/8)) \\
&= n + n + n + 8T(n/8) \\
&\quad \ldots \\
&= in + 2^iT(n/2^i) \\
&= kn + 2^kT(1) \\
&= n\lg n + nT(1) = \Theta(n\lg n)
\end{aligned}$$

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

$$\begin{aligned}
T(b^k) &= aT(b^{k-1}) + f(b^k) \\
&= a[aT(b^{k-2}) + f(b^{k-1})] + f(b^k) = a^2T(b^{k-2}) + af(b^{k-1}) + f(b^k) \\
&= a^2[aT(b^{k-3}) + f(b^{k-2})] + af(b^{k-1}) + f(b^k) \\
&= a^3T(b^{k-3}) + a^2f(b^{k-2}) + af(b^{k-1}) + f(b^k) \\
&= \cdots \\
&= a^kT(1) + a^{k-1}f(b^1) + a^{k-2}f(b^2) + \cdots + a^0f(b^k) \\
&= a^k[T(1) + \sum_{j=1}^{k} f(b^j)/a^j].
\end{aligned}$$

Since $a^k = a^{\log_b n} = n^{\log_b a}$, we get the following formula for the solution to recurrence (B.14) for $n = b^k$:

$$T(n) = n^{\log_b a}[T(1) + \sum_{j=1}^{\log_b n} f(b^j)/a^j].$$

Common Recurrence Types in Algorithm Analysis

- **Divide-and-Conquer $T(n) = aT(n/b) + f(n)$**

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
    if p < r
       then q ← ⌊(p+r)/2⌋
            MergeSort (A, p, q)
            MergeSort (A, q+1, r)
            Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

```
Merge(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
    for i ← 1 to n1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
           then A[k] ← L[i]
                i ← i + 1
           else A[k] ← R[j]
```

Floor and ceilings are a pain to deal with. If n is assumed to be a power of 2 ($n = 2^k$), this will simplify the recurrence to:

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

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

Analysis: Substitution Method

$$T(n) = 2.T(\frac{n}{2}) + n$$

$$T(\frac{n}{2}) = 2.T(\frac{n}{2^2}) + \frac{n}{2}$$

$$T(\frac{n}{2^2}) = 2.T(\frac{n}{2^3}) + \frac{n}{2^2}$$

$$T(\frac{n}{2^3}) = 2.T(\frac{n}{2^4}) + \frac{n}{2^3} \quad \ldots$$

$$T(\frac{n}{2^{k-1}}) = 2.T(\frac{n}{2^k}) + \frac{n}{2^{k-1}}$$

Analysis of Merge-sort Algorithm

$$T(n) = 2.T(\frac{n}{2}) + \Theta(n) = 2^2.T(\frac{n}{2^2}) + n + n$$

$$T(n) = 2^2.T(\frac{n}{2^2}) + n + n$$

$$T(n) = 2^3.T(\frac{n}{2^3}) + n + n + n$$

$$\cdots$$

$$T(n) = 2^k.T(\frac{n}{2^k}) + \underbrace{n + n + \ldots + \mathrm{n}}_{k-times}$$

Analysis of Merge-sort Algorithm

$$T(n) = 2^k.T(\frac{n}{2^k}) + \underbrace{n + n + \ldots + \mathrm{n}}_{k-times}$$

$$T(n) = 2^k.T(\frac{n}{2^k}) + k.n$$

$$\text{Let us suppose that}: n = 2^k \Rightarrow \log_2 n = k$$

$$\text{Hence}, T(n) = n.T(1) + n.\log_2 n = n + n.\log_2 n$$

$$T(n) = \Theta(n.\log_2 n)$$

Important Recurrence Types

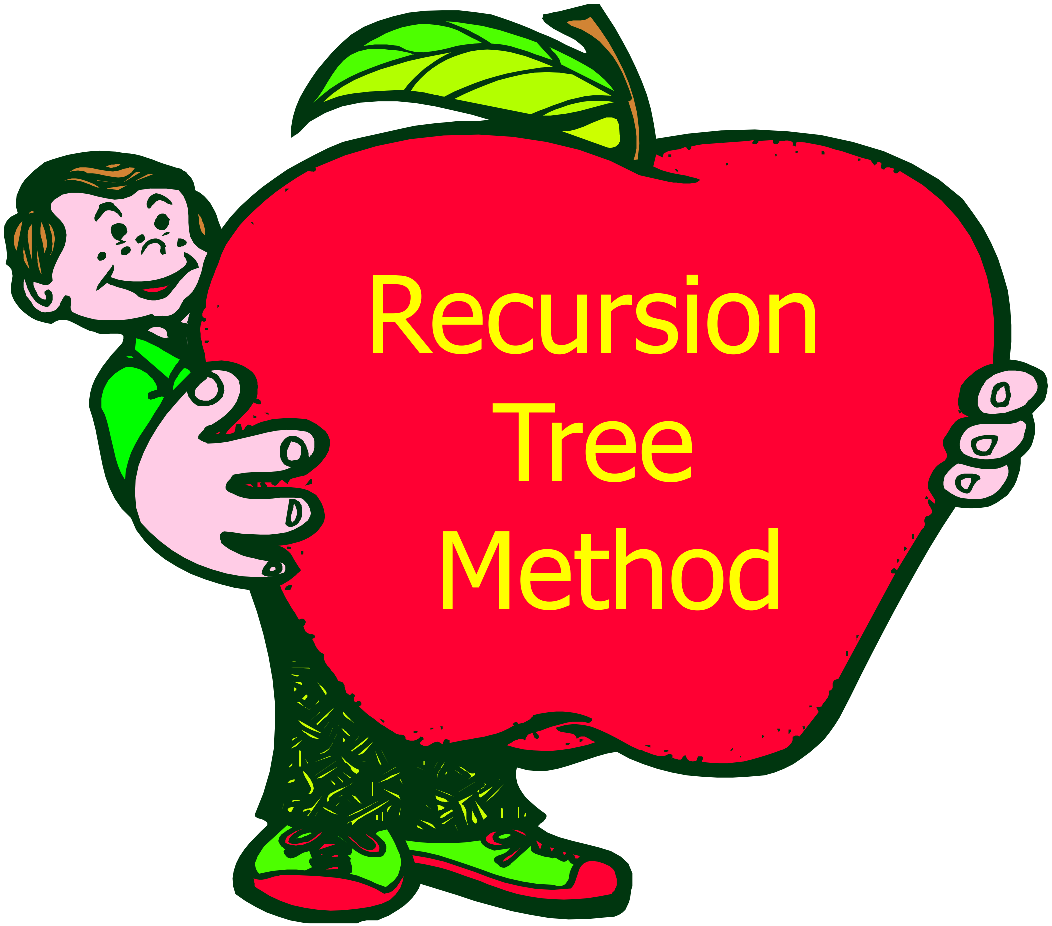
- One (constant) operation reduces problem size by one.
  $T(n) = T(n-1) + c$ $T(1) = d$
  Solution: $T(n) = (n-1)c + d$ ***linear***

- A pass through input reduces problem size by one.
  $T(n) = T(n-1) + cn$ $T(1) = d$
  Solution: $T(n) = [n(n+1)/2 - 1]\, c + d$ ***quadratic***

- One (constant) operation reduces problem size by half.
  $T(n) = T(n/2) + c$ $T(1) = d$
  Solution: $T(n) = c \log n + d$ ***logarithmic***

- A pass through input reduces problem size by half.
  $T(n) = 2T(n/2) + cn$ $T(1) = d$
  Solution: $T(n) = cn \log n + d\, n$ ***n log n***

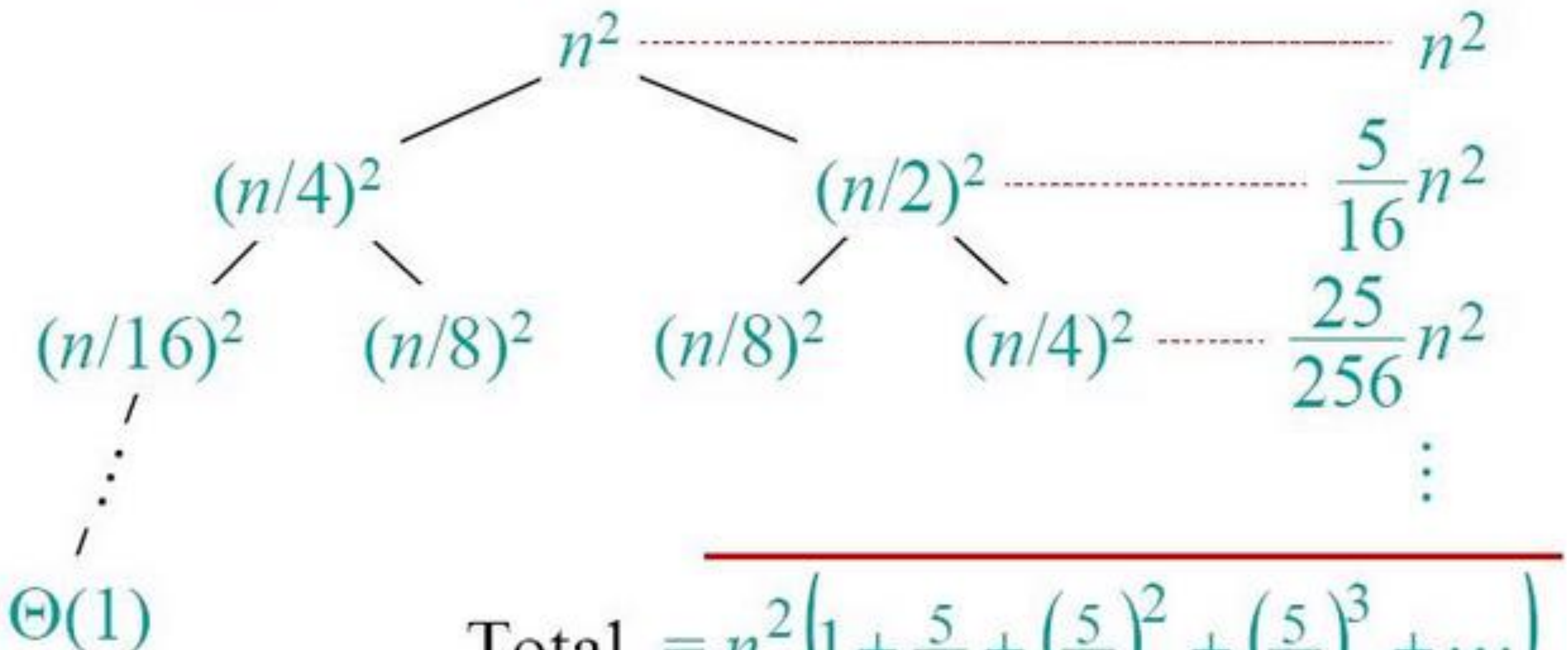
Recursion Tree Method



Recursion Tree Method

- Expanding the recurrence into a tree
  - In a recursion tree, each node represents the cost of a **single sub-problem** somewhere in the set of recursive function invocations.
- Summing the cost at each level
  - We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

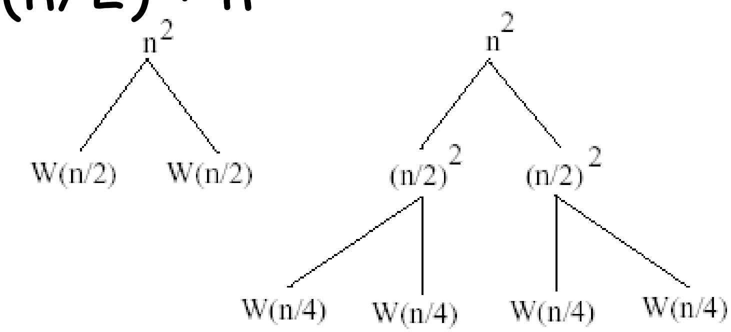
Solve $T(n) = T(n/4) + T(n/2) + n^2$:



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

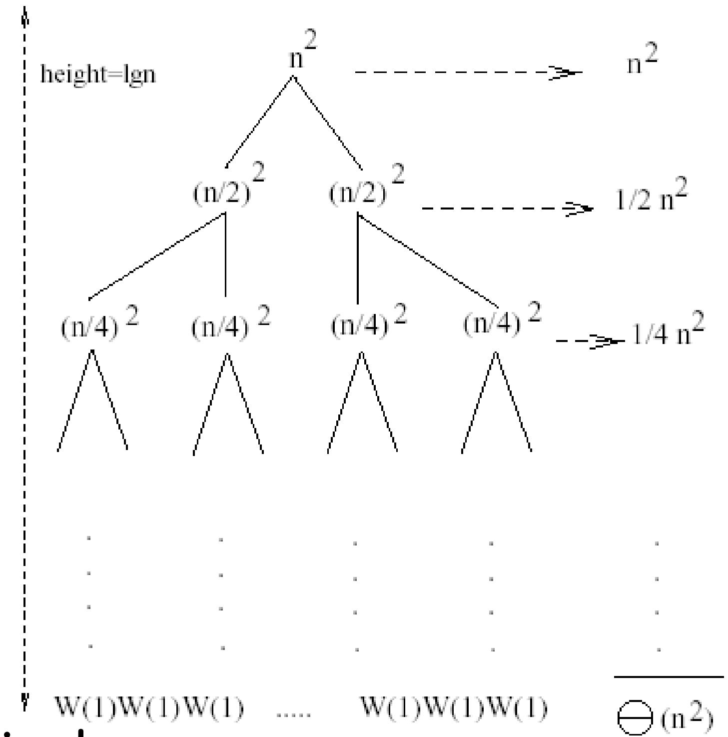
$$= \Theta(n^2) \quad \textit{geometric series}$$

$W(n) = 2W(n/2) + n^2$



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

$W(n/4)=2W(n/8)+(n/4)^2$

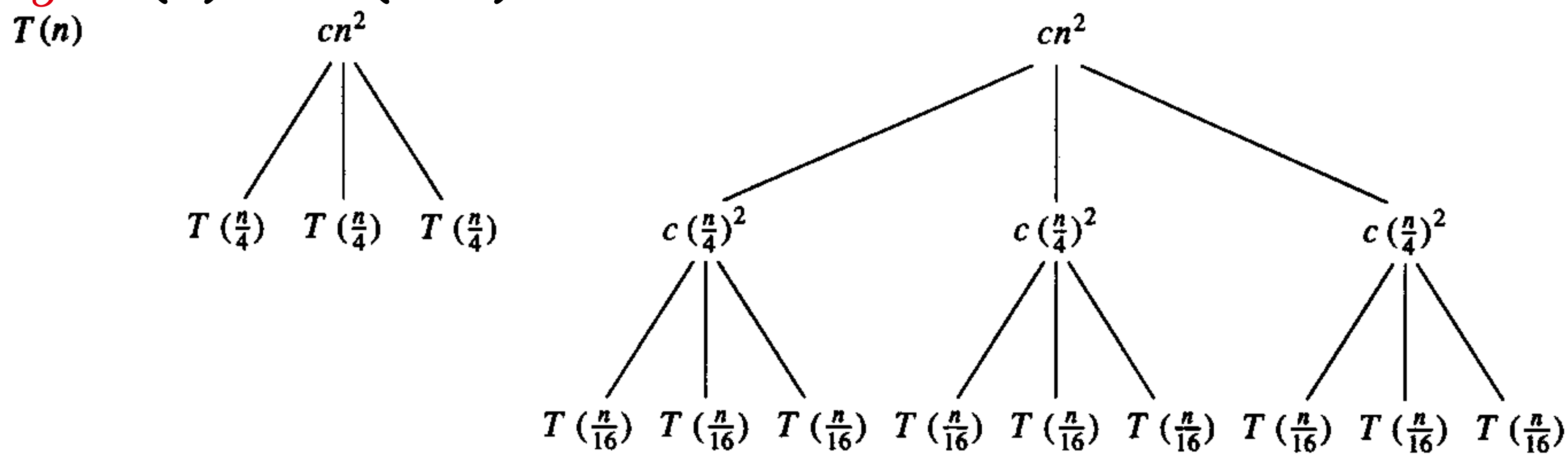


- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$
- Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level i = 2^i
- Total cost:

$$W(n) = \sum_{i=0}^{\lg n-1} \frac{n^2}{2^i} + 2^{\lg n} W(1) = n^2 \sum_{i=0}^{\lg n-1} \left(\frac{1}{2}\right)^i + n \le n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i + O(n) = n^2 \frac{1}{1-\frac{1}{2}} + O(n) = 2n^2$$

$\Rightarrow W(n) = O(n^2)$

E.g.: $T(n) = 3T(n/4) + cn^2$



- Subproblem size at level i is: $n/4^i$
- Subproblem size hits 1 when $1 = n/4^i \Rightarrow i = \log_4 n$
- Cost of a node at level $i = c(n/4^i)^2$
- Number of nodes at level $i = 3^i \Rightarrow$ last level has $3^{\log_4 n} = n^{\log_4 3}$ nodes
- Total cost:

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

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

Upshot

Convert the recurrence into a tree:

- Each node represents the cost incurred at various levels of recursion
- Sum up the costs of all levels

Used to "guess" a solution for the recurrence

The substitution method

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

Substitution method

- Guess a solution
  - $T(n) = O(g(n))$
  - Induction goal: apply the definition of the asymptotic notation
    - $T(n) \le d\, g(n)$, for some $d > 0$ and $n \ge n_0$
  - Induction hypothesis: $T(k) \le d\, g(k)$ for all $k < n$ (strong induction)
- Prove the induction goal
  - Use the **induction hypothesis** to find some values of the constants d and n_0 for which the **induction goal** holds

Example: Binary Search

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

- Guess: $T(n) = O(\lg n)$
  - Induction goal: $T(n) \le d \lg n$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le d \lg(n/2)$
- Proof of induction goal:

$$T(n) = T(n/2) + c \le d \lg(n/2) + c$$

$$= d \lg n - d + c \le d \lg n$$

if: $-d + c \le 0,\ d \ge c$

- Base case?

Example 2

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

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le c\,n^2$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n-1) \le c(n-1)^2$ for all $k < n$
- Proof of induction goal:

$T(n) = T(n-1) + n \le c\,(n-1)^2 + n$

$= cn^2 - (2cn - c - n) \le cn^2$

if: $2cn - c - n \ge 0 \Leftrightarrow c \ge n/(2n-1) \Leftrightarrow c \ge 1/(2 - 1/n)$

  - For $n \ge 1 \Rightarrow 2 - 1/n \ge 1 \Rightarrow$ any $c \ge 1$ will work

Example 3

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

- Guess: $T(n) = O(n\lg n)$
  - Induction goal: $T(n) \le cn \lg n$, for some c and $n \ge n_0$
  - Induction hypothesis: $T(n/2) \le cn/2 \lg(n/2)$
- Proof of induction goal:

$$T(n) = 2T(n/2) + n \le 2c\,(n/2)\lg(n/2) + n$$

$$= cn \lg n - cn + n \le cn \lg n$$

$$\text{if: } -cn + n \le 0 \Rightarrow c \ge 1$$

- Base case?

Changing variables

$$T(n) = 2T(\sqrt{n}\ ) + \lg n$$

- Rename: $m = \lg n \Rightarrow n = 2^m$

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

- Rename: $S(m) = T(2^m)$

$S(m) = 2S(m/2) + m \Rightarrow S(m) = O(m\lg m)$ (demonstrated before)

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

Idea: transform the recurrence to one that you have seen before

Example 2 - Substitution

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

- Guess: $T(n) = O(n^2)$
  - Induction goal: $T(n) \le dn^2$, for some d and $n \ge n_0$
  - Induction hypothesis: $T(n/4) \le d\,(n/4)^2$
- Proof of induction goal:

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

$\le 3d\,(n/4)^2 + cn^2$

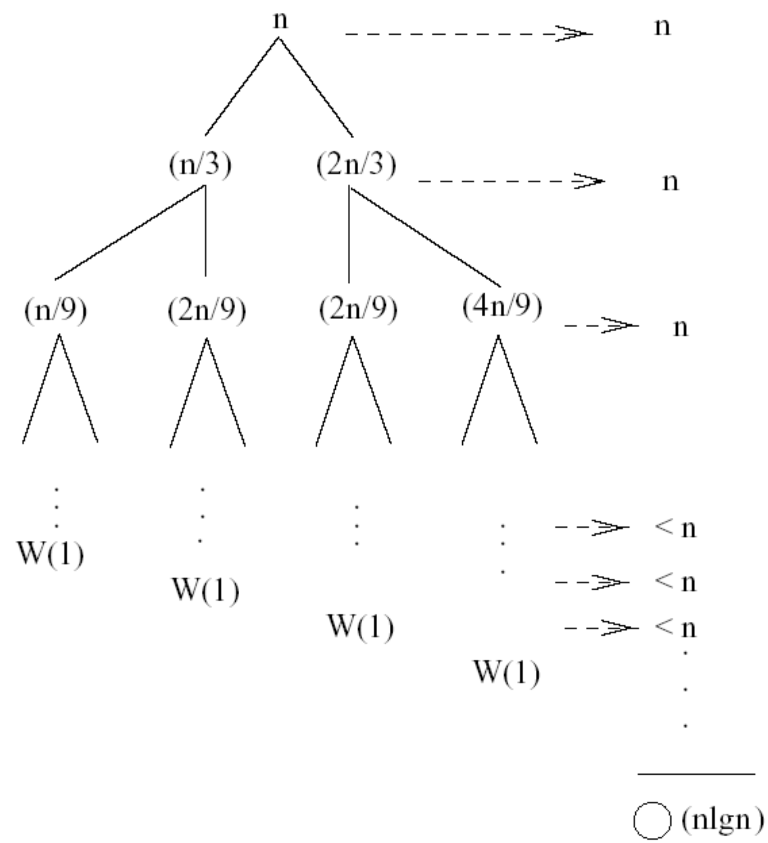
$= (3/16)\,d\,n^2 + cn^2$

$\le d\,n^2$ if: $d \ge (16/13)c$

- Therefore: $T(n) = O(n^2)$

Example 3 (simpler proof)

$W(n) = W(n/3) + W(2n/3) + n$



- The longest path from the root to a leaf is:

  $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \ldots \rightarrow 1$

- Subproblem size hits 1 when

  $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$

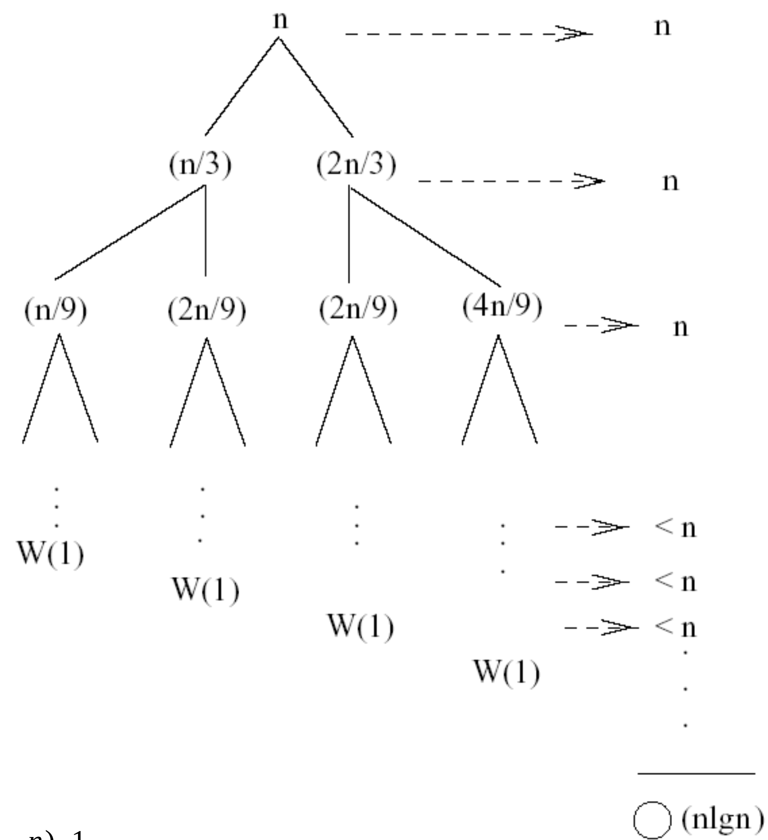
- Cost of the problem at level i = n

- Total cost:

$$W(n) < n + n + \ldots = n(\log_{3/2} n) = n \frac{\lg n}{\lg \frac{3}{2}} = O(n \lg n)$$

Example 3

$W(n) = W(n/3) + W(2n/3) + n$



- The longest path from the root to a leaf is:

  $n \rightarrow (2/3)n \rightarrow (2/3)^2 n \rightarrow \ldots \rightarrow 1$

- Subproblem size hits 1 when

  $1 = (2/3)^i n \Leftrightarrow i = \log_{3/2} n$

- Cost of the problem at level i = n

- Total cost: $W(n) < n + n + \ldots = \sum_{i=0}^{(\log_{3/2} n)-1} n + 2^{(\log_{3/2} n)} W(1) <$

$$< n \sum_{i=0}^{\log_{3/2} n} 1 + n^{\log_{3/2} 2} = n \log_{3/2} n + O(n) = n \frac{\lg n}{\lg 3/2} + O(n) = \frac{1}{\lg 3/2} n \lg n + O(n)$$

General Recurrence Relation of Divide and Conq.

Base of recursion

Running time for base

$$T(n) = \begin{cases} c & \text{if } n = n_0 \\ a.T(f(n)) + g(n) & \text{otherwise} \end{cases}$$

Number of times recursive call is made

All other processing not counting recursive calls

Size of problem solved by recursive call

Master Theorem: A general divide-and-conquer recurrence

$T(n) = aT(n/b) + g(n)$ where $g(n) \in \Theta(n^k)$

$a < b^k$	$T(n) \in \Theta(n^k)$
$a = b^k$	$T(n) \in \Theta(n^k \log n)$
$a > b^k$	$T(n) \in \Theta(n$ to the power of $(\log_b a))$

Note: the same results hold with O instead of Θ.

Div & Conq. (contd.)

- $T(n) = aT(n/b)+g(n)$, $a \geq 1$, $b > 1$
- Master Theorem:

  If $g(n) \in \Theta(n^d)$ where $d \geq 0$ then

What if a = 1?
Have we seen it?

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So, $A(n) \in \Theta(n^{\log_2 2})$
Or, $A(n) \in \Theta(n)$

Without going through back-subs. we got it, but not quite…

For adding n numbers with divide and conquer technique, the number of additions A(n) is:
$A(n) = 2A(n/2)+1$

Here, $a = ?$, $b = ?$, $d = ?$ $a = 2$, $b = 2$, $d = 0$

Which of the 3 cases holds ? $a = 2 > b^d = 2^0$, case 3

Div. & Conq. (contd.)

$T(n) = aT(n/b)+f(n),\ a \geq 1,\ b > 1$

If $f(n) \in \Theta(n^d)$ where $d \geq 0$, then

$$T(n) \in \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \lg n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

$T(n) = 2T(n/2)+6n-1$?

$T(n) = 3\,T(n/2) + n$ $\quad$ $a = 3,\ b = 2,\ f(n) \in \Theta(n^1)$, so $d = 1$

$a=3 > b^d=2^1$ $\quad$ Case 3: $T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.5850})$

$T(n) = 3\,T(n/2) + n^2$ $\quad$ $a = 3,\ b = 2,\ f(n) \in \Theta(n^2)$, so $d = 2$

$a=3 < b^d=2^2$ $\quad$ Case 1: $T(n) \in \Theta(n^2)$

$T(n) = 4\,T(n/2) + n^2$ $\quad$ $a = 4,\ b = 2,\ f(n) \in \Theta(n^2)$, so $d = 2$

$a=4 = b^d=2^2$ $\quad$ Case 2: $T(n) \in \Theta(n^2 \lg n)$

$T(n) = 0.5\,T(n/2) + 1/n$ $\leftarrow$ Master th^m doesn't apply, $a<1$, $d<0$

$T(n) = 2\,T(n/2) + n/\lg n$ $\leftarrow$ Master th^m doesn't apply $f(n)$ not polynomial

$T(n) = 64\,T(n/8) - n^2 \lg n$ $f(n)$ is not positive, doesn't apply

$T(n) = 2^n\,T(n/8) + n$ a is not constant, doesn't apply

Important Recurrence Types

Recurrence	Algorithm	Big-Oh Solution
$T(n) = T(n/2) + O(1)$	Binary Search	$O(\log n)$
$T(n) = T(n-1) + O(1)$	Sequential Search	$O(n)$
$T(n) = 2\,T(n/2) + O(1)$	Tree Traversal	$O(n)$
$T(n) = T(n-1) + O(n)$	Selection Sort (other n^2 sorts)	$O(n^2)$
$T(n) = 2\,T(n/2) + O(n)$	Mergesort (average case Quicksort)	$O(n \log n)$