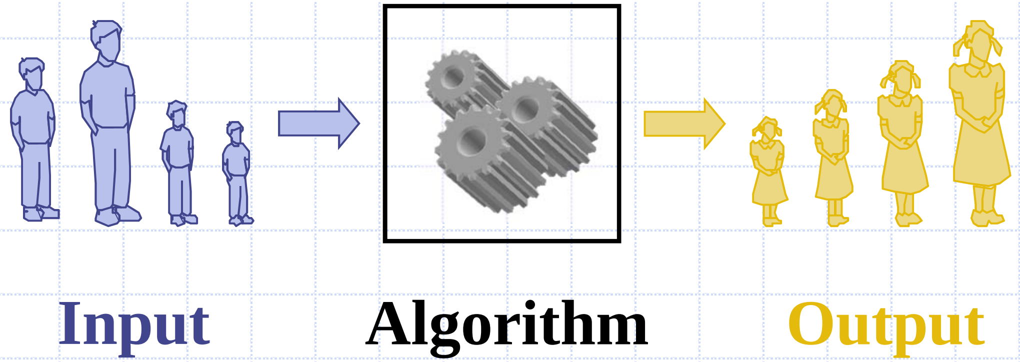
Lesson 2: Introduction To Analysis of Algorithms: *Discovering the Laws Governing Nature's Computation*



Input Algorithm Output

An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time. Some familiar problems solved with algorithms: Sorting an array of numbers, finding a specific string in a list, computing the shortest path between two locations, finding prime factors of a given integer.

Natural Things To Ask

- How can we determine whether an algorithm is *efficient* ?
- *Correct* ?
- Given two algorithms that achieve the same goal, how can we decide which one is better? (E.g. sorting) Can our analysis be independent of a particular operating system or implementation in a language?
- How can we express the steps of an algorithm without depending on a particular implementation?

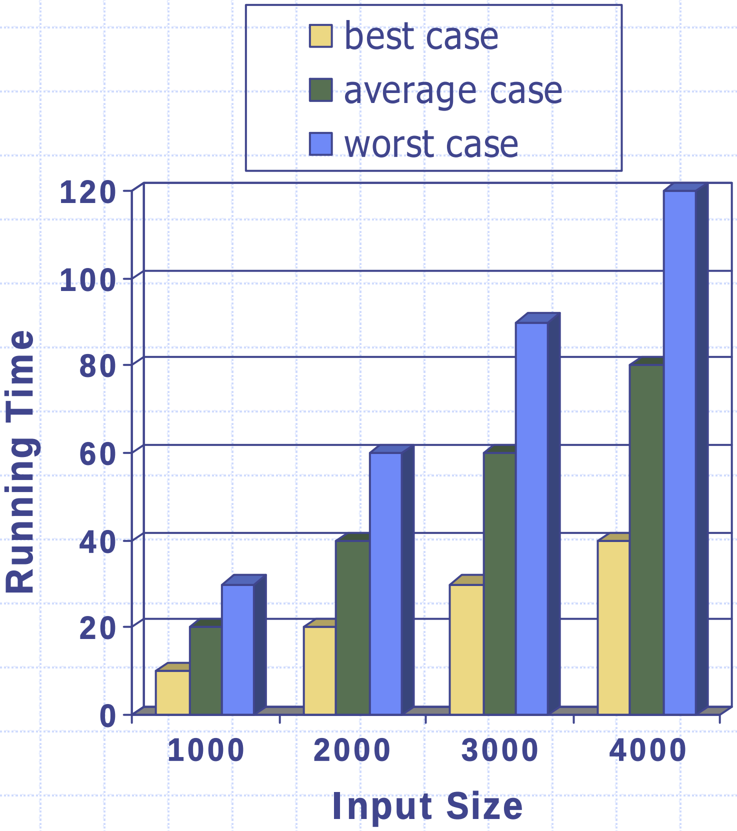
A Framework For Analysis of Algorithms

We will specify:

- A simple neutral language for describing algorithms
- A simple, general computational model in which algorithms execute
- Procedures for measuring running time
- A classification system that will allow us to categorize algorithms (a precise way of saying “fast”, “slow”, “medium”, etc)
- Techniques for proving correctness of an algorithm

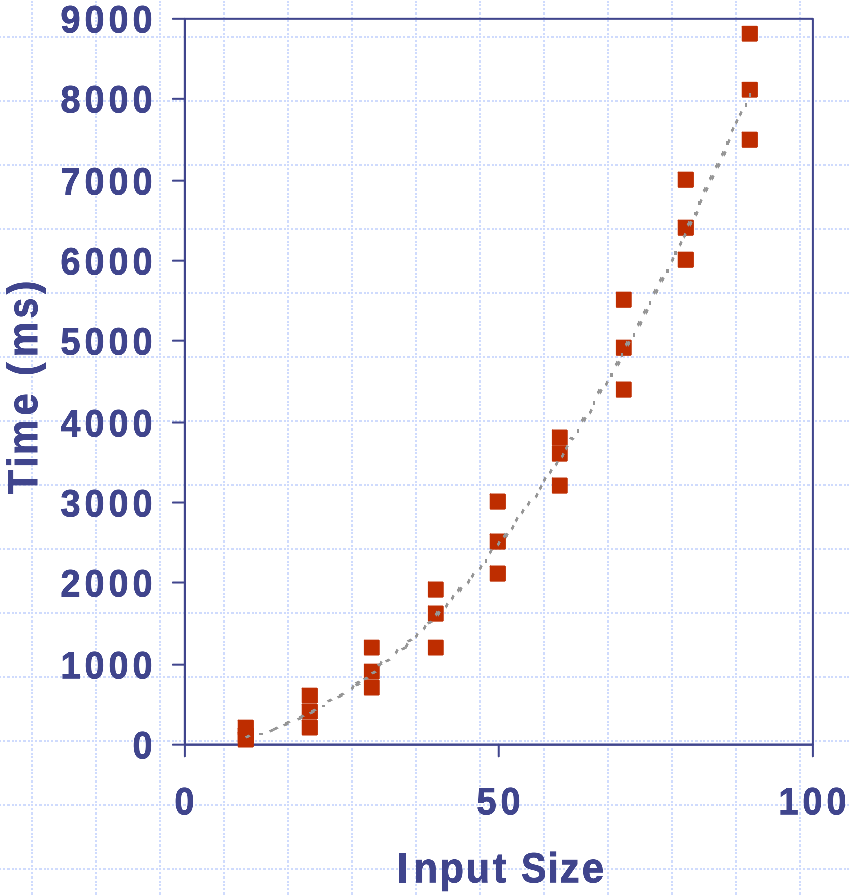
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- Often, we focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



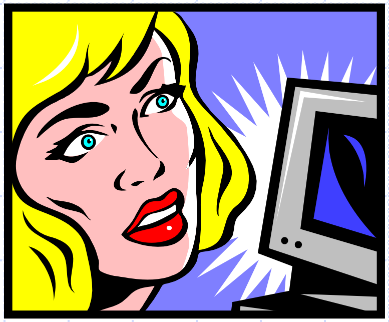
Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Optionally, plot the results

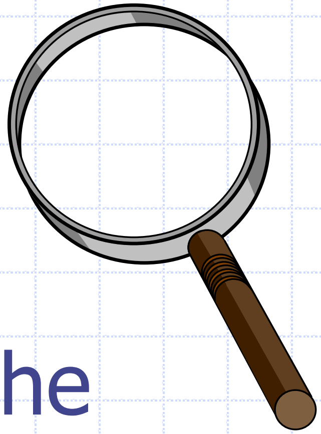


Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult
- Results may not be indicative of the running time on other inputs not included in the experiment.
- In order to compare two algorithms, the same hardware and software environments must be used



Theoretical Analysis



- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, *n*.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

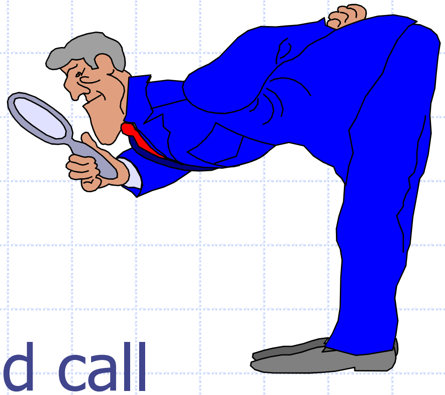
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax($A$, $n$)
  Input array $A$ of $n$ integers
  Output maximum element of $A$

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ to $n - 1$ do
    if $A[i] > currentMax$ then
      $currentMax \leftarrow A[i]$
  return $currentMax$
```

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …
- Method call

  var.method (***arg*** [, ***arg***…])
- Return value

  return ***expression***
- Expressions

  $\leftarrow$ Assignment (like = in Java)

  $=$ Equality testing (like == in Java)

  n^2 Superscripts and other mathematical formatting allowed

Exercises

- Sorting algorithm: Translate into pseudo-code:

Given an array arr, *create a second array* arr2 *of the same length*.

To begin, find the min *of* arr, *place it in position 0 of* arr2, *and remove* min *from* arr

At the i*th step, find the* min *of* arr, *place it in the next available position in* arr2, *and remove* min *from* arr

(Assume that min and remove functions are available)

- Palindrome algorithm: Transform into pseudo-code:

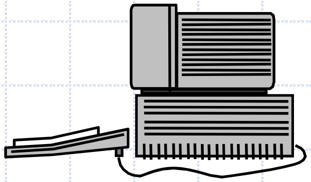
```
public boolean isPal(String s) {
   if(s==null || s.length() <= 1) {
      return true;
   }
   while(s.length() > 1){
      if(s.charAt(0) !=
         s.charAt(s.length()-1)){
          return false;
      }
      s = s.substring(1,s.length()-1);
   }
   return true;
}
```

Main Point

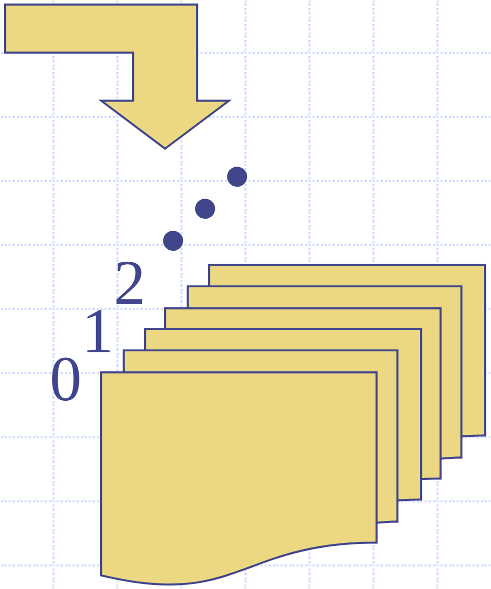
For purposes of examining, analyzing, and comparing algorithms, a neutral algorithm language is used, independent of the particularities of programming languages, operating systems, and system hardware. Doing so makes it possible to study the inherent performance attributes of algorithms, which are present regardless of implementation details. This illustrates the SCI principles that more abstract levels of intelligence are more comprehensive and unifying.

The Random Access Machine (RAM) Model

- A **CPU**



- A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



- Memory cells are numbered and accessing any cell in memory takes unit time.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Primitive Operations In This Course

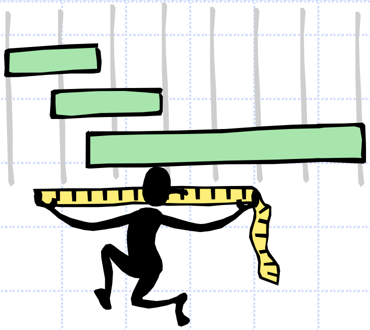
- Performing an arithmetic operation (+, *, etc)
- Comparing two numbers
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
- Following an object reference

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

```
Algorithm arrayMax(A, n)                              # operations
    currentMax ← A[0]                                     2
    m ← n − 1                                             2
    for i ← 1 to m do                                   1 + n
        if A[i] > currentMax then                       2(n − 1)
            currentMax ← A[i]                           2(n − 1)
        { increment counter i }                         2(n − 1)
    return currentMax                                     1
                                             Total        7n
```

Estimating Running Time



- Algorithm ***arrayMax*** executes $7\boldsymbol{n}$ primitive operations in the worst case. Define:

  $\boldsymbol{a}$ = Time taken by the fastest primitive operation

  $\boldsymbol{b}$ = Time taken by the slowest primitive operation

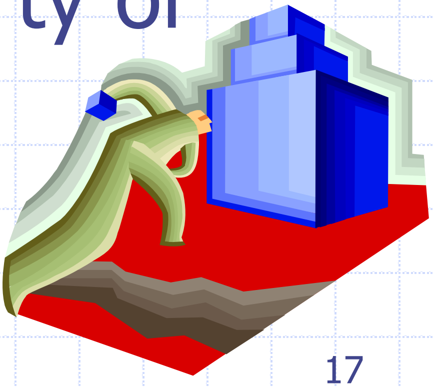
- Let $\boldsymbol{T}(\boldsymbol{n})$ be worst-case time of ***arrayMax*.** Then

$$\boldsymbol{a}\,(7\boldsymbol{n}) \le \boldsymbol{T}(\boldsymbol{n}) \le \boldsymbol{b}(7\boldsymbol{n})$$

- Hence, the running time $\boldsymbol{T}(\boldsymbol{n})$ is bounded by two linear functions

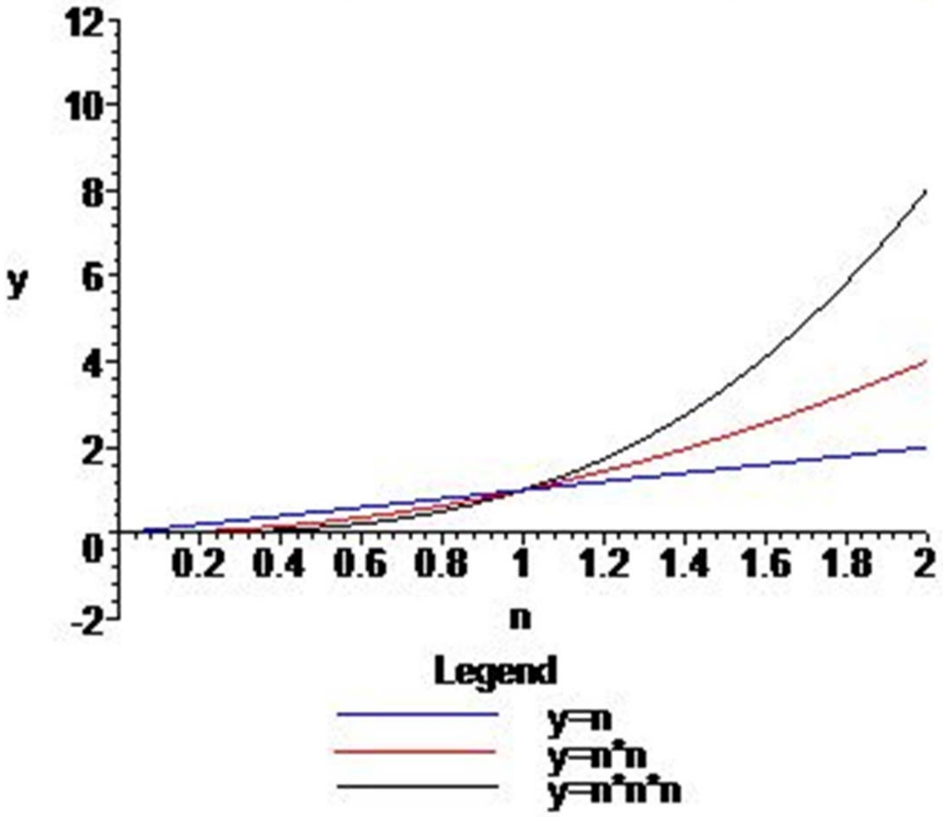
Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm ***arrayMax***



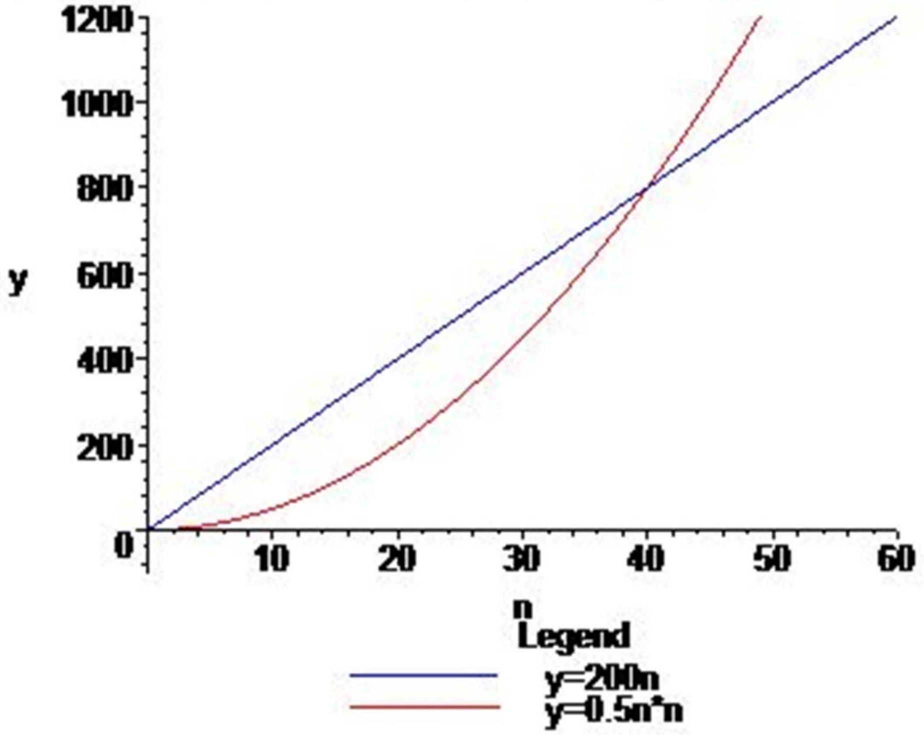
Growth Rates

- Growth rates of functions:
  - Linear ≈ n
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
- The graph of the cubic begins as the slowest but eventually overtakes the quadratic and linear graphs
- Important factor for growth rates is the behavior as *n* gets large



Constant Factors & Lower-order Terms

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Example
  - Compare 200*n with 0.5n*n
  - Quadratic growth rate must eventually dominate linear growth



Big-Oh Notation (§1.2)

- Given functions $f(n)$ and $g(n)$ defined on non-negative integers n, we say that $f(n)$ is $O(g(n))$ (or "f(n) belongs to $O(g(n))$") if there are positive constants c and n_0 such that

  $f(n) \le cg(n)$ for all $n \ge n_0$

- Example: $2n + 10$ is $O(n)$

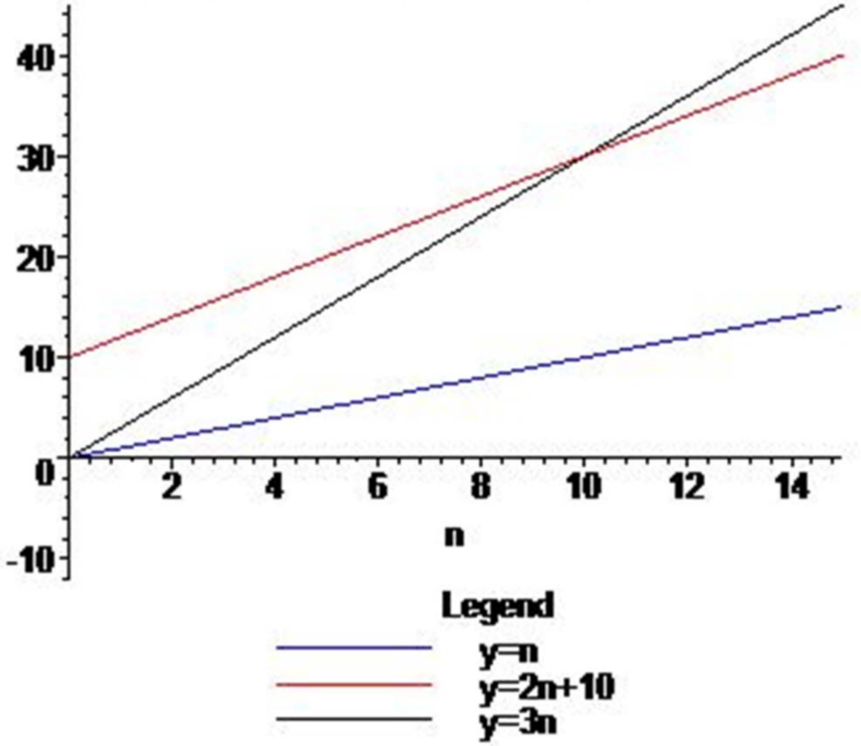
  Scratchwork

  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Try: $c = 3$ and $n_0 = 10$

  Solution

  Let n_0 =10. For any n ≥ n_0, we show 2n+10 ≤ 3n:

  But this is true iff 10 ≤ n, which holds by choice of n.



Big-Oh Example

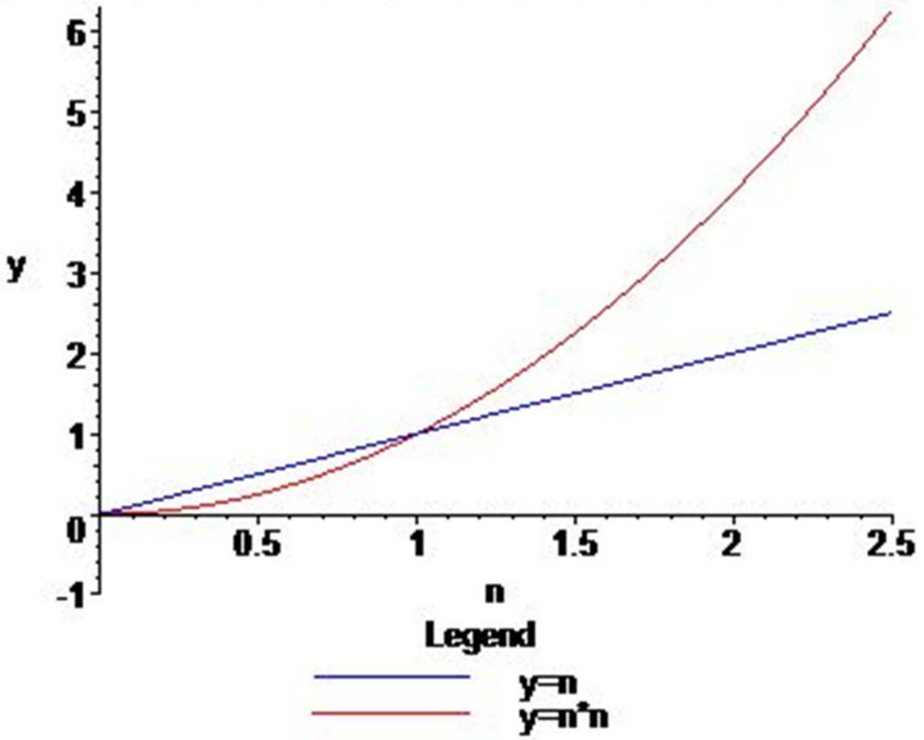
- Example: n^2 is not $O(n)$

Proof

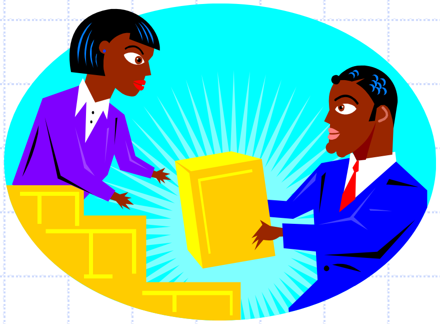
For each c and n_0, we need to find an $n \geq n_0$ such that $n^2 > cn$.

Can do this by letting n be any integer bigger than both n_0 and c. Then

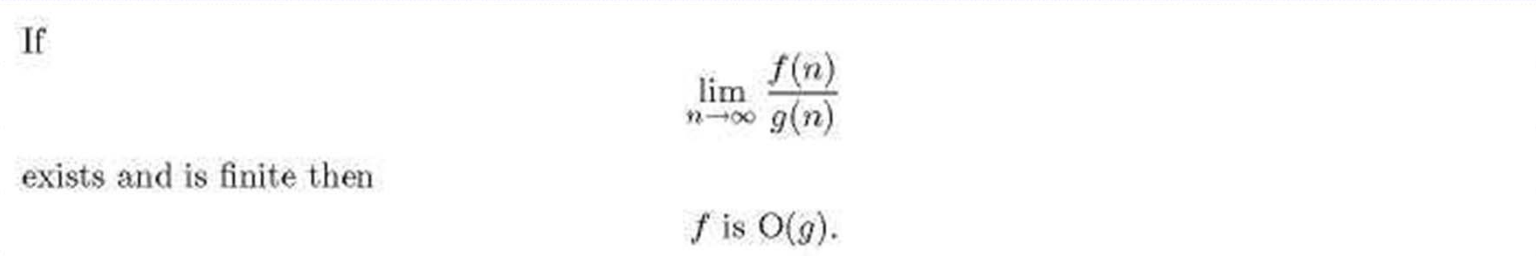
$$n^2 = n * n > c * n$$



More Big-Oh Examples



- n is O(2n+1)
- nlog n + n is O(nlog n)
- FACT:



If

$$\lim_{n\to\infty} \frac{f(n)}{g(n)}$$

exists and is finite then

$$f \text{ is } O(g).$$

- Example:

  $3n^2 + 1$ is $O(2n^2 + n)$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- Example: Neither of the functions $2n$ nor $2n^2$ grows any faster (asymptotically) than n^2. Therefore, both functions belong to $O(n^2)$

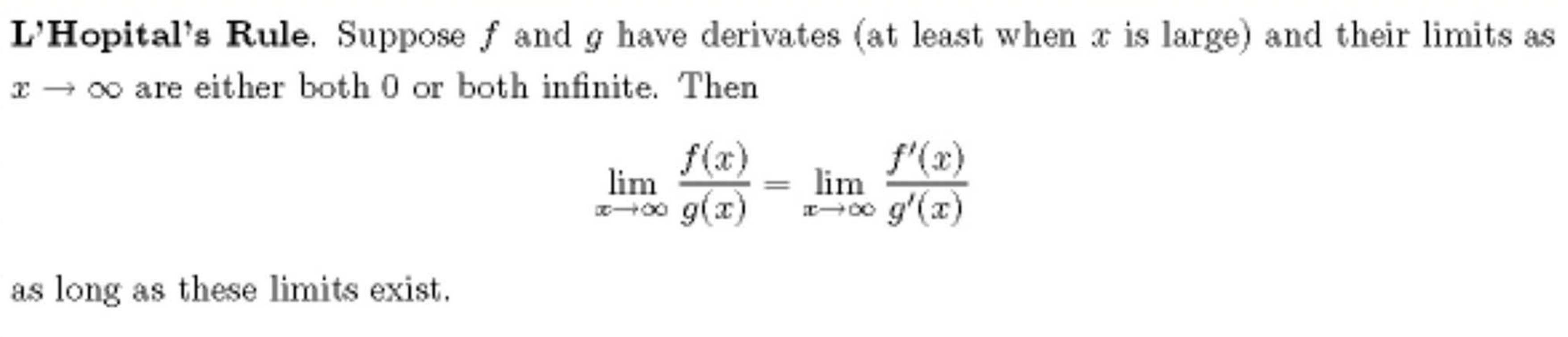
Standard Complexity Classes

- The most common complexity classes used in analysis of algorithms are, in increasing order of growth rate:

  $O(1)$, $O(\log n)$, $O(n^{1/k})$, $O(n)$, $O(n\log n)$, $O(n^k)$ $(k > 1)$,

  $O(2^n)$, $O(n!)$, $O(n^n)$

- Functions that belong to classes in the first row are known as *polynomial time bounded.*
- Verification of the relationships between these classes can be done most easily using limits, sometimes with L'Hopital's Rule



L'Hopital's Rule. Suppose f and g have derivates (at least when x is large) and their limits as $x \to \infty$ are either both 0 or both infinite. Then

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

as long as these limits exist.

Optional Examples Using Limits

- $\lim(3n^2 + 2n - 4 / n^2)$ is finite (= 3), so $3n^2 + 2n - 4$ is $O(n^2)$.
- $\lim (n^{1/2} / n) = 0$ so $n^{1/2}$ is $O(n)$
  but $\lim(n / n^{1/2})$ is infinite, so n is not $O(n^{1/2})$
- $\lim (\log n / n^{1/2}) = 0$, so log n is $O(n^{1/2})$
  but $\lim (n^{1/2} / \log n)$ is infinite, so $n^{1/2}$ is not $O(\log n)$
- $\lim (n^k / 2^n) = 0$, so n^k is $O(2^n)$ (for any $k > 0$)
  but $\lim (2^n / n^k)$ is infinite, so 2^n is not $O(n^k)$

Big-Oh Rules

- These rules can be verified using limits – can use the rules even if you are unfamiliar with limits
- If is $f(n)$ a polynomial of degree d, say, $f(n) = a_0 + a_1 n + \ldots + a_d n^d$, then $f(n)$ is $O(n^d)$:
  1. Drop lower-order terms (those of degree less than d)
  2. Drop constant factors (in this case, a_d)
  3. See first example on previous slide
- 2 Guidelines:
  - Use the smallest possible class of functions
  - E.g. Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
  - Use the simplest expression of the class
  - E.g. Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the (worst-case) asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We determined that algorithm ***arrayMax*** executes at most $7\boldsymbol{n}$ primitive operations
  - Since $7\boldsymbol{n}$ is $\boldsymbol{O}(\boldsymbol{n})$, we say that algorithm ***arrayMax*** "runs in $\boldsymbol{O}(\boldsymbol{n})$ time"

Basic Rules For Computing Asymptotic Running Times

Rule-1: For Loops

The running time of a for loop is at most the running time of the statements inside the loop (including tests) times the number of iterations (see ***arrayMax***)

Rule-2: Nested Loops

Analyze from inside out. The total running time of a statement inside a group of nested loops is the running time of the statement times the sizes of all the loops

```
for i ← 0 to n-1 do
    for j ← 0 to n-1 do
        k ← i + j
```

(Runs in $O(n^2)$)

(continued)

Rule-3: Consecutive Statements

Running times of consecutive statements should be added in order to compute running time of the whole

```
for i ← 0 to n-1 do
    a[i] ←  0
for i ← 0 to n-1 do
    for j ← 0 to i  do
        a[i] ← a[i] + i + j
```

(Running time is $O(n) + O(n^2)$. By an exercise, this is $O(n^2)$)

(continued)

- Rule-4: If/Else

For the fragment

```
    **if** *condition* **then**
        S1
    **else**
        S2
```

the running time is never more than the running time of the *condition* plus the larger of the running times of S1 and S2.

Example: Removing Duplicates From An Array

The problem: Given an array of n integers that lie in the range 1..2n, return an array in which all duplicates have been removed.

Remove Dups, Algorithm #1

```
Algorithm removeDups1(A,n)
    Input: An array A with n > 0 integers in the
    range 0..2n-1
    Output: An array B with all duplicates in A removed
    for i ← A.length–1 to 0 do
        for j ← A.length –2 to i + 1 do
            if A[j] = A[i] then
                A ← removeLast(A, A[i])
Algorithm removeLast(A,a)
    Input: An array A of integers and an array element a
    Output: The array A modified by removing last occurrence of a
    pos ← -1
    k ← A.length
    B ← new Array(k - 1)
    i ← k - 1
    while pos < 0 do           //must eventually terminate
        if a = A[i] then pos ← i
        else i ← i – 1
    for j ← 0 to k-2 do
        if j < pos then B[j] ← A[j]
        else B[j] ← A[j+1]
    return B
```

Analysis

T_{rl} = running time of removeLast

T_{rd} = running time of removeDups1

- $T_{rl}(k)$ is $O(2k) = O(k)$
- $T_{rd}(n)$ is $O(n^3)$

Therefore, *the running time of Algorithm #1 is* $O(n^3)$

One way to improve: Insert non-dups into an auxiliary array.

Remove Dups, Algorithm #2

```
Algorithm removeDups2(A,n)
    Input: An array A with n > 0 integers in the
    range 0..2n-1
    Output: An array B with all duplicates in A removed
    B ← new Array(n)    //assume initialized with 0's
    index ← 0
    for i ← 0 to n-1 do
        dupFound ← false
        for j ← 0 to i –1 do
            if A[j] = A[i] then
                dupFound ← true
                break  //exit to outer loop
        //if no dup found up to i, add A[i] to new array
        if !dupFound then
            B[index] ← A[i]
            index ← index + 1
    //end outer for loop
    //next: eliminate extra 0's at the end
    C ← new Array(index)
    for j ← 0 to index do
        C[j] ← B[j]
    return C
```

Analysis

T = running time of removeDups2

O(n) initialization +
nested for loops bounded by n +
O(n) copy operation

=>

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

Therefore, *the running time of Algorithm #2 is* $O(n^2)$

One way to improve: Use bookkeeping device to keep track of duplicates and eliminate inner loop

Remove Dups, Algorithm #3

```
Algorithm removeDups3(A,n)
    Input: An array A with n > 0 integers in the range 0..2n-1
    Output: An array with all duplicates in A removed
    W ← new Array(2n)     //for bookkeeping
    B ← new Array(n)      //assume both initialized with 0's
    index ← 0
    for i ← 0 to n-1 do
        u ← A[i]
        W[u] ← W[u] + 1
        if W[u] = 1 then   //means a new value
            B[index] ← A[i]
            index ← index + 1

    //next: eliminate extra 0's at the end
    C ← new Array(index)
    for j ← 0 to index do
        C[j] ← B[j]
    return C
```

Analysis

T = running time of removeDups3

O(n) initialization +
single for loop of size n +
O(n) copy operation

=>

T(n) is 3 * O(n) = O(n)

Therefore, *the running time of Algorithm #3 is* O(n)

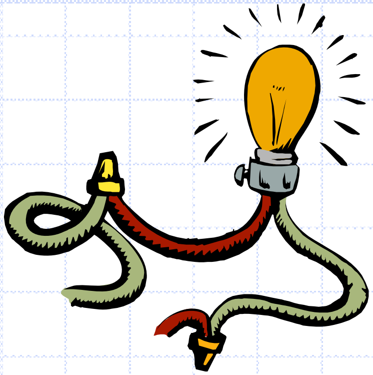
Main Point

One can improve the running time of both the remove duplicates algorithm and the "sum of two equals third" algorithm from Ω(n^2) to O(n) by introducing a a tracking array as a means to encapsulate "bookkeeping" operations (this can be done more generally using a hashtable). This technique is reminiscent of the Principle of the Second Element from SCI: To remove the darkness, struggling at the level of darkness is ineffective; instead, introduce a *second element* – namely, *light*. As soon as the light is introduced, the problem of darkness disappears.

Relatives of Big-Oh

- **big-Omega**
  - f(n) is Ω(g(n)) if g(n) is O(f(n)).
- **big-Theta**
  - f(n) is Θ(g(n)) if f(n) is both O(g(n)) and Ω(g(n)).

Intuition for Asymptotic Notation



big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically **less than or equal** to g(n)

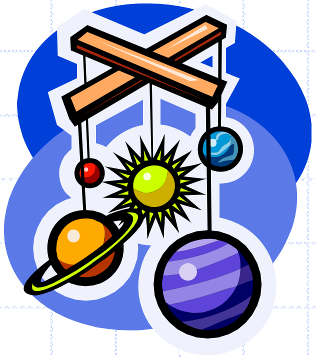
big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically **greater than or equal** to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically **equal** to g(n)

Examples of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$ and therefore, $5n^2$ is $\Theta(n^2)$**

  f(n) is $\Omega(g(n))$ iff g(n) is O(f(n)) iff there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $g(n) \leq c \bullet f(n)$ for all $n \geq n_0$

  Try $c = 5$ and $n_0 = 1$. This proves $5n^2$ is $\Omega(n^2)$**.**

  To show $5n^2$ is $\Theta(n^2)$**,** must show also that n^2 is $O(5n^2)$ – which is obvious.

  Therefore $5n^2$ is $\Theta(n^2)$

- **$5n^2$ is $\Omega(n)$ but $5n^2$ is not** $\Theta(n)$

  f(n) is $\Omega(g(n))$ iff g(n) is O(f(n)) iff there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $g(n) \leq c \bullet f(n)$ for all $n \geq n_0$

  Try $c = 1$ and $n_0 = 1$. This proves $5n^2$ is $\Omega(n)$**.**

  However, $5n^2$ is *not* O(n)**:** for any c and n_0**,** let n be greater than both. Then $5n^2 > n^2 = n*n > c * n$. Therefore, $5n^2$ is not $\Theta(n)$

Running Time of Recursive Algorithms

```
Algorithm search(A,x)
    Input: An already sorted array A with n
            elements and search value x
    Output: true or false

    return binSearch(A, x, 0,  A.length-1)
```

(continued)

```
Algorithm binSearch(A,x,lower,upper)
    Input: Already sorted array A, value x to be
           searched for in interval [lower,upper]
    Output: true or false

if lower > upper then  return false               +1
mid ← (upper + lower)/2                           +3
if x = A[mid] then  return true                   +2
if x < A[mid]  then                               +2
    return binSearch(A, x, lower, mid -1)         +2 + T(n/2)
else
    return binSearch(A, x, mid + 1, upper)
```

For the worst case (x not found), running time is given by the ***Recurrence Relation***:

$T(1) = 12$; $T(n) = 10 + T(n/2)$

Solving A Recurrence Relation: The Guessing Method

$T(1) = 12$

$T(2) = 10 + 12$

$T(4) = 10 + 10 + 12$

$T(8) = 10 + 10 + 10 + 12$

$T(2^m) = 10 * m + 12$

$T(n) = 10 * \log n + 12$, which is $O(\log n)$

- Guessing method requires us to guess the general formula from a few small values. This is not always possible to do (we will discuss an alternative method in a later lecture)
- When using the guessing method, final formula should be verified. Often this is done by induction, though in simple cases, a direct verification is possible.

Verification of Formula

Claim The function f(n) = 10 * log n + 12 is a solution to the recurrence

$T(1) = 12;\quad T(n) = T(n/2) + 10$ (n a power of 2)

Proof: Prove that whenever n is a power of 2, f(n) satisfies the recurrence. Since n is a power of 2, we write $n = 2^m$. Treating n as a power of 2, f(n) can be written as

$$f(2^m) = 10 * m + 12.$$

We must show that $f(1) = 12$ and $f(2^m) = f(2^{m-1}) + 10$

For n = 1, we have:

$f(1) = 10 * \log 1 + 12 = 12$

In general,

$$\begin{aligned} f(2^m) &= 10 * m + 12 \\ &= 10 * (m-1) + 10 + 12 \\ &= (10 * (m-1) + 12) + 10 \\ &= f(2^{m-1}) + 10, \end{aligned}$$

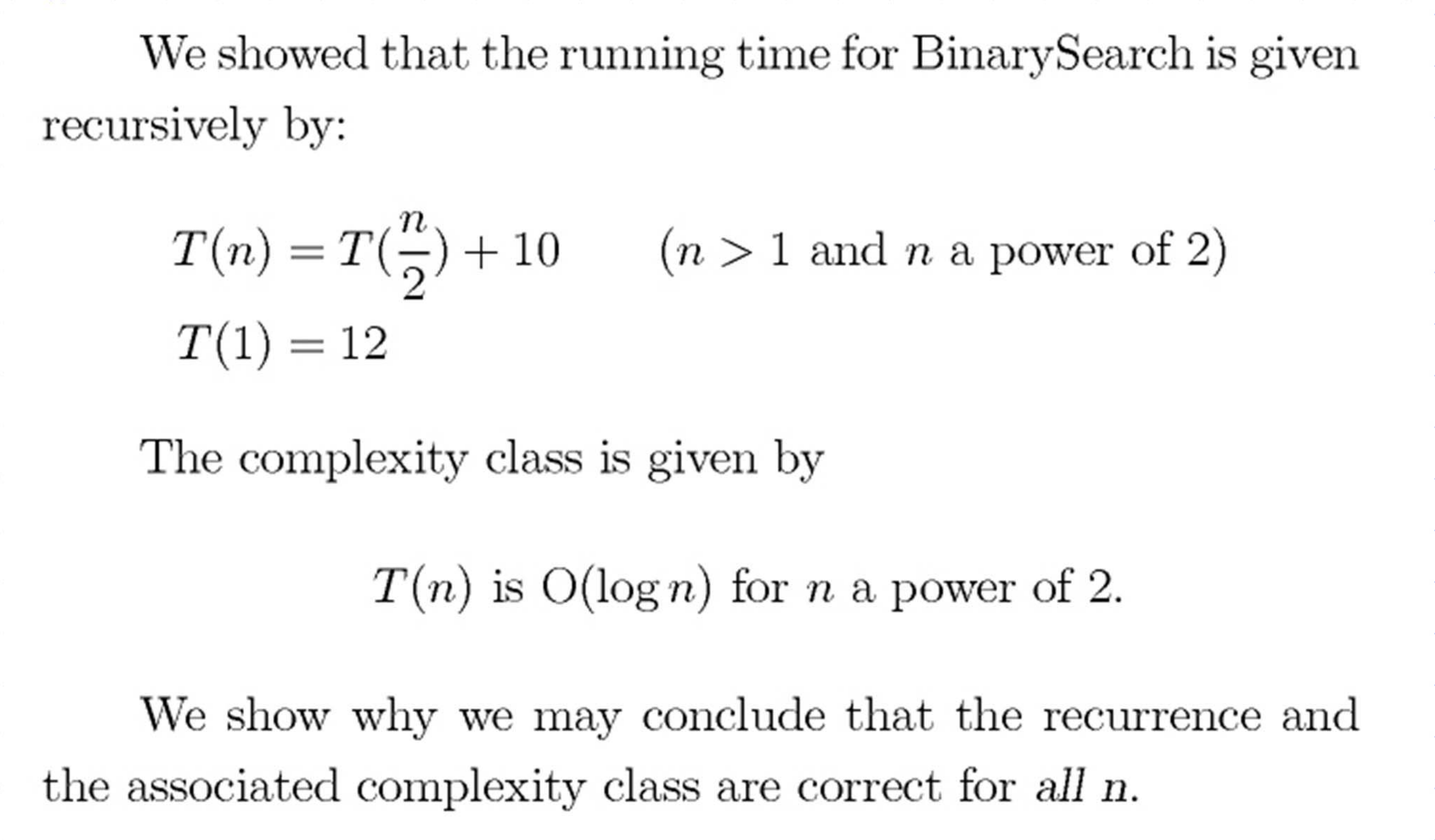
as required.

Additional Points About the Guessing Method

- To state results (so far) correctly, it’s necessary for n to be a power of 2. But we wish to have a bound on running time for any n – what can be done?

- In all cases that concern us, T is well enough behaved (as is the complexity function log n) between powers of 2 to permit us to conclude that, if we know the asymptotic running time, assuming n is a power of 2, the running time for all n, not necessarily a power of 2, must belong to this same complexity class. We illustrate the technique in this case.

Computing T(n) For Recursive Algorithms, n Not A Power of 2



We showed that the running time for BinarySearch is given recursively by:

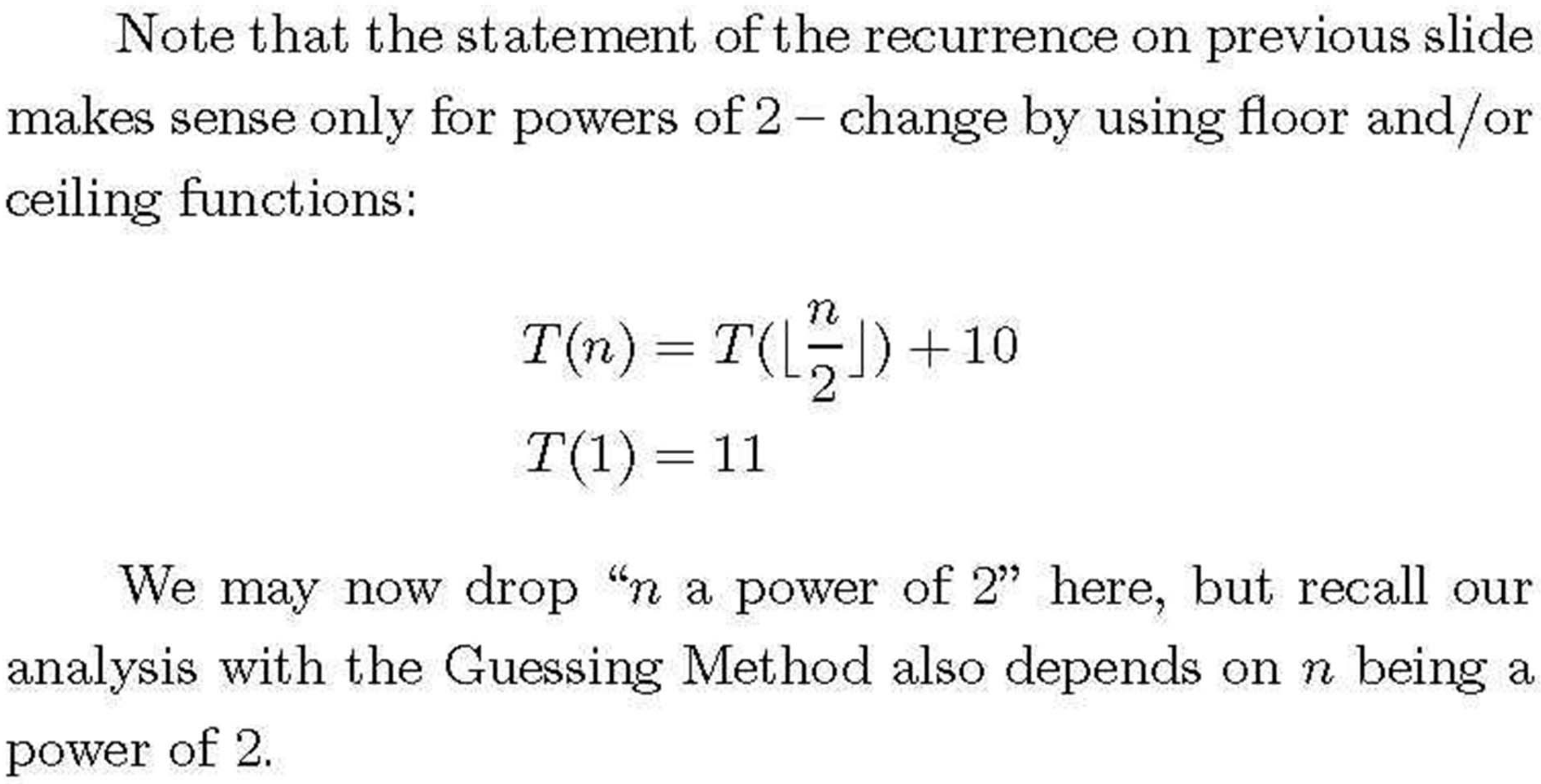
$$T(n) = T(\frac{n}{2}) + 10 \qquad (n > 1 \text{ and } n \text{ a power of } 2)$$
$$T(1) = 12$$

The complexity class is given by

$$T(n) \text{ is } \mathrm{O}(\log n) \text{ for } n \text{ a power of } 2.$$

We show why we may conclude that the recurrence and the associated complexity class are correct for *all n*.

Not A Power of 2 (continued)

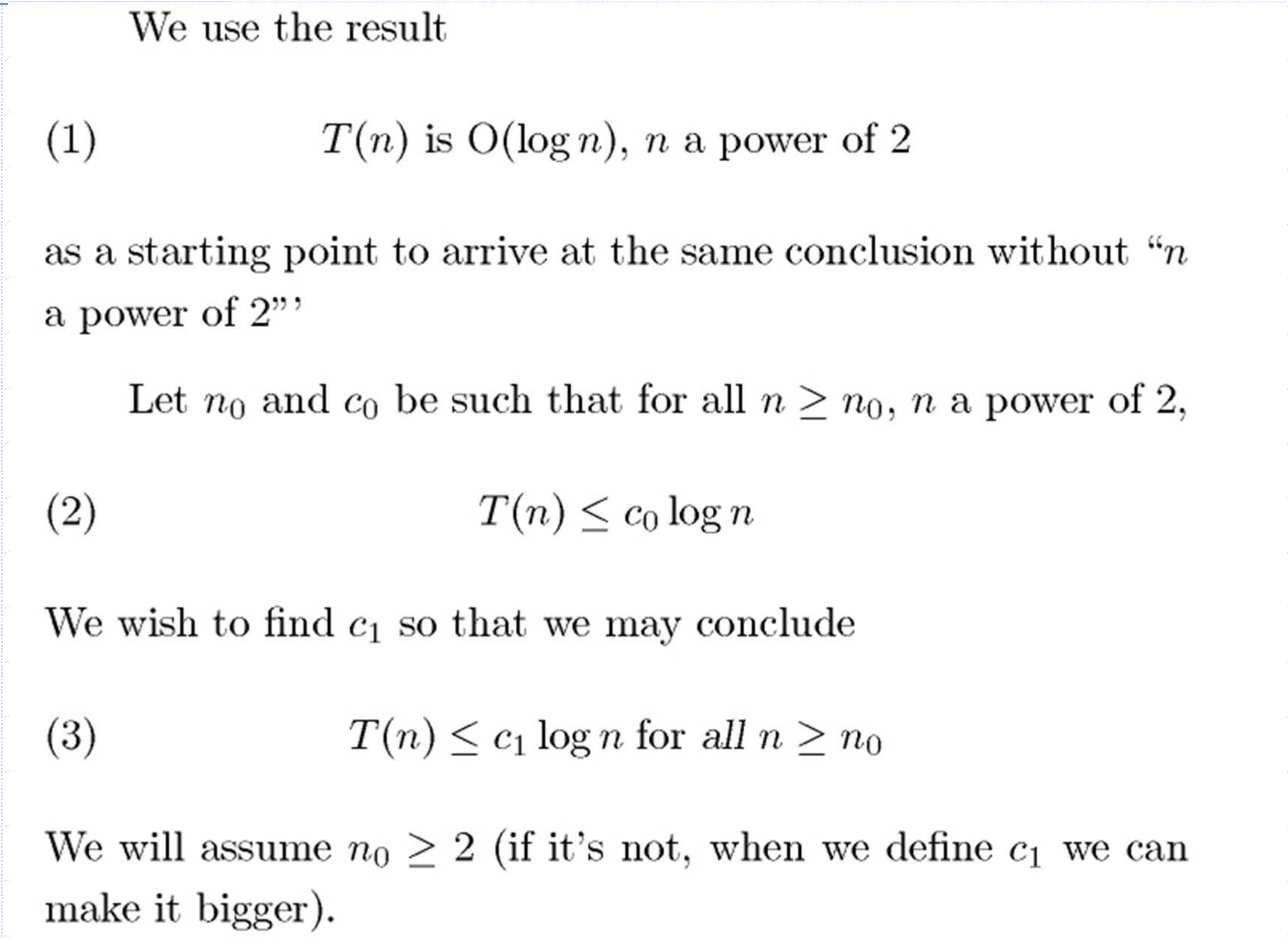


Note that the statement of the recurrence on previous slide makes sense only for powers of 2 – change by using floor and/or ceiling functions:

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + 10$$
$$T(1) = 11$$

We may now drop "n a power of 2" here, but recall our analysis with the Guessing Method also depends on n being a power of 2.

Not A Power of 2 (continued)



We use the result

(1) $T(n)$ is $O(\log n)$, n a power of 2

as a starting point to arrive at the same conclusion without “n a power of 2”’

Let n_0 and c_0 be such that for all $n \geq n_0$, n a power of 2,

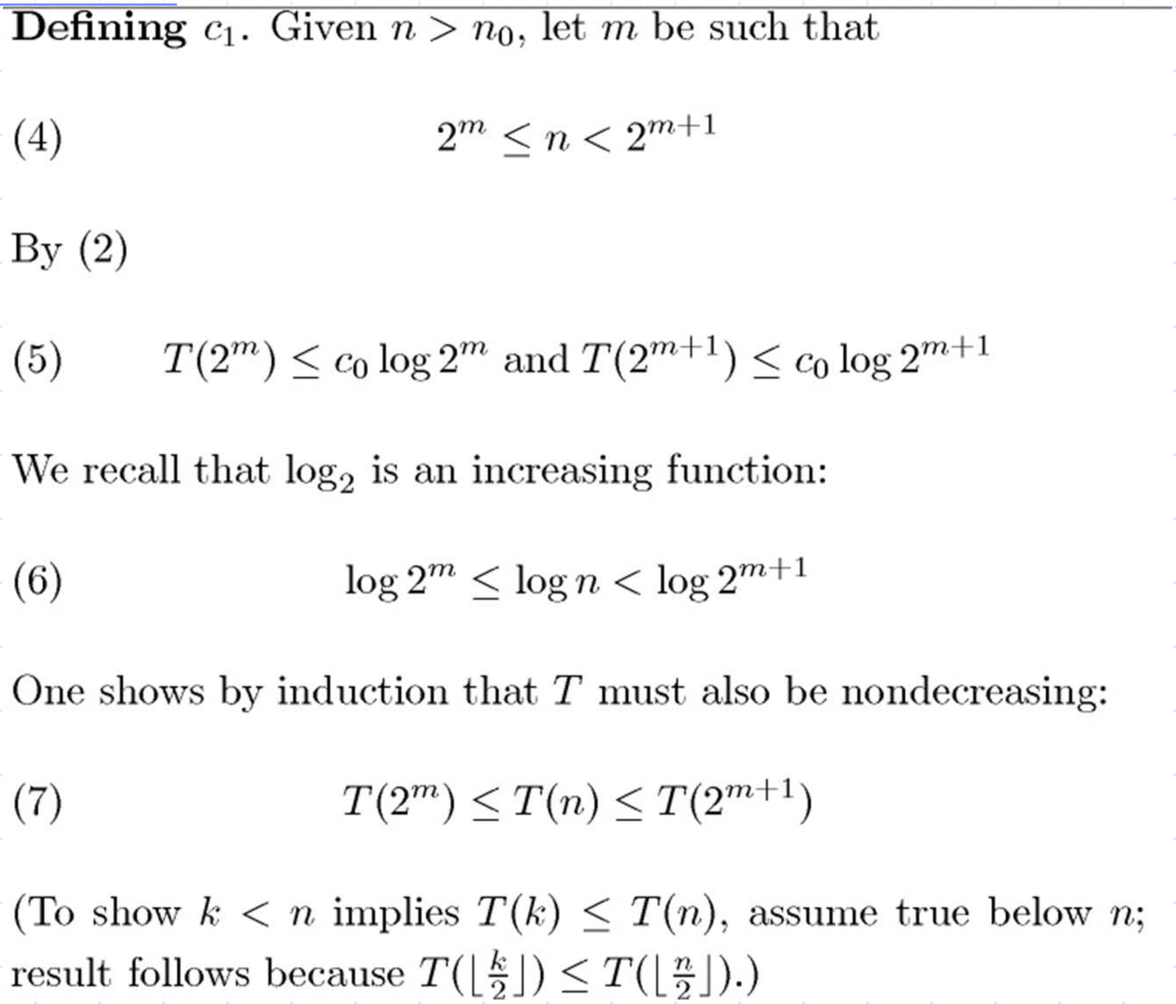
(2) $$T(n) \leq c_0 \log n$$

We wish to find c_1 so that we may conclude

(3) $T(n) \leq c_1 \log n$ for *all* $n \geq n_0$

We will assume $n_0 \geq 2$ (if it’s not, when we define c_1 we can make it bigger).

Not A Power of 2 (continued)



Defining c_1. Given $n > n_0$, let m be such that

$$(4) \qquad 2^m \le n < 2^{m+1}$$

By (2)

$$(5) \qquad T(2^m) \le c_0 \log 2^m \text{ and } T(2^{m+1}) \le c_0 \log 2^{m+1}$$

We recall that $\log_2$ is an increasing function:

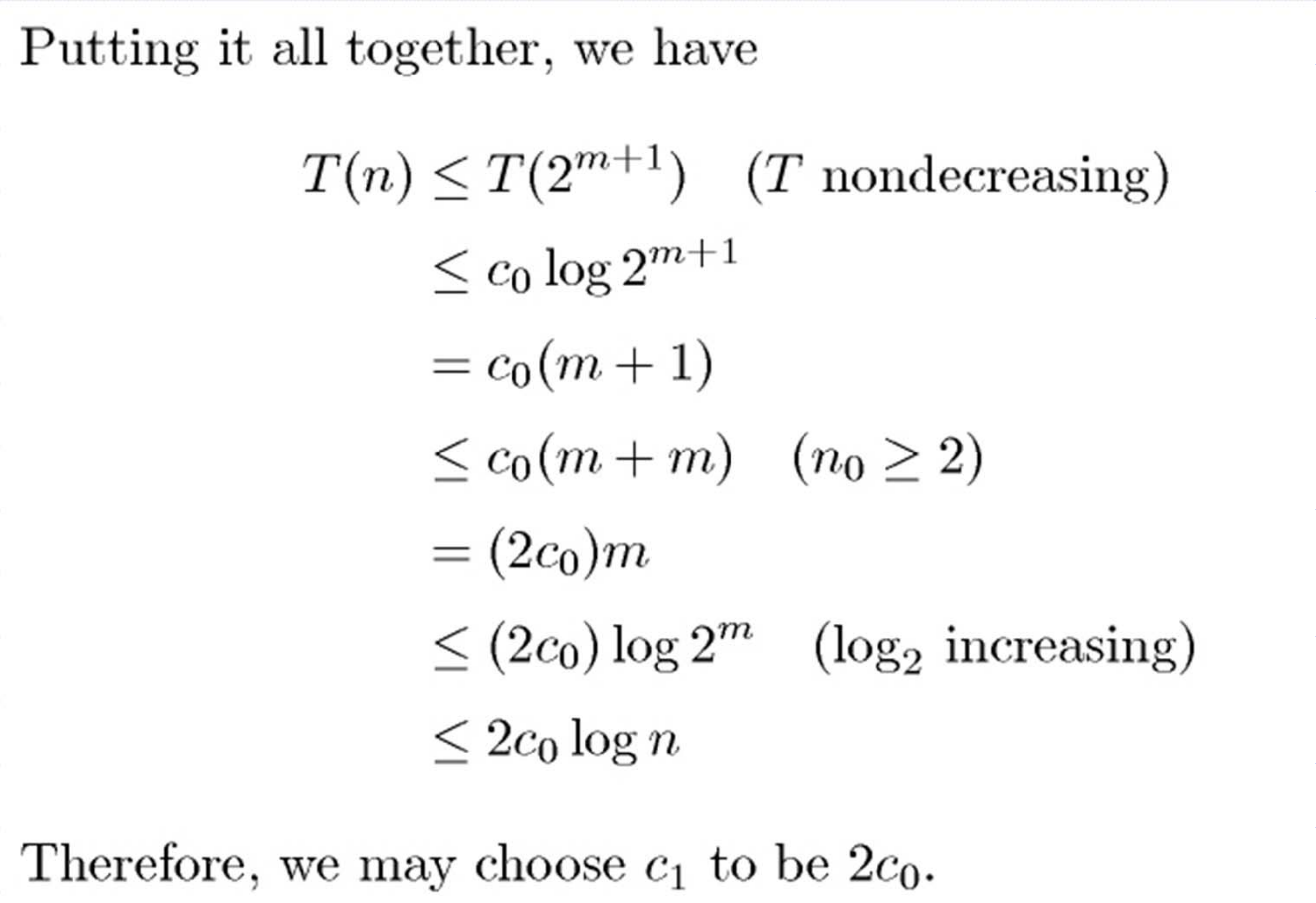
$$(6) \qquad \log 2^m \le \log n < \log 2^{m+1}$$

One shows by induction that T must also be nondecreasing:

$$(7) \qquad T(2^m) \le T(n) \le T(2^{m+1})$$

(To show $k < n$ implies $T(k) \le T(n)$, assume true below n; result follows because $T(\lfloor \frac{k}{2} \rfloor) \le T(\lfloor \frac{n}{2} \rfloor)$.)

Not A Power of 2 (continued)

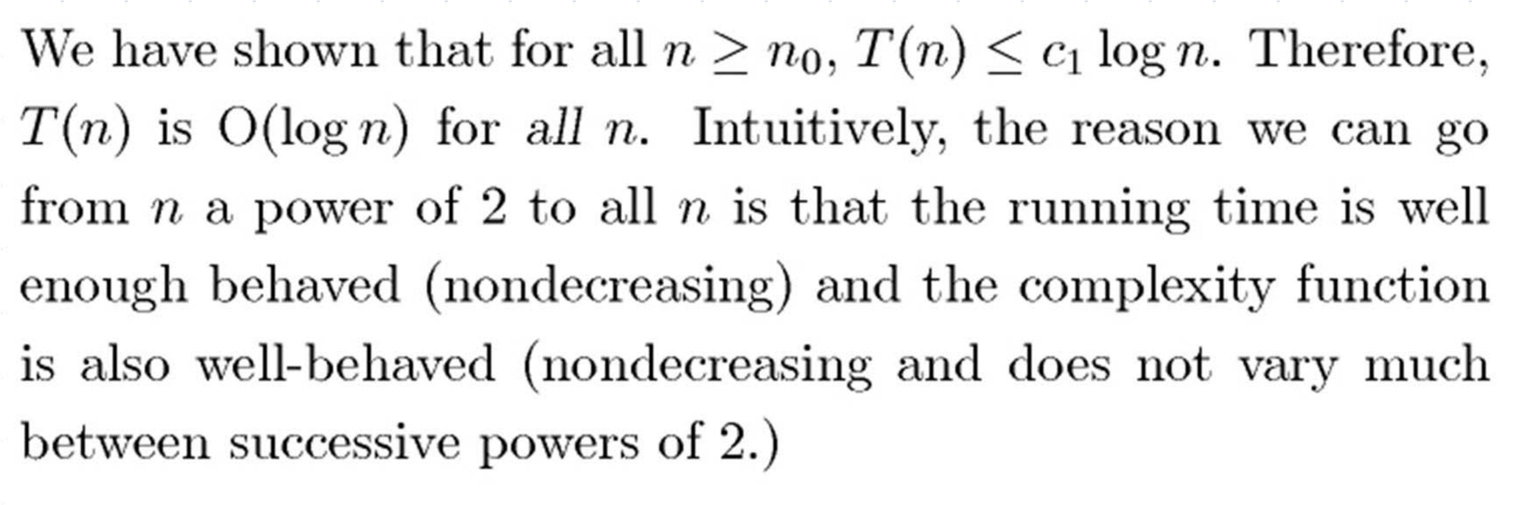


Putting it all together, we have

$$
\begin{aligned}
T(n) &\le T(2^{m+1}) \quad (T \text{ nondecreasing}) \\
&\le c_0 \log 2^{m+1} \\
&= c_0(m+1) \\
&\le c_0(m+m) \quad (n_0 \ge 2) \\
&= (2c_0)m \\
&\le (2c_0) \log 2^m \quad (\log_2 \text{ increasing}) \\
&\le 2c_0 \log n
\end{aligned}
$$

Therefore, we may choose c_1 to be $2c_0$.

Not A Power of 2 (continued)



We have shown that for all $n \geq n_0$, $T(n) \leq c_1 \log n$. Therefore, $T(n)$ is $\mathrm{O}(\log n)$ for *all* n. Intuitively, the reason we can go from n a power of 2 to all n is that the running time is well enough behaved (nondecreasing) and the complexity function is also well-behaved (nondecreasing and does not vary much between successive powers of 2.)

The Divide And Conquer Algorithm Strategy

- The binary search algorithm is an example of a “Divide And Conquer” algorithm, which is typical strategy when recursion is used.
- The method:
  - **Divide** the problem into subproblems (divide input array into left and right halves)
  - **Conquer** the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
  - **Combine** the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

Main Point

Recurrence relations are used to analyze recursively defined algorithms. Just as recursion involves repeated self-calls by an algorithm, so the complexity function $T(n)$ is defined in terms of itself in a recurrence relation. Recursion is a reflection of the self-referral dynamics of consciousness, on the basis of which all creation emerges. Recall: "Curving back on my own nature, I create again and again." (Gita, 9.8).

Another Technique To Solve Recurrences: Counting Self-Calls

- To determine the running time of a recursive algorithm, another often-used technique is *counting self-calls.*
- Often, processing time in a recursion, apart from self-calls, is constant. In such cases, running time is proportional to the number of self-calls.

Example of Counting Self-Calls: The Fib Algorithm

- The Fibonacci numbers are defined recursively by:

  $F(0) = 0, F(1) = 1, F(n) = F(n-1)+F(n-2)$

- This is a recursive algorithm for computing the nth Fibonacci number:

```
Algorithm fib(n)
    Input: a natural number n
    Output: F(n)

if (n = 0 || n = 1) then return n

return fib(n-1) + fib(n-2)
```

(continued)

Lemma. For n>1, the number of self-calls in fib(n) is $\geq F(n)$

Proof. By induction.

Base Case n=2. In this case there are two self-calls, but $F(2) = 1$.

Induction Step

- Assume the result for all $k < n$.
- Therefore, number of self-calls for fib(n-1) is $\geq F(n-1)$, for fib(n-2) is $\geq F(n-2)$.
- Therefore, number of self-calls for fib(n) is
  $\geq F(n-1)+F(n-2)$
  $=F(n)$

Lemma. For all $n > 4$, $F(n) > (4/3)^n$ **Proof**. Exercise!

==================================

Therefore, the running time of the fib algorithm is $\Omega(r^n)$ for some $r > 1$. In other words, fib is an *exponentially slow* algorithm!

Proving Correctness: Iterative Algorithms

- To prove that an iteratively defined algorithm (typically this means that loops are used instead of recursion) is correct, one uses the technique of *loop invariants.*
- The main idea is to prove that a certain relationship is preserved by each iteration through one or more loops. This fact then allows us to establish the correctness of the algorithm

Technique of Loop Invariants

- The technique of loop invariants is a form of *finite induction.*
- We wish to prove a statement *S* of the form:

  for all i < n, S_i *is true at the completion*
  of iteration #i of the loop

  (where S_i is a statement that depends on i).
- To prove S it suffices to perform these steps:
  - Pick a starting point *k* – typically, begin at end of iteration #1, but can start at a later iteration if necessary. Verify S_k for this starting value k.
  - Assume i is greater than or equal to k and that S_i holds at completion of iteration #i. Prove that S_{i+1} holds at the end of iteration #i+1.

Example of Technique of Loop Invariants

```
Algorithm iterativeFactorial(n)
    Input: A non-negative integer n
    Output: n!

    if (n = 0 || n = 1) then
        return 1
    accum ← 1
    for i ← 1 to n do
        accum ← accum * i
    return accum
```

Proof of correctness:

We establish the following loop invariant:
S_i: the value of accum is i!

Starting value is k = 1. At the end of iteration #1, S_1 is clearly true.

Assuming S_i (namely, that accum = i!) holds after iteration #i, it is clear that after iteration #i+1, accum = (i+1)!, and so S_{i+1}holds, as required.

Therefore, S_n holds, which means the final value of accum (which is the return value) is n!

Proving Correctness: Recursive Algorithms

The strategy for proving correctness of a recursive algorithm involves the following steps:

1. Verify that the recursion is valid: there should be a base case and recursive calls must eventually lead to the base case
2. Show that the values given by the base case are correct outputs for the function
3. Show that, if you assume the output value of the algorithm on input j < n is correct, then output value on input n is correct.

Example of Correctness Proof for a Recursive Algorithm

```
Algorithm recursiveFactorial(n)
    Input: A non-negative integer n
    Output: n!

    if (n = 0 || n = 1) then
        return 1
    return n * recursiveFactorial(n-1)
```

Proof of Correctness

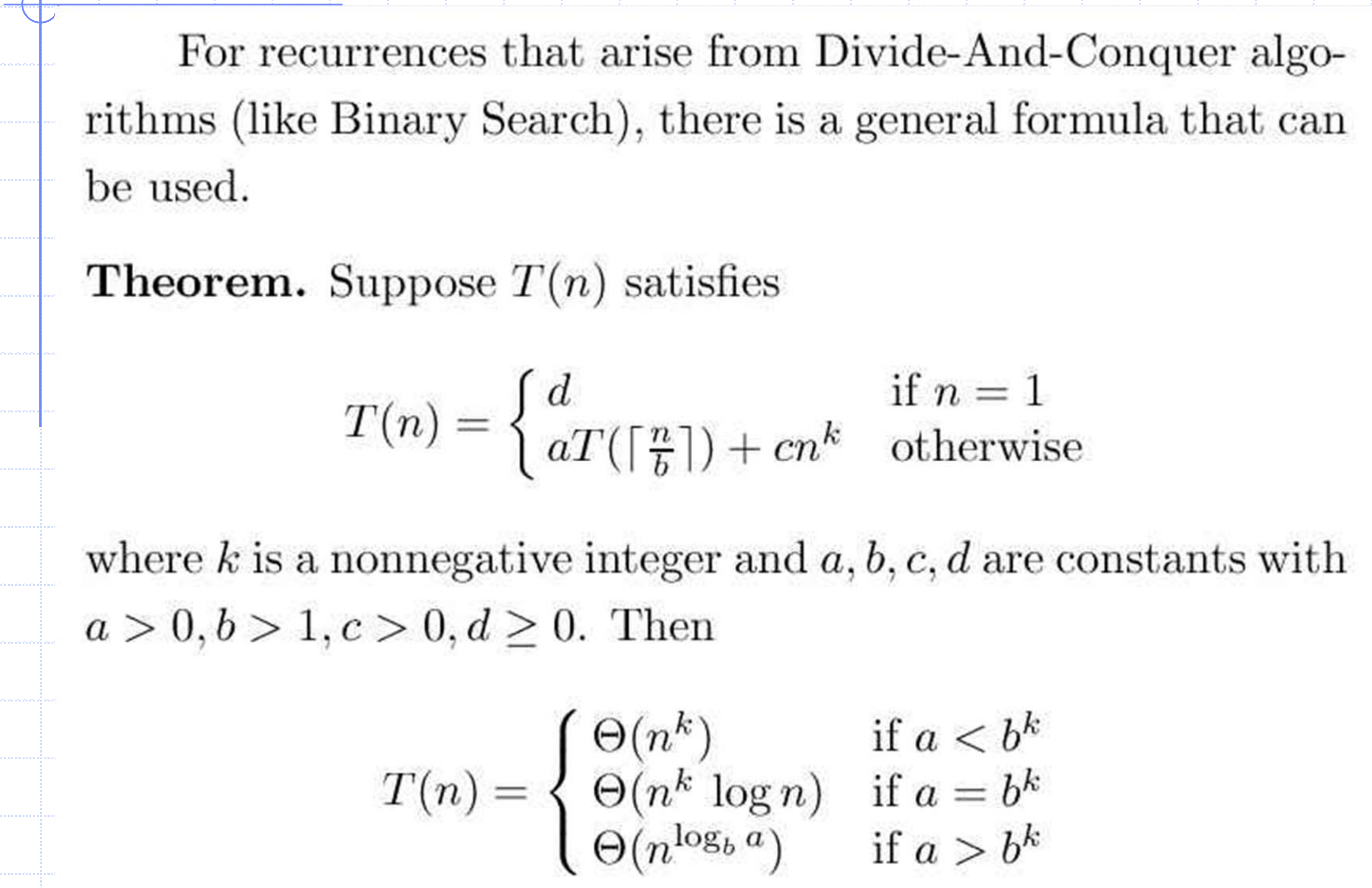
Valid Recursion Base case is “n= 0 or n=1”. Each self-call reduces input size by 1, so eventually base case is reached

Base The values 0! and 1! are correctly computed by the base case

Recursion Assuming recursiveFactorial(n-1) correctly computes n-1!, we must show output of recursiveFactorial(n) is correct. But output of recursiveFactorial(n) is n * recursiveFactorial(n-1) = n!, as required.

Therefore, the algorithm correctly computes n! for every n.

The Master Formula



For recurrences that arise from Divide-And-Conquer algorithms (like Binary Search), there is a general formula that can be used.

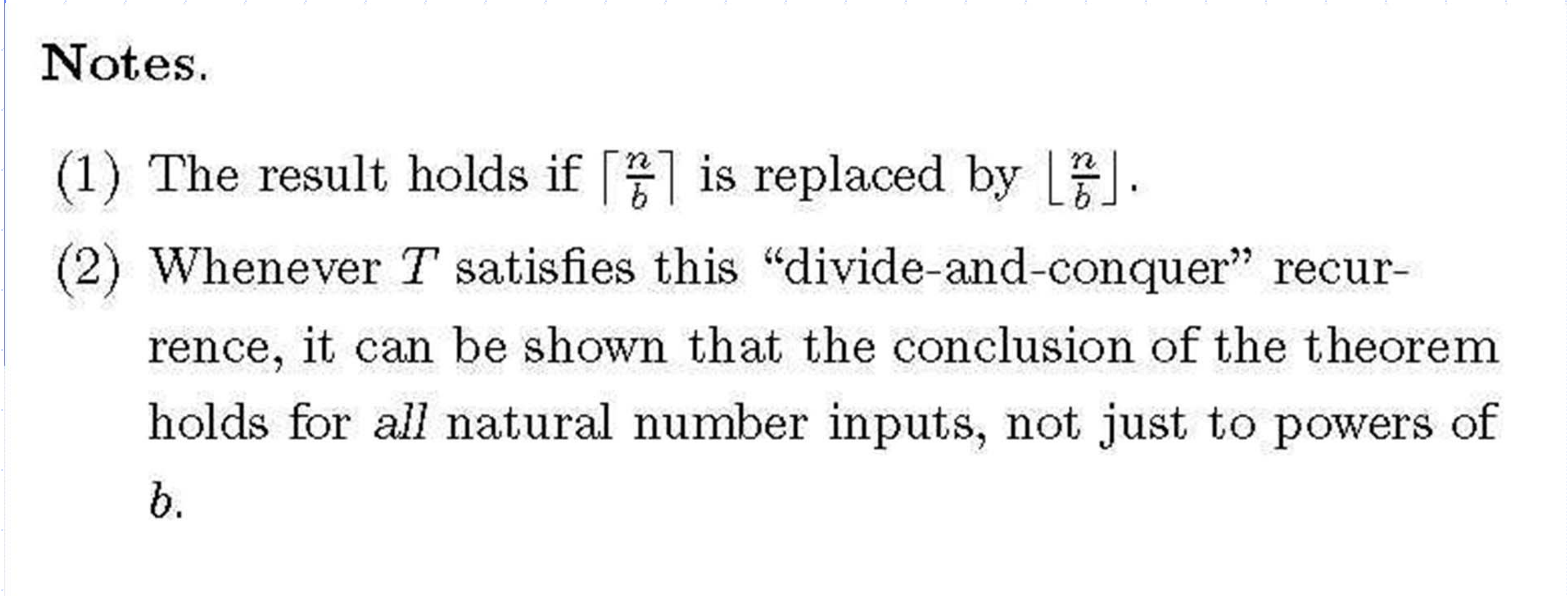
Theorem. Suppose $T(n)$ satisfies

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT(\lceil \frac{n}{b} \rceil) + cn^k & \text{otherwise} \end{cases}$$

where k is a nonnegative integer and a, b, c, d are constants with $a > 0, b > 1, c > 0, d \geq 0$. Then

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

Master Formula (continued)

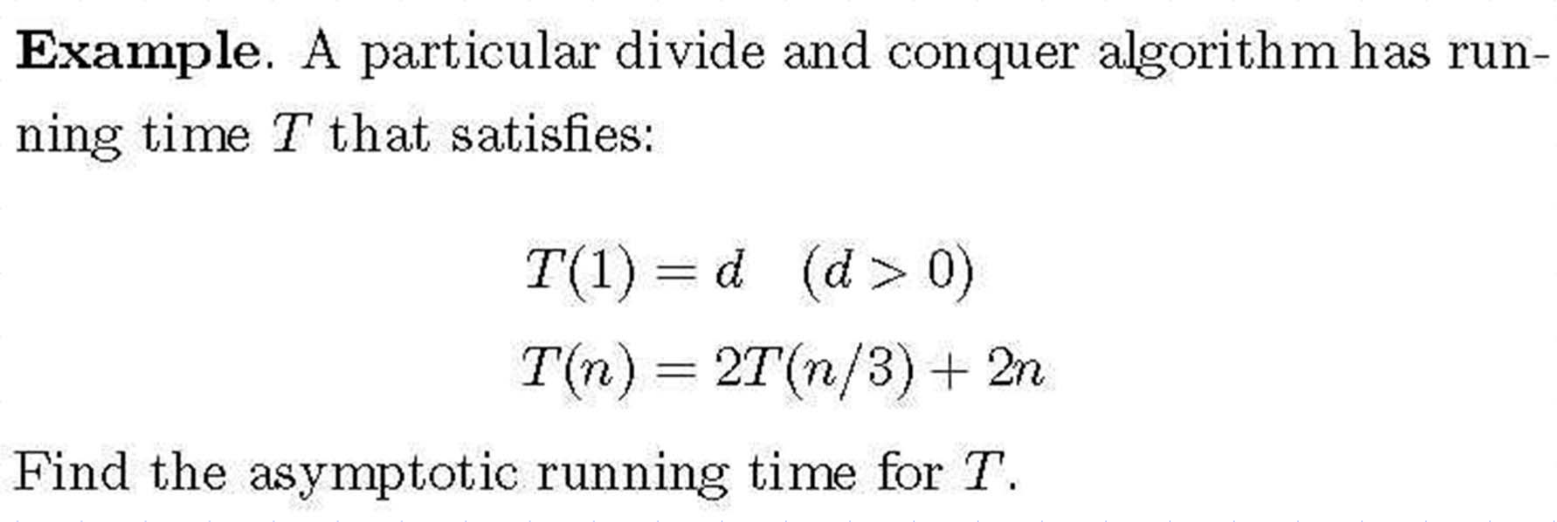


Notes.

(1) The result holds if $\lceil \frac{n}{b} \rceil$ is replaced by $\lfloor \frac{n}{b} \rfloor$.

(2) Whenever T satisfies this “divide-and-conquer” recurrence, it can be shown that the conclusion of the theorem holds for *all* natural number inputs, not just to powers of b.

Master Formula (continued)



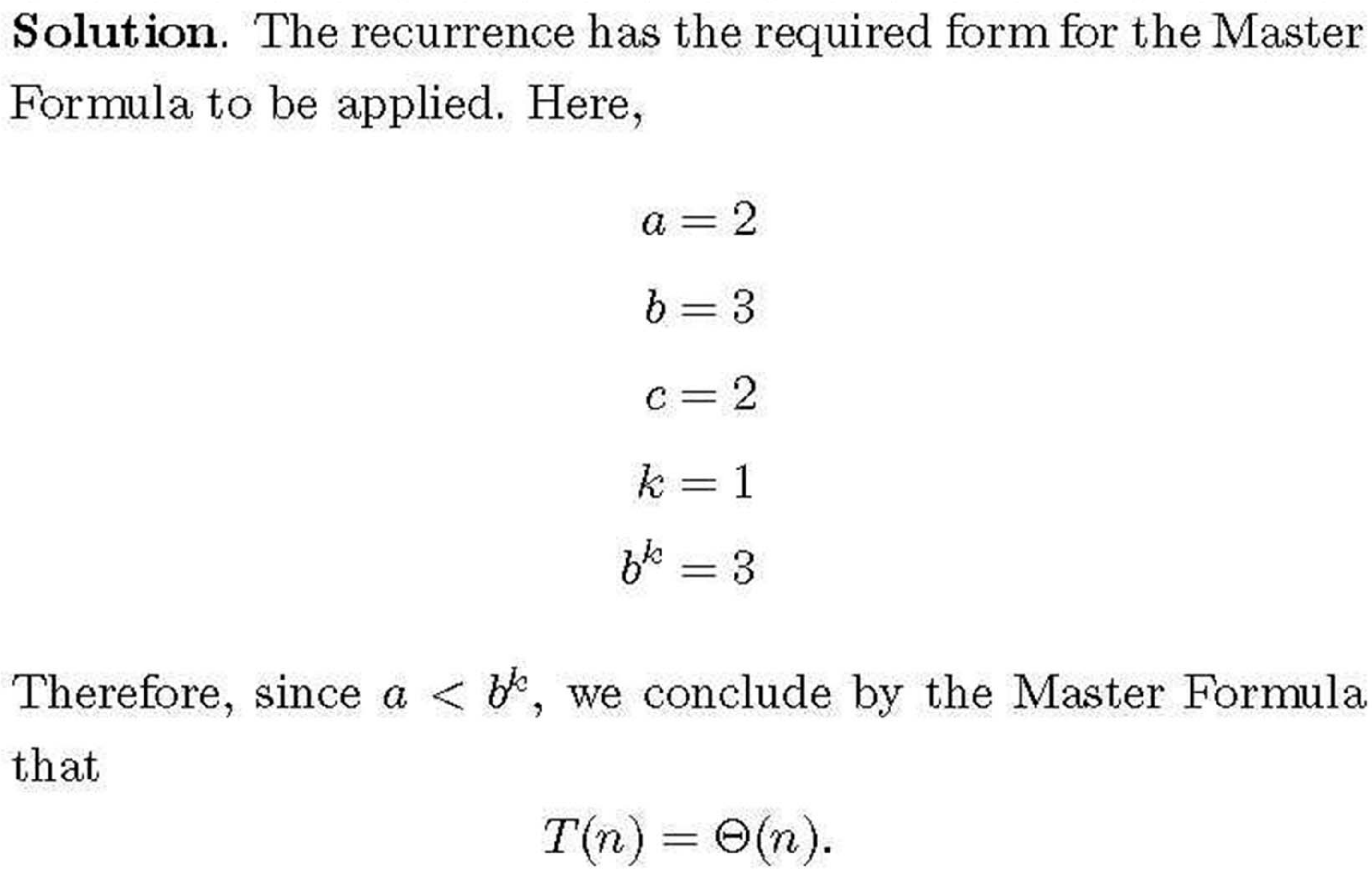
Example. A particular divide and conquer algorithm has running time T that satisfies:

$$T(1) = d \quad (d > 0)$$

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

Find the asymptotic running time for T.

Master Formula (continued)



Solution. The recurrence has the required form for the Master Formula to be applied. Here,

$$a = 2$$

$$b = 3$$

$$c = 2$$

$$k = 1$$

$$b^k = 3$$

Therefore, since $a < b^k$, we conclude by the Master Formula that

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

Connecting the Parts of Knowledge With The Wholeness of Knowledge

1. There are many techniques for analyzing the running time of a recursive algorithm. Most require special handling for special requirements (e.g. n not a power of 2, counting self-calls, verifying a guess).
2. The Master Formula combines all the intelligence required to handle analysis of a wide variety of recursive algorithm into a single simple formula, which, with minimal computation, produces the exact complexity class for algorithm at hand.

3. *Transcendental Consciousness* is the field beyond diversity, beyond problems, and therefore is the field of solutions.
4. *Impulses Within The Transcendental Field.* Impulses within this field naturally form the blueprint for unfoldment of the highly complex universe. This blueprint is called *Ved.*
5. *Wholeness Moving Within Itself.* In Unity Consciousness, solutions to problems arise naturally as expressions of one's own unbounded nature.