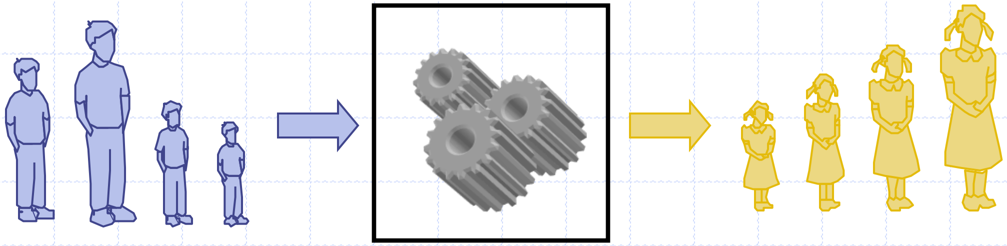
Lesson 2: Introduction to Analysis of Algorithms: *Discovering the Laws Governing Nature’s Computation*



Input Algorithm Output

Wholeness of the Lesson

An algorithm is a procedure for performing a computation or deriving an output from a given set of inputs according to a specified rule. By representing algorithms in a neutral language, it is possible to determine, in mathematical terms, the efficiency of an algorithm and whether one algorithm typically performs better than another. Efficiency of computation is the hallmark of Nature’s self-referral performance. Contact with the home of all the laws of nature at the source of thought results in action that is maximally efficient and less prone to error.

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?

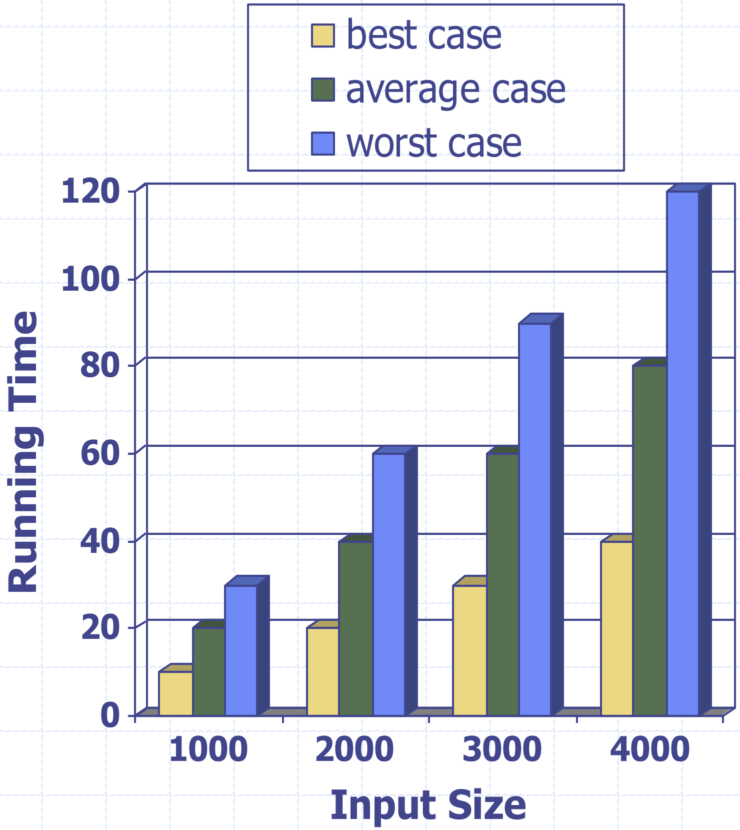
A Framework For Analysis of Algorithms

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

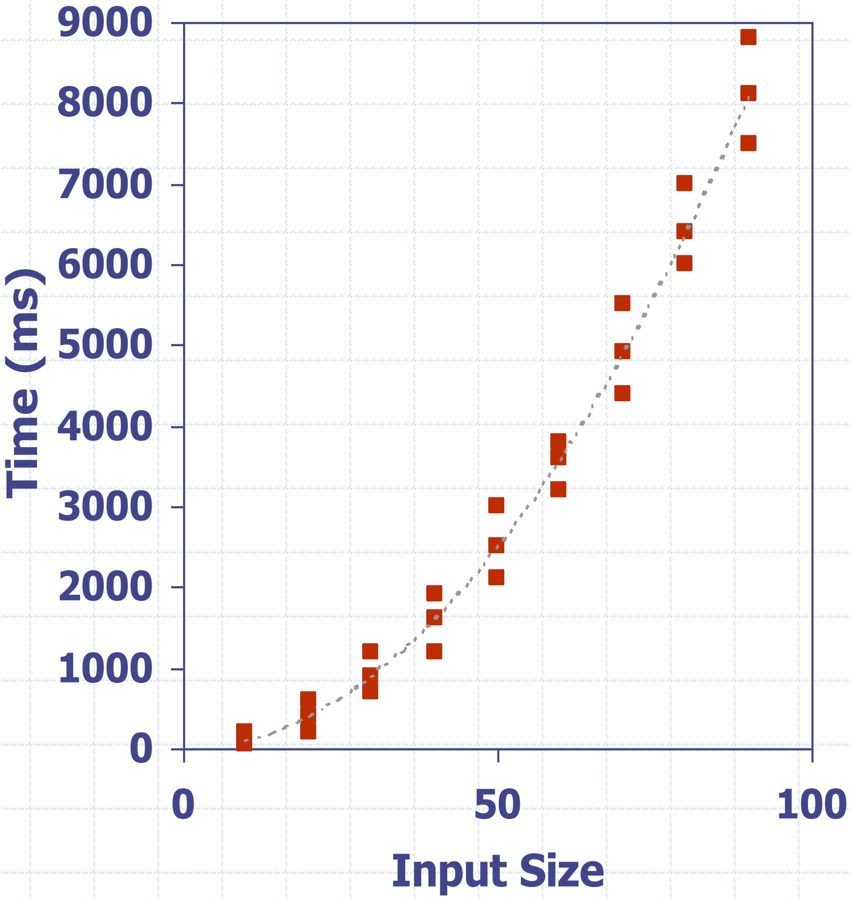
Running Time

- 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



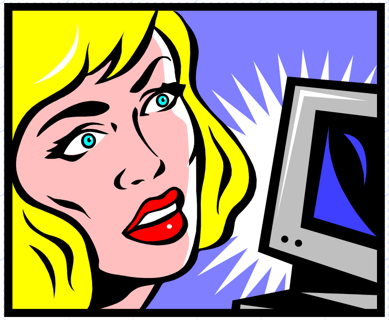
Running Time by Experiment

- 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

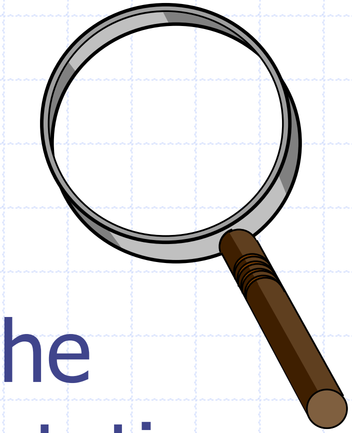


Limitations of Experiments

- 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 $A$

  $m \leftarrow n - 1$
  $currentMax \leftarrow A[0]$
  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

  ← 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

Return arr2

(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 principle that more abstract levels of intelligence are more comprehensive and unifying.

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

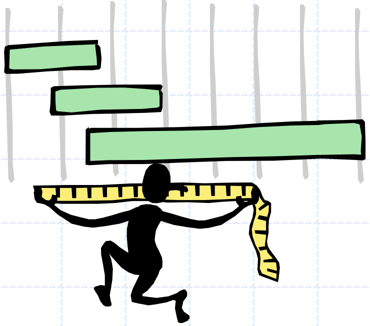
Counting PrimitiveOperations

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                                        $1 + n$
    //one assignment and m+1 comparisons (i = 1, ..., m+1.)
    //Note m + 1 = n. Thus 1 assignment and n comparisons.
        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 $7n$ primitive operations in the worst case. Define:

  a = Time taken by the fastest primitive operation

  b = Time taken by the slowest primitive operation

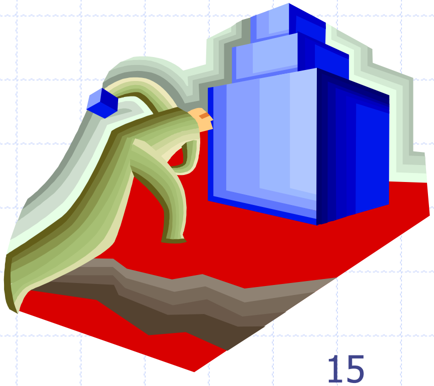
- Let $T(n)$ be worst-case time of ***arrayMax***. Then

$$a^* (7n) \leq T(n) \leq b^*(7n)$$

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

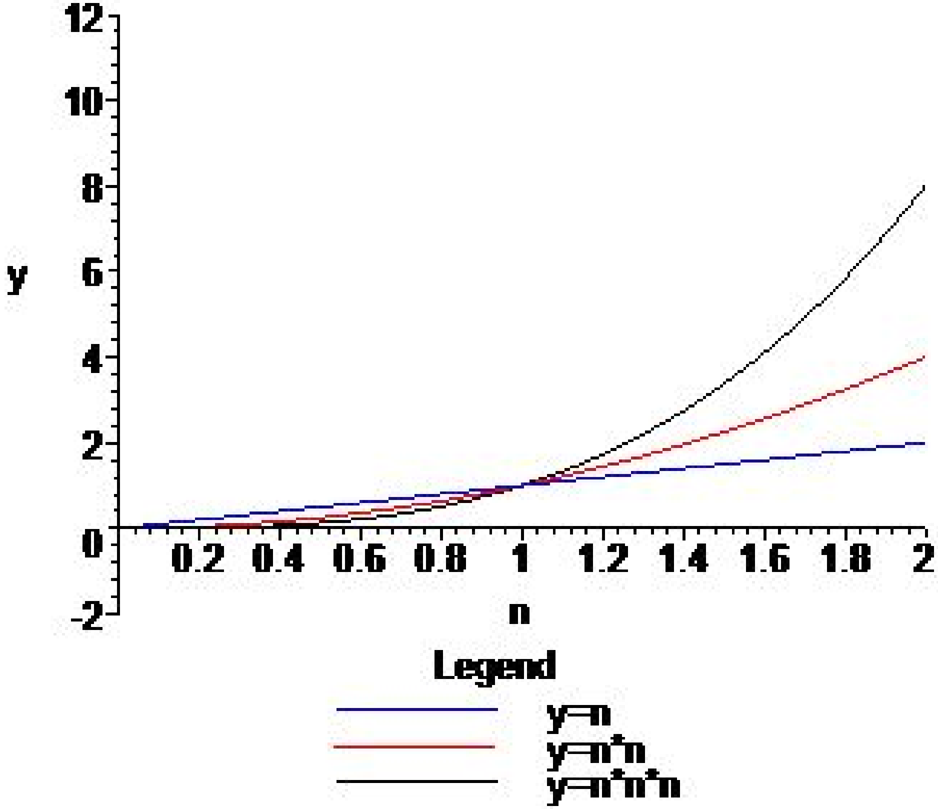
Growth Rate of Running Time

- 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*



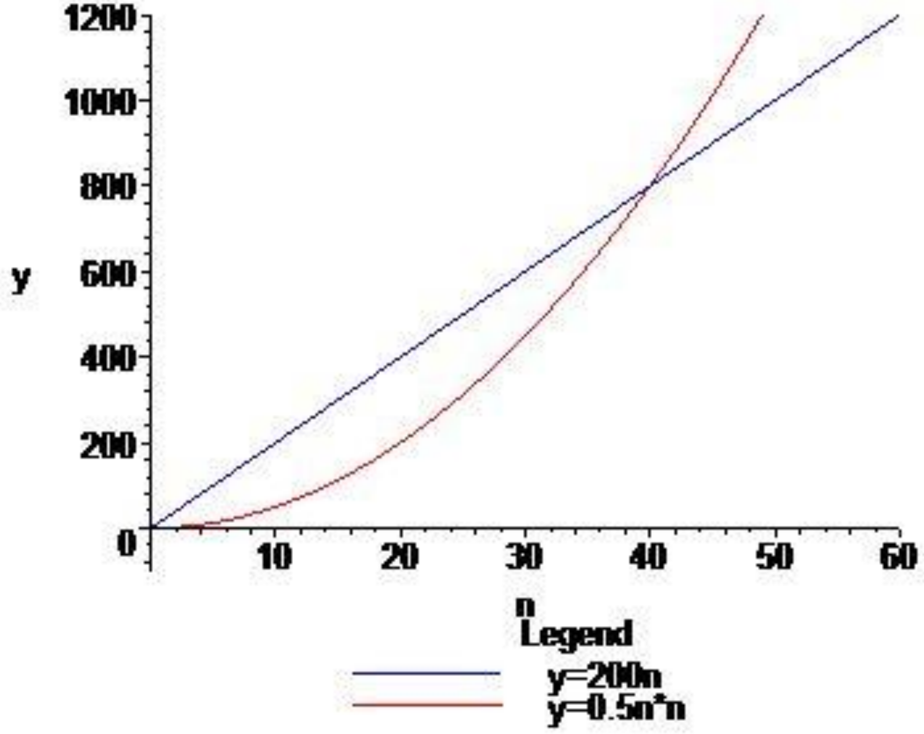
Growth Rates

- 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

  $f(n) \leq cg(n)$ for all $n \geq n_0$

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

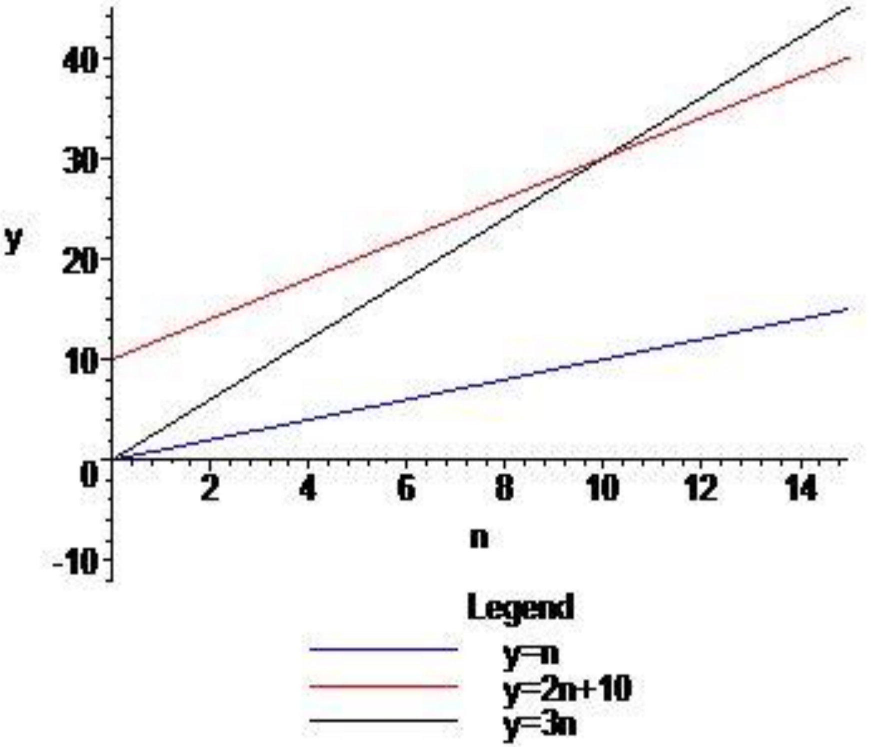
  f(n) = 2n + 10

  If g(n) = n, 3g(n) will eventually get bigger than f(n). We look for n_0, the point where the two graphs meet:

  3n = 2n + 10

  n = 10

  It follows that for all n ≥ 10, f(n) ≤ 3g(n)



Big-Oh Example

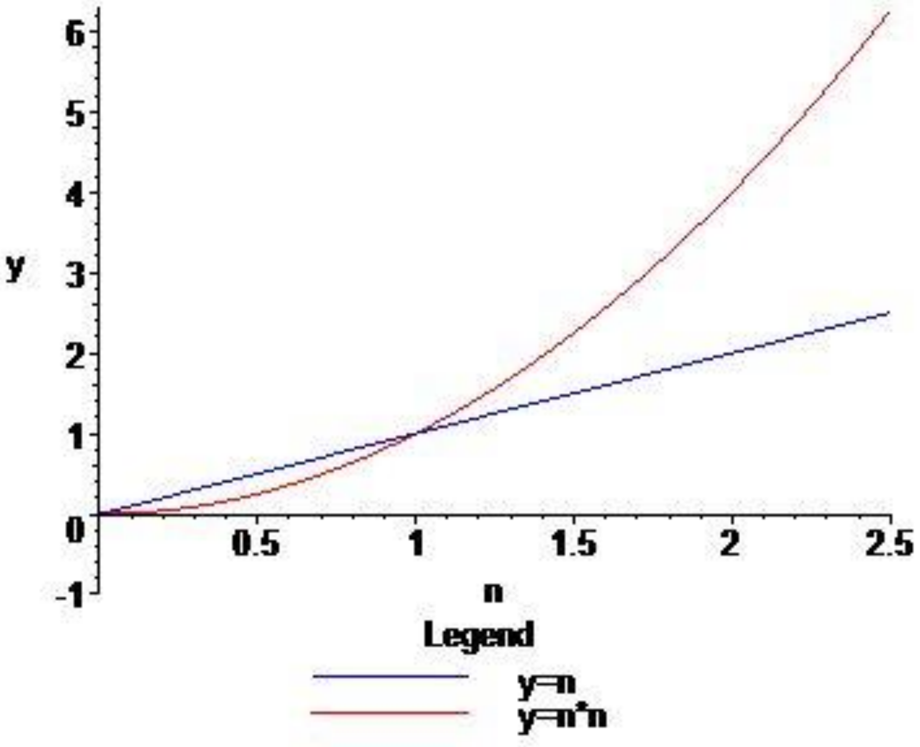
- Example: n^2 is not $O(n)$

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

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



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 $O(n^2)$

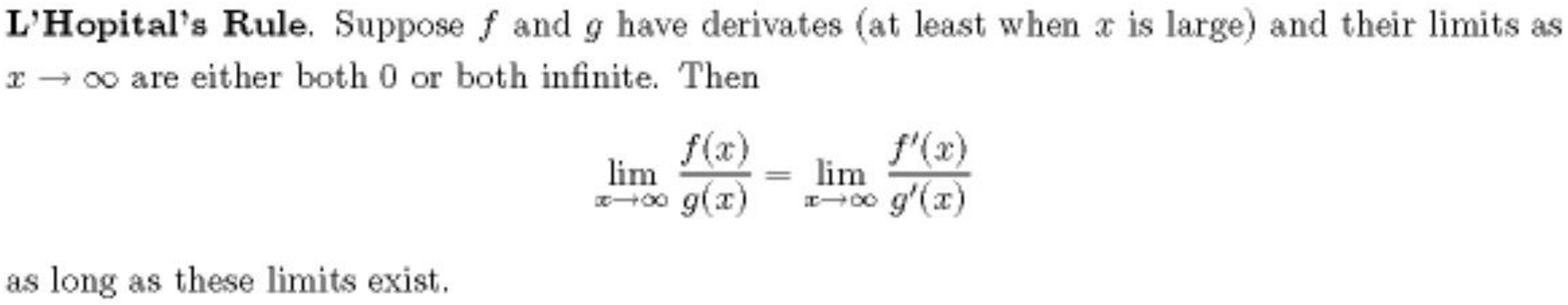
Standard Complexity Classes

- 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.

Big-Oh Rules

- 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

- 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 $7n$ primitive operations
  - Since $7n$ is $O(n)$, we say that algorithm ***arrayMax*** “runs in $O(n)$ 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 0..2n - 1, 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 –1 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 bound to 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]
        if W[u] = 0 then   //means a new value
            B[index] ← A[i]
            index ← index + 1
            W[u] ← 1

    //next: eliminate extra 0s 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)

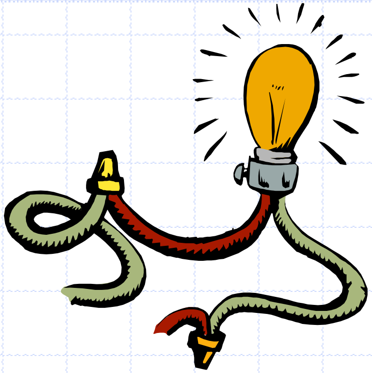
Main Point

One can improve the running time of the "remove duplicates" algorithm from O(n^2) to O(n) by introducing 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 $\Omega(g(n))$ if g(n) is O(f(n)).
- **big-Theta**
  - f(n) is $\Theta(g(n))$ if f(n) is both O(g(n)) and $\Omega(g(n))$.
- **little-oh**
  - f(n) is o(g(n)) if, for any constant c > 0, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$
  - In case $\lim_n(f(n)/g(n))$ exists,
    - f(n) is o(g(n)) if and only if the limit = 0.
    - f(n) is $\omega(g(n))$ if and only if the limit = ∞.
    - f(n) is $\Theta(g(n))$ if and only if the limit = c, a non-zero constant
    - f(n) is O(g(n)) if and only if limit exists and is finite.

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)

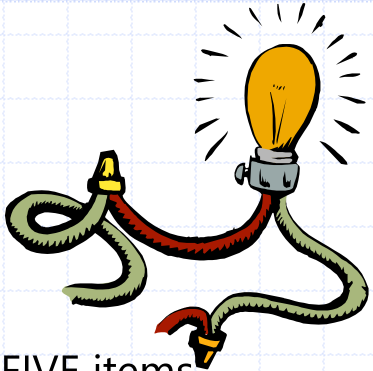
little-oh

- f(n) is o(g(n)) if f(n) is **asymptotically strictly less** than g(n)

little-omega

- f(n) is ω(g(n)) if f(n) is **asymptotically strictly greater** than g(n)

Intuition for Asymptotic Notation



A story for you. Today I went to Walmart and bought FIVE items.

Item	Price
1	7.89
2	9.99
3	6.29
4	8.56
5	9.21

Let us call the prices p1, p2, p3, p4 and p5.

We want to estimate amountSpent = p1 + p2 + p3 + p4 + p5

Case 1: Upper estimate

p1 <= 8. p2 <= 10. p3 <= 7. p4 <= 9. p5 <= 10.

amountSpent = p1 + p2 + p3 + p4 + p5 <= 8 + 10 + 7 + 9 + 10 = 44.

This an upper estimate. This is what we do in the case of Big-O

Case 2: Lower estimate

p1 >= 7. p2 >= 9. p3 >= 6. p4 >= 8. p5 >= 9.

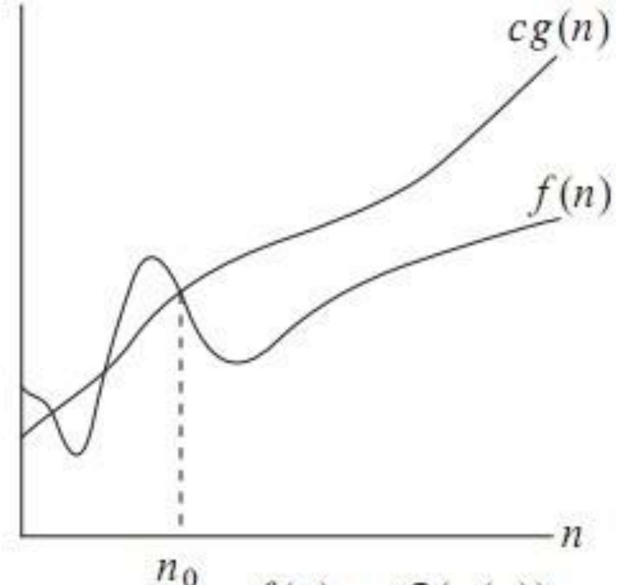
amountSpent = p1 + p2 + p3 + p4 + p5 >= 7 + 9 + 6 + 8 + 9 = 39.

This a lower estimate. This is what we do in the case of Big-Omega,

Big Oh (*O*)

f(n)= O(g(n)) *iff* there exist positive constants c and n_0 such that f(n) ≤ cg(n) for all n ≥ n_0

O-notation to give an upper bound on a function

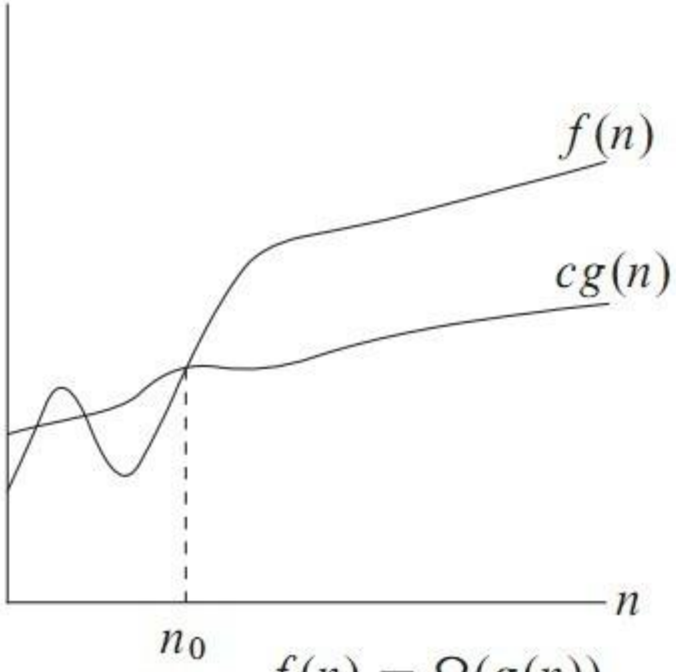


$f(n) = O(g(n))$

Omega Notation

Big oh provides an asymptotic upper bound on a function.
Omega provides an asymptotic lower bound on a function.

$$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \} .$$

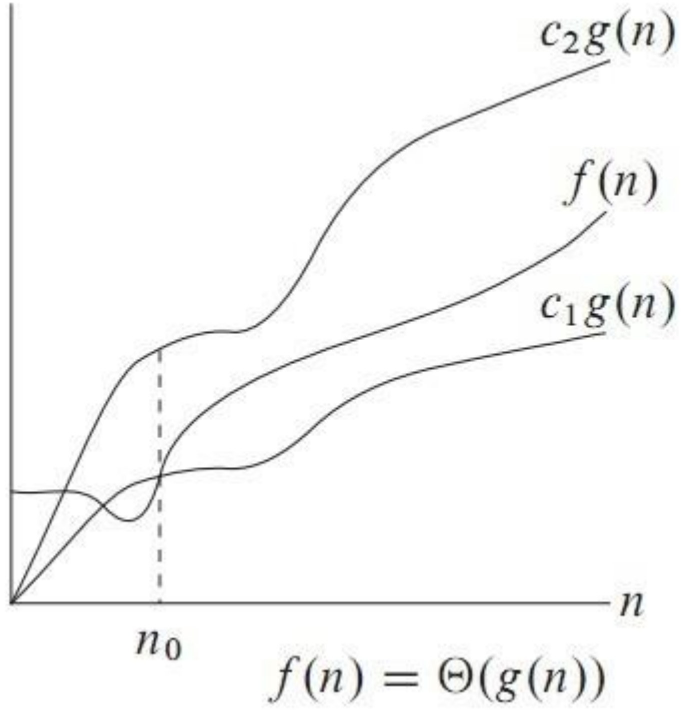


$f(n) = \Omega(g(n))$

Theta Notation

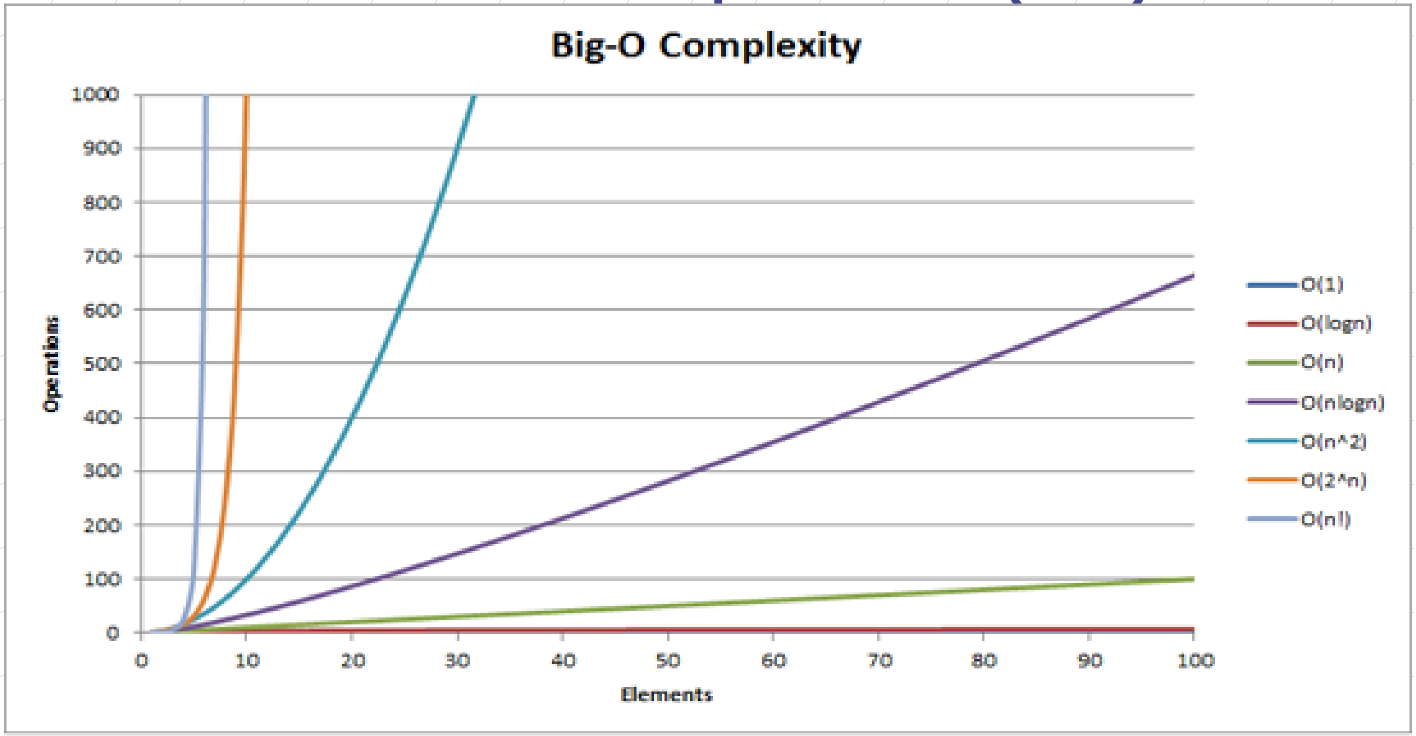
Theta notation is used when function *f* can be bounded both from above and below by the same function *g*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .^1$$



How bad is exponential complexity

- Fibonacci example – the recursive fib cannot even compute fib(50)



Running Time of Recursive Algorithms

- Problem: Given an array of integers in sorted order, is it possible to perform a search for an element in such a way that no more than half the elements of the array are examined? (Assume the array has 8 or more elements.)

Binary Search

```
**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 of size n, value x to be searched for in array section A[lower]..A[upper]

Output: true or false

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

For the worst case (x is above all elements of A and n a power of 2), running time is given by the ***Recurrence Relation:*** (In this case, right half is always half the size of the original.)

$$T(1) = d; \quad T(n) = c + T(n/2)$$

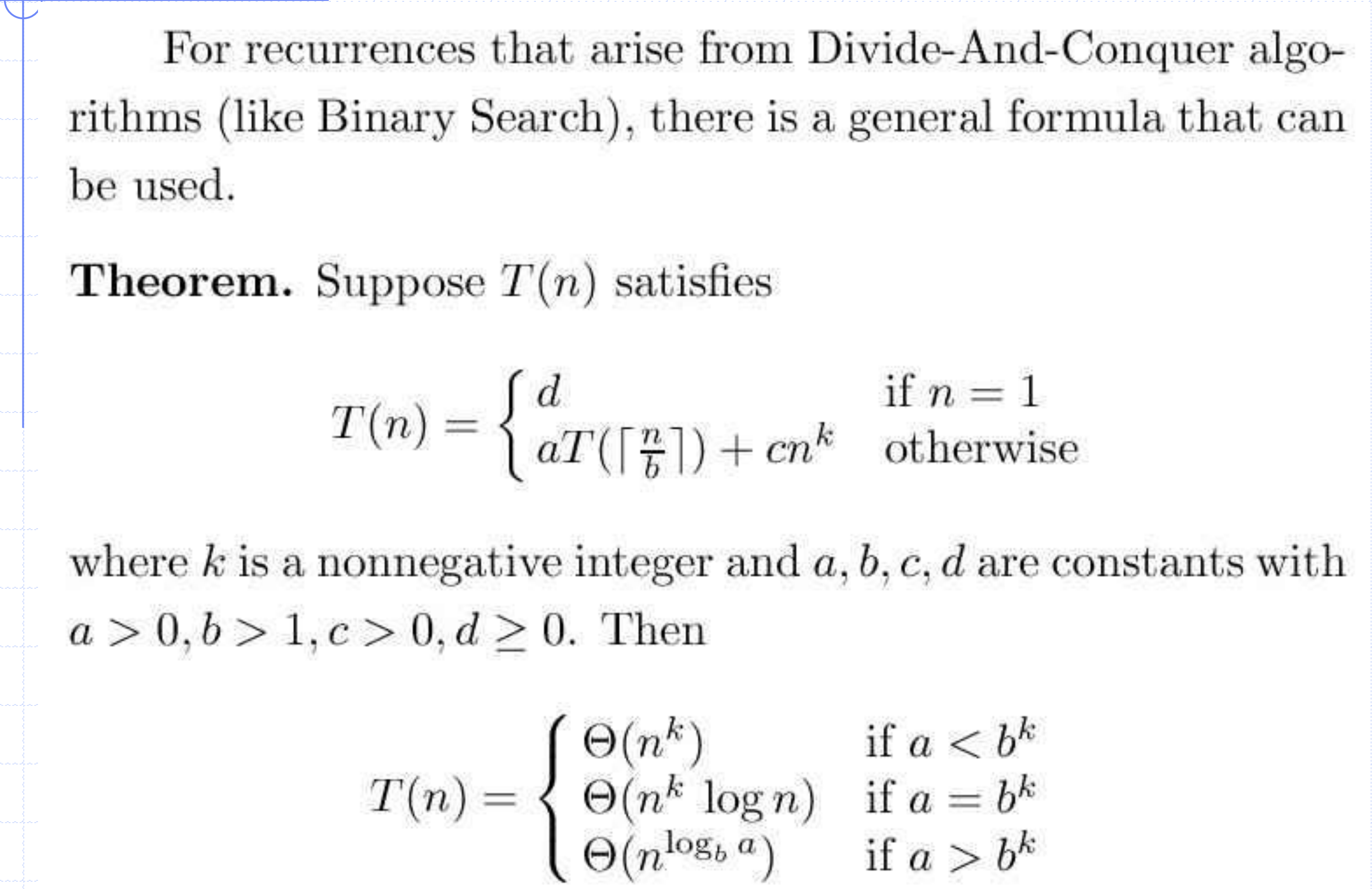
Solving A Recurrence Relation

$T(1) = d$
$T(2) = c + d$
$T(4) = c + c + d$
$T(8) = c + c + c + d$
$T(2^m) = c * m + d$
$T(n) = c * \log n + d$, which is $O(\log n)$

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)

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

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

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 at the basis of Nature’s functioning. Ultimately, there is just one field of existence; everything that happens therefore is just the dynamics of this one field interacting with itself. When individual awareness opens to this level of Nature’s functioning, great accomplishments become possible.

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 algorithms 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.