Basic Mathematics Quick Review

Logarithms

- $a^m = x \Rightarrow m = log_a(x)$ (where $a > 0, a \neq 1$ and $x > 0$)
- $log_a a = 1$ and $log_a 1 = 0$
- $log_a(x^k) = k(log_a x)$
- $log_a(m * n) = log_a m + log_a n$
- $log_a \frac{m}{n} = log_a m - log_a n$
- $log_a x = \frac{log_k x}{log_k a} = 1/(log_x a)$
- $x^{log_a y} = y^{log_a x}$

Exponentials

- $(ab)^n = a^n b^n$
- $(a^m)^n = a^{m*n}$
- $a^m * a^n = a^{m+n}$
- $\lim_{n\to\infty}(\frac{n^b}{a^n}) = 0$ (where $a > 1$, $b > 0$)

Summations

$$\sum_{i=1}^{n} 1 = \underbrace{1 + 1 + 1 + \cdots + 1}_{\text{n times}} = n$$

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} i^3 = 1 + 8 + 27 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$$

$$\sum_{i=0}^{k} x^i = 1 + x + x^2 + \cdots + x^k = \frac{x^{(k+1)} - 1}{x - 1} \text{ (when } |x| \neq 1)$$

$$\sum_{i=0}^{\infty} x^i = 1 + x + x^2 + x^3 \cdots\cdots = \frac{1}{1 - x} \text{ (when } |x| < 1)$$

What is an Algorithm?

(And how do we analyze one?)

Algorithms

- ***Informally,***
  - A tool for solving a well-specified computational problem.

Input → ***Algorithm*** → ***Output***

- **Example: sorting**

  input: A sequence of numbers.

  output: An ordered permutation of the input.

  issues: correctness, efficiency, storage, etc.

Strengthening the Informal Definiton

- An algorithm is a **finite** sequence of **unambiguous** instructions for solving a well-specified computational problem.
- Important Features:
  - Finiteness.
  - Definiteness.
  - Input.
  - Output.
  - Effectiveness.

Algorithm Analysis

- Determining performance characteristics. (Predicting the resource requirements.)
  - Time, memory, communication bandwidth etc.
  - **Computation time** (running time) is of primary concern.
- Why analyze algorithms?
  - **Choose** the **most efficient** of several possible algorithms for the same problem.
  - Is the best possible **running time** for a problem ***reasonably finite*** for practical purposes?
  - Is the algorithm **optimal** (best in some sense)? – Is something better possible?

Running Time

- Run time expression should be machine-independent.
  - Use a model of computation or "hypothetical" computer.
  - Our choice – **RAM model** (most commonly-used).
- Model should be
  - Simple.
  - Applicable.

RAM Model

- Generic single-processor model.
- **Supports simple constant-time instructions** found in real computers.
  - Arithmetic (+, –, *, /, %, floor, ceiling).
  - Data Movement (load, store, copy).
  - Control (branch, subroutine call).
- Run time (**cost**) is uniform (**1 time unit**) for all simple instructions.
- Memory is unlimited.
- Flat memory model – no hierarchy.
- Access to a word of memory takes **1 time unit**.
- Sequential execution – **no concurrent operations**.

Model of Computation

- Should be simple, or even simplistic.
  - Assign uniform cost for all simple operations and memory accesses. (Not true in practice.)
  - **Question:** **Is this OK?**
- Should be widely applicable.
  - Can't assume the model to support complex operations. **Ex:** **No SORT instruction.**
  - Size of a word of data is finite.
  - **Why?**

Running Time – Definition

- Call each simple instruction and access to a word of memory a “primitive operation” or “step.”
- Running time of an algorithm for a given input is
  - The **number of steps** executed by the algorithm on that **input**.
- Often referred to as the ***complexity*** of the algorithm.

Complexity and Input

- Complexity of an algorithm generally depends on
  - **Size of input**.
    - Input size depends on the problem.
      - Examples: No. of items to be sorted.
      - No. of vertices and edges in a graph.
  - **Other characteristics of the input data**.
    - Are the items already sorted?
    - Are there cycles in the graph?

Worst, Average, and Best-case Complexity

- Worst-case Complexity
  - **Maximum** steps the algorithm takes for any possible input.
  - Most tractable measure.
- Average-case Complexity
  - **Average** of the running times of all ***possible* inputs.**
  - Demands a definition of probability of each input, which is usually difficult to provide and to analyze.
- Best-case Complexity
  - **Minimum** number of steps for any possible input.
  - Not a useful measure. Why?

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
$i \leftarrow 1$	c_1	1
while $i \le n$ **and** A[i] != *key*	c_2	x
do i++	c_3	x-1
if $i \le n$	c_4	1
then return *true*	c_5	1
else return *false*	c_6	1

x ranges between 1 and n+1.

So, the running time ranges between

$c_1 + c_2 + c_4 + c_5$ – **best case**

and

$c_1 + c_2(n+1) + c_3 n + c_4 + c_6$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
1 $i \leftarrow 1$	1	1
2 **while** $i \leq n$ **and** A[i] != *key*	1	x
3 **do** i++	1	x-1
4 **if** $i \leq n$	1	1
5 **then return** *true*	1	1
6 **else return** *false*	1	1

Assign a cost of 1 to all statement executions.

Now, the running time ranges between

$1+ 1+ 1 + 1 = 4$ – **best case**

and

$1+ (n+1)+ n + 1 + 1 = 2n+4$ – **worst case**

A Simple Example – *Linear Search*

INPUT: a sequence of *n* numbers, *key* to search for.

OUTPUT: *true* if *key* occurs in the sequence, *false* otherwise.

LinearSearch(A, *key*)	*cost*	*times*
1 $i \leftarrow 1$	1	1
2 **while** $i \leq n$ **and** A[i] != *key*	1	x
3 **do** i++	1	x-1
4 **if** $i \leq n$	1	1
5 **then return** *true*	1	1
6 **else return** *false*	1	1

If we assume that we search for a random item in the list, on an average, Statements 2 and 3 will be executed $n/2$ times. Running times of other statements are independent of input. Hence, **average-case complexity** is

$1+ n/2+ n/2 + 1 + 1 = n+3$

Order of growth

- Principal interest is to determine
  - how running time grows with input size – **Order of growth**.
  - the running time for large inputs – **Asymptotic complexity**.
- In determining the above,
  - **Lower-order terms and coefficient of the highest-order term are insignificant.**
  - **Ex: In $7n^5+6n^3+n+10$, which term dominates the running time for very large n?**
- Complexity of an algorithm is denoted by the highest-order term in the expression for running time.
  - **Ex: $O(n)$, $\Theta(1)$, $\Omega(n^2)$**, etc.
  - Constant complexity when running time is independent of the input size – denoted $O(1)$.
  - **Linear Search**: **Best case $\Theta(1)$, Worst and Average cases: $\Theta(n)$.**
- More on O, Θ, and Ω in next class. Use Θ for the present.

Comparison of Algorithms

- Complexity function can be used to compare the performance of algorithms.
- Algorithm A is more efficient than Algorithm B for solving a problem, if the complexity function of A is of lower order than that of B.
- Examples:
  - **Linear Search** – $\Theta(n)$ vs. **Binary Search** – $\Theta(\lg n)$
  - **Insertion Sort** – $\Theta(n^2)$ vs. **Quick Sort** – $\Theta(n \lg n)$

Comparisons of Algorithms

- **Multiplication**
  - classical technique: $O(nm)$
  - divide-and-conquer: $O(nm^{\ln 1.5}) \sim O(nm^{0.59})$

    For operands of size 1000, takes 40 & 15 seconds respectively on a Cyber 835.
- **Sorting**
  - insertion sort: $\Theta(n^2)$
  - merge sort: $\Theta(n \lg n)$

    For 10^6 numbers, it took 5.56 hrs on a supercomputer using machine language and 16.67 min on a PC using C/C++.

Why Order of Growth Matters?

- Computer speeds double every two years, so why worry about algorithm speed?
- When speed doubles, what happens to the amount of work you can do?
- What about the demands of applications?

Why Efficiency Matters?

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Source: Algorithm Design Book (chapter 2) by Jon Kleinberg and Éva Tardos

We say that an algorithm is efficient if has a polynomial running time.

Exceptions. Some poly-time algorithms do have high constants and/or exponents, and/or are useless in practice.

Q. Which would you prefer $20 n^{100}$ vs. $n^{1 + 0.02 \ln n}$?

Example Algorithm

```
FIBONACCI(n)
if n == 0
    return 0
elseif n == 1
    return 1
else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

1. Is the algorithm *correct?*
   - for every valid input, does it terminate?
   - if so, does it do the right thing?
2. How much *time* does it take to complete?
3. Can we do better?

- The algorithm is clearly correct
  - assuming $n \geq 0$

Better Fibonacci Algorithm

- Again, the sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, . . .
- Idea: we can build F_n from the ground up!

```
SMARTFIBONACCI(n)
if n == 0
      return 0
elseif n == 1
      return 1
else pprev = 0
      prev = 1
      for i = 2 to n
            f = prev + pprev
            pprev = prev
            prev = f
return f
```

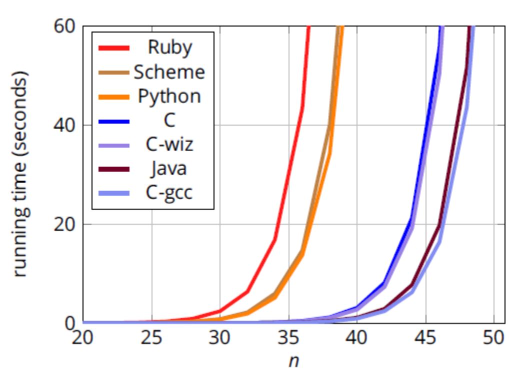
$T(n) = 6 + 6(n - 1) = 6n$

The *complexity* of **SMARTFIBONACCI**(n) is ***linear*** in n

Performance

- How long does it take?

Let's try it out. . .



Comments regarding Performance

- Different implementations perform differently
  - it is better to let the compiler do the optimization
  - simple language tricks don't seem to pay off
- However, the differences are not substantial
  - *all* implementations sooner or later seem to hit a wall...
- Conclusion: ***the problem is with the algorithm***
- We need a mathematical characterization of the performance of the algorithm

  We'll call it the algorithm's ***computational complexity***

Computational Complexity of Fibonacci Algorithm

- Let $T(n)$ be the number of ***basic steps*** needed to compute **Fibonacci**(n)

```
FIBONACCI(n)
if n == 0
      return 0
elseif n == 1
      return 1
else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

$T(0) = 2; T(1) = 3$

$T(n) = T(n-1) + T(n-2) + 3$

- So, let's try to understand how F_n grows with n

Computational Complexity of Fibonacci Algorithm

- So, let's try to understand how F_n grows with n

$$T(n) \geq F_n = F_{n-1} + F_{n-2}$$

Now, since $F_n \geq F_{n-1} \geq F_{n-2} \geq F_{n-3} \geq \ldots$

$$F_n \geq 2F_{n-2} \geq 2(2F_{n-4}) \geq 2(2(2F_{n-6})) \geq \ldots \geq 2^{\frac{n}{2}}$$

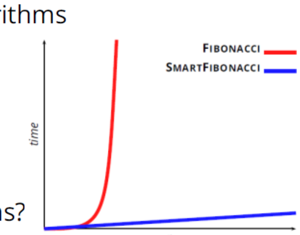
This means that

$$T(n) \geq (\sqrt{2})^n \approx (1.4)^n$$

- $T(n)$ ***grows exponentially*** with n
- Can we do better?

Slow Vs Fast Fibonacci

- We informally characterized our two Fibonacci algorithms
  - **FIBONACCI** is ***exponential*** in n
  - **SMARTFIBONACCI** is (almost) ***linear*** in n



- How do we characterize the complexity of algorithms?
  - in general
  - in a way that is ***specific to the algorithms***
  - but ***independent of implementation details***

Why models?

- What is a machine model?
  - A abstraction describes the operation of a machine.
  - Allowing to associate a value (cost) to each machine operation.
- Why do we need models?
  - Make it easy to reason algorithms
  - Hide the machine implementation details so that general results that apply to a broad class of machines to be obtained.
  - Analyze the achievable complexity (time, space, etc) bounds
  - Analyze maximum parallelism
  - Models are directly related to algorithms.

RAM (Random Access Machine) model

- Memory consists of infinite array (memory cells).
- Each memory cell holds an infinitely large number.
- Instructions execute sequentially one at a time.
- All instructions take unit time
  - Load/store
  - Arithmetic
  - Logic
- Running time of an algorithm is the number of instructions executed.
- Memory requirement is the number of memory cells used in the algorithm.

RAM model

- An informal model of the ***random-access machine (RAM)***
- ***Basic types*** in the RAM model
  - integer and floating-point numbers
  - limited size of each "word" of data (e.g., 64 bits)
- ***Basic steps*** in the RAM model
  - ***operations involving basic types***
  - load/store: assignment, use of a variable
  - arithmetic operations: addition, multiplication, division, etc.
  - branch operations: conditional branch, jump
  - subroutine call
- A ***basic step*** in the RAM model takes a ***constant time***

RAM (Random Access Machine) model

- The RAM model is the base of algorithm analysis for sequential algorithms although it is not perfect.
  - Memory not infinite
  - Not all memory access take the same time
  - Not all arithmetic operations take the same time
  - Instruction pipelining is not taken into consideration
- The RAM model (with asymptotic analysis) often gives relatively realistic results.

Analysis in the RAM Model

```
SMARTFIBONACCI(n)
if n == 0
    return 0
elseif n == 1
    return 1
else pprev = 0
    prev = 1
    for i = 2 to n
        f = prev + pprev
        pprev = prev
        prev = f
return f
```

cost	*times* ($n > 1$)
c_1	1
c_2	0
c_3	1
c_4	0
c_5	1
c_6	1
c_7	n
c_8	$n-1$
c_9	$n-1$
c_{10}	$n-1$
c_{11}	1

$$T(n) = c_1 + c_3 + c_5 + c_6 + c_{11} + nc_7 + (n-1)(c_8 + c_9 + c_{10})$$

$$\boxed{T(n) = nC_1 + C_2 \quad \Rightarrow T(n) \text{ is a } \textit{linear function} \text{ of } n}$$

Worst-Case Time Complexity

- In general we measure the complexity of an algorithm *as a function of the **size** of the input*
    - size measured in bits
- In general we measure the complexity of an algorithm ***in the worst case***
- **Example:** given a sequence $A = \langle a_1, a_2, \ldots, a_n \rangle$, output TRUE if A contains two equal values $a_i = a_j$ (with $i \neq j$)

```
FINDEQUALS(A)
for i = 1 to length(A) − 1
    for j = i + 1 to length(A)
        if A[i] == A[j]
            return TRUE
return FALSE
```

$$T(n) = C\frac{n(n-1)}{2}$$

Worst case. Running time guarantee for any input of size n.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Exceptions: *Some exponential time algorithms are used in practice. Why ?*

Complexity Analysis

- Does a load/store operation cost more than, say, an arithmetic operation?

$x = 0$ vs. $y + z$

- ***We do not care about the specific costs of each basic step***
  - these costs are likely to vary significantly with languages, implementations, and processors
  - so, we assume $c_1 = c_2 = c_3 = \cdots = c_i$
  - we also ignore the specific *value* c_i, and in fact ***we ignore every constant cost factor***
- We care only about the ***order of growth*** or *rate of growth* of $T(n)$
  - so we ignore lower-order terms

- Algorithm:
  - A set of **explicit, unambiguous finite steps**, which when carried out for a given set of **initial condition** to produce the corresponding **output** and terminate in **finite time**.
- Program:
  - An implementation of an algorithm in some programming languages
- Data Structure:
  - **Organization** of data needed to solve the problem

- An informal model of the ***random-access machine (RAM)***
- ***Basic types*** in the RAM model
  - integer and floating-point numbers
  - limited size of each "word" of data (e.g., 64 bits)
- ***Basic steps*** in the RAM model
  - ***operations involving basic types***
  - load/store: assignment, use of a variable
  - arithmetic operations: addition, multiplication, division, etc.
  - branch operations: conditional branch, jump
  - subroutine call
- A ***basic step*** in the RAM model takes a ***constant time***

Good Algo.'?

- Efficient
  - Running Time
  - Space Used
- Running time depends on
  - Single vs Multi processor
  - Read or Write speed to Memory
  - 32 bit vs 64 bit
  - Input -> rate of growth of time, Efficiency as a function of input (number of bits in an input number, number of data elements...)

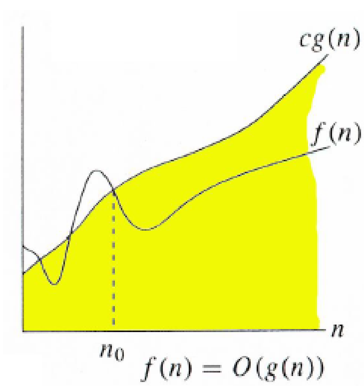
Asymptotic Notations

- ***O*, Ω, Θ, *o*, ω**
- Defined for functions over the natural numbers.
  - Ex: $f(n) = \Theta(n^2)$.
  - Describes how $f(n)$ grows in comparison to n^2.
- Define a ***set*** of functions; in practice used to compare two function sizes.
- The notations describe different rate-of-growth relations between the defining function and the defined set of functions.

Big-Oh Notation (Formal Definition)

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

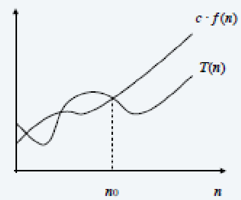
  $f(n) \le cg(n)$ for $n \ge n_0$
- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \le cn$
  - $(c - 2)\, n \ge 10$
  - $n \ge 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



$$O(g(n)) = \{ f(n) : \text{there exist constants } c > 0,\ n_0 > 0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \}$$

Big-Oh notation

Upper bounds. $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq c \cdot f(n)$ for all $n \geq n_0$.



Ex. $T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is $O(n^2)$. ← choose c = 50, n_0 = 1
- $T(n)$ is also $O(n^3)$.
- $T(n)$ is neither $O(n)$ nor $O(n \log n)$.

Typical usage. Insertion makes $O(n^2)$ compares to sort n elements.

Alternate definition. $T(n)$ is $O(f(n))$ if $\limsup_{n \to \infty} \frac{T(n)}{f(n)} < \infty$.

Notational abuses

Equals sign. $O(f(n))$ is a set of functions, but computer scientists often write $T(n) = O(f(n))$ instead of $T(n) \in O(f(n))$.

Ex. Consider $f(n) = 5n^3$ and $g(n) = 3n^2$.

- We have $f(n) = O(n^3) = g(n)$.
- Thus, $f(n) = g(n)$.

Domain. The domain of $f(n)$ is typically the natural numbers $\{ 0, 1, 2, \ldots \}$.

- Sometimes we restrict to a subset of the natural numbers. Other times we extend to the reals.

Nonnegative functions. When using big-Oh notation, we assume that the functions involved are (asymptotically) nonnegative.

Bottom line. OK to abuse notation; not OK to misuse it.

More Big-Oh Examples

- 7n - 2

7n-2 is O(n)
need c > 0 and $n_0 \geq 1$ such that $7n - 2 \leq cn$ for $n \geq n_0$
this is true for c = 7 and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

$3n^3 + 20n^2 + 5$ is $O(n^3)$
need c > 0 and $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
this is true for c = 4 and $n_0 = 21$

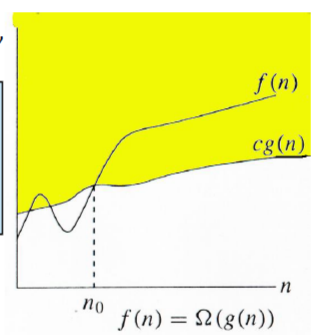
- 3 log n + 5

3 log n + 5 is O(log n)
need c > 0 and $n_0 \geq 1$ such that $3 \log n + 5 \leq c \log n$ for $n \geq n_0$
this is true for c = 8 and $n_0 = 2$

Ω -notation

For function $g(n)$, we define $\Omega(g(n))$, big-Omega of n, as the set:

**$\Omega(g(n)) = \{f(n)$:
$\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
we have $0 \leq cg(n) \leq f(n)\}$**



Intuitively: Set of all functions whose *rate of growth* is the same as or higher than that of $g(n)$.

$g(n)$ is an *asymptotic lower bound* for $f(n)$.

Example

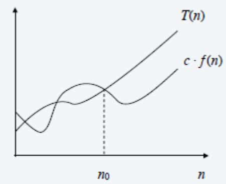
$\Omega(g(n)) = \{f(n)$: $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$, we have $0 \leq cg(n) \leq f(n)\}$

- $\sqrt{n} = \Omega(\log n)$. Choose c and n_0.
  for c=1 and n_0 =16,

$$c * \log n \leq \sqrt{n} \quad , \forall n \geq 16$$

Big-Omega notation

Lower bounds. $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that $T(n) \geq c \cdot f(n)$ for all $n \geq n_0$.



Ex. $T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is both $\Omega(n^2)$ and $\Omega(n)$. ⟵ choose c = 32, n₀ = 1
- $T(n)$ is neither $\Omega(n^3)$ nor $\Omega(n^3 \log n)$.

Typical usage. Any compare-based sorting algorithm requires $\Omega(n \log n)$ compares in the worst case.

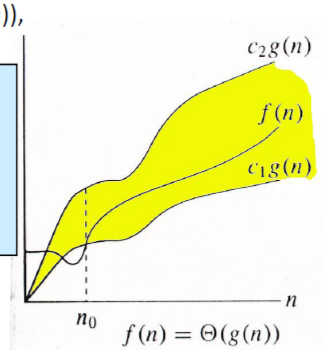
Meaningless statement. Any compare-based sorting algorithm requires at least $O(n \log n)$ compares in the worst case.

Θ-notation

For function $g(n)$, we define $\Theta(g(n))$, big-Theta of n, as the set:

$\Theta(g(n))$ = {$f(n)$:
∃ positive constants c_1, c_2, and n_0,
such that $\forall n \geq n_0$,
we have $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
}

Intuitively: Set of all functions that have the same *rate of growth* as $g(n)$.



$g(n)$ is an *asymptotically tight bound* for $f(n)$.

$f(n)$ and $g(n)$ are nonnegative, for large n.

Example

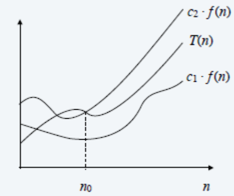
$\Theta(g(n))$ = {$f(n)$: ∃ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$}

- $10n^2 - 3n = \Theta(n^2)$

- Is $3n^3 \in \Theta(n^4)$??
- How about $2^{2n} \in \Theta(2^n)$??

Big-Theta notation

Tight bounds. $T(n)$ is $\Theta(f(n))$ if there exist constants $c_1 > 0$, $c_2 > 0$, and $n_0 \geq 0$ such that $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$ for all $n \geq n_0$.

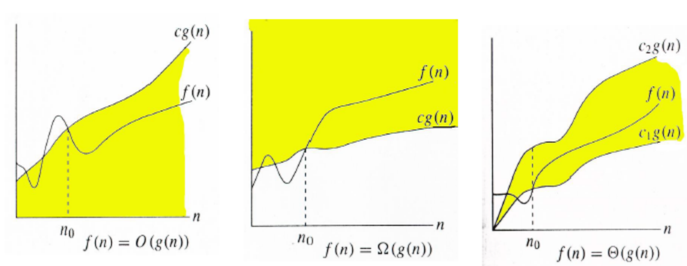


Ex. $T(n) = 32n^2 + 17n + 1$.

- $T(n)$ is $\Theta(n^2)$. ⟵ choose $c_1 = 32$, $c_2 = 50$, $n_0 = 1$
- $T(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

Typical usage. Mergesort makes $\Theta(n \log n)$ compares to sort n elements.

Relations Between *O*, Ω, Θ



- ❖Big-Oh says, "Your algorithm is at least this good."
- ❖Omega says, "Your algorithm is at least this bad."

Limits

- $\lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in O(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] < \infty \Rightarrow f(n) \in \Theta(g(n))$
- $0 < \lim_{n\to\infty} [f(n) / g(n)] \Rightarrow f(n) \in \Omega(g(n))$
- $\lim_{n\to\infty} [f(n) / g(n)]$ undefined $\Rightarrow$ can't say

Useful facts

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c > 0$, then $f(n)$ is $\Theta(g(n))$.

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$.

Asymptotic bounds for some common functions

Polynomials. Let $T(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $T(n)$ is $\Theta(n^d)$.

Pf. $$\lim_{n \to \infty} \frac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$$

Logarithms. $\Theta(\log_a n)$ is $\Theta(\log_b n)$ for any constants $a, b > 0$. ← no need to specify base (assuming it is a constant)

Logarithms and polynomials. For every $d > 0$, $\log n$ is $O(n^d)$.

Exponentials and polynomials. For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$.

Pf. $$\lim_{n \to \infty} \frac{n^d}{r^n} = 0$$

Rank the following functions by increasing order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{20}$ of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, $\ldots$, $g_{19} = O(g_{20})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\binom{n}{2}$	$n \log n$	$\sqrt{n} \cdot 10^{100}$	$8n^2$	$\log \sqrt{\log n}$
$n!$	$\log \log n$	$n^{\log n}$	$\log n!$	$4^{\log n}$
$\sum_{k=0}^{n} \binom{n}{k}$	$2^{\log^2 n}$	10^{100}	3^n	$\log n$
$(\sqrt{2})^{\log n}$	$(n-1)!$	$3n^3$	2^n	$5\sqrt{n}$

Exercise

Express functions in A in asymptotic notation using functions in B.

A	B	
$5n^2 + 100n$	$3n^2 + 2$	$A \in \Theta(B)$

$A \in \Theta(n^2)$, $n^2 \in \Theta(B) \Rightarrow A \in \Theta(B)$

A	B	
$\log_3(n^2)$	$\log_2(n^3)$	$A \in \Theta(B)$

$\log_b a = \log_c a / \log_c b$; $A = 2\lg n / \lg 3$, $B = 3\lg n$, $A/B = 2/(3\lg 3)$

A	B	
$n^{\lg 4}$	$3^{\lg n}$	$A \in \omega(B)$

$a^{\log b} = b^{\log a}$; $B = 3^{\lg n} = n^{\lg 3}$; $A/B = n^{\lg(4/3)} \rightarrow \infty$ as $n \rightarrow \infty$

A	B	
$\lg^2 n$	$n^{1/2}$	$A \in o(B)$

$\lim_{n\rightarrow\infty} (\lg^a n / n^b) = 0$ (here $a = 2$ and $b = 1/2$) $\Rightarrow A \in o(B)$

Big-Oh notation with multiple variables

Upper bounds. $T(m, n)$ is $O(f(m, n))$ if there exist constants $c > 0$, $m_0 \geq 0$, and $n_0 \geq 0$ such that $T(m, n) \leq c \cdot f(m, n)$ for all $n \geq n_0$ and $m \geq m_0$.

Ex. $T(m, n) = 32mn^2 + 17mn + 32n^3$.

- $T(m, n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
- $T(m, n)$ is neither $O(n^3)$ nor $O(mn^2)$.

Typical usage. Breadth-first search takes $O(m + n)$ time to find the shortest path from s to t in a digraph.

Linear time: O(n)

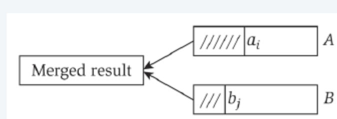
Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of n numbers $a_1, \ldots, a_n$.

```
max ← a1
for i = 2 to n {
   if (ai > max)
      max ← ai
}
```

Linear time: O(n)

Merge. Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.




```
i = 1, j = 1
while (both lists are nonempty) {
   if (ai ≤ bj) append ai to output list and increment i
   else         append bj to output list and increment j
}
append remainder of nonempty list to output list
```

Claim. Merging two lists of size n takes $O(n)$ time.
Pf. After each compare, the length of output list increases by 1.

Linearithmic time: O(n log n)

O(n log n) time. Arises in divide-and-conquer algorithms.

Sorting. Mergesort and heapsort are sorting algorithms that perform $O(n \log n)$ compares.

Largest empty interval. Given n time-stamps $x_1, \ldots, x_n$ on which copies of a file arrive at a server, what is largest interval when no copies of file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.

Quadratic time: $O(n^2)$

Ex. Enumerate all pairs of elements.

Closest pair of points. Given a list of n points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x1 - x2)² + (y1 - y2)²
for i = 1 to n {
   for j = i+1 to n {
      d ← (xi - xj)² + (yi - yj)²
      if (d < min)
         min ← d
   }
}
```

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion. [see Chapter 5]

Cubic time: $O(n^3)$

Cubic time. Enumerate all triples of elements.

Set disjointness. Given n sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

$O(n^3)$ solution. For each pair of sets, determine if they are disjoint.

```
foreach set Si {
   foreach other set Sj {
      foreach element p of Si {
         determine whether p also belongs to Sj
      }
      if (no element of Si belongs to Sj)
         report that Si and Sj are disjoint
   }
}
```

Polynomial time: $O(n^k)$

Independent set of size k. Given a graph, are there k nodes such that no two are joined by an edge?

k is a constant

$O(n^k)$ solution. Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
   check whether S in an independent set
   if (S is an independent set)
      report S is an independent set
   }
}
```

- Check whether S is an independent set takes $O(k^2)$ time.
- Number of k element subsets = $\binom{n}{k} = \frac{n(n-1)(n-2)\times\cdots\times(n-k+1)}{k(k-1)(k-2)\times\cdots\times 1} \leq \frac{n^k}{k!}$
- $O(k^2 n^k / k!) = O(n^k)$.

poly-time for k=17,
but not practical

Exponential time

Independent set. Given a graph, what is maximum cardinality of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← φ
foreach subset S of nodes {
   check whether S in an independent set
   if (S is largest independent set seen so far)
      update S* ← S
   }
}
```

Sublinear time

Search in a sorted array. Given a sorted array A of n numbers, is a given number x in the array?

$O(\log n)$ solution. Binary search.

```
lo ← 1, hi ← n
while (lo ≤ hi) {
   mid ← (lo + hi) / 2
   if      (x < A[mid]) hi ← mid - 1
   else if (x > A[mid]) lo ← mid + 1
   else return yes
}
return no
```

Divide and Conquer
(Merge Sort)

Divide and Conquer

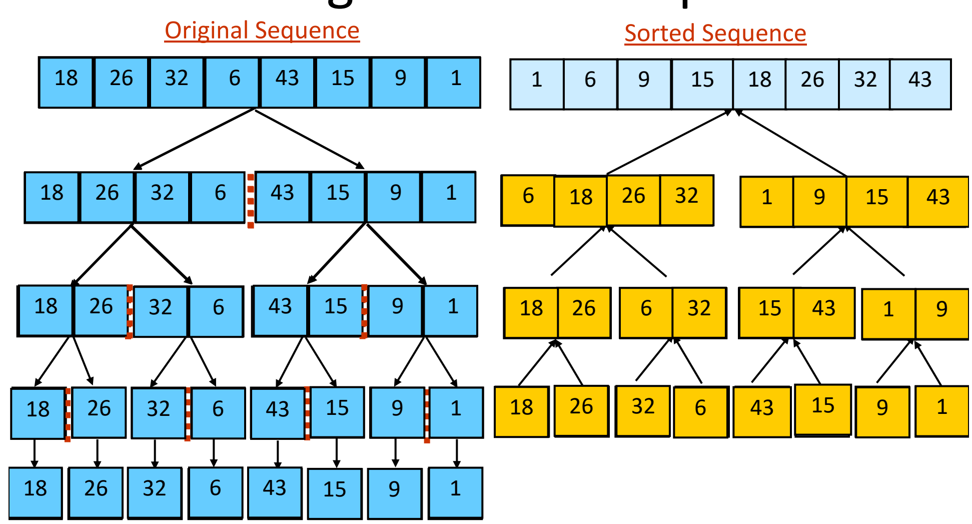
- Recursive in structure
  - ***Divide*** the problem into sub-problems that are similar to the original but smaller in size
  - ***Conquer*** the sub-problems by solving them recursively. If they are small enough, just solve them in a straightforward manner.
  - ***Combine*** the solutions to create a solution to the original problem

An Example: Merge Sort

Sorting Problem: Sort a sequence of n elements into non-decreasing order.

- ***Divide***: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each
- ***Conquer:*** Sort the two subsequences recursively using merge sort.
- ***Combine***: Merge the two sorted subsequences to produce the sorted answer.

Merge Sort – Example



Merge-Sort (A, p, r)

INPUT: a sequence of *n* numbers stored in array A

OUTPUT: an ordered sequence of *n* numbers

```
MergeSort (A, p, r)   // sort A[p..r] by divide & conquer
    if p < r
      then q ← ⌊(p+r)/2⌋
           MergeSort (A, p, q)
           MergeSort (A, q+1, r)
           Merge (A, p, q, r) // merges A[p..q] with A[q+1..r]
```

Initial Call: MergeSort(A, 1, n)

Procedure Merge

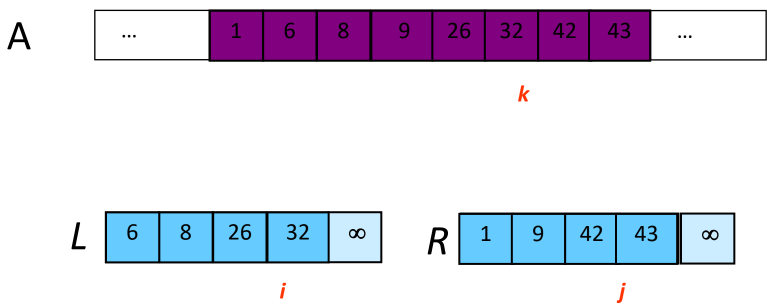
```
Merge(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
    for i ← 1 to n1
      do L[i] ← A[p + i − 1]
    for j ← 1 to n2
      do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
      do if L[i] ≤ R[j]
        then A[k] ← L[i]
             i ← i + 1
        else A[k] ← R[j]
             j ← j + 1
```

Input: Array containing sorted subarrays $A[p..q]$ and $A[q+1..r]$.

Output: Merged sorted subarray in $A[p..r]$.

Sentinels, to avoid having to check if either subarray is fully copied at each step.

Merge – Example



Correctness of Merge

```
Merge(A, p, q, r)
n1 ← q − p + 1
n2 ← r − q
    for i ← 1 to n1
        do L[i] ← A[p + i − 1]
    for j ← 1 to n2
        do R[j] ← A[q + j]
    L[n1+1] ← ∞
    R[n2+1] ← ∞
    i ← 1
    j ← 1
    for k ← p to r
        do if L[i] ≤ R[j]
            then A[k] ← L[i]
                 i ← i + 1
            else A[k] ← R[j]
                 j ← j + 1
```

Loop Invariant for the *for* loop

At the start of each iteration of the for loop:

Subarray $A[p..k-1]$

contains the $k - p$ smallest elements of L and R in sorted order.

$L[i]$ and $R[j]$ are the smallest elements of L and R that have not been copied back into A.

Initialization:

Before the first iteration:

- $A[p..k-1]$ is empty.
- $i = j = 1$.
- $L[1]$ and $R[1]$ are the smallest elements of L and R not copied to A.

Correctness of Merge

```
Merge(A, p, q, r)
1  n1 ← q − p + 1
2  n2 ← r − q
3      for i ← 1 to n1
4          do L[i] ← A[p + i − 1]
5      for j ← 1 to n2
6          do R[j] ← A[q + j]
7      L[n1+1] ← ∞
8      R[n2+1] ← ∞
9      i ← 1
10     j ← 1
11     for k ← p to r
12         do if L[i] ≤ R[j]
13             then A[k] ← L[i]
14                  i ← i + 1
15             else A[k] ← R[j]
16                  j ← j + 1
```

Maintenance:

Case 1: $L[i] \le R[j]$

- By LI, A contains $p - k$ smallest elements of L and R in sorted order.
- By LI, $L[i]$ and $R[j]$ are the smallest elements of L and R not yet copied into A.
- Line 13 results in A containing $p - k + 1$ smallest elements (again in sorted order). Incrementing i and k reestablishes the LI for the next iteration.

Similarly for $L[i] > R[j]$.

Termination:

- On termination, $k = r + 1$.
- By LI, A contains $r - p + 1$ smallest elements of L and R in sorted order.
- L and R together contain $r - p + 3$ elements. All but the two sentinels have been copied back into A.

Analysis of Merge Sort

- Running time $T(n)$ of Merge Sort:
- Divide: computing the middle takes $\Theta(1)$
- Conquer: solving 2 subproblems takes $2T(n/2)$
- Combine: merging n elements takes $\Theta(n)$
- Total:

$$T(n) = \Theta(1) \qquad \text{if } n = 1$$
$$T(n) = 2T(n/2) + \Theta(n) \qquad \text{if } n > 1$$

$\Rightarrow T(n) = \Theta(n \lg n)$ (CLRS, Chapter 4)