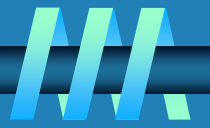
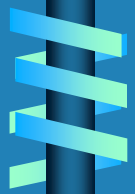
Analysis of algorithms

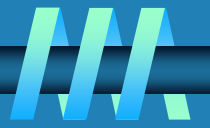


- **Issues:**
  - correctness
  - time efficiency
  - space efficiency
  - optimality

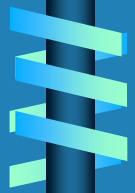
- **Approaches:**
  - empirical analysis – less useful
  - theoretical analysis – most important



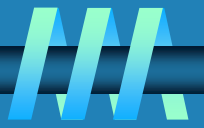
Empirical analysis of time efficiency



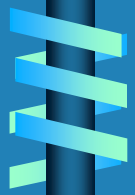
- Select a specific sample of inputs
- Use physical unit of time (e.g., milliseconds) or
  Count actual number of basic operation's executions
- Analyze the empirical data
- Problems:



Empirical analysis of time efficiency



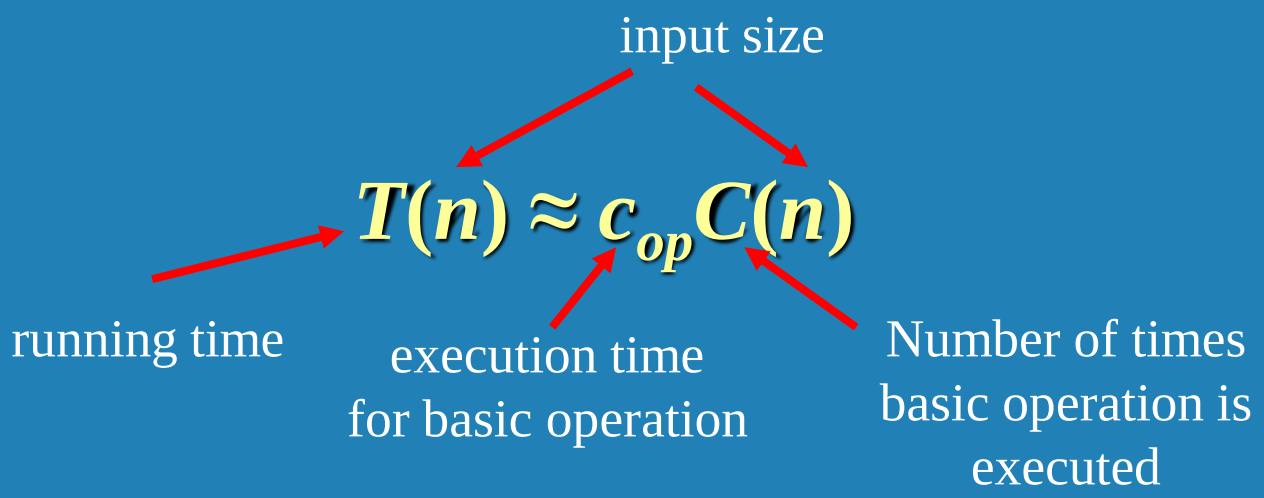
- Select a specific sample of inputs
- Use physical unit of time (e.g., milliseconds) or
  Count actual number of basic operation's executions
- Analyze the empirical data
- Problem - Inefficient:
  - Must implement algorithm
  - Must run on many data sets to see effects of scaling
  - Hard to see patterns in actual data



Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the *basic operation* as a function of *input size*

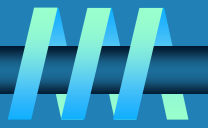
- ***Basic operation*: the operation that contributes most towards the running time of the algorithm**



Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

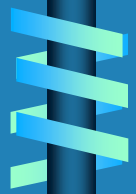
Counting Operations



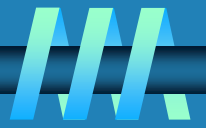
Consider counting steps in FindMax

- **Problems:**
  - **Hard to analyze**
  - **May not need precise information**
    - **Precise details less relevant than order growth**
    - **May not know times (or relative times) of steps**
    - **Only gives results within constants (constants relevant later)**
      - $aC(n) < T(n) < bC(n)$

- **More interested in growth rates (ie Big O), but**
- **Careful analysis can compare algs with same growth rate**
  - **If needed**
  - **Knuth examples**



Best-case, average-case, worst-case



For a given input size, how does algorithm perform on different datasets of that size

For datasets of size *n* identify different datasets that give:

- Worst case: $C_{worst}(n)$ – maximum over all inputs of size *n*
- Best case: $C_{best}(n)$ – minimum over all inputs of size *n*

- Average case: $C_{avg}(n)$ – "average" over inputs of size *n*
  - Typical input, NOT the average of worst and best case
  - Aanalysis requires knowing distribution of all possible inputs of size n
  - Can consider ALL POSSIBLE input sets of size n, average over all sets

- Some algs are same for all three (eg all case performance)

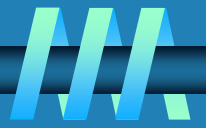
Example: Sequential search

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

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- **Worst case**
- **Best case**
- **Average case: depends on assumputions about input: ?**

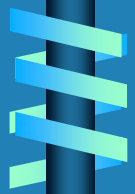
Example: Sequential search



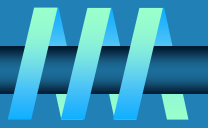
ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

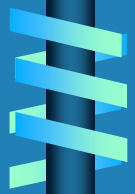
- **Worst case**
- **Best case**
- **Average case: depends on assumputions about input: proportion of found vs not-found keys**



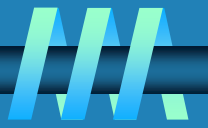
Example: Find maximum



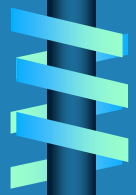
- **Worst case:**
- **Best case:**
- **Average case:**
- **All case:**



Critical factor for analysis: Growth rate



- **Most important: *Order of growth* as $n \to \infty$**
  - **What is the growth rate of time as input size increases**
  - **How does time increase as input size increases**



Growth rate: critical for performance performance

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

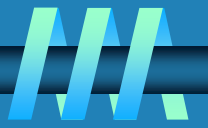
Table 2.1 Values (some approximate) of several functions important for analysis of algorithms

Focus: asymptotic order of growth:

Main concern: which function describes behavior.

Less concerned with constants

Asymptotic order of growth



Critical factor for problem size *n*:

- IS NOT the exact number of basic ops executed for given *n*

- IS how number of basic ops GROWS as *n* increases

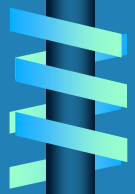
- Constant factors and constants do not change growth RATE

- Rate most relevant for large input sizes, so ignore small sizes

- Informally: 5*n*^2 and 100*n*^2 +1000 are both *n*^2

- Define formally later

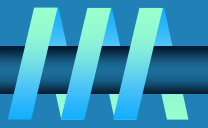
Call this: Asymptotic Order of Growth



Basic asymptotic efficiency classes

1	constant	Best case
log *n*	logarithmic	Divide Ignore part
n	linear	Examine each Online/Stream Algs
n log *n*	*n*-log-*n* or linearithmic	Divide Use all parts
n^2	quadratic	Nested loops
n^3 n^k	cubic	Nested loops Examine all *k*-tuples
2^n	exponential	All subsets
n!	factorial	All permutations

Order of growth: Sets of functions



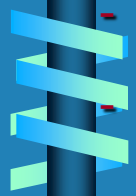
Approach: Group functions based on their *growth rates*

Example: Θ(*n*^2) is set of functions whose growth rate is *n*^2
(ie group all functions with *n*^2 growth rate)

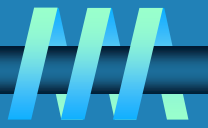
These are all in Θ(*n*^2):

- ***f*(*n*) = 100n^2 + 1000**
- ***f*(*n*) = n^2 + 1**
- ***f*(*n*) = 0.001n^2 + 1000000**

- **They are also Θ(10*n*^2), but we rarely say that**



Order of growth: Upper, tight, lower bounds



More formally:

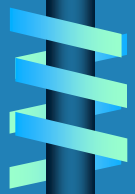
- $\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$

Upper, tight, and lower bounds on performance:

- **$O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$**
  - **[ie f 's speed is same as or faster than g, f bounded above by g]**

- **$\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$**

- **$\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$**
  - **[ie f 's speed is same as or slower than g, f bounded below by g]**



t(n) ∈ O(g(n)) iff t(n) <=cg(n) for n > n_0

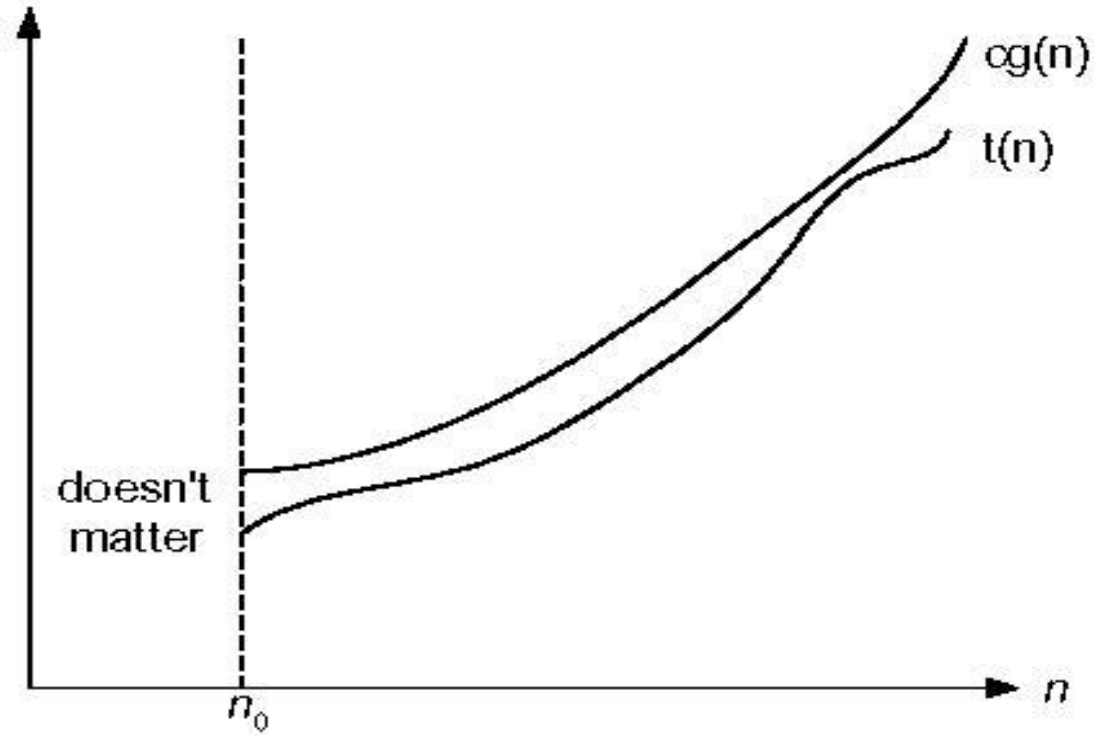


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

t(n) = 10n^3 in $O(n^3)$ and in $O(n^5)$. What c and n0? More later.

$t(n) \in \Omega(g(n))$ iff $t(n) >= cg(n)$ for $n > n_0$

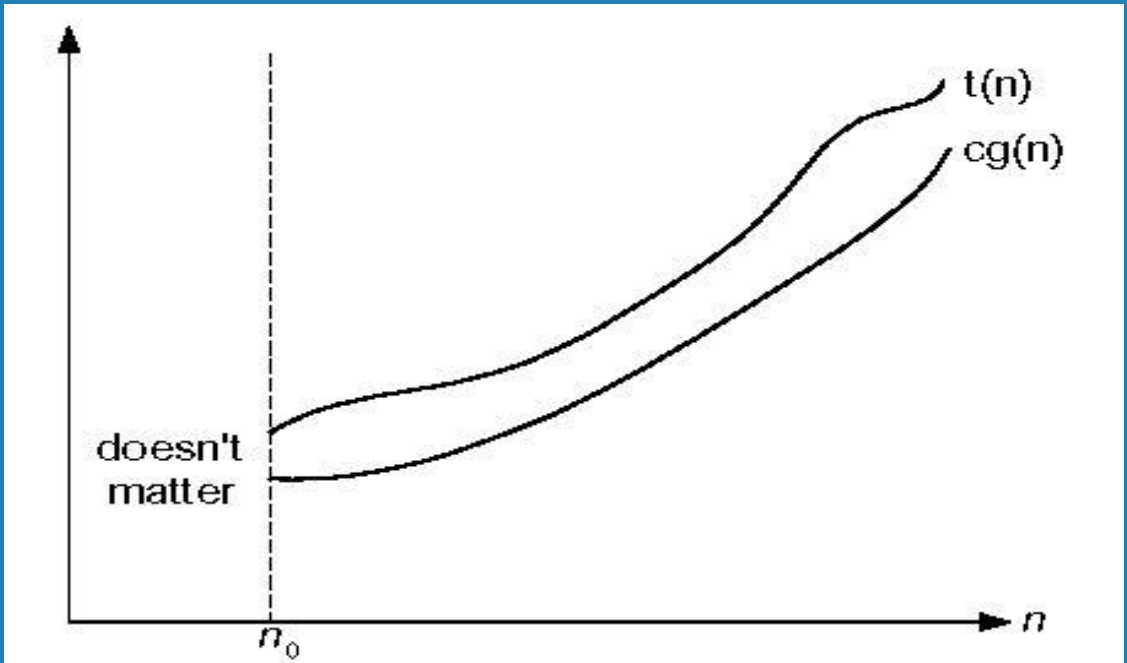


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

$t(n) = 10n^3$ in $\Omega(n^2)$ and in $\Omega(n^3)$

$t(n) \in \Theta(g(n))$ iff $t(n) \in O(g(n))$ and $\in \Omega(g(n))$

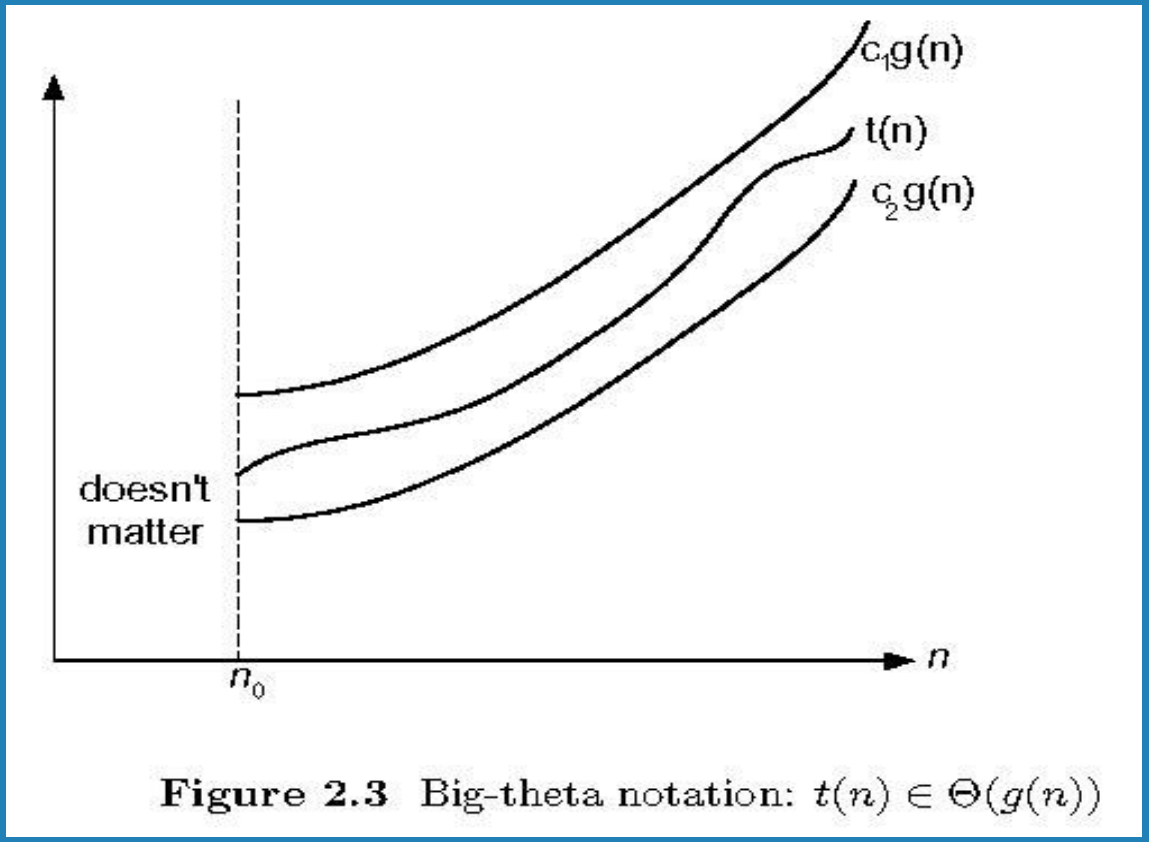
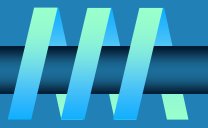


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

$t(n) = 10n^3$ in $\Theta(n^3)$ but NOT in $\Theta(n^2)$ or $\Theta(n^4)$

Terminology



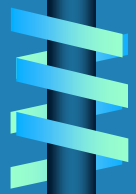
Informally: f(n) is O(g) means f(n) ∈ O(g)

Example: 10*n* is O(n^2)

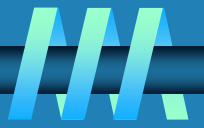
Informally: f(n) = O(g) means f(n) ∈ O(g)

Example: 10*n* = O(n^2)

Use similar terms for Big Omega and Big Theta



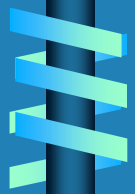
Big O, Ω, Θ : Informal Definitions



$f(n)$ is in $O(g(n))$ if order of growth of $f(n) \leq$ order of growth of $g(n)$ (within constant multiple and for large n).

$f(n)$ is in $\Omega(g(n))$ if order of growth of $f(n) \geq$ order of growth of $g(n)$ (within constant multiple and for large n).

$f(n)$ is in $\Theta(g(n))$ if order of growth of $f(n) =$ order of growth of $g(n)$ (within constant multiples and for large n).



Big O, Ω, Θ : Formal Definitions

$f(n) \in O(g(n))$ iff there exist positive constant c and non-negative integer n_0 such that

$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

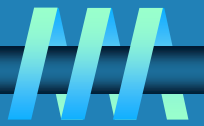
$f(n) \in \Omega(g(n))$ iff there exist positive constant c and non-negative integer n_0 such that

$$f(n) \geq c\, g(n) \text{ for every } n \geq n_0$$

$f(n) \in \Theta(g(n))$ iff there exist positive constants c_1 and c_2 and non-negative integer n_0 such that

$$c_1\, g(n) \leq f(n) \leq c_2\, g(n) \text{ for every } n \geq n_0$$

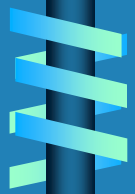
Big O, Ω, Θ: Why Constant c



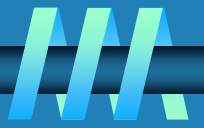
What is gained by the constant c:

$f(n) \leq c\, g(n)$ **for every** $n \geq n_0$

What do you think?



Big O, Ω, Θ: Why Constant c



What is gained by the constant c:

$f(n) \leq c\, g(n)$ **for every** $n \geq n_0$

What do you think?

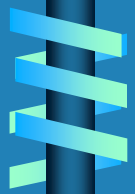
Allows $c\, g(n)$ **to exceed** $f(n)$

Allows us to ignore leading constants in $f(n)$

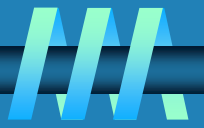
For example, 0.1n^2, n^2, and 10n^2

All have same growth rate: n^2

Meaning what: as input size doubles, time is 4x increase



Using the Definition: Big O



Informal Definition: $f(n)$ **is in** $O(g(n))$ **if order of growth of** $f(n) \leq$ **order of growth of** $g(n)$ **(within constant multiple),**

Definition: $f(n) \in O(g(n))$ **iff there exist positive constant** c **and non-negative integer** n_0 **such that**

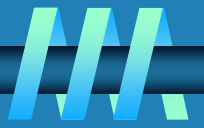
$$f(n) \leq c\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n$ **is** $O(n^2)$
  - **[Can choose** c **and** n_0**. Solve for 2 different** c**'s]**

- $5n + 20$ **is** $O(n)$ **[Solve for** c **and** n_0**]**

Using the Definition: Big Omega



Informal Definition: $f(n)$ **is in** $\Omega(g(n))$ **iff order of growth of** $f(n) \geq$ **order of growth of** $g(n)$ **(within constant multiple),**

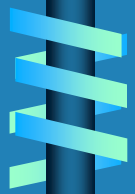
Definition: $f(n) \in \Omega(g(n))$ **iff there exist positive constant** c **and non-negative integer** n_0 **such that**

$$f(n) \geq c\, g(n) \text{ for every } n \geq n_0$$

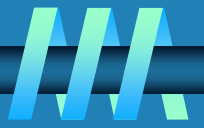
Examples:

- $10n^2$ **is** $\Omega(n)$

- $5n + 20$ **is** $\Omega(n)$



Using the Definition: Theta



Informal Definition: $f(n)$ **is in** $\Theta(g(n))$ **if order of growth of** $f(n)$ **= order of growth of** $g(n)$ **(within constant multiple),**

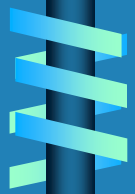
Definition: $f(n) \in \Theta(g(n))$ **iff there exist positive constants** c_1 **and** c_2 **and non-negative integer** n_0 **such that**

$$c_1\, g(n) \leq f(n) \leq c_2\, g(n) \text{ for every } n \geq n_0$$

Examples:

- $10n^2$ **is** $\Theta(n^2)$

- $5n + 20$ **is** $\Theta(n)$



Some properties of asymptotic order of growth

- $f(n) \in O(f(n))$
- $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$
- If $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$

  Note similarity with $a \leq b$
- If $f_1(n) \in O(g_1(n))$ and $f_2(n) \in O(g_2(n))$, then

  $f_1(n) + f_2(n) \in O(\max\{g_1(n), g_2(n)\})$

  $O(g_1(n) + g_2(n))$?

Establishing order of growth using limits

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

Examples:

- $10n$ vs. n^2

- $n(n+1)/2$ vs. n^2

Big O, Little o. Big and little Omega

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c > 0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

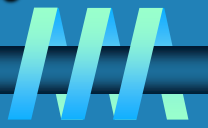
Examples:

- **$10n$ vs. n^2**
- 10n is O(n^2) and o(n^2)
- 10n is O(n) but not o(n)

Big O: upper bound or the same

•Little o: upper bound and not the same

L'Hôpital's rule and Stirling's formula



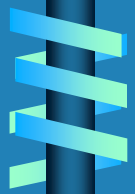
L'Hôpital's rule: If $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

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

Example: $\log n$ vs. n

Stirling's formula: $n! \approx (2\pi n)^{1/2} (n/e)^n$

Example: 2^n vs. $n!$



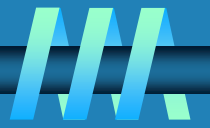
Orders of growth of some important functions

- All polynomials of the same degree k belong to the same class: $a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)$ [Why?]
- All logarithmic functions $\log_a n$ belong to the same class $\Theta(\log n)$ no matter what the logarithm's base $a > 1$ is [Why?]
- Exponential functions a^n have different orders of growth for different a's
  - *Frequently just say Exponential time, ignoring exponent*
- order $\log n$ < order n^α ($\alpha>0$) < order a^n < order $n!$ < order n
  - Order $n \log n$?

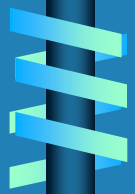
Basic asymptotic efficiency classes

1	constant	Best case
log *n*	logarithmic	Divide Ignore part
n	linear	Examine each Online/Stream Algs
n log *n*	*n*-log-*n* or linearithmic	Divide Use all parts
n^2	quadratic	Nested loops
n^3 n^k	cubic	Nested loops Examine all *k*-tuples
2^n	exponential	All subsets
n!	factorial	All permutations

Polynomial=Efficient. Exponential=Not Efficient



- In general, we say that
  - Functions with polynomial growth are efficient
  - Functions with exponential growth are NOT efficient
- But, what about n^100 vs 2^{0.2 log n}
  - Yes, exponential is faster for all reasonable n
  - But, such algorithms don't happen in practice
- In practice, polynomials are faster than exponential



Time efficiency of nonrecursive algorithms

General Plan for Analysis

- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

Useful summation formulas and rules

$\Sigma_{l \le i \le u} 1 = 1+1+ \cdots +1 = u - l + 1$

In particular, $\Sigma_{1 \le i \le u} 1 = n - 1 + 1 = n \in \Theta(n)$

$\Sigma_{1 \le i \le n} i = 1+2+ \cdots +n = n(n+1)/2 \approx n^2/2 \in \Theta(n^2)$

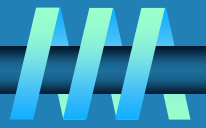
$\Sigma_{1 \le i \le n} i^2 = 1^2+2^2+ \cdots +n^2 = n(n+1)(2n+1)/6 \approx n^3/3 \in \Theta(n^3)$

$\Sigma_{0 \le i \le n} a^i = 1 + a + \cdots + a^n = (a^{n+1} - 1)/(a - 1)$ for any $a \neq 1$

In particular, $\Sigma_{0 \le i \le n} 2^i = 2^0 + 2^1 + \cdots + 2^n = 2^{n+1} - 1 \in \Theta(2^n)$

$\Sigma(a_i \pm b_i) = \Sigma a_i \pm \Sigma b_i \qquad \Sigma c a_i = c \Sigma a_i \qquad \Sigma_{l \le i \le u} a_i = \Sigma_{l \le i \le m} a_i + \Sigma_{m+1 \le i \le u} a_i$

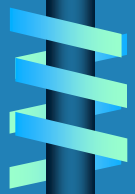
Example: Sequential search



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

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

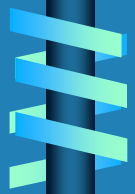
- **Worst case: Omega, Theta, O?**
- **Best case: Omega, Theta, O?**
- **Average case: Omega, Theta, O?**



Example 1: Maximum element

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 ← A[0]
for i ← 1 to n − 1 do
    if A[i] > maxval
        maxval ← A[i]
return maxval
```



Example 2: Element uniqueness problem

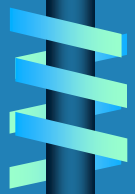
ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//                and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

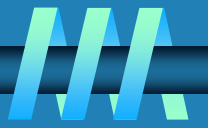
Example 3: Matrix multiplication

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C



Example 4: Gaussian elimination



Algorithm *GaussianElimination*($A[0..n\text{-}1,0..n]$)

//Implements Gaussian elimination of an n-by-$(n+1)$ matrix $\boldsymbol{A}$

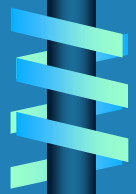
for $i \leftarrow 0$ **to** $n - 2$ **do**

  for $j \leftarrow i + 1$ **to** $n - 1$ **do**

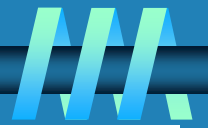
    for $k \leftarrow i$ **to** n **do**

      $A[j,k] \leftarrow A[j,k] - A[i,k] * A[j,i] / A[i,i]$

Find the efficiency class and a constant factor improvement.



Example 5: Counting binary digits



ALGORITHM *Binary*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

$count \leftarrow 1$

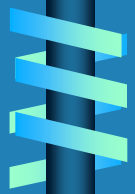
while $n > 1$ **do**

    $count \leftarrow count + 1$

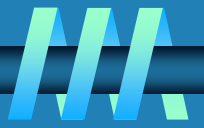
    $n \leftarrow \lfloor n/2 \rfloor$

return *count*

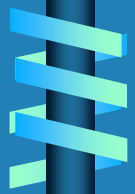
?



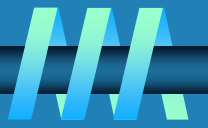
Plan for Analysis of Recursive Algorithms



- Decide on a parameter indicating an input's size.
- Identify the algorithm's basic operation.
- Check whether the number of times the basic op. is executed may vary on different inputs of the same size. (If it may, the worst, average, and best cases must be investigated separately.)
- Set up a **recurrence relation** with an appropriate **initial condition** expressing the number of times the basic op. is executed.
- **Solve** the recurrence (ie find a closed form or, at the very least, establish its solution's order of growth) by backward substitutions or another method.



Recurrences



Terminology (used interchangably):

- **Recurrence**
- **Recurrence Relation**
- **Recurrence Equation**

Express T(n) in terms of T(smaller n)

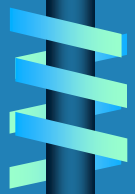
- **Example: $M(n) = M(n-1) + 1,\ M(0) = 0$**

Solve: find a closed form:

- **ie T(n) in terms of n, alone (ie not T)**

Learn: 1. how to describe an alg w/ a recurrence

2. how to solve a recurrence (ie find a closed form)



Example 1: Recursive evaluation of $n!$

Definition: $n! = 1 \cdot 2 \cdot \ldots \cdot (n\text{-}1) \cdot n$ for $n \geq 1$ and $0! = 1$

Recursive definition of $n!$: $F(n) = F(n\text{-}1) \cdot n$ for $n \geq 1$ and $F(0) = 1$

ALGORITHM $F(n)$

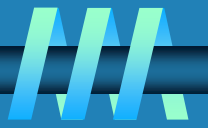
```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) ∗ n
```

Size:

Basic operation:

Recurrence relation:

Solving the recurrence for M(n)



Recurrence: M(n) = M(n-1) + 1,

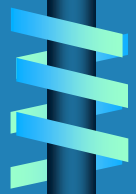
Initial Condition: M(0) = 0

Solution Methods:

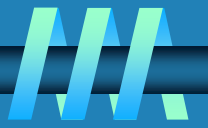
- **Guess?**
- **Forward Substitution?**
- **Backward Substitution?**
- **General Methods?**

Guess closed form: ?

How to check a possible solution?



Check a possible solution with Substitution



Recurrence: M(*n*) = M(*n*-1) + 1

Initial Condition: M(0) = 0

Possible solution: M(n) = n. Substitute:

n=0: M(0) = n = 0

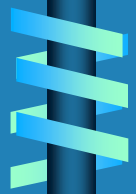
n = k > 0: M(k) = M(k-1) + 1

k = (k-1) + 1

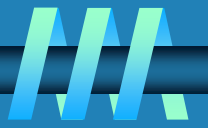
= k (Correct)

Or: M(k) = M(k-1) + 1 = (k-1) + 1 = k. (Correct)

N.B. This is actually a hand-wavy proof by induction



Recurrences, Initial Conditions, Solutions



$M(n) = M(n-1) + 1, \quad M(0) = 1$

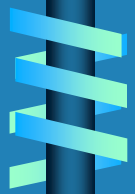
M(n) = ??

A Recurrence has an infinite number of solutions

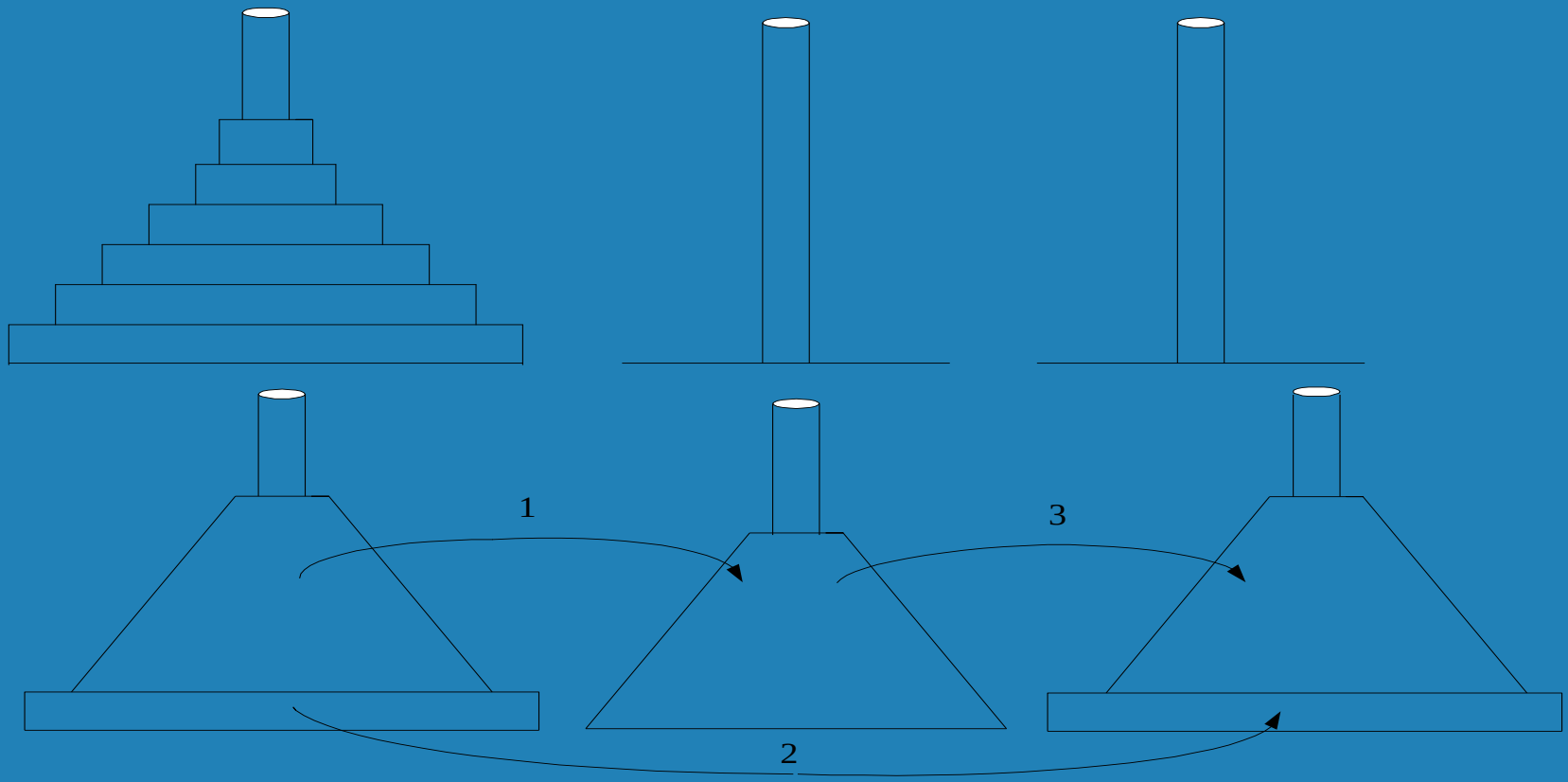
- Initial conditions eliminate all but 1

(Anyone know what other kind of equation is similar?)

(Sort of similar to y = x + 1, then add x=3)

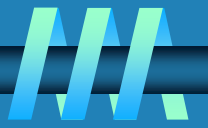


Example 2: The Tower of Hanoi Puzzle



Recurrence for number of moves:

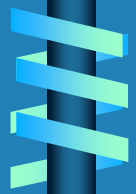
Solving recurrence for number of moves



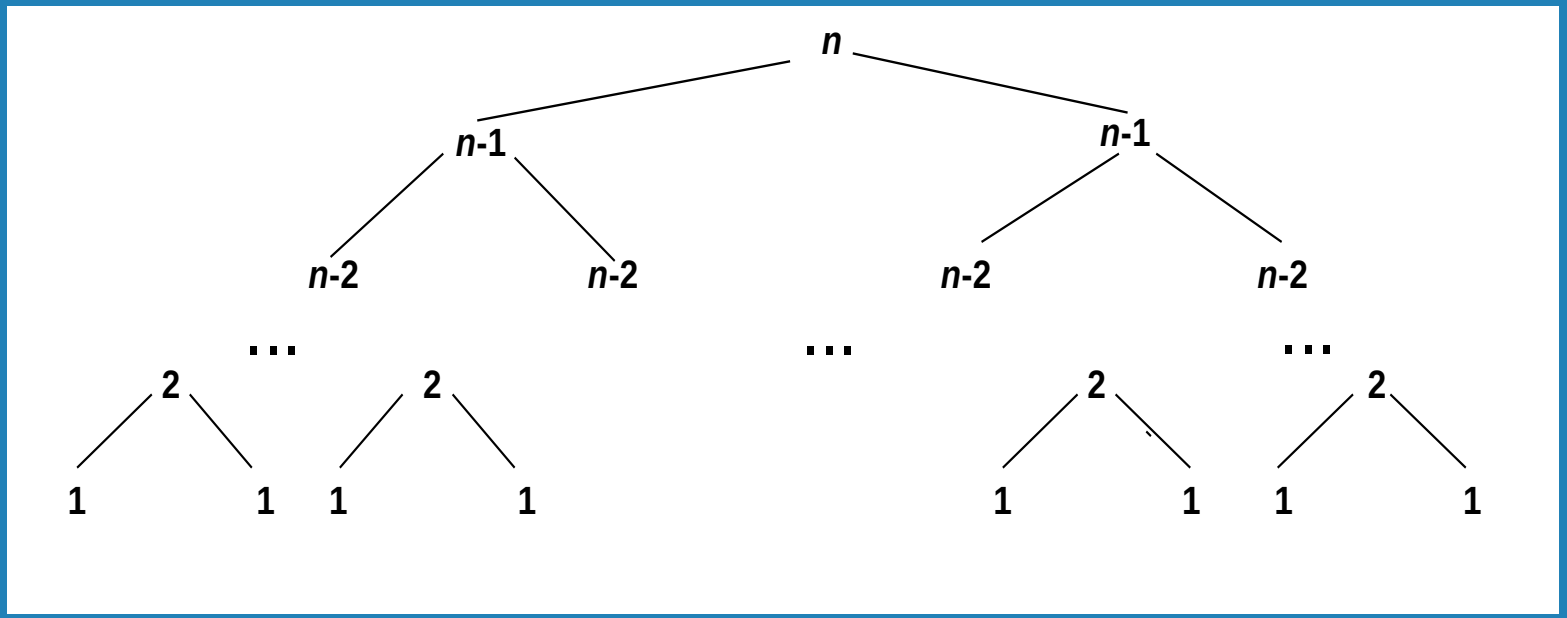
$M(n) = 2M(n-1) + 1, \quad M(1) = 1$

Guess closed form?

(What happens to number of moves when …)



Tree of calls for the Tower of Hanoi Puzzle



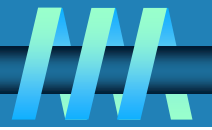
$M(n) = 2M(n\text{-}1) + 1, \quad M(1) = 1$

What happens when we add another disk?

Another level of the tree?

What proportion of the nodes are leaves?

Example 3: Counting #bits



ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n

//Output: The number of binary digits in n's binary representation

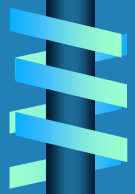
if $n = 1$ **return** 1

else return $BinRec(\lfloor n/2 \rfloor) + 1$

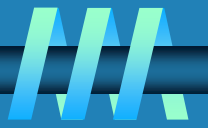
M(n) = M(?) ?,

M(?) = ?

Guess closed form?



General Methods



Forward and Backward Substitution problems:

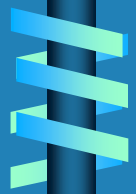
- solution must be proved
- solution may be difficult to find

General Methods:

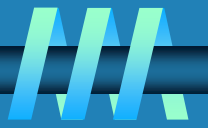
- Apply to certain classes of recurrences
- Solution always possible within the class

Two Common General Methods:

1. Master Method
2. Create and solve Characteristic Equation



Fibonacci numbers



The Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, …

The Fibonacci recurrence:

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

$$F(0) = 0$$

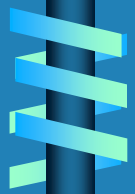
$$F(1) = 1$$

General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n-1) + cX(n-2) = 0$$

Key idea: *Assume* $X(n) = r^n$ for some value *r*.

Steps: Change recurrence to equation. Solve for r.



Solving $aX(n) + bX(n-1) + cX(n-2) = 0$

- Set up the characteristic equation (quadratic)

$$ar^2 + br + c = 0$$

- Solve to obtain roots r_1 and r_2

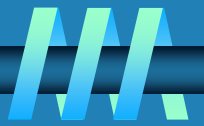
- General solution to the recurrence

  if r_1 and r_2 are two distinct real roots: $X(n) = \alpha r_1^n + \beta r_2^n$

  if $r_1 = r_2 = r$ are two equal real roots: $X(n) = \alpha r^n + \beta n r^n$

- Particular solution can be found by using initial conditions

Application to the Fibonacci numbers



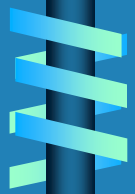
$F(n) = F(n-1) + F(n-2)$ or $F(n) - F(n-1) - F(n-2) = 0$

Characteristic equation:

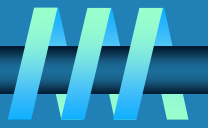
Roots of the characteristic equation:

General solution to the recurrence:

Particular solution for F(0) =0, F(1)=1:



Computing Fibonacci numbers



1. **Definition-based recursive algorithm**

2. **Nonrecursive definition-based algorithm**

3. **Explicit formula algorithm**

4. **Logarithmic algorithm based on formula:**

$$\begin{bmatrix} F(n\text{-}1) & F(n) \\ F(n) & F(n\text{+}1) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$$

for $n \geq 1$, assuming an efficient way of computing matrix powers.