Analysis of Algorithms & Orders of Growth

Analysis of Algorithms

- An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.
- What is the goal of analysis of algorithms?
  - To compare algorithms mainly in terms of running time but also in terms of other factors (e.g., memory requirements, programmer's effort etc.)
- What do we mean by running time analysis?
  - **Determine how running time increases as the size of the problem increases**.

Example: Searching

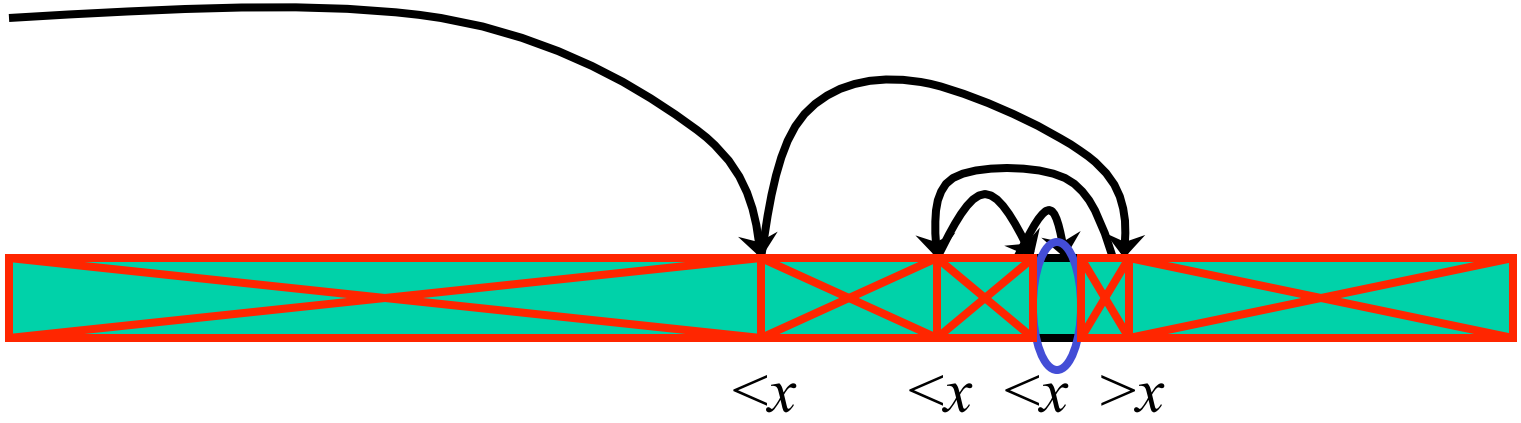
- Problem of *searching an ordered list*.
  - Given a list L of n elements that are sorted into a definite order (*e.g.*, numeric, alphabetical),
  - And given a particular element x,
  - Determine whether x appears in the list, and if so, return its index (position) in the list.

Search alg. #1: Linear Search

procedure *linear search*
(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while $(i \leq n \wedge x \neq a_i)$
$i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location* {index or 0 if not found}

Search alg. #2: Binary Search

- Basic idea: On each step, look at the *middle* element of the remaining list to eliminate half of it, and quickly zero in on the desired element.



Search alg. #2: Binary Search

procedure *binary search*
(x:integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$ {left endpoint of search interval}
$j := n$ {right endpoint of search interval}
while $i<j$ **begin** {while interval has >1 item}
$m := \lfloor (i+j)/2 \rfloor$ {midpoint}
if $x > a_m$ **then** $i := m+1$ **else** $j := m$
end
if $x = a_i$ **then** *location* $:= i$ **else** *location* $:= 0$
return *location*

Is Binary Search more efficient?

- **Number of iterations**:
  - For a list of n elements, Binary Search can execute at most $\log_2 n$ times!!
  - Linear Search, on the other hand, can execute up to n times !!

Average Number of Iterations

Length	Linear Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1,000	500.5	9.0
10,000	5000.5	12.0

Is Binary Search more efficient?

- **Number of computations per iteration**:
  - Binary search does more computations than Linear Search per iteration.
- **Overall**:
  - If the number of components is small (say, less than 20), then Linear Search is faster.
  - If the number of components is large, then Binary Search is faster.

How do we analyze algorithms?

- We need to define a number of objective measures.

(1) Compare execution times?

Not good: times are specific to a particular computer !!

(2) Count the number of statements executed?

Not good: number of statements vary with the programming language as well as the style of the individual programmer.

Example (# of statements)

Algorithm 1

```
arr[0] = 0;
arr[1] = 0;
arr[2] = 0;
 ...
arr[N-1] = 0;
```

Algorithm 2

```
for(i=0; i<N; i++)
  arr[i] = 0;
```

How do we analyze algorithms?

(3) Express running time as a function of the input size n (i.e., $f(n)$).

- To compare two algorithms with running times $f(n)$ and $g(n)$, we need a **rough measure** of **how fast a function grows**.
- Such an analysis is independent of machine time, programming style, etc.

Computing running time

- Associate a "cost" with each statement and find the "total cost“ by finding the total number of times each statement is executed.
- Express running time in terms of the size of the problem.

Algorithm 1

	Cost
arr[0] = 0;	c1
arr[1] = 0;	c1
arr[2] = 0;	c1
...	
arr[N-1] = 0;	c1

c1+c1+...+c1 = c1 x N

Algorithm 2

	Cost
for(i=0; i<N; i++)	c2
arr[i] = 0;	c1

(N+1) x c2 + N x c1 =
(c2 + c1) x N + c2

Computing running time (cont.)

```
                                        Cost
sum = 0;                                 c1
for(i=0; i<N; i++)                       c2
   for(j=0; j<N; j++)                    c2
        sum += arr[i][j];                c3
                                        ----------
```

$$c1 + c2 \; x \; (N+1) + c2 \; x \; N \; x \; (N+1) + c3 \; x \; N \; x \; N$$

Comparing Functions Using Rate of Growth

- Consider the example of buying *elephants* and *goldfish*:

  Cost: cost_of_elephants + cost_of_goldfish

  Cost ~ cost_of_elephants (**approximation**)

- The low order terms in a function are relatively insignificant for **large** n

$$n^4 + 100n^2 + 10n + 50 \quad \sim \quad n^4$$

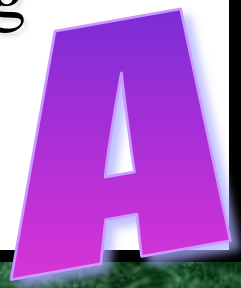
i.e., $n^4 + 100n^2 + 10n + 50$ and n^4 have the same rate of growth

Rate of Growth ≡Asymptotic Analysis

- Using *rate of growth* as a measure to compare different functions implies comparing them **asymptotically**.
- If $f(x)$ is *faster growing* than $g(x)$, then $f(x)$ always eventually becomes larger than $g(x)$ **in the limit** (for large enough values of x).

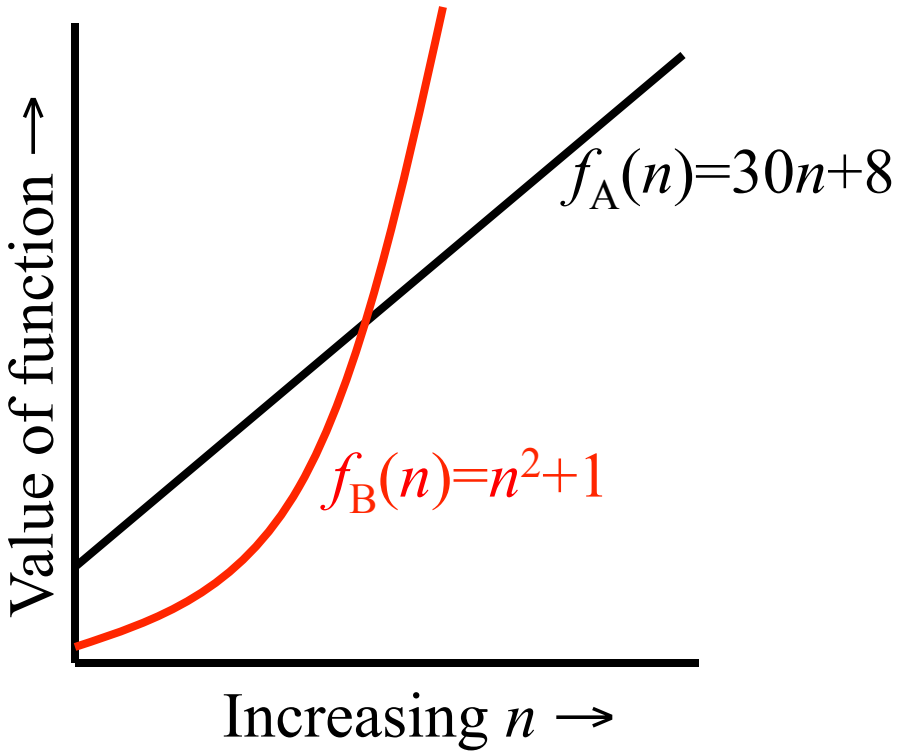
Example

- Suppose you are designing a web site to process user data (*e.g.*, financial records).
- Suppose program A takes $f_A(n)=30n+8$ microseconds to process any n records, while program B takes $f_B(n)=n^2+1$ microseconds to process the n records.
- Which program would you choose, knowing you'll want to support millions of users?



Visualizing Orders of Growth

- On a graph, as you go to the right, a faster growing function eventually becomes larger...



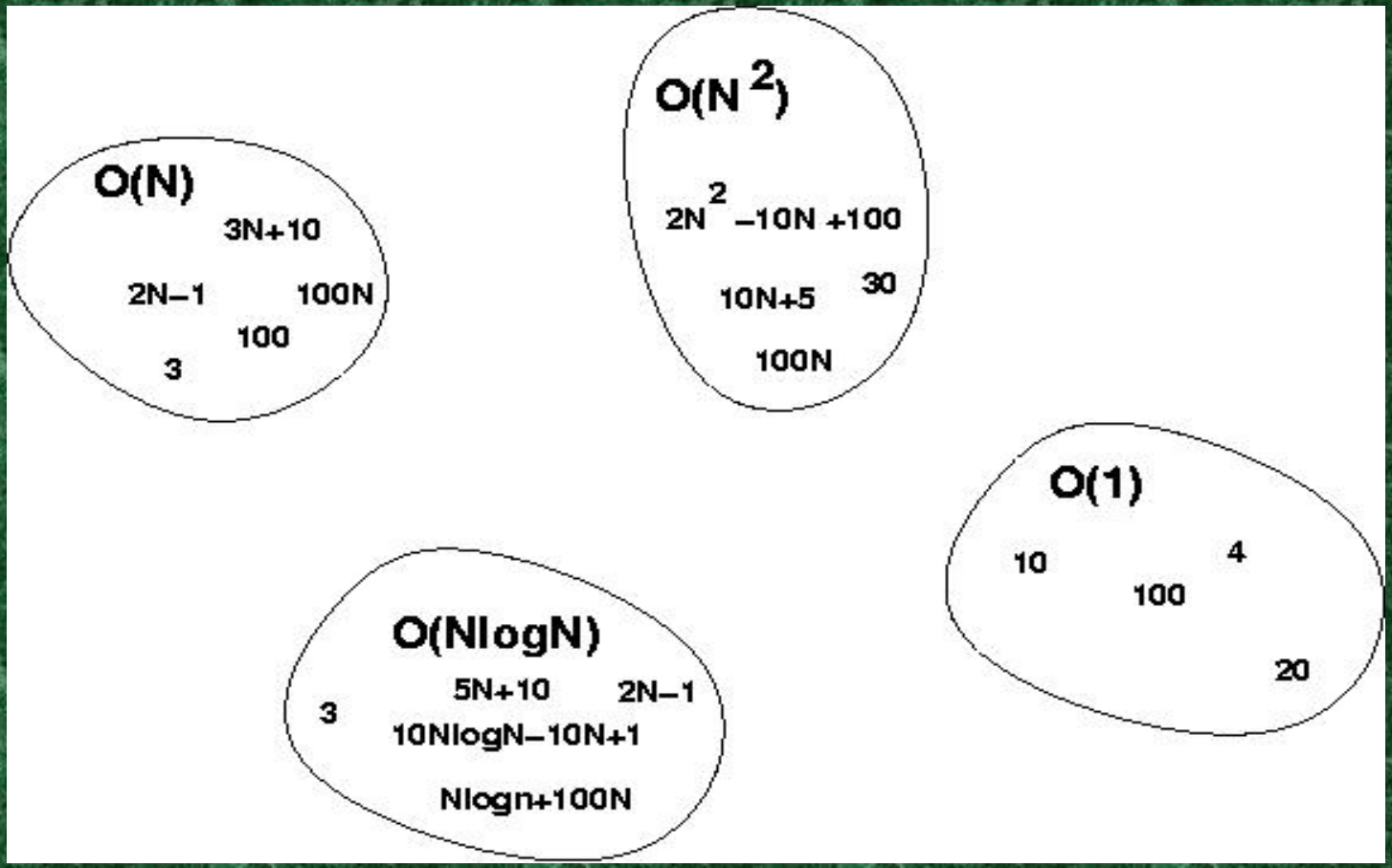
Big-O Notation

- We say $f_A(n)=30n+8$ is *order* n, or $O(n)$. It is, **at most**, roughly *proportional* to n.
- $f_B(n)=n^2+1$ is *order* n^2, or $O(n^2)$. It is, **at most**, roughly proportional to n^2.
- In general, an $O(n^2)$ algorithm will be slower than $O(n)$ algorithm.
- **Warning**: an $O(n^2)$ function will grow faster than an $O(n)$ function.

More Examples …

- We say that $n^4 + 100n^2 + 10n + 50$ is of the order of n^4 or $O(n^4)$
- We say that $10n^3 + 2n^2$ is $O(n^3)$
- We say that $n^3 - n^2$ is $O(n^3)$
- We say that 10 is $O(1)$,
- We say that 1273 is $O(1)$

Big-O Visualization



Computing running time

Algorithm 1

	Cost
arr[0] = 0;	c1
arr[1] = 0;	c1
arr[2] = 0;	c1
...	
arr[N-1] = 0;	c1

c1+c1+...+c1 = c1 x N

Algorithm 2

	Cost
for(i=0; i<N; i++)	c2
arr[i] = 0;	c1

(N+1) x c2 + N x c1 =
(c2 + c1) x N + c2

O(n)

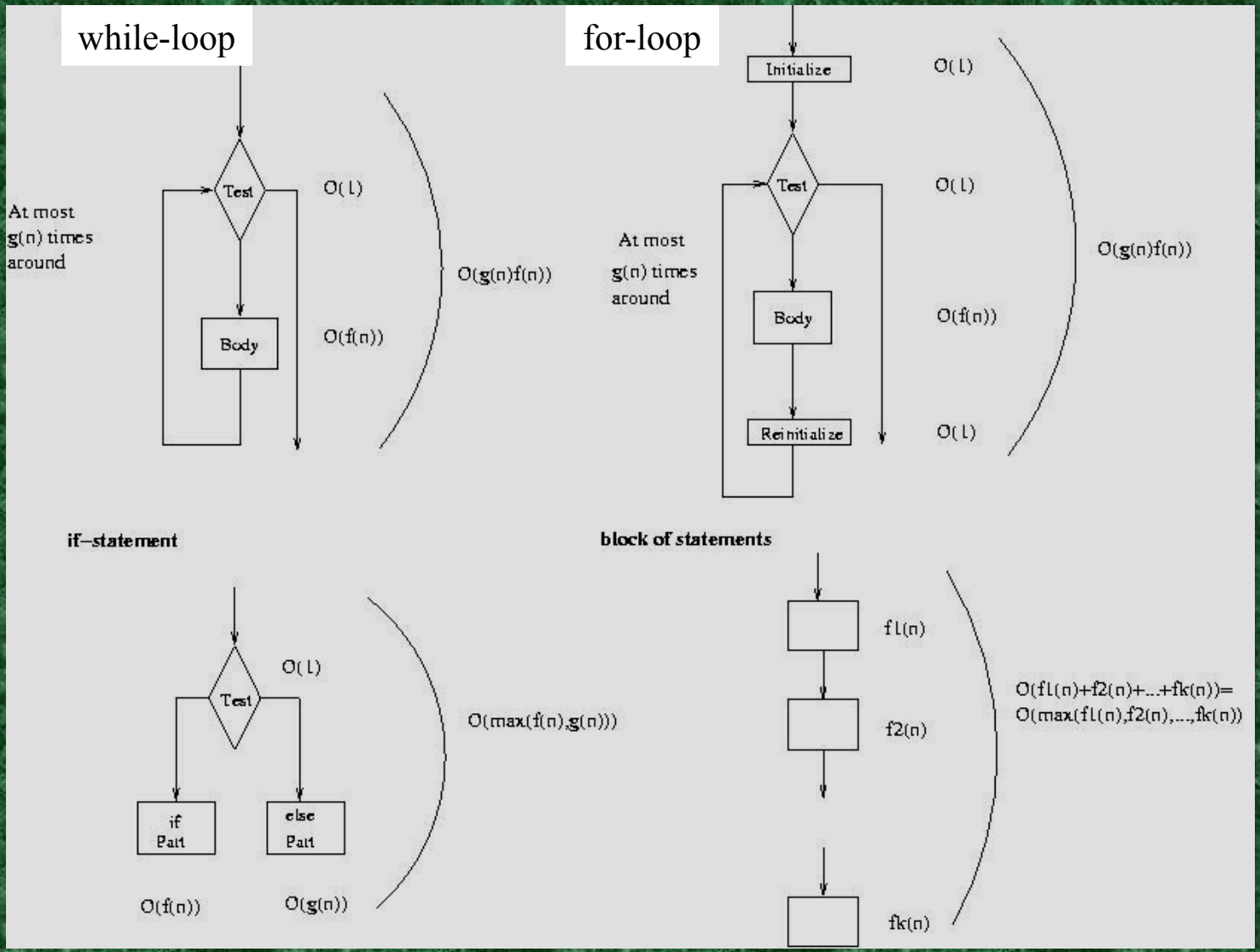
Computing running time (cont.)

```
                                        Cost
sum = 0;                                 c1
for(i=0; i<N; i++)                       c2
   for(j=0; j<N; j++)                    c2
        sum += arr[i][j];                c3
                                        ------------
```

$c1 + c2 \; x \; (N+1) + c2 \; x \; N \; x \; (N+1) + c3 \; x \; N \; x \; N$

$O(n^2)$

Running time of various statements



Examples

```
i = 0;
while (i<N) {
    X=X+Y;                    // O(1)
    result = mystery(X);      // O(N), just an example...
    i++;                      // O(1)
}
```

- The body of the while loop: O(N)
- Loop is executed: N times

$N \times O(N) = O(N^2)$

Examples (cont.' d)

```
if (i<j)
   for ( i=0; i<N; i++ )
        X = X+i;
else
   X=0;
```

O(N)

O(1)

Max (O(N), O(1)) = O (N)

Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n)=O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n)= Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality":
  - f(n)= Θ (g(n)) implies: f(n) "=" g(n)

Definition: *O(g)*, *at most order g*

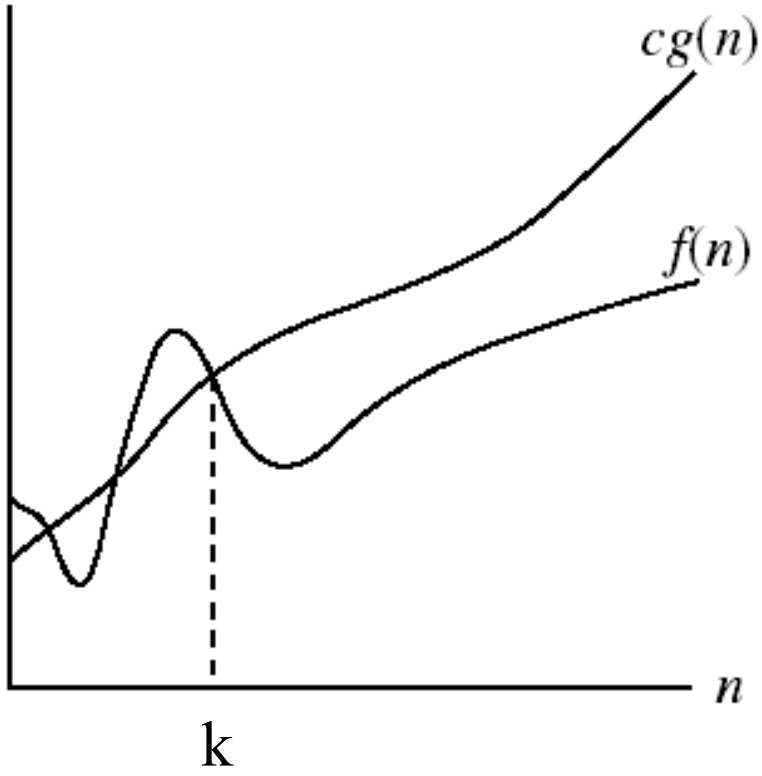
Let f,g are functions $\mathbf{R}\rightarrow\mathbf{R}$.

- We say that "*f* is *at most order g*", if:

$$\exists c,k: f(x) \leq cg(x), \forall x>k$$

  - "Beyond some point *k*, function *f* is at most a constant *c* times *g* (*i.e.*, proportional to *g*)."
- "*f* is *at most order g*", or "*f* is O(*g*)", or "f=O(g)" all just mean that $f\in$O(g).
- Sometimes the phrase "at most" is omitted.

Big-O Visualization



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

Points about the definition

- Note that f is $O(g)$ as long as ***any*** values of c and k exist that satisfy the definition.
- But: The particular c, k, values that make the statement true are *not* unique: **Any larger value of c and/or k will also work.**
- You are **not** required to find the smallest c and k values that work. (Indeed, in some cases, there may be no smallest values!)

However, you should **prove** that the values you choose do work.

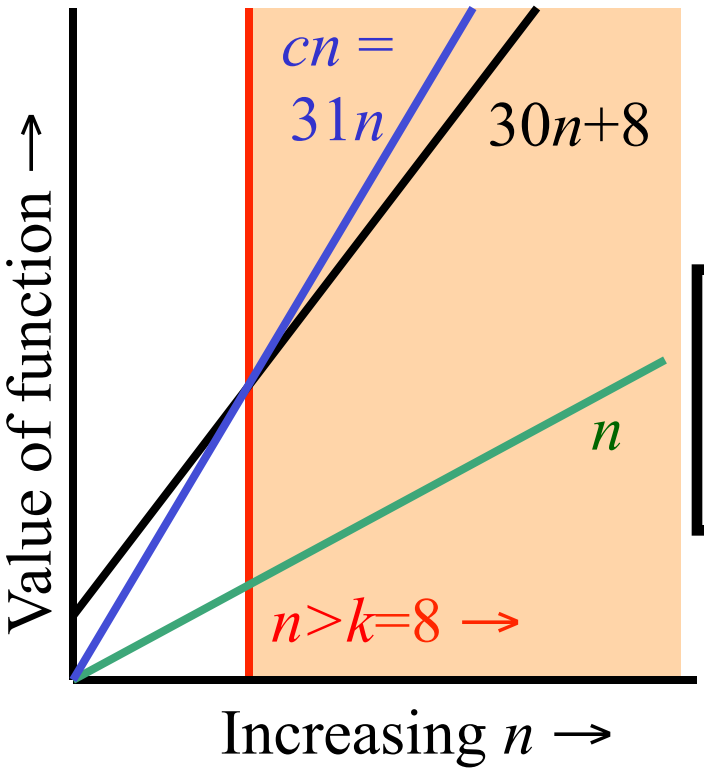
"Big-O" Proof Examples

- Show that $30n+8$ is $O(n)$.
  - Show $\exists c,k$: $30n+8 \leq cn$, $\forall n>k$.
    - Let $c=31$, $k=8$. Assume $n>k=8$. Then $cn = 31n = 30n + n > 30n+8$, so $30n+8 < cn$.
- Show that n^2+1 is $O(n^2)$.
  - Show $\exists c,k$: $n^2+1 \leq cn^2$, $\forall n>k$: .
    - Let $c=2$, $k=1$. Assume $n>1$. Then

    $cn^2 = 2n^2 = n^2+n^2 > n^2+1$, or $n^2+1< cn^2$.

Big-O example, graphically

- Note 30*n*+8 isn't less than *n* *anywhere* (*n*>0).
- It isn't even less than 31*n* *everywhere*.
- But it *is* less than 31*n* <u>everywhere to the right of *n*=8</u>.



$30n+8 \in O(n)$

Common orders of magnitude

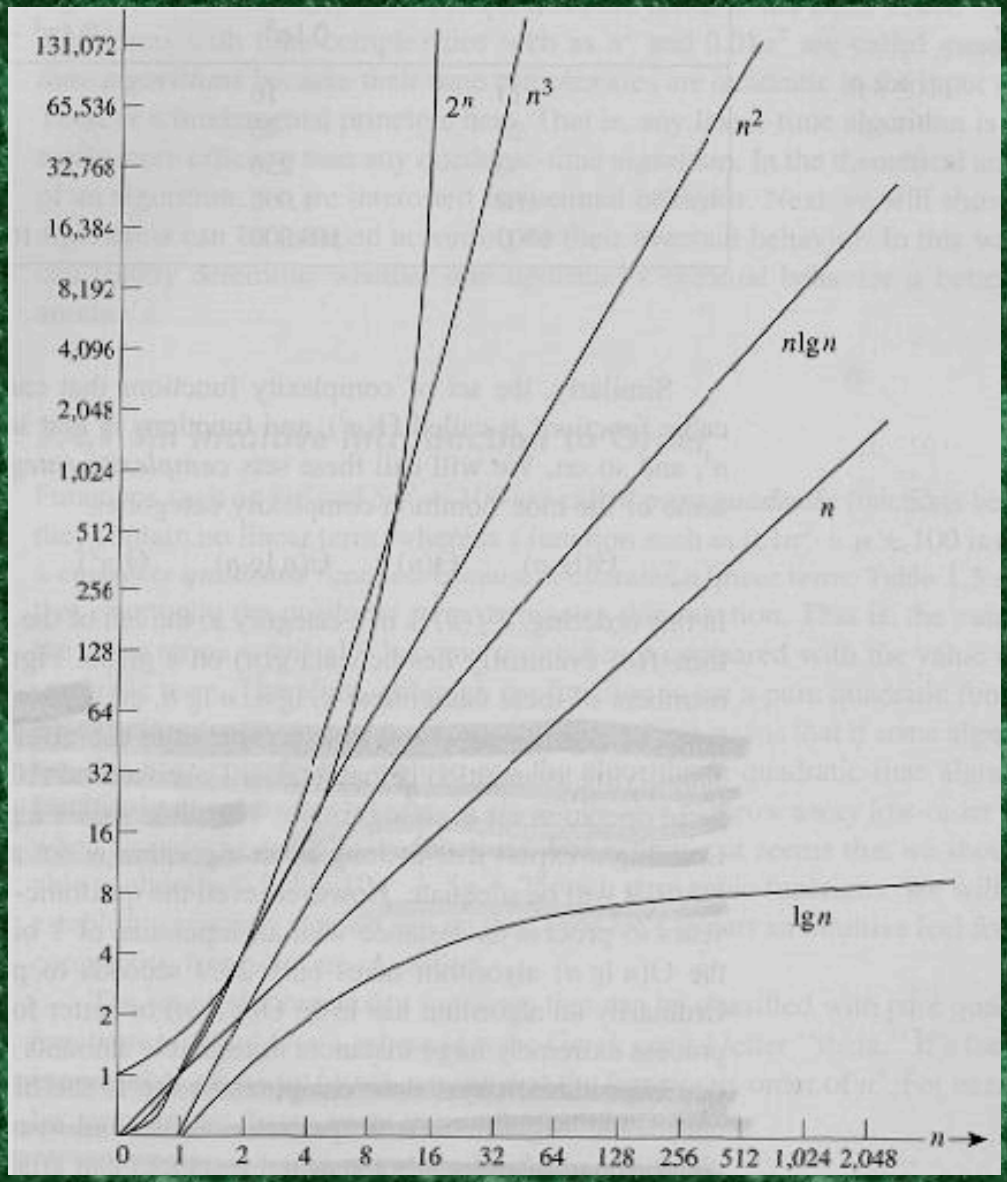


Table 1.4 Execution times for algorithms with the given time complexities

n	$f(n) = \lg n$	$f(n) = n$	$f(n) = n \lg n$	$f(n) = n^2$	$f(n) = n^3$	$f(n) = 2^n$
10	0.003 μs*	0.01 μs	0.033 μs	0.1 μs	1 μs	1 μs
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	8 μs	1 ms†
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	27 μs	1 s
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	64 μs	18.3 min
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	125 μs	13 days
10^2	0.007 μs	0.10 μs	0.664 μs	10 μs	1 ms	4×10^{15} years
10^3	0.010 μs	1.00 μs	9.966 μs	1 ms	1 s	
10^4	0.013 μs	10 μs	130 μs	100 ms	16.7 min	
10^5	0.017 μs	0.10 ms	1.67 ms	10 s	11.6 days	
10^6	0.020 μs	1 ms	19.93 ms	16.7 min	31.7 years	
10^7	0.023 μs	0.01 s	0.23 s	1.16 days	31,709 years	
10^8	0.027 μs	0.10 s	2.66 s	115.7 days	3.17×10^7 years	
10^9	0.030 μs	1 s	29.90 s	31.7 years		

*1 μs = 10^{-6} second.
†1 ms = 10^{-3} second.

Order-of-Growth in Expressions

- "O(f)" can be used as a term in an arithmetic expression .

E.g.: we can write "x^2+x+1" as "x^2+O(x)" meaning "x^2 plus some function that is O(x)".

- Formally, you can think of any such expression as denoting a set of functions:

$$\text{“}x^2+O(x)\text{”} :\equiv \{g \mid \exists f \in O(x): g(x) = x^2 + f(x)\}$$

Useful Facts about Big O

- Constants ...

  $\forall c>0,\ O(cf)=O(f+c)=O(f-c)=O(f)$

- Sums:

  - If $g \in O(f)$ and $h \in O(f)$, then $g+h \in O(f)$.

  - If $g \in O(f_1)$ and $h \in O(f_2)$, then

    $g+h \in O(f_1+f_2) = O(\max(f_1,f_2))$

    (Very useful!)

More Big-O facts

- Products:

  If $g \in O(f_1)$ and $h \in O(f_2)$, then $gh \in O(f_1 f_2)$

- Big O, as a relation, is *transitive*:

  $f \in O(g) \wedge g \in O(h) \rightarrow f \in O(h)$

More Big O facts

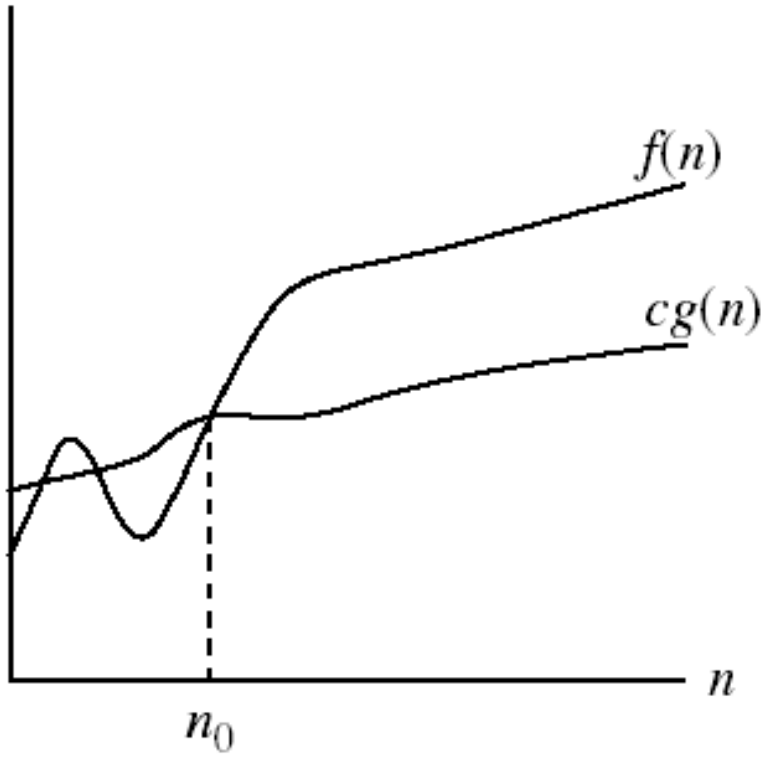
- $\forall$ f,g & constants $a,b \in \mathbf{R}$, with $b \geq 0$,
  - $af = O(f)$ (*e.g.* $3x^2 = O(x^2)$)
  - $f + O(f) = O(f)$ (*e.g.* $x^2 + x = O(x^2)$)
  - $|f|^{1-b} = O(f)$ (*e.g.* $x^{-1} = O(x)$)
  - $(\log_b |f|)^a = O(f)$ (*e.g.* $\log x = O(x)$)
  - $g = O(fg)$ (*e.g.* $x = O(x \log x)$)
  - $fg \neq O(g)$ (*e.g.* $x \log x \neq O(x)$)
  - $a = O(f)$ (*e.g.* $3 = O(x)$)

Definition: $\Omega(g)$, *at least order g*

Let f,g be any function $\mathbf{R}\rightarrow\mathbf{R}$.

- We say that "*f is at least order g*", written $\Omega(g)$, if
$$\exists c,k: f(x) \geq cg(x), \forall x>k$$
  - "Beyond some point k, function f is at least a constant c times g (*i.e.,* proportional to g)."
  - Often, one deals only with positive functions and can ignore absolute value symbols.
- "*f* is *at least order g*", or "*f* is $\Omega(g)$", or "$f=\Omega(g)$" all just mean that $f\in\Omega(g)$.

Big- Ω Visualization



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

Definition: $\Theta(g)$, *exactly order g*

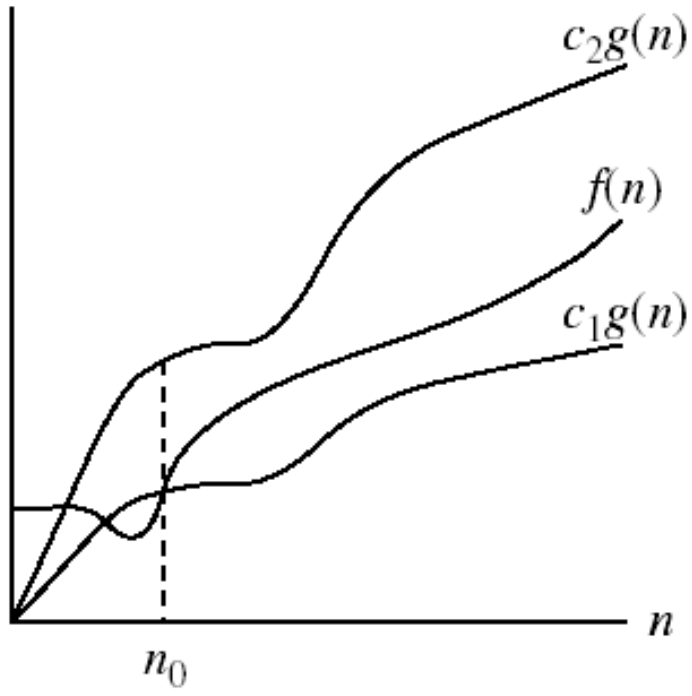
- If $f \in O(g)$ and $g \in O(f)$ then we say "*g and f are of the same order*" or "*f is (exactly) order g*" and write $f \in \Theta(g)$.
- Another equivalent definition:

  $$\exists c_1 c_2, k: c_1 g(x) \leq f(x) \leq c_2 g(x), \forall x > k$$

- "Everywhere beyond some point k, $f(x)$ lies in between two multiples of $g(x)$."
- $\Theta(g) \equiv O(g) \cap \Omega(g)$

  (i.e., $f \in O(g)$ and $f \in \Omega(g)$)

Big- Θ Visualization



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

Rules for Θ

- Mostly like rules for O( ), except:
- $\forall f,g>0$ & constants $a,b \in \mathbf{R}$, with $b>0$,
  $af \in \Theta(f)$ ← Same as with O.
  $f \notin \Theta(fg)$ unless $g=\Theta(1)$ ← Unlike O.
  $|f|^{1-b} \notin \Theta(f)$, and ← Unlike with O.
  $(\log_b |f|)^c \notin \Theta(f)$. ← Unlike with O.
- The functions in the latter two cases we say are *strictly of lower order* than $\Theta(f)$.

Θ example

- Determine whether: $\left(\sum_{i=1}^{n} i\right) \stackrel{?}{\in} \Theta(n^2)$
- Quick solution:

$$\left(\sum_{i=1}^{n} i\right) = n(n+1)/2$$

$$= n \cdot \Theta(n)/2$$

$$= n \cdot \Theta(n)$$

$$= \Theta(n^2)$$

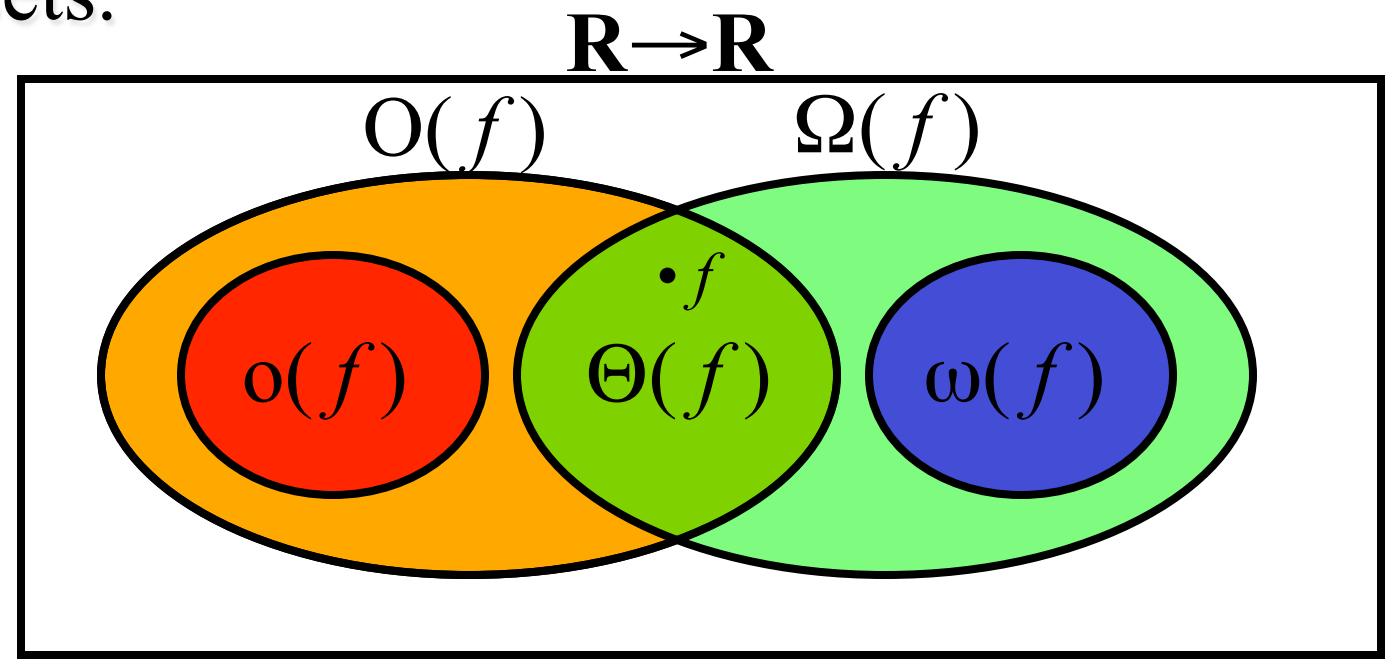
Other Order-of-Growth Relations

- $o(g) = \{f \mid \forall c\, \exists k: f(x) < cg(x), \forall x > k\}$
  "The functions that are *strictly lower order than g*." $o(g) \subset O(g) - \Theta(g)$.

- $\omega(g) = \{f \mid \forall c\, \exists k: cg(x) < f(x), \forall x > k\}$
  "The functions that are *strictly higher order than g*." $\omega(g) \subset \Omega(g) - \Theta(g)$.

Relations Between the Relations

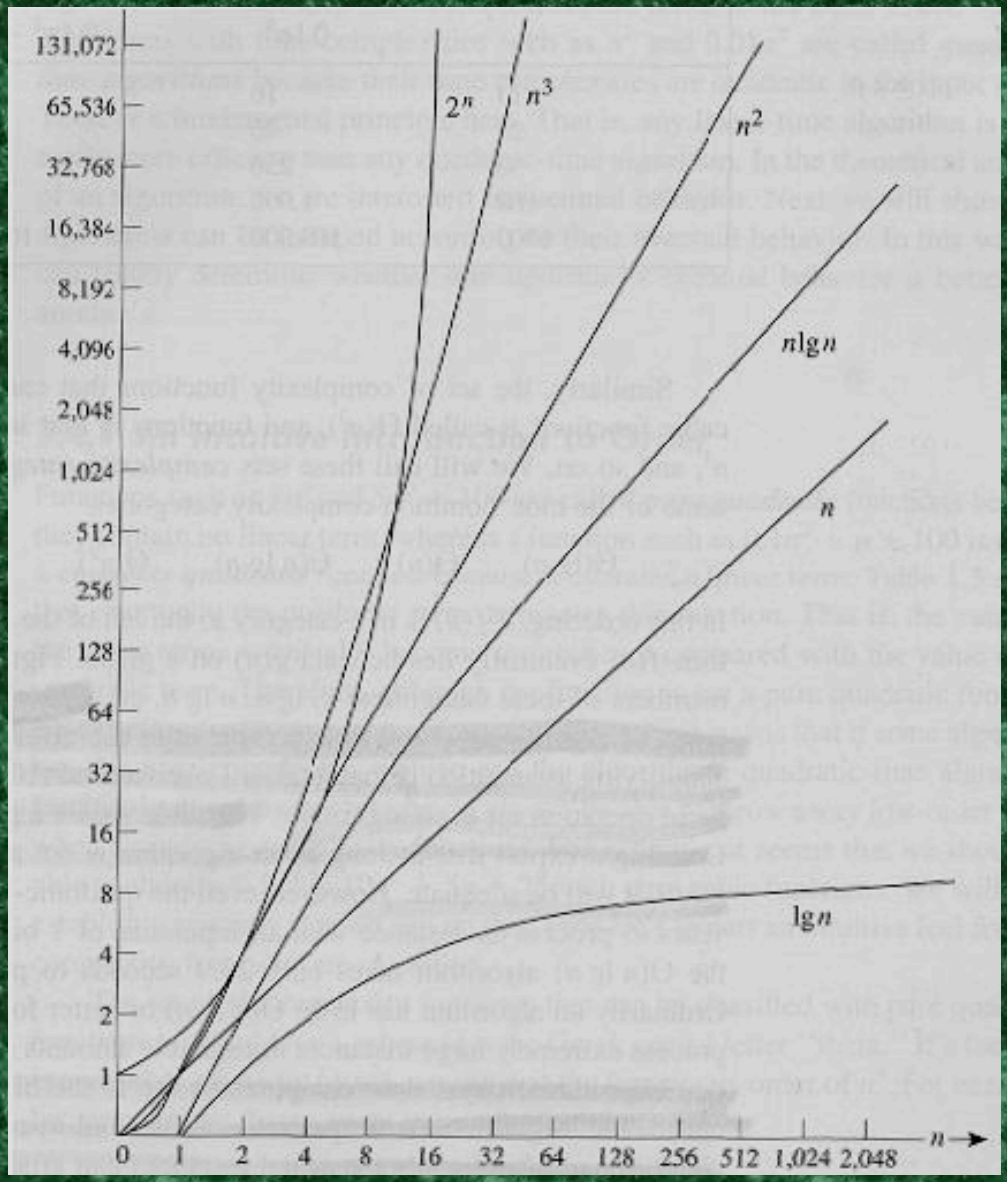
- Subset relations between order-of-growth sets.



Strict Ordering of Functions

- Temporarily let's write $f \prec g$ to mean $f \in o(g)$,
  $f \sim g$ to mean $f \in \Theta(g)$
- Note that $f \prec g \Leftrightarrow \lim_{x \to \infty} \frac{f(x)}{g(x)} = 0.$
- Let $k>1$. Then the following are true:
  $1 \prec \log\log n \prec \log n \sim \log_k n \prec \log^k n$
  $\prec n^{1/k} \prec n \prec n \log n \prec n^k \prec k^n \prec n! \prec n^n \ \ldots$

Common orders of magnitude



Review: Orders of Growth

Definitions of order-of-growth sets, $\forall g: \mathbf{R} \rightarrow \mathbf{R}$

- $O(g) \equiv \{f \mid \exists\, c,k: f(x) \leq cg(x),\ \forall x>k\ \}$
- $o(g) \equiv \{f \mid \forall c\ \exists k: f(x) < cg(x), \forall x>k\ \}$
- $\Omega(g) \equiv \{f \mid \mid \exists c,k : f(x) \geq cg(x), \forall x>k\ \}$
- $\omega(g) \equiv \{f \mid \forall\ c\ \exists k: f(x) > cg(x),\ \forall x>k\}$
- $\Theta(g) \equiv \{f \mid \exists c_1 c_2, k: c_1 g(x) \leq f(x) \mid \leq c_2 g(x),\ \forall x>k\}$

Algorithmic and Problem Complexity

Algorithmic Complexity

- The *algorithmic complexity* of a computation is some measure of how *difficult* it is to perform the computation.
- Measures some aspect of *cost* of computation (in a general sense of cost).

Problem Complexity

- The complexity of a computational *problem* or *task* is the complexity of the algorithm with the lowest order of growth of complexity for solving that problem or performing that task.
- *E.g.* the problem of searching an ordered list has *at most logarithmic* time complexity. (Complexity is O(log *n*).)

Tractable *vs.* Intractable Problems

- A problem or algorithm with <u>at most</u> polynomial time complexity is considered *tractable* (or *feasible*). **P** is the set of all tractable problems.
- A problem or algorithm that has more than polynomial complexity is considered *intractable* (or *infeasible*).
- **Note**
  - $n^{1,000,000}$ is *technically* tractable, but really impossible. $n^{\log \log \log n}$ is *technically* intractable, but easy.
  - Such cases are rare though.

Dealing with Intractable Problems

- Many times, a problem is intractable for a small number of input cases that do not arise in practice very often.
  - Average running time is a better measure of problem complexity in this case.
  - Find approximate solutions instead of exact solutions.

Unsolvable problems

- It can be shown that there exist problems that no algorithm exists for solving them.
- Turing discovered in the 1930's that there are problems unsolvable by *any* algorithm.
- Example: the *halting problem*
  - Given an arbitrary algorithm and its input, will that algorithm eventually halt, or will it continue forever in an "*infinite loop*?"

NP and NP-complete

- **NP** is the set of problems for which there exists a tractable algorithm for *checking solutions* to see if they are correct.
- **NP-complete** is a class of problems with the property that if any one of them can be solved by a polynomial worst-case algorithm, then all of them can be solved by polynomial worst-case algorithms.
  - *Satisfiability problem*: find an assignment of truth values that makes a compound proposition true.

P *vs.* **NP**

- We know **P⊆NP**, but the most famous unproven conjecture in computer science is that this inclusion is *proper* (*i.e.*, that **P⊂NP** rather than **P=NP**).
- It is generally accepted that no **NP-complete** problem can be solved in polynomial time.
- Whoever first proves it will be famous!

Questions

- Find the best big-O notation to describe the complexity of following algorithms:
  - A binary search algorithm.
  - A linear search algorithm.
  - An algorithm that finds the maximum element in a finite sequence.

Time Complexity

We can use Big-O to find the time complexity of algorithms (i.e., how long they take in terms of the number of operations performed).

There are two types of complexity normally considered.

- Worst-case complexity. The largest number of operations needed for a problem of a given size in the worst case. The number of operations that will guarantee a solution.
- Average-case complexity. The average number of operations used to solve a problem over all inputs of a given size.

Complexity of the Linear Search Algorithm

Worst-case complexity

- The algorithm loops over all the values in a list of n values.
  - At each step, two comparisons are made. One to see whether the end of the loop is reached, and one to compare the search element x with the element in the list.
  - Finally, one more comparison is made outside the loop.
  - If x is equal to list element a_i, then $2i + 1$ comparisons are made.
  - If x is not in the list, then $2n + 2$ comparisons are made. ($2n$ comparisons to determine that x is not in the list, an additional comparison is used to exit the loop and one comparison is made outside the loop.
  - The worst-case complexity is thus $2n$ and is $O(n)$.

Complexity of the Linear Search Algorithm

Average-case complexity

- The algorithm loops over all the values in a list of n values.
  - If x is the ith term of the list, 2 comparisons will be used at each of the i steps of the loop, and one outside the loop , so that a total of 2i+1 comparisons are needed.
  - On average, the number of comparisons is
  - $\frac{3 + 5 + 7 + \ldots + (2n+1)}{n} = \frac{2(1 + 2 + 3 + \ldots\ldots + n) + n}{n} = \frac{2[n\,(n+1)/2]}{n} + 1 = n + 2$

What's the numerator? $2\left(\sum_{k=1}^{n} k = \frac{n(n+1)}{2}\right)$

Average case is O(n)