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 $currentMax$

Primitive Operations In This Course

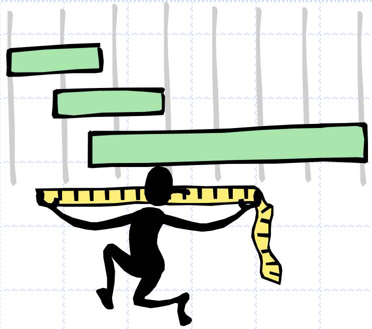
- Performing an arithmetic operation (+, *, etc)
- Comparing two numbers
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
- Following an object reference

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
$currentMax \leftarrow A[0]$	2
$m \leftarrow n - 1$	2
for $i \leftarrow 1$ **to** m **do**	$1 + n$
//one assignment and m+1 comparisons (i = 1, …, m+1.)	
//Note m + 1 = n. Thus, Thus 1 assignment and n coparisons.	
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ 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

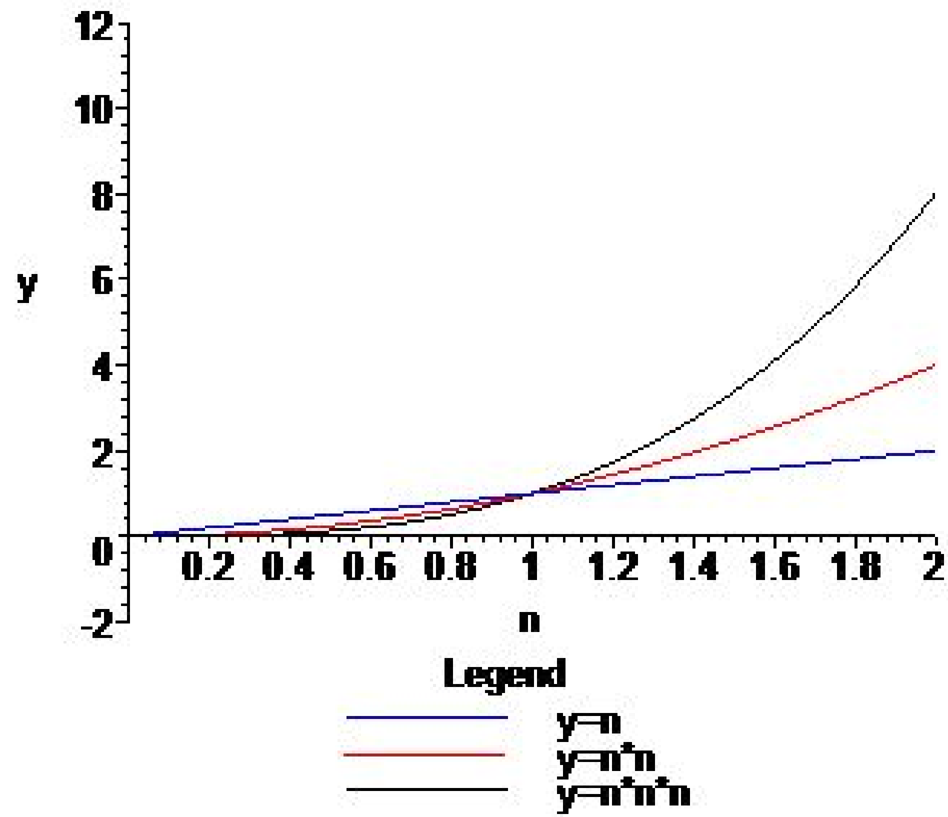
- Let $T(n)$ be worst-case time of ***arrayMax.*** Then

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

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

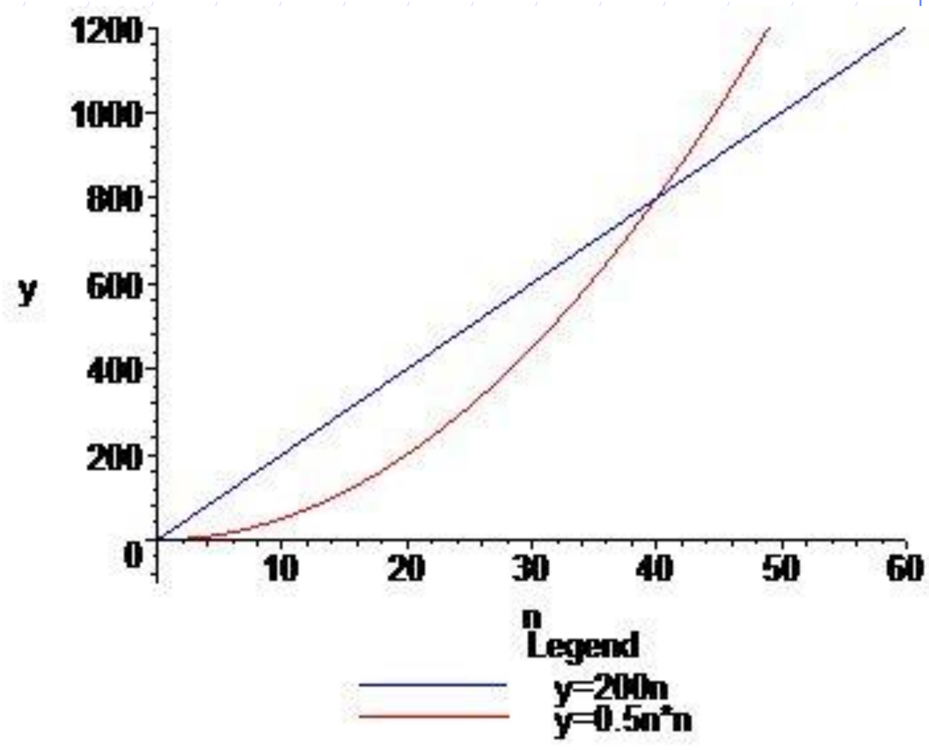
Growth Rates

- 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 $O(n)$

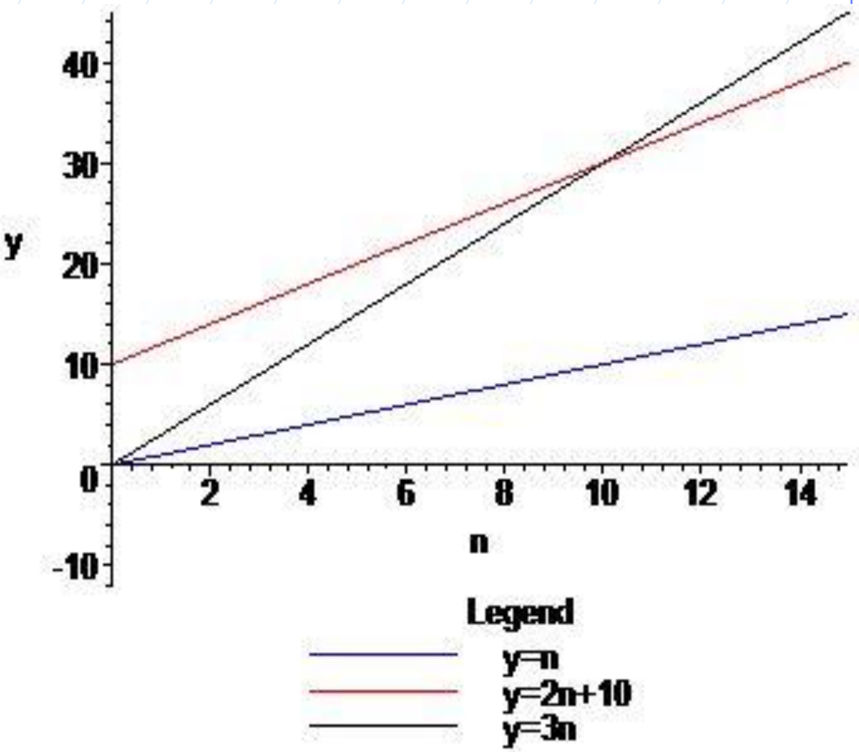
  f(n) = 2n + 10

  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 $n \geq 10$, $f(n) \leq 3g(n)$



Big-Oh Example

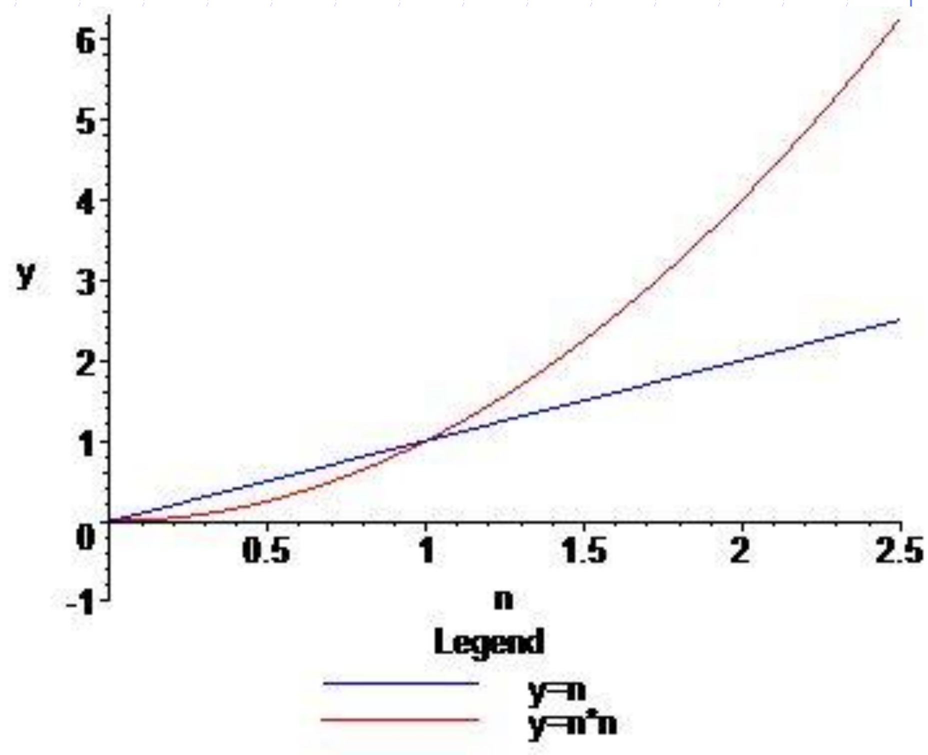
- Example: n^2 is not $O(n)$

Proof

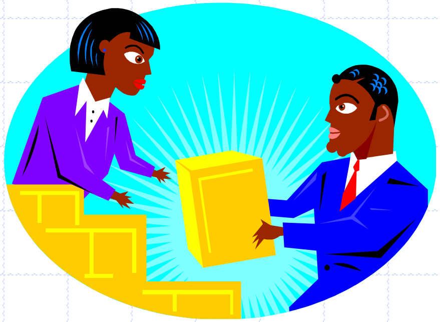
For each c and n_0, we need to find an $n \geq n_0$ such that $n^2 > cn$.

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

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



More Big-Oh Examples



- n is O(2n+1)
- nlog n + n is O(nlog n)
- Fact (for students who are familiar with "limits"):

If

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

exists and is finite then

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

- Example:

$$3n^2 + 1 \text{ is } O(2n^2 + n)$$

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.

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

=>

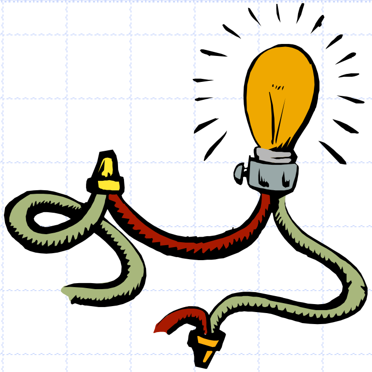
T(n) is 3 * O(n) = O(n)

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

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)).
- **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 ω(g(n)) if and only if the limit = ∞.
    - f(n) is Θ(g(n)) if and only if the limit = c, a non-zero constant

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)

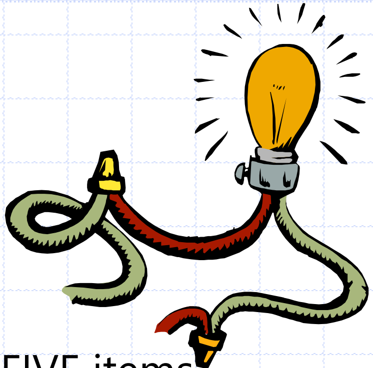
little-oh

- 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

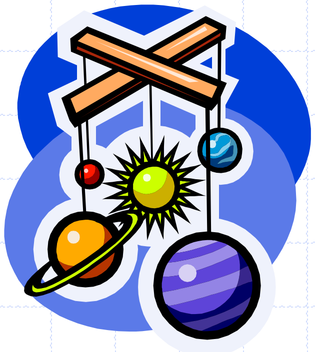
Case 2: Lower estimate

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,

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 $n \geq n_0$

  So $5n^2$ is $\Omega(n^2)$ iff n^2 is $O(5n^2)$, which is obviously true.

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

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

- **$5n$ is $o(n^2)$ but $5n$ is not $o(n)$**

Need to show that for any positive c, $5n \leq cn^2$ for large enough n. This inequality holds whenever $n \geq 5/c$.

Therefore, to prove that $5n$ is $o(n^2)$, given any positive c, pick n_0 bigger than 5/c. Then for all $n \geq n_0$, $5n \leq cn^2$

To show $5n$ is not $o(n)$, we must find positive c so that for every choice of n_0, there is an $n \geq n_0$ for which $5n > cn$.

This is obviously true: let $c = 1$, and given n_0, choose $n = n_0$.

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

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 S(n) of self-calls in fib(n) is ≥ F(n)

Proof. By (strong) induction on n.

Base Cases:

n=2. In this case S(2) = 2 ≥ 1 = F(2). Thus S(n) ≥ F(n).

n=3. In this case S(3) = 4 ≥ 2 = F(3). Thus S(n) ≥ F(n).

Induction Hypothesis:

Assume the result for all values of n in the interval [2, m].

Thus S(n) ≥ F(n) for 2 ≤ n ≤ m.

In particular, S(m) ≥ F(m) and S(m-1) ≥ F(m-1).

Induction Step:

$$
\begin{aligned}
S(m+1) &= 2 + S(m) + S(m - 1) \\
&\geq 2 + F(m) + F(m - 1) \quad \text{(by Induction Hypothesis)} \\
&\geq F(m + 1)
\end{aligned}
$$

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!

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

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