بسم الله الرحمن الرحيم

Design and Analysis of Algorithm

Department of Computer Science
COMSATS Institute of Information Technology, Islamabad

Tanveer Ahmed Siddiqui

Recap

Lecture No 10

Mathematical Analysis of non recursive Algorithm

Steps in mathematical analysis of non recursive algorithms

- Decide on parameter *n* indicating input size
- Identify algorithm's basic operation
- Determine worst, average, and best case for input of size *n*
- Set up summation for *C(n)* reflecting algorithm's loop structure
- Simplify summation using standard formulas

- Which algorithm is best?

ALGORITHM factorial(n)
// Input: A positive integer
// Output: The factorial of the positive integer n.
factorial ← 1 // initialize answer
i ← 1
while(i ≤ n) **do**
  factorial ← factorial * i
  i ← i + 1
return factorial

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

Lecture No 11

Analysis of Recursive Algorithm

(Analysis Framework.)

After completing this lecture you will be able

- To solve problem with recursive algorithm
- To convert an iterative algorithm into recursive algorithm.
- To compute Time complexity of recursive algorithm
- To compare iterative version with recursive version

What is Recursion?

- Dictionary definition:
- A problem-solving method of
  "decomposing bigger problems into smaller sub-problems that *are identical to itself.*"
- Recursion:
  - Process of solving a problem by reducing it to smaller versions of itself
- General Idea:
  - Solve simplest (smallest) cases DIRECTLY
    - usually these are very easy to solve
  - Solve bigger problems using smaller sub-problems
    - that are identical to itself (but smaller and simpler)
- Abstraction:
  - To solve a given problem, we first assume that we ALREADY know how to solve it for smaller instances!!

- Recursive algorithm:
  - Algorithm that finds the solution to a given problem by reducing the problem to smaller versions of itself.
  - Has one or more base cases.
  - Implemented using recursive methods.
- Recursive method:
  - Method that calls itself.
- Base case:
  - Case in recursive definition in which the solution is obtained directly.
  - Stops the recursion.
- General case:
  - Case in recursive definition in which a smaller version of itself is called.
  - Must eventually be reduced to a base case.

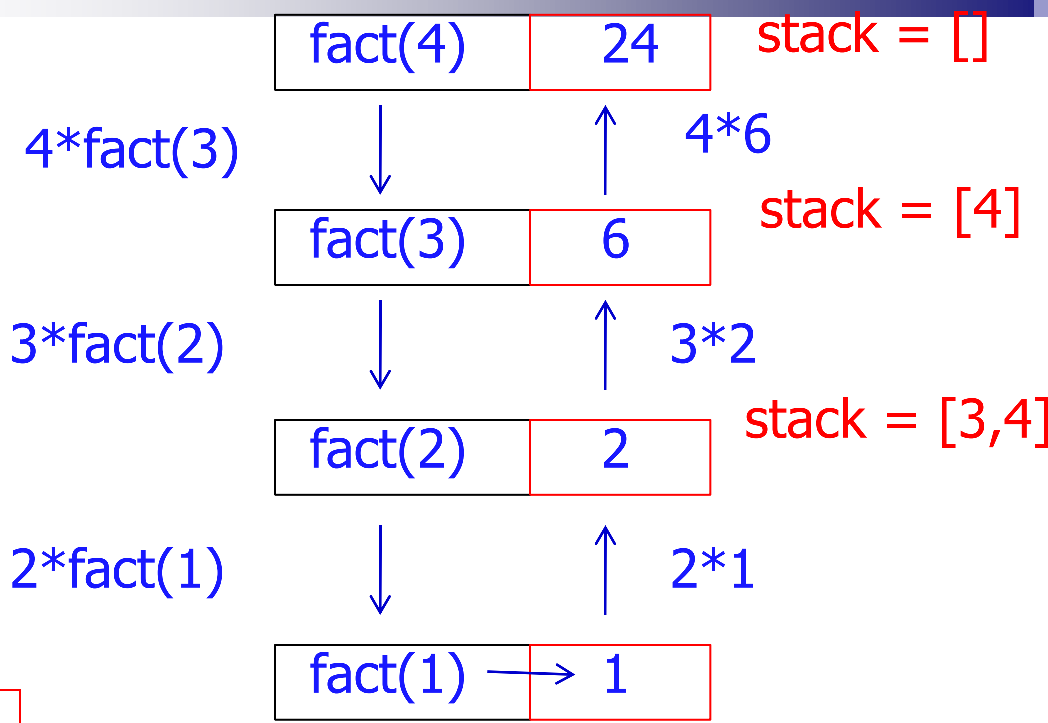
- Understand problem requirements.
- Determine limiting conditions.
- Identify base cases.
- Provide direct solution to each base case.
- Identify general cases.
- Provide solutions to general cases in terms of smaller versions of general cases.

fact(4): stack = [4]

fact(3): stack = [3,4]

fact(2): stack = [2,3,4]

fact(1): stack = [1,2,3,4]




```
fact(n)
   If n<=1 then return←1
            else return ←fact(n-1)*n
   endif
return return
```

Sequence of recursive calls

Back to the calling function

- Definition of Fibonacci numbers
  1. $F_1 = 1$,
  2. $F_2 = 1$,
  3. for $n>2$, $F_n = F_{n-1} + F_{n-2}$
- Problem: Compute F_n for any n.
- The above is a recursive definition.
  - F_n is computed in-terms of itself
  - actually, smaller copies of itself – F_{n-1} and F_{n-2}
- Actually, Not difficult:

$F_3 = 1 + 1 = 2$ $F_6 = 5 + 3 = 8$ $F_9 = 21 + 13 = 34$

$F_4 = 2 + 1 = 3$ $F_7 = 8 + 5 = 13$ $F_{10} = 34 + 21 = 55$

$F_5 = 3 + 2 = 5$ $F_8 = 13 + 8 = 21$ $F_{11} = 55 + 34 = 89$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

ALGORITHM $Fib(n)$

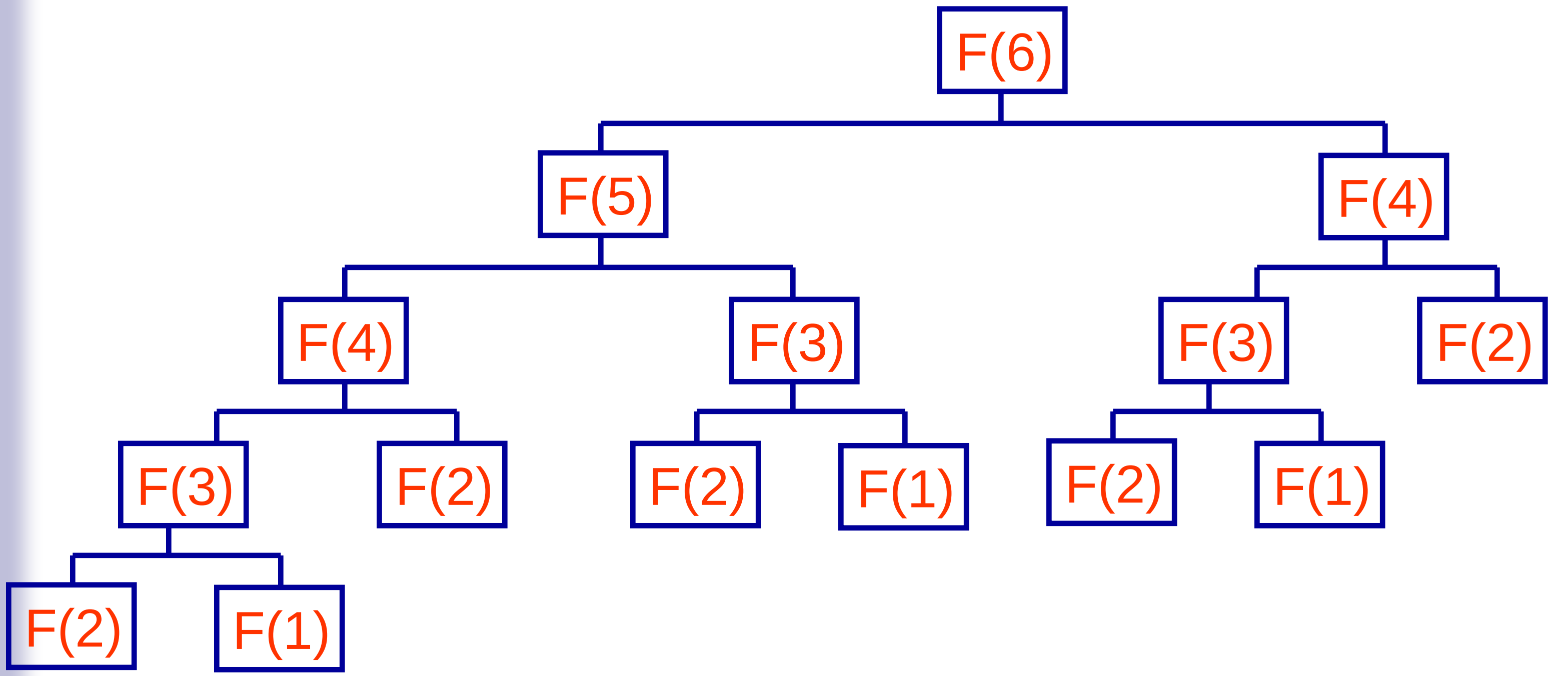
//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
$F[0] \leftarrow 0;\ F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

- The below is a recursive algorithm
- It is simple to understand and elegant!
- But, very SLOW(**WHY**)

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \leq 1$ **return** n
else return $F(n-1) + F(n-2)$

- How slow is it?
  - E.g. To compute F(6)…



HW: Can we compute it faster?

Decide on parameter n indicating input size

Identify algorithm's basic operation

Determine worst, average, and best case for input of size n

Set up a recurrence relation and initial condition(s)

Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution

Determine Order of growth of C(n)

- A recurrence relation is an equation which is defined in terms of itself.
- It expresses the value of a function for an argument *n* in terms of the values of function for arguments less than *n*.
- Examples:

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

2. $$T(n)=\begin{cases}1 & if\ n=1\\ 2T(n-1) & otherwise\end{cases}$$

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*

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 $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \le 1$ **return** n
else return $F(n-1) + F(n-2)$

that can be defined by the simple recurrence

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1$$

and two initial conditions

$$F(0) = 0, \qquad F(1) = 1.$$

Deriving Recurrence Relations

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.

```
procedure bugs(n)
  if n = 1 then do something
    else
      bugs(n − 1);
      bugs(n − 2);
      for i := 1 to n do
        something
```

$$T(n) = \left\{ \right.$$

Deriving Recurrence Relations

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.

```
procedure daffy(n)
  if n = 1 or n = 2 then do something
    else
      daffy(n − 1);
      for i := 1 to n do
        do something new
      daffy(n − 1);
```

$$T(n) = \begin{cases} \\ \\ \end{cases}$$

Deriving Recurrence Relations

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.

```
procedure elmer(n)
  if n = 1 then do something
    else if n = 2 then do something else
    else
      for i := 1 to n do
        elmer(n − 1);
        do something different
```

$$T(n) = \begin{cases} \\ \\ \end{cases}$$

Deriving Recurrence Relations

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.

```
procedure yosemite(n)
  if n = 1 then do something
    else
      for i := 1 to n − 1 do
        yosemite(i);
        do something completely different
```

$$T(n) = \begin{cases} \\ \\ \end{cases}$$

Deriving Recurrence Relations

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

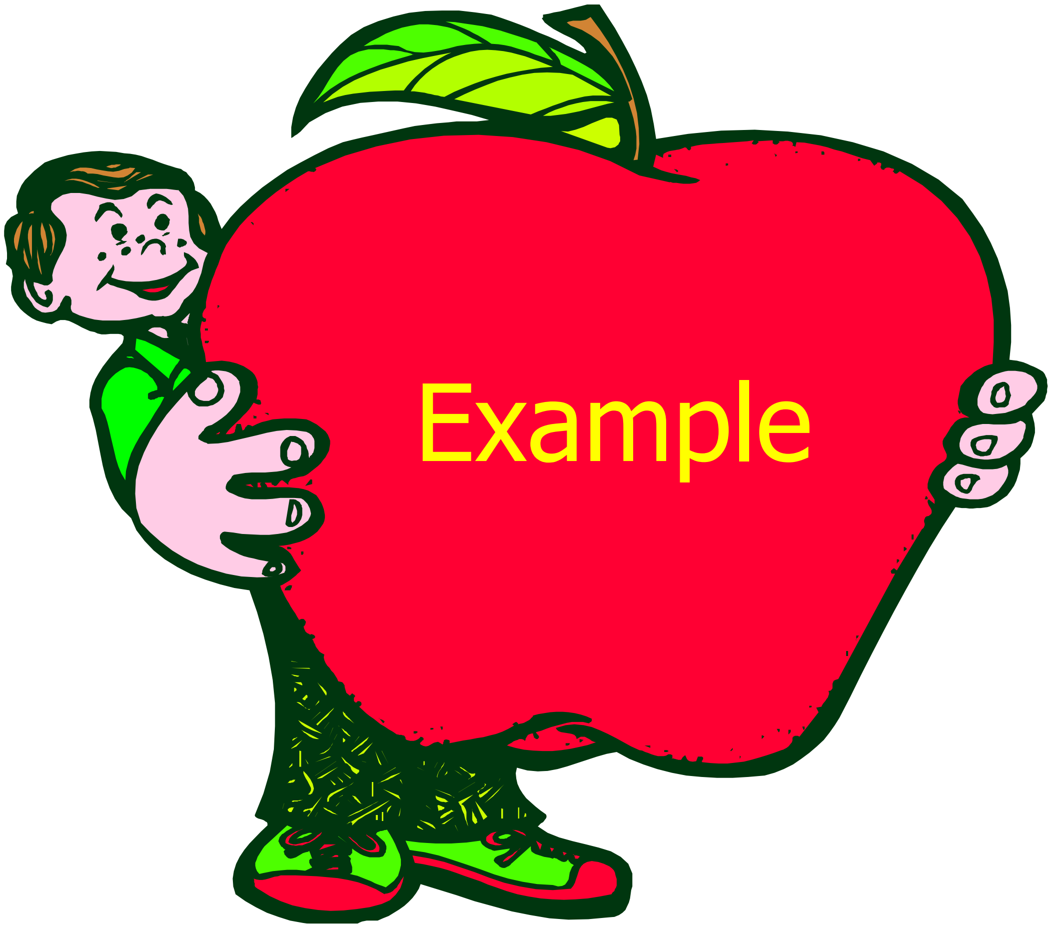
1) Figure out what "n", the ***problem size***, is.
2) See what value of n is used as the ***base of the recursion***. It will usually be a single value (e.g. $n = 1$), but may be multiple values. Suppose it is n_0.
3) Figure out what $T(n_0)$ is. You can usually use **"some constant c"**, but sometimes 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.

```
function multiply(y, z)
    comment return the product yz
1.  if z = 0 then return(0) else
2.  if z is odd
3.      then return(multiply(2y, ⌊z/2⌋)+y)
4.      else return(multiply(2y, ⌊z/2⌋))
```

D.Y.S = Do Your Self

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One** A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size *n* and a smaller instance of size *n* − 1. Specific examples include recursive evaluation of *n*! and insertion sort. The recurrence equation for investigating the time efficiency of such algorithms typically has the following form:
  - $T(n) = T(n-1) + f(n)$
- where function **f(n)** accounts for the time needed to reduce an instance to a smaller one and to extend the solution of the smaller instance to a solution of the larger instance.


Example

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One:** $T(n) = T(n-1) + f(n)$

```
ALGORITHM  factorial(n)
// Input: A positive integer
// Output: The factorial of the positive integer n.
factorial ← 1   // initialize answer
i ← 1
while( i ≤ n ) do
   factorial ← factorial * i
   i  ← i + 1
return  factorial
```

Convert iterative into recursive

n! = n*(n-1)!
0! = 1

ALGORITHM $F(n)$

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

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One: T (n) = T (n – 1) + f (n)**

ALGORITHM $F(n)$

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

- What is measure of an input's size?
  - The number of elements in the array, i.e., n.
- What is its basic operation/Primitive Operation?
  - Multiplication(*)

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One: T (n) = T (n – 1) + f (n)**

ALGORITHM $F(n)$

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

- The basic operation of the algorithm is multiplication, whose number of executions we denote M(n).

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

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One: T (n) = T (n – 1) + f (n)**

ALGORITHM $F(n)$

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

- Now we obtain initial condition by inspecting the condition that makes the algorithm stop its recursive calls:
  - **if** $n = 0$ **return** 1.

Common Recurrence Types in Algorithm Analysis

ALGORITHM $F(n)$

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

- **if** n = 0 **return** 1.

$M(0) = 0.$

the calls stop when $n = 0$ — no multiplications when $n = 0$

- This tells us two things.
  - First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence $M(n)$ defined is 0.
  - Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.

Common Recurrence Types in Algorithm Analysis

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
```

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

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

Common Recurrence Types in Algorithm Analysis

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

Recurrence relation:

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

T(1) = 1

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

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

YOUR TURN

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-One: T (n) = T (n – 1) + f (n)**

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in ascending order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Convert into recursive version

Recurrence relation?

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

ALGORITHM BubbleSort(A, *n*)
 for i ← 0 to n-2 **do**
  if A[i] > A[i+1] **then**
   swap A[i] and A[i+1]
 BubbleSort(A, *n-1*)

Common Recurrence Types in Algorithm Analysis

- **Decrease-by-a-Constant-Factor** A decrease-by-a-constant-factor algorithm solves a problem by reducing its instance of size *n* to an instance of size *n/b* (*b* = 2 for most but not all such algorithms), solving the smaller instance recursively, and then, if necessary, extending the solution of the smaller instance to a solution of the given instance. The most important example is binary search; other examples include exponentiation by squaring. The recurrence equation for investigating the time efficiency of such algorithms typically has the form

- **$T(n) = T(n/b) + f(n)$,**

Common Recurrence Types in Algorithm Analysis

- **$T(n) = T(n/b) + f(n)$,**

- Where $b > 1$ and function $f(n)$ accounts for the time needed to reduce an instance to a smaller one and to extend the solution of the smaller instance to a solution of the larger instance. Strictly speaking, equation is valid only for $n = b^k$, $k = 0, 1, \ldots$. For values of n that are not powers of b, there is typically some round off, usually involving the floor and/or ceiling functions.

Common Recurrence Types in Algorithm Analysis

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

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

//Implements nonrecursive binary search
//Input: An array $A[0..n-1]$ sorted in ascending order and
//   a search key K
//Output: An index of the array's element that is equal to K
//   or -1 if there is no such element
$l \leftarrow 0;\quad r \leftarrow n-1$
while $l \le r$ **do**
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$ **return** m
    else if $K < A[m]$ $r \leftarrow m-1$
    else $l \leftarrow m+1$
return -1

$$\underbrace{A[0]\ldots A[m-1]}_{\text{search here if } K<A[m]}\ \overset{\overset{K}{\updownarrow}}{A[m]}\ \underbrace{A[m+1]\ldots A[n-1]}_{\text{search here if } K>A[m]}.$$

- Let us apply binary search to searching for $K = 70$ in the array

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration 1	l						m						r
iteration 2								l		m			r
iteration 3								l,m	r				

Common Recurrence Types in Algorithm Analysis

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

ALGORITHM *BinarySearch*$(A[0..n-1], K)$

```
//Implements nonrecursive binary search
//Input: An array A[0..n − 1] sorted in ascending order and
//       a search key K
//Output: An index of the array's element that is equal to K
//       or −1 if there is no such element
l ← 0;   r ← n − 1
while l ≤ r do
    m ← ⌊(l + r)/2⌋
    if K = A[m] return m
    else if K < A[m] r ← m − 1
    else l ← m + 1
return −1
```

Convert this iterative version into recursive version

←

```
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)
```

Common Recurrence Types in Algorithm Analysis

- **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)
```

- The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array.
- For the sake of simplicity, we will count the so-called three-way comparisons.
- This assumes that after one comparison of *key* with *A*[*m*], the algorithm can determine whether *key* is smaller, equal to, or larger than *A*[*m*].

Common Recurrence Types in Algorithm Analysis

- **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)
```

- How many such comparisons does the algorithm make on an array of *n* elements?
- The answer obviously depends not only on *n* but also on the specifics of a particular instance of the problem.
- Let us find the number of key comparisons in the worst case $C_{worst}(n)$.

Common Recurrence Types in Algorithm Analysis

- **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)
```

- The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches.
- Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for $C_{worst}(n)$:

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

YOUR TURN

- Analyze the following algorithm

ALGORITHM *Bisection*($f(x), a, b, eps, N$)

//Implements the bisection method for finding a root of $f(x) = 0$
//Input: Two real numbers a and b, $a < b$,
// a continuous function $f(x)$ on $[a, b]$, $f(a)f(b) < 0$,
// an upper bound on the absolute error $eps > 0$,
// an upper bound on the number of iterations N
//Output: An approximate (or exact) value x of a root in (a, b)
//or an interval bracketing the root if the iteration number limit is reached
$n \leftarrow 1$ //iteration count
while $n \le N$ **do**
    $x \leftarrow (a + b)/2$
    if $x - a < eps$ **return** x
    $fval \leftarrow f(x)$
    if $fval = 0$ **return** x
    if $fval * f(a) < 0$
        $b \leftarrow x$
    else $a \leftarrow x$
    $n \leftarrow n + 1$
return "iteration limit", a, b

Common Recurrence Types in Algorithm Analysis

- **Divide-and-Conquer** A divide-and-conquer algorithm solves a problem by dividing its given instance into several smaller instances, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance. Assuming that all smaller instances have the same size **n/b**, with *a* of them being actually solved, we get the following recurrence valid for $n = b^k$, $k = 1, 2, \ldots$:
  - **$T(n) = aT(n/b) + f(n)$,**
- where $a \geq 1$, $b \geq 2$, and $f(n)$ is a function that accounts for the time spent on dividing the problem into smaller ones and combining their solutions. Recurrence (B.14) is called the ***general divide-and-conquer recurrence.***

Common Recurrence Types in Algorithm Analysis

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

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
  if $A[i] > maxval$
    $maxval \leftarrow A[i]$
return $maxval$

Convert into recursive version

The recurrence for the number of element comparisons is

$$C(n) = \begin{cases} 0 & \text{for } n = 1 \\ C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 & \text{for } n > 1, \end{cases}$$

Recurrence relation?

Algorithm $MaxIndex(A[l..r])$
//Input: A portion of array $A[0..n-1]$ between indices l and r $(l \leq r)$
//Output: The index of the largest element in $A[l..r]$
if $l = r$ return l
else $temp1 \leftarrow MaxIndex(A[l..\lfloor (l+r)/2 \rfloor])$
  $temp2 \leftarrow MaxIndex(A[\lfloor (l+r)/2 \rfloor + 1..r])$
  if $A[temp1] \geq A[temp2]$
    return $temp1$
  else return $temp2$

SOLVING RECURRENCE RELATION

Solving Recurrence Relation

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

Solving Recurrence Relations

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 – 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 $f(n) = n^k$, $\sum_{j=1}^{n} f(j) \in \Theta(n^{k+1})$.

Solving Recurrence Relations

```
function multiply(y, z)
    comment return the product yz
    if z = 0 then return(0) else
    if z is odd
        then return(multiply(2y, ⌊z/2⌋)+y)
        else return(multiply(2y, ⌊z/2⌋))
```

Let $T(n)$ be the running time of multiply(y, z), where z is an n-bit natural number.

Then for some $c, d \in \mathbb{R}$,

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

Therefore, for large enough n,

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

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

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

Warning

This is not a proof. There is a gap in the logic. Where did

$$\boxed{T(n) = T(n-i) + id}$$

come from? Hand-waving!

What would make it a proof? Either

- Prove that statement by induction on i, or
- Prove the result by induction on n.

- 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/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,

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

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

$$T(n/2) = c + T(n/4)$$
$$T(n/4) = c + T(n/8)$$

$$\begin{aligned} T(n) &= c + T(n/2) \\ &= c + c + T(n/4) \\ &= c + c + c + T(n/8) \end{aligned}$$

Assume $n = 2^k$

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

$$= c\lg n + T(1)$$
$$= \Theta(\lg n)$$

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 $n = b^k$:

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

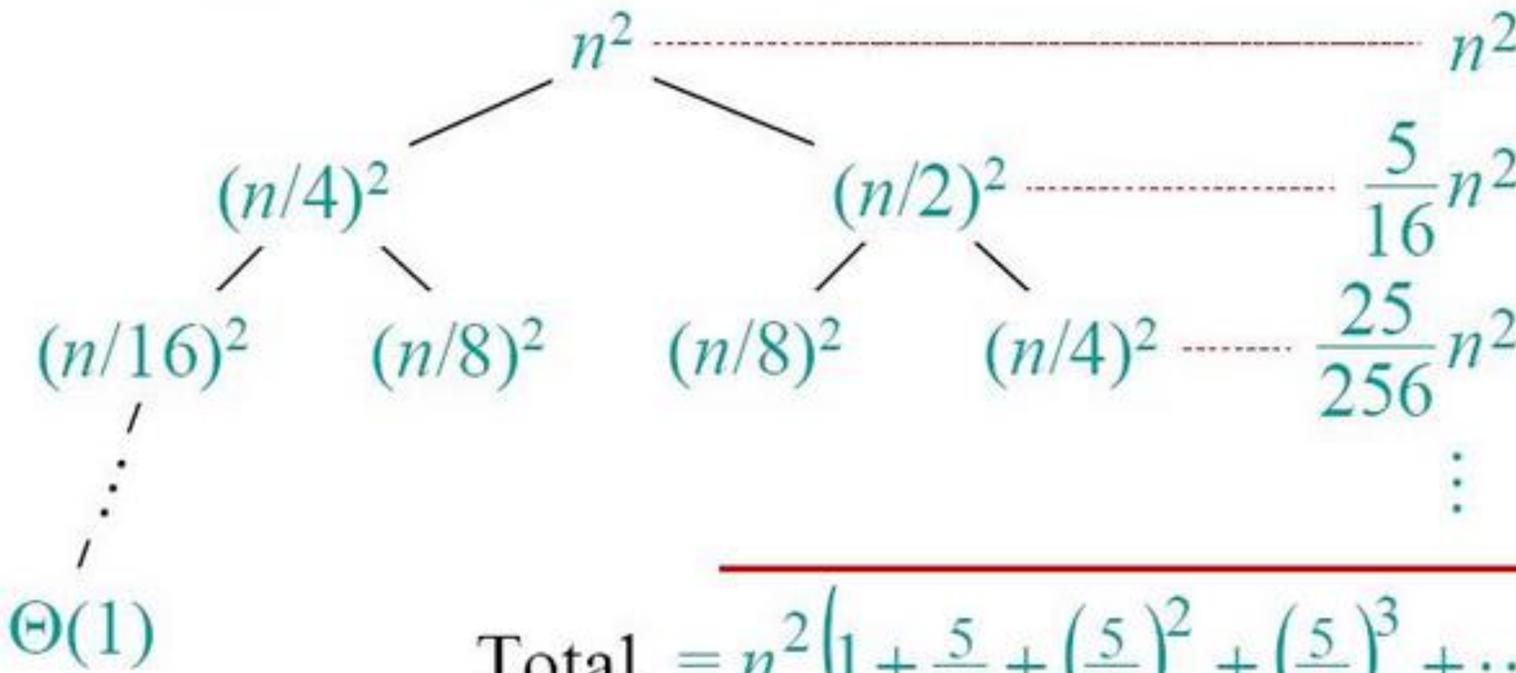
$$\mathbf{T(n) = n + 2T(n/2)} \qquad \text{Assume: } n = 2^k$$

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

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

- 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.

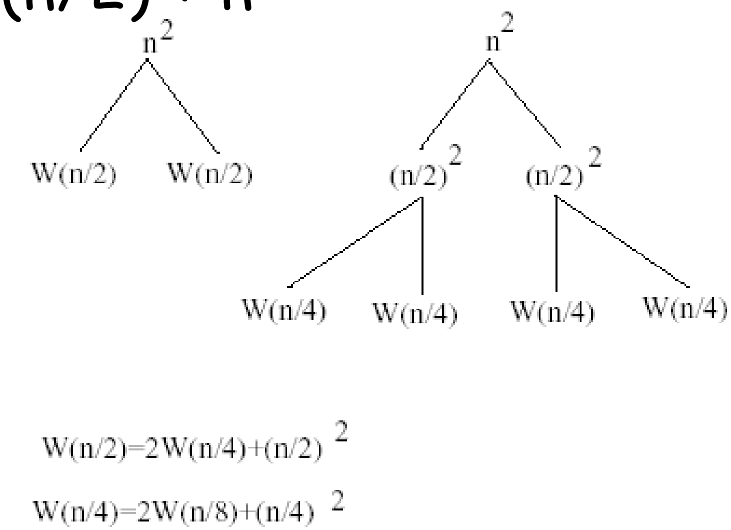
Solve $T(n) = T(n/4) + T(n/2) + n^2$:

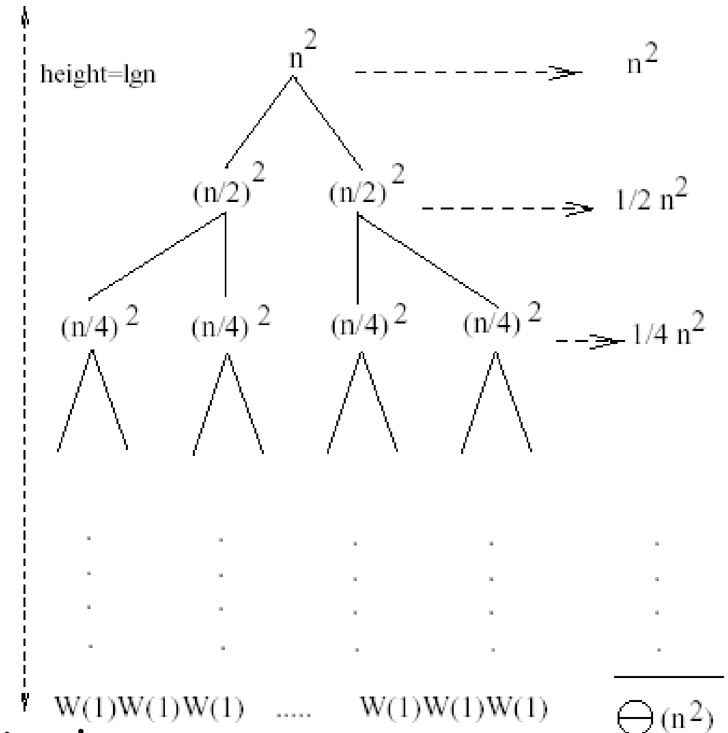


$$\text{Total} = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \left(\frac{5}{16}\right)^3 + \cdots\right)$$

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

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



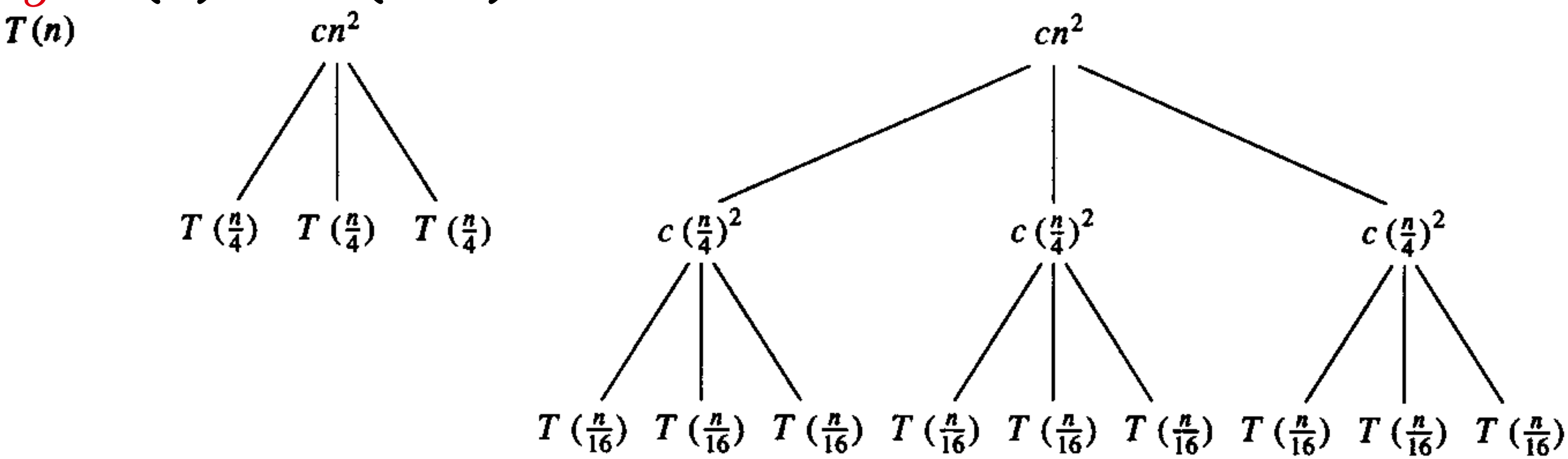


- Subproblem size at level i is: $n/2^i$
- Subproblem size hits 1 when $1 = n/2^i \Rightarrow i = \lg n$
- Cost of the problem at level i = $(n/2^i)^2$ No. of nodes at level i = 2^i
- Total cost:

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

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

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:

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

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

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:

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

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

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

$\le d\,n^2 \qquad$ 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:

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

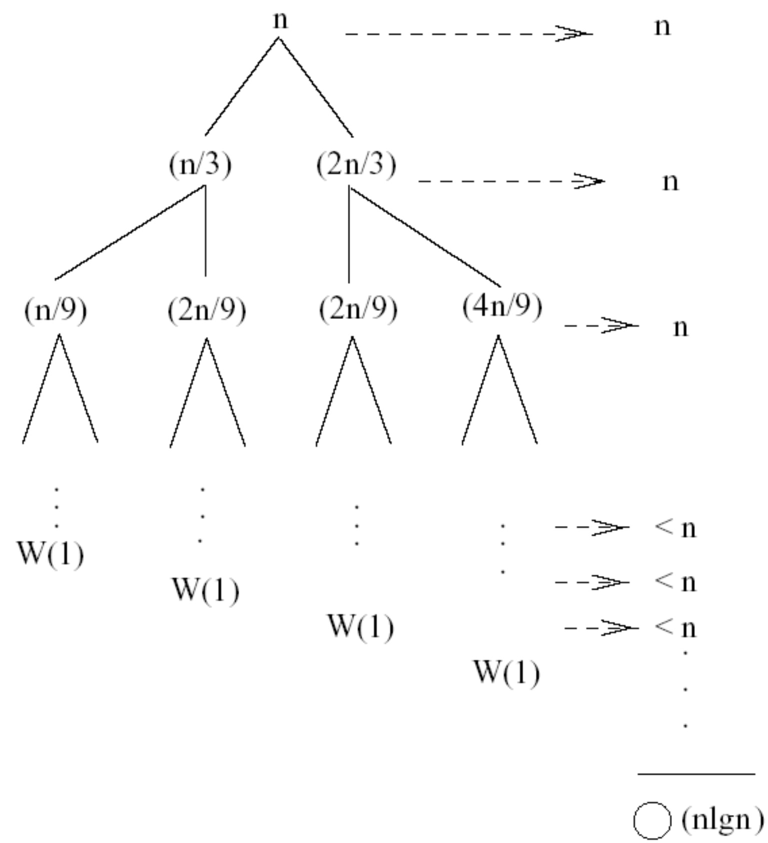
- Subproblem size hits 1 when

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

- 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

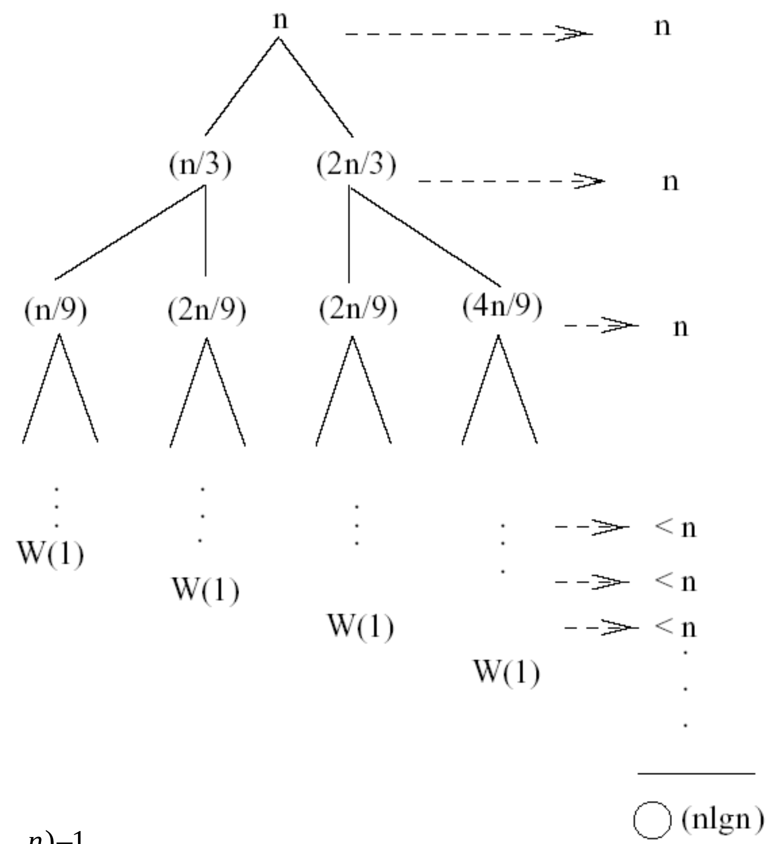
$W(n) = W(n/3) + W(2n/3) + n$

- 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

- 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: $T(n) = cn \log n + d\, n$ *$n \log n$*

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)	$O(n \log n)$