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 $O(3n)$"

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 $\leftarrow$ 0 to n-1 **do**

  for j $\leftarrow$ 0 to n-1 **do**

  k $\leftarrow$ 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

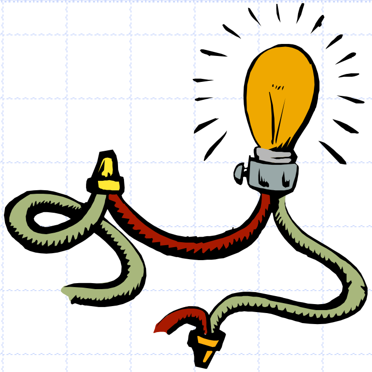
S2

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

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

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)

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 $\geq$ F(n)

Proof. By (strong) induction on n.

<u>Base Cases:</u>

n=2. In this case $S(2) = 2 \geq 1 = F(2)$. Thus $S(n) \geq F(n)$.

n=3. In this case $S(3) = 4 \geq 2 = F(3)$. Thus $S(n) \geq F(n)$.

<u>Induction Hypothesis:</u>

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

Thus $S(n) \geq F(n)$ for $2 \leq n \leq m$.

In particular, $S(m) \geq F(m)$ and $S(m-1) \geq F(m-1)$.

<u>Induction Step:</u>

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

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

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