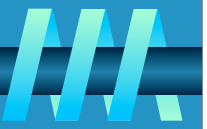
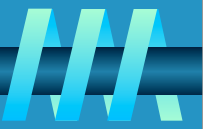


Chapter 2

Fundamentals of the Analysis of Algorithm Efficiency

Analysis of algorithms

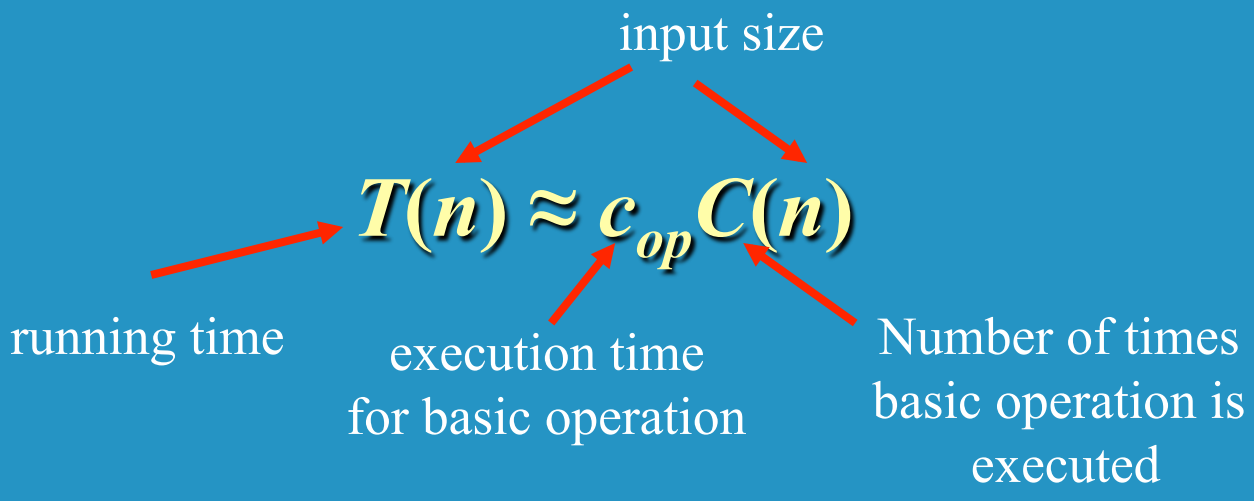


- **Issues:**
  - correctness
  - time efficiency
  - space efficiency (aka space complexity)
  - optimality
- **Approaches:**
  - theoretical analysis
  - empirical analysis

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**

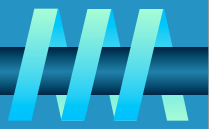


Input size and basic operation examples

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

Remember that other operations may also adversely impact performance

Empirical analysis of time efficiency



The best data for empirical analysis is the actual application running the actual data set. Without that, we try to do our best...

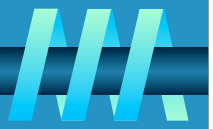
- **Select a specific (typical) sample of inputs**
- **Use physical unit of time (e.g., milliseconds)**

  or

  Count actual number of basic operation's executions

  Preview: You may have more than one answer depending on path through code for that data set. May exhibit best, worst, average case of operations. (i.e. consider searching a list for a match if list is ordered or unordered in increasing or decreasing key order)
- **Analyze the empirical data**

Best-case, average-case, worst-case



For some algorithms efficiency depends on form of input:

- **Worst case:** $C_{worst}(n)$ – maximum over inputs of size n
- **Best case:** $C_{best}(n)$ – minimum over inputs of size n
- **Average case:** $C_{avg}(n)$ – "average" over inputs of size n
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations considered as a random variable under some assumption about the probability distribution of all possible inputs

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
```

basic operation

- **Worst case**
  - $C_w(n)=n$ (unlucky as find as last element in list)
- **Best case**
  - $C_b(n)=1$ (lucky find as first element in list)
- **Average case**
  - $C_a(n)=(n+1)/2$ (will inspect half of elements on average)

Types of formulas for basic operation's count

- **Exact formula** (sequential search)

  e.g., $C(n) = n(n-1)/2$

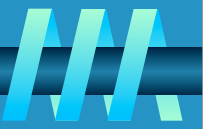
- **Formula indicating order of growth with specific multiplicative constant (0.5 = k)**

  e.g., $C(n) \approx 0.5\, n^2$ $\quad$ kn^2

- **Formula indicating order of growth with unknown multiplicative constant**

  e.g., $C(n) \approx cn^2$

Order of growth



- Most important: Order of growth within a constant multiple as $n \to \infty$

- Examples:
  - How much faster will algorithm run on computer that is twice as fast?
    - will run 2x as fast
    - will run in ½ the time
  - How much longer does it take to solve problem of double input size?
    - will run in 2x the time
    - will run in ½ as fast (note, computer just as fast but double work...)

(beware of "faster" vs "time")

Values of some important functions as $n \to \infty$

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}		

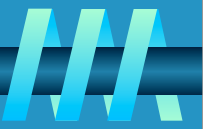
look at that growth!

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

both 2^n and n! are considered "exponential growth" functions...

(yes, only 2^n is truly mathematically exponential though...)

Asymptotic order of growth

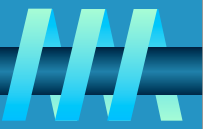


A way of comparing functions that ignores constant factors and small input sizes

- **$O(g(n))$: class of functions $f(n)$ that grow no faster than $g(n)$**
  - **Big-oh**
- **$\Theta(g(n))$: class of functions $f(n)$ that grow at same rate as $g(n)$**
  - **Big-theta**
- **$\Omega(g(n))$: class of functions $f(n)$ that grow at least as fast as $g(n)$**
  - **Big-omega**

Remember that constant factors (k) may still negatively impact performance of real world implementations.

Formal mathematical definitions:



- $f(n) = O(g(n))$ means there are positive constants c and k, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

- $f(n) = \Theta(g(n))$ means there are positive constants c1, c2, and k, such that $0 \leq c1g(n) \leq f(n) \leq c2g(n)$ for all $n \geq k$. The values of c1, c2, and k must be fixed for the function f and must not depend on n.

- $f(n) = \Omega(g(n))$ means there are positive constants c and k, such that $0 \leq cg(n) \leq f(n)$ for all $n \geq k$. The values of c and k must be fixed for the function f and must not depend on n.

Big-oh O

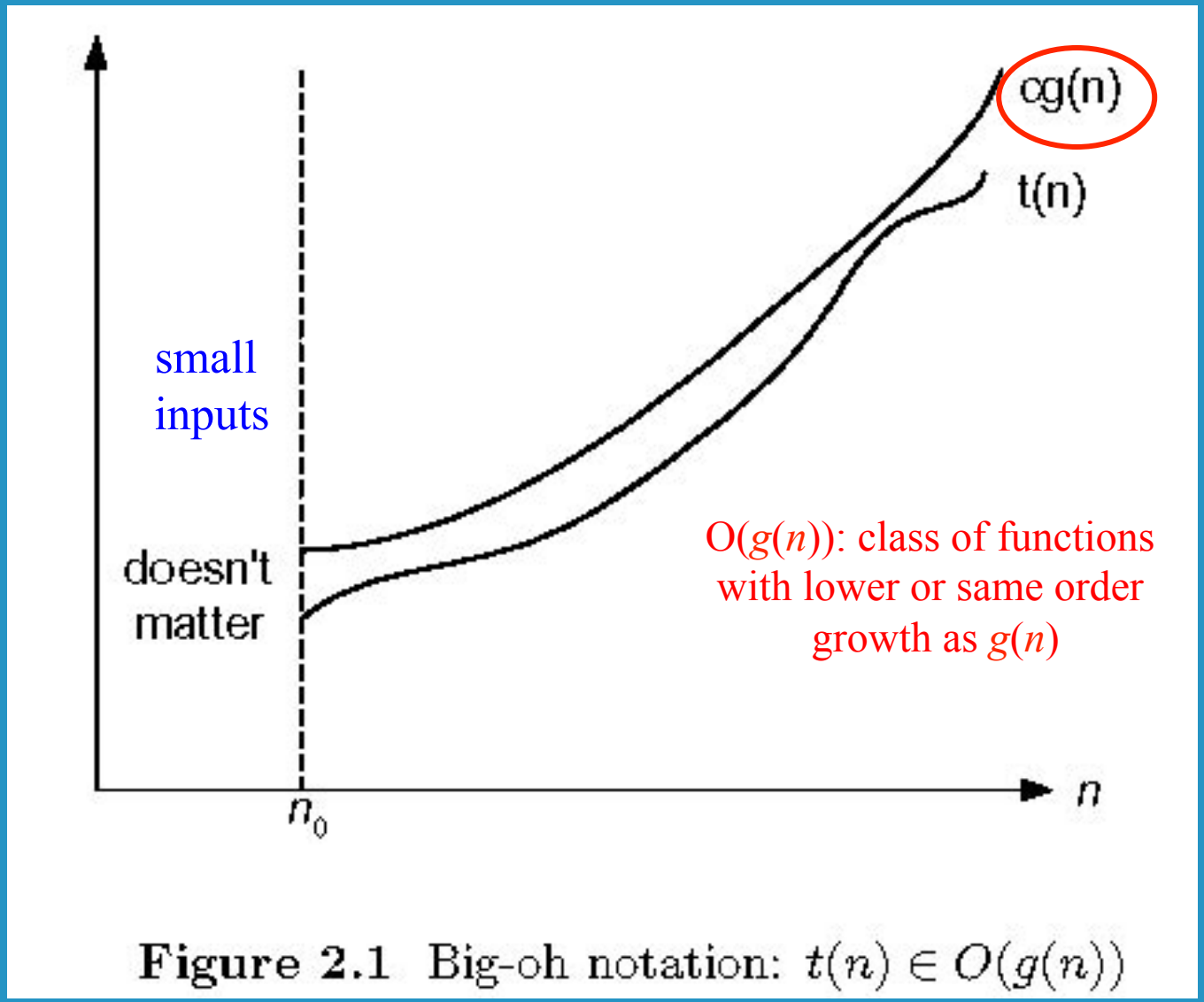
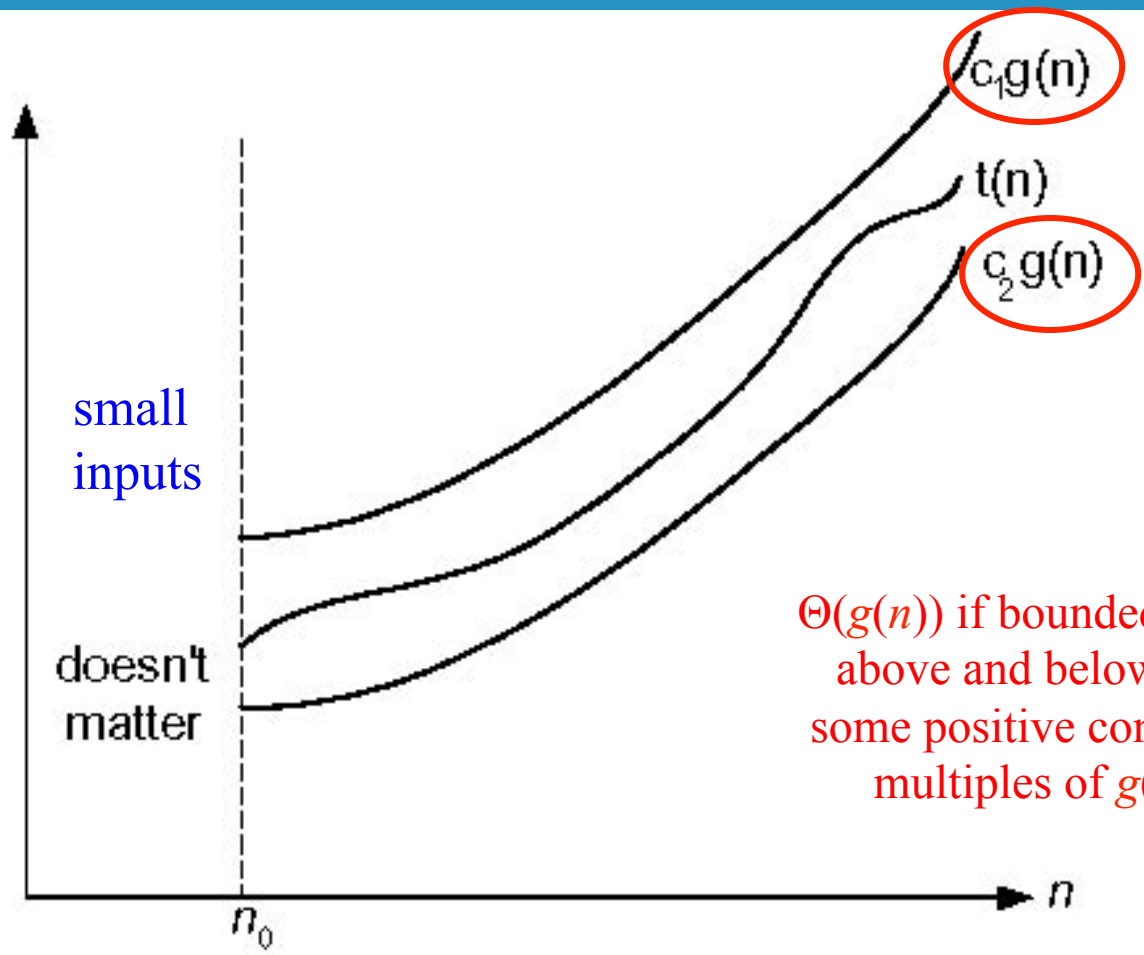


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

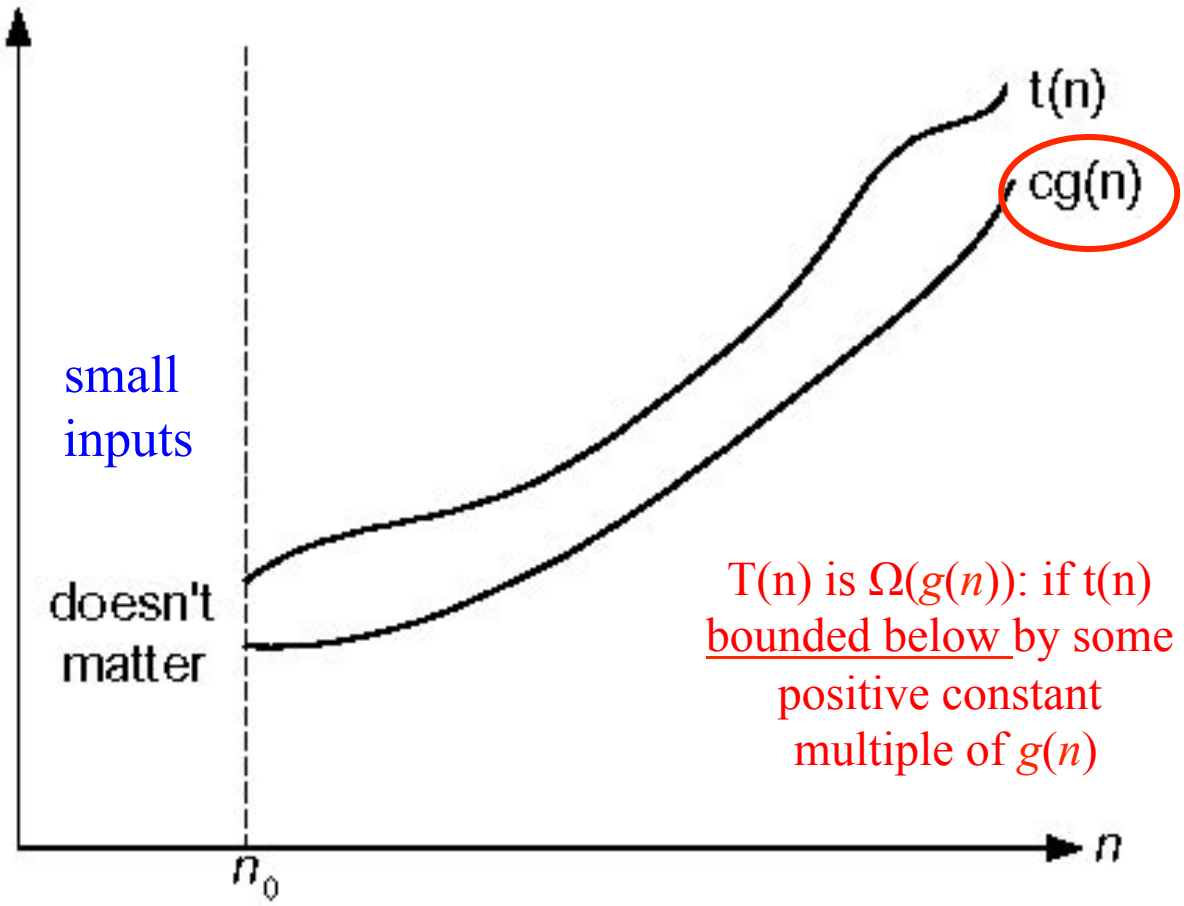
Big-theta Θ



Θ($g(n)$) if bounded both above and below by some positive constant multiples of $g(n)$

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

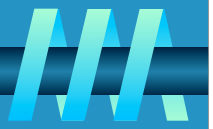
Big-omega Ω



T(n) is Ω($g(n)$): if t(n) bounded below by some positive constant multiple of $g(n)$

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

Big O notation:



- ***Landau's symbol***: comes from the name of the German number theoretician Edmund Landau who invented the notation. Letter O is used because the rate of growth of a function is also called its *order*.

- Example: analyze algorithm to find time (or number of steps) to complete a problem of size n is $T(n) = 6n^2 - 4n + 6$
  - Ignore constants (as they depend on particular hardware program is run on) and slower growing terms, we say $T(n)$ grows at the order of n^2 and write it as $T(n) = O(n^2)$.

Order of common functions:

O(1)	constant
O(log *n*)	logarithmic
$O((\log(n))^c)$	polylogarithmic
O(n)	linear
O(n log *n)*	*n*-log-*n* or linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(n^c)$	polynomial
$O(c^n)$	exponential
O(n!)	factorial

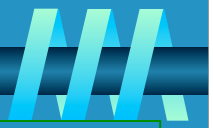
c is considered an actual number constant here while "n" is considered a varying amount.
e.g. do c=8 times versus do n=until_some_condition times

Time efficiency of nonrecursive algorithms

General Plan for Analysis

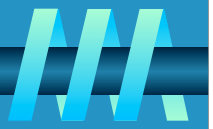
- Decide on parameter *n* indicating *input size*
- Identify algorithm's *basic operation*
- Determine *worst*, *average*, and *best* cases for input of size *n*
- Set up a sum for the number of times the basic operation is executed
- Simplify the sum using standard formulas and rules (see Appendix A)

How efficient is an algorithm



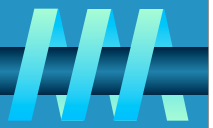
- Must consider many resources:
  - CPU (time) use
  - coprocessor (GPU for example)
  - memory use
  - disk use
  - network use
  - others...
- Typically (by default) time complexity is CPU use

Performance vs Complexity



- **Performance:**
  - how much time/memory/disk/etc... is actually used when a program is run. Depends on machine, compiler, etc... as well as implementation code.
- **Complexity:**
  - how do the resource requirements of a program or algorithm scale. (i.e. what happens as the size of the problem grows?)
- **Complexity affects Performance**
- **Performance does NOT affect Complexity**

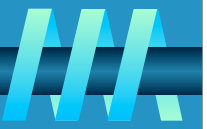
Basic operations



- Time required by function/procedure is proportional to the number of “basic operations” that it performs

- Basic operations – examples of...
  - one arithmetic operation (e.g. +, -, *,/)
  - one assignment (e.g. x ← 0)
  - one test (e.g. x = 0?)
  - one read (of a primitive type: integer, float, character, boolean)
  - one write (of a primitive type: integer, float, character, boolean)
  - others...

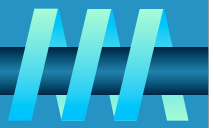
Operations to time



- **Constant time operations:**
  - these operations take the same amount of time every time they are called (regardless of any parameters).
  - example: StackSize function call returns either the number of elements in the stack or that the stack is empty.

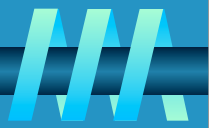
- **Problem or Input size dependent:**
  - the number of operations performed is dependent on the size of the problem or the number of inputs.
  - example: sorting a list of elements. number of elements in the list determines the number of operations performed by the algorithm.

Complexity



- Want the ***number of operations*** relative to the ***problem size***
  - we are not interested in the exact number of operations being performed
- Typically want the ***worst case***, the ***maximum*** number of operations that may be performed for a give problem size.

How to determine complexities (1)

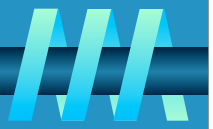


- **How to determine the running time of code?**
  - **depends on what type of statements are used.**

```
statement 1
statement 2
    ...
statement k
```

  - **total time = time(statement 1) + time (statement 2) +...+ time(statement k)**
  - **if each statement is considered "simple" (only involves basic operations) then time for each is constant and total time is then considered constant: O(1), and called an order constant or constant time algorithm.**

How to determine complexities (2)

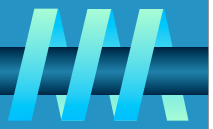


- **If-Then-Else**

```
if (condition) then
        block 1 (sequence of statements)
else
        block 2 (sequence of statements)
end if
```

  - either block 1 or block 2 will execute, not both
    - worst-case time is the slower of either
    - max(time(block 1), time(block 2))
    - if block 1 takes O(1) and block 2 takes O(n), then complexity of this if-then-else would be O(n)

How to determine complexities (3)

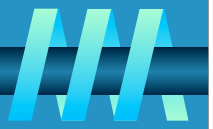


- Loops

```
for i = 1 to N loop
        sequence of statements
end loop
```

  - loop will execute N times
    - assume: sequence of statements is O(1)
    - total time for loop is then N * O(1) which is O(N)

How to determine complexities (4)

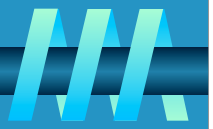


- **Nested Loops**

```
for i = 1 to N loop
        for j = 1 to M loop
                sequence of statements
        end loop
end loop
```

- outer loop will execute N times
- every time the outer loop executes, the inner loop executes M times
- inner loop statements will execute a total of N * M times
- complexity is thus O(N * M)
  - if N and M are of same relative size, then $O(N^2)$ or $O(M^2)$

How to determine complexities (4)



- **Function or Procedure calls**
  - where f(w) takes constant time (regardless of input *w*)
  - where g(w) takes time proportional (linear) to the value of input *w*
  - we have...
    - f(w) has O(1)
    - g(w) has O(N)
  - same rules apply when function is in a loop

```
for i = 1 to N loop
        g(i)
end loop
```

    - Complexity here is $O(N^2)$ as the loop executes N times and each call to g(N) is complexity O(N). $N*O(N) = O(N^2)$

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                ← basic operation
        maxval ← A[i]
return maxval
```

- Comparison is basic operation as it will be executed every time the loop is executed.
- Same number of comparisons for all arrays size n.
  - same number of operations for worst, average, best cases

It is... $\Theta(g(n))$: Big-theta, class of functions $f(n)$ that grow at same rate as $g(n)$

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      ← basic operation
return true
```

- 2 possible worst case:
  - does not exit early as no equal elements
  - exits when last array elements match

It is... $\Theta(n^2)$: Big-theta, class of functions $f(n)$ that grow at same rate as $g(n)$

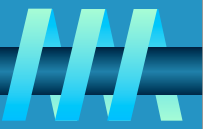
Example 3: Matrix multiplication

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**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$ ← basic operation
return C

- Row by column multiplication
  - element by element multiplication and summation is basic operation
- It is... $\Theta(n^3)$: Big-theta, class of functions $f(n)$ that grow at same rate as $g(n)$

Recurrence Relations:



- M(n) is defined in terms of n
  - e.x. M(n) defined in terms of n-1 until M(n)=0
- a function is defined in terms of itself
- note that there has to be a terminating condition or it is unsolvable (infinite)

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 operation is executed.
- Solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or another method.

Example 1: Recursive evaluation of *n*!

Definition: compute the factorial function F(n)=n! for an arbitrary nonnegative integer n.

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

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

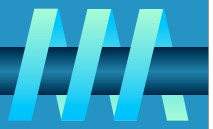
ALGORITHM $F(n)$

```
//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: n

Basic operation: multiplication

backwards substitutions

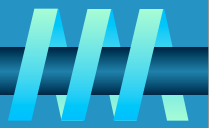


find n! where n=5:

f(n)=f(n-1)*n

Push unresolved problems on stack, when resolved pop off stack and pass back result to be used in resolving the next one on the stack... until done.

backwards substitutions

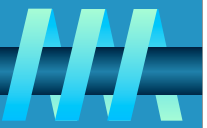


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5

backwards substitutions



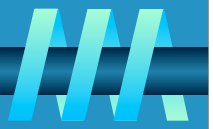
find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5

f(4)=f(4-1)*4

backwards substitutions

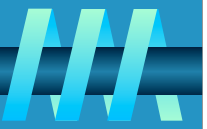


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4
f(3)=f(3-1)*3

backwards substitutions

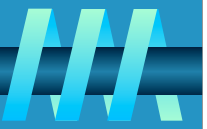


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4
f(3)=f(3-1)*3
f(2)=f(2-1)*2

backwards substitutions

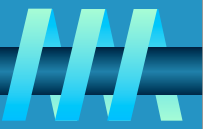


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4
f(3)=f(3-1)*3
f(2)=f(2-1)*2
f(1)=f(1-1)*1

backwards substitutions



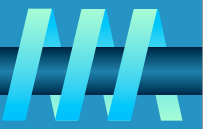
find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4
f(3)=f(3-1)*3
f(2)=f(2-1)*2
f(1)=f(1-1)*1 → f(0)*1 → 1*1 → 1

Rem. by definition f(0)=n!=1

backwards substitutions

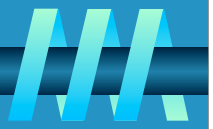


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4
f(3)=f(3-1)*3
f(2)=f(2-1)*2 → f(1)*2 → 1*2 → 2
f(1)=f(1-1)*1 → f(0)*1 → 1*1 → 1

backwards substitutions

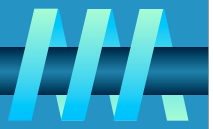


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4
f(3)=f(3-1)*3 → f(2)*3 → 2*3 → 6
f(2)=f(2-1)*2 → f(1)*2 → 1*2 → 2
f(1)=f(1-1)*1 → f(0)*1 → 1*1 → 1

backwards substitutions

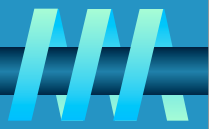


find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5
f(4)=f(4-1)*4 → f(3)*4 → 6*4 → 24
f(3)=f(3-1)*3 → f(2)*3 → 2*3 → 6
f(2)=f(2-1)*2 → f(1)*2 → 1*2 → 2
f(1)=f(1-1)*1 → f(0)*1 → 1*1 → 1

backwards substitutions



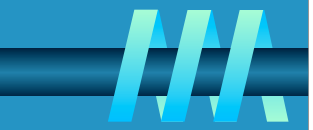
find n! where n=5:

f(n)=f(n-1)*n

f(5)=f(5-1)*5 → f(4)*5 → 24*5 → 120
f(4)=f(4-1)*4 → f(3)*4 → 6*4 → 24
f(3)=f(3-1)*3 → f(2)*3 → 2*3 → 6
f(2)=f(2-1)*2 → f(1)*2 → 1*2 → 2
f(1)=f(1-1)*1 → f(0)*1 → 1*1 → 1

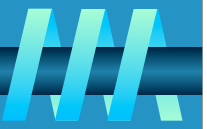
f(n)=f(n-1)*n for every n>0
f(0)=1
results in n steps of function f() for solution

Example 2: The Tower of Hanoi Puzzle



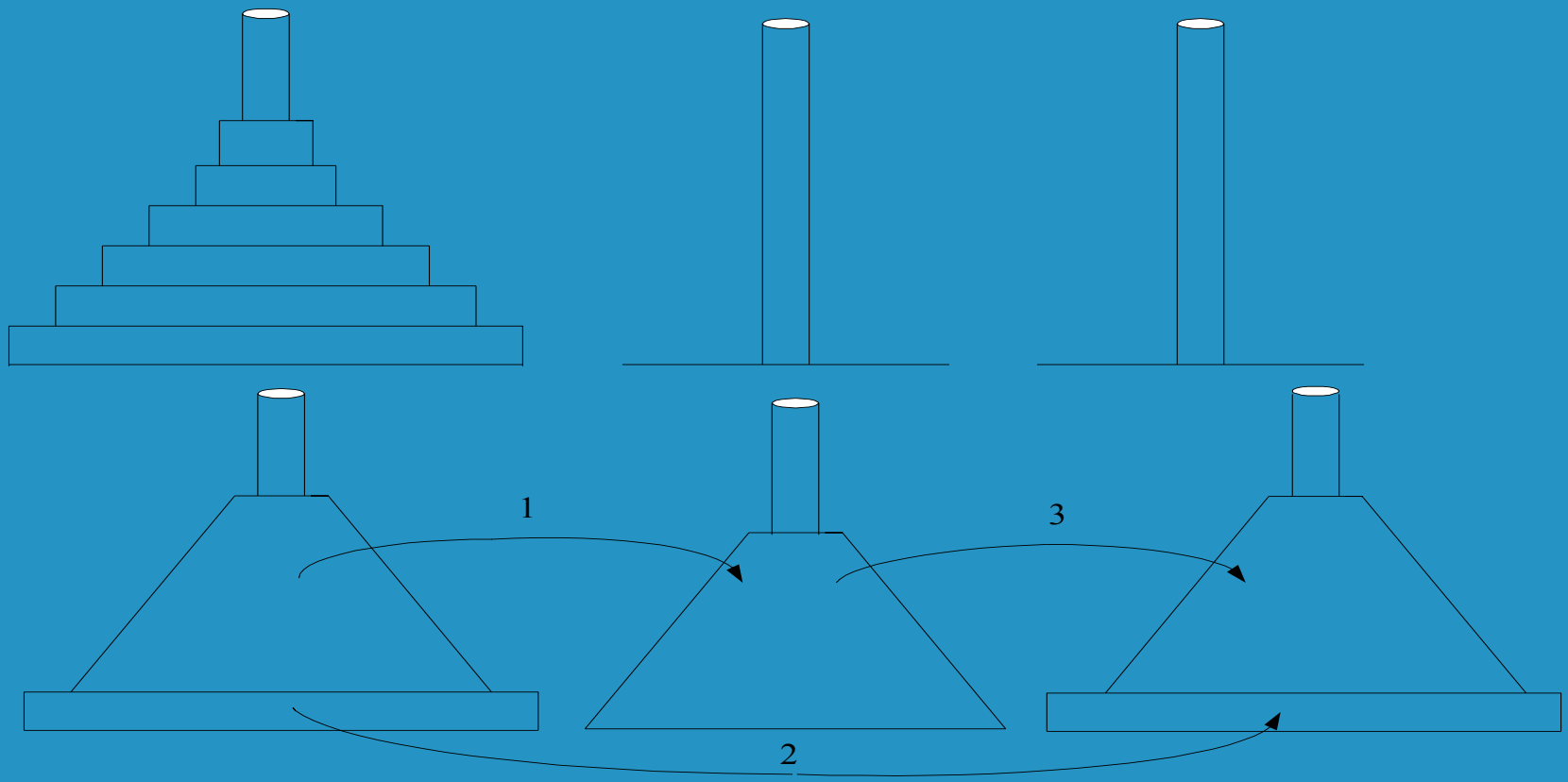
- n disks of different sizes that can slide onto any of 3 pegs
- initially all disks are on the first peg in size order – largest at bottom, smallest on top
- goal is to move all disks to 3rd peg using the second one as an auxiliary hold location where necessary
- only one disk may be moved at a time
- a larger disk may not be put on top of a smaller disk

Example 2: The Tower of Hanoi Puzzle



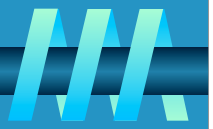
- to move n>1 disks from peg 1 to peg 3 (peg 2 as auxiliary)
- move recursively n-1 disks from peg 1 to peg 2 (with 3 as auxiliary)
- then move largest disk directly from peg 1 to peg 3
- finally move recursively n-1 disks from peg 2 to peg 3 (using peg 1 as auxiliary)
- where n=1 move single disk directly from source peg to destination peg

Example 2: The Tower of Hanoi Puzzle



What the motion looks like...

backwards substitutions

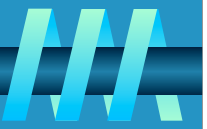


Use backwards substitutions to see the trend of the algorithm...

find that it takes 2^n-1

Thus this solution is an exponential algorithm - one that grows rapidly as n increases in size...

Fibonacci numbers



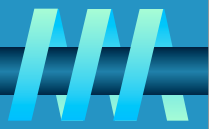
Fibonacci numbers are characterized by the fact that every number in the series is the sum of the two preceding numbers in the series.

By definition, the first two numbers in the Fibonacci sequence are either 1 and 1, or 0 and 1, depending on chosen starting point of sequence.

The Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Fibonacci numbers



The Fibonacci recurrence:

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$

$$F(0) = 0$$

$$F(1) = 1$$

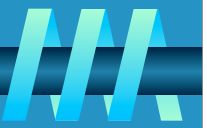
General 2nd order linear homogeneous recurrence with constant coefficients:

$$aX(n) + bX(n\text{-}1) + cX(n\text{-}2) = 0$$

(where a, b, c are some fixed real numbers, a not 0)

exponential growth algorithm...

Empirical Analysis of Algorithms



General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. understand the experiment's purpose
2. decide on the efficiency metric M to be measured and the measurement unit (i.e. operation count vs time unit)
3. decide on characteristics of the input sample (range, size, etc)
4. prepare a program implementing the algorithm(s) for the experimentation
5. generate a sample of inputs
6. run the algorithm(s) on the sample's inputs and record the data observed (i.e. operation count or time unit)
7. analyze the data obtained