Data Structures and Algorithms - L2

Prof.N.L.Bhanu Murthy

BITS Pilani

Hyderabad Campus

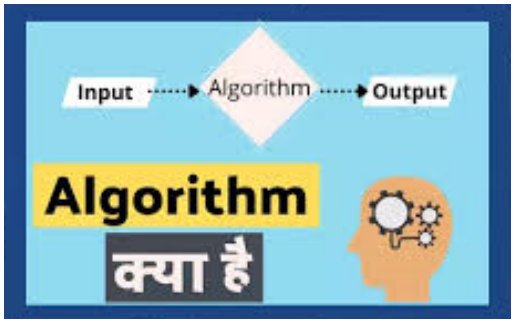
What is an algorithm?

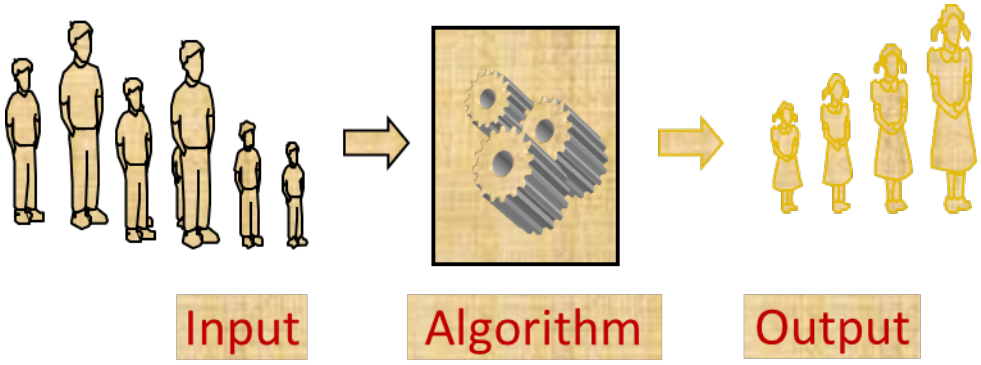
A clearly specified set of simple instructions to be followed to solve a problem

- ✓Takes a set of values, as input and
- ✓ produces a value, or set of values, as output

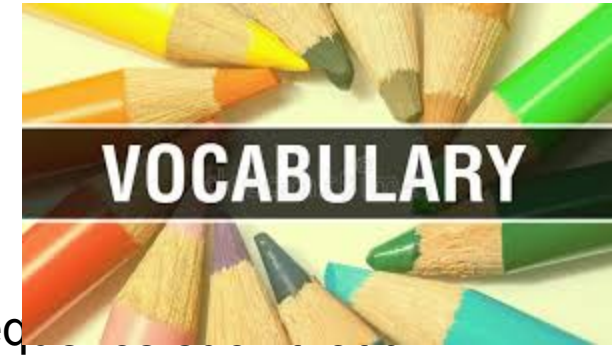
May be specified

- ✓In English or Hindi or Telugu
- ✓As a computer program
- ✓As a pseudo-code





Some Vocabulary with an example



Problem: Sorting of given keys

Input: A sequence of *n* keys *a*1, . . . , *an*.

Output: The permutation (reordering) of the input seq
≤ *a*2 ≤ · · · ≤ *an*−1 ≤ *an*.

Instance: An *instance* of sorting might be an array of names, like *{Mike, Bob, Sally, Jill ,Jan}*, or a list of numbers like *{154, 245, 568, 324, 654, 324}*

Algorithm: An *algorithm* is a procedure that takes any of the possible input instances and transforms it to the desired output.

Which algorithm is better?

Who is topper in BPHC?

Algorithm 1:

Sort *A* into decreasing order

Output *A*[1].

Which is better?

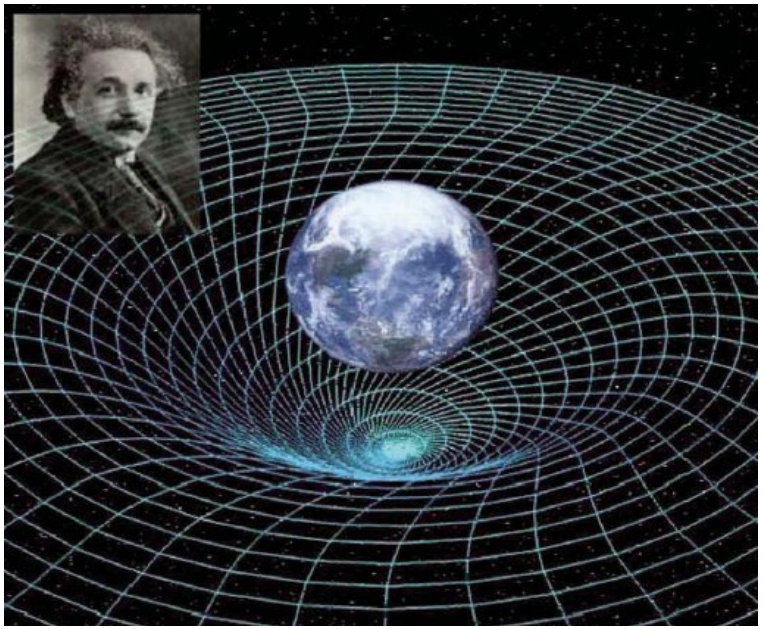
Algorithm 2:

```
int i;
int m = A[1];
for (i = 2; i <= n; i ++)
    if (A[i] > m)
        m = A[i];
return m;
```

Who's the champion?

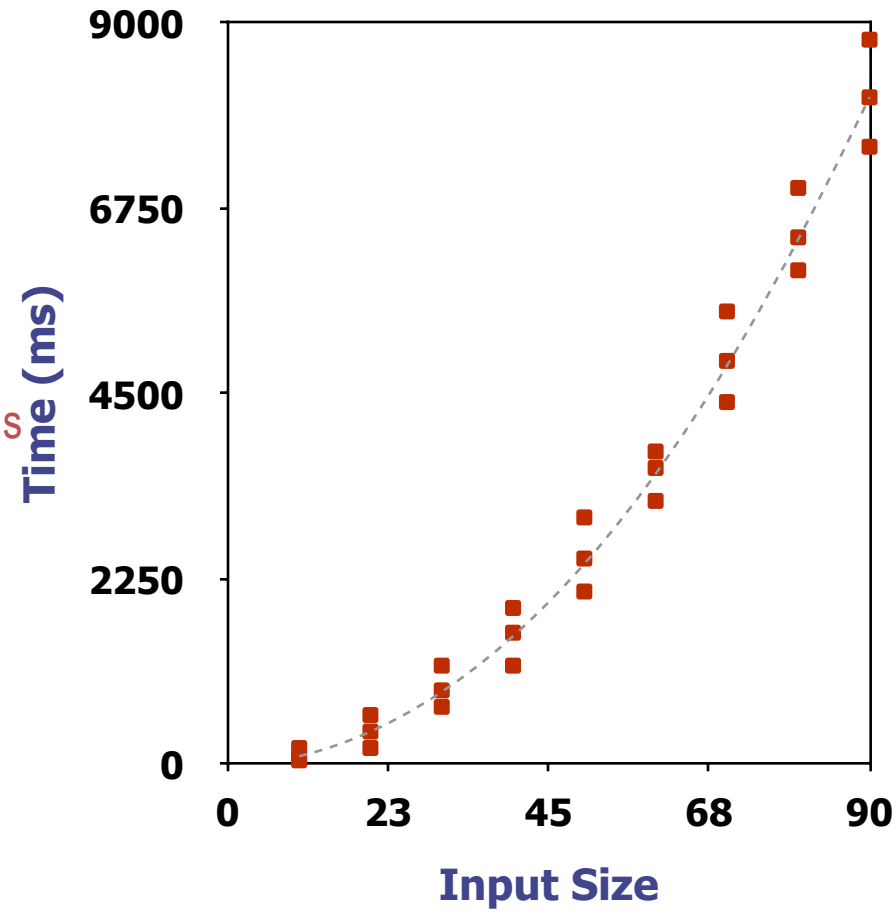
“Better” = more efficient

- ✓ Time
- ✓ Space

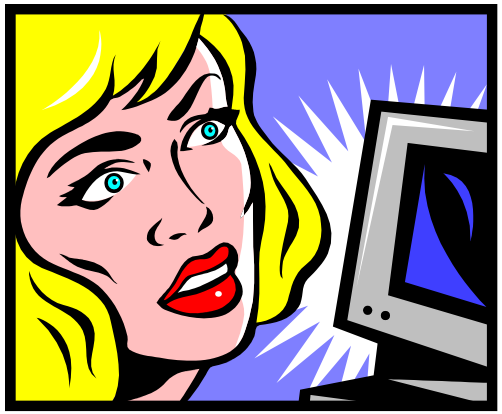


Experimental Studies

- ✓ Write a program implementing the algorithm
- ✓ Run the program with inputs of varying size and composition
- ✓ Use a method like System.currentTimeMillis() or Clock functions to get an accurate measure of the actual running time
- ✓ Plot the results

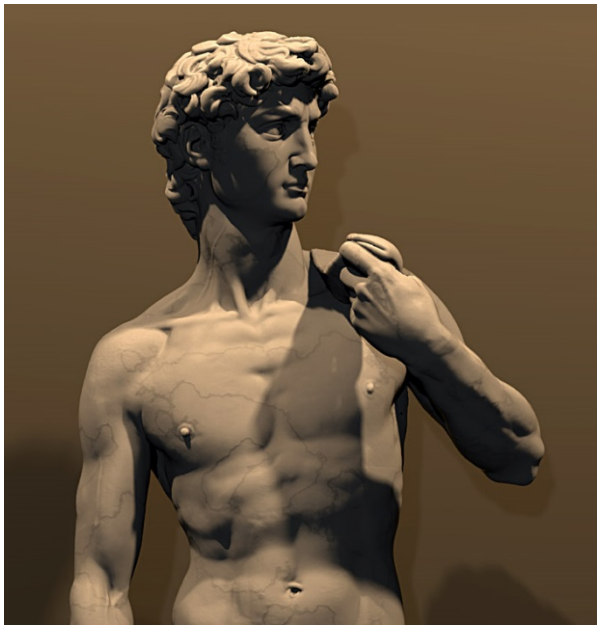


Limitations of Experiments



- ✓ Results may not be indicative of the running time **as all inputs may not be included** in the experiment.
- ✓ In order to compare two algorithms, the same hardware and software environments must be used
- ✓ It is necessary to implement the algorithm, and the efficiency of implementation varies

Asymptotic notation



$O(n)$

$o(n)$

$\Omega(n)$

$\omega(n)$

$\Theta(n)$

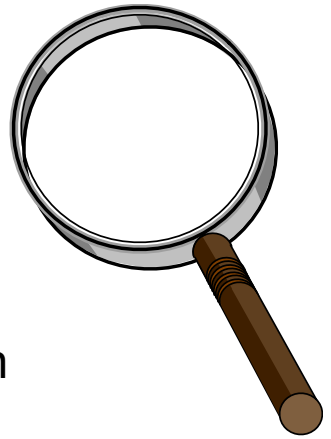
Edmund Landau

- 1877~1938
- Inventor of the asymptotic notation

Donald E. Knuth

- 1938 ~
- Turing Award, 1974.
- Father of the analysis of algorithms
- Popularizing the asymptotic notation

Theoretical Analysis



- ✓ Uses a high-level description of the algorithm instead of an implementation
- ✓ Characterizes running time as a function of the input size, *n*.
- ✓ Takes into account all possible inputs
- ✓ Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

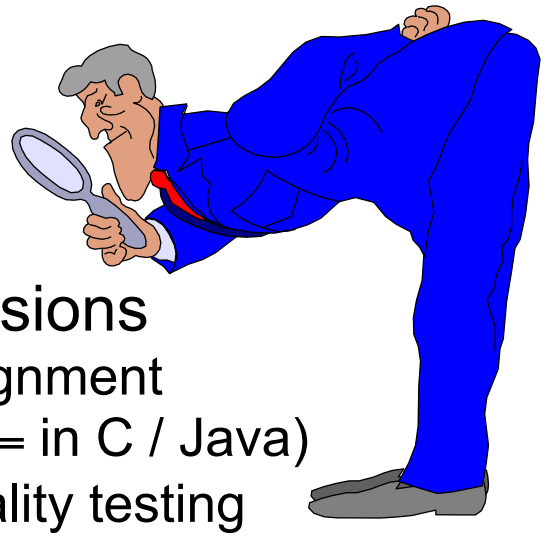
- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

```
Algorithm arrayMax(A, n)
    Input array A of n integers
    Output maximum element of A

    currentMax ← A[0]
    for i ← 1 to n – 1 do
        if A[i] > currentMax then
            currentMax ← A[i]
    return currentMax
```

Pseudocode Details



- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (***arg*** [, ***arg***…])

  Input …

  Output …

- Expressions
  - ← Assignment (like = in C / Java)
  - = Equality testing (like == in C / Java)
  - n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

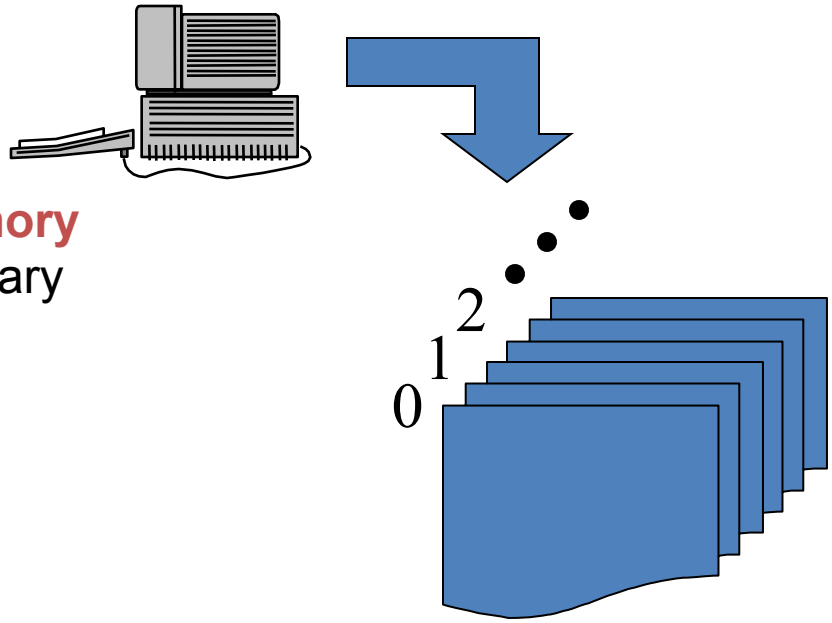
- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

Examples:

- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

The Random Access Machine (RAM) Model (*)

- A **CPU**
- A potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



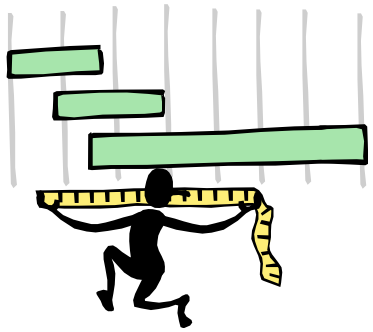
- ✓ Memory cells are numbered and accessing any cell in memory takes unit time.

Counting Primitive Operations

➢ By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm *arrayMax*(*A*, *n*)	# operations
currentMax ← *A*[0]	2
for (*i* =1; i<n; i++)	2*n*
(i=1 once, i<n n times, i++ (n-1) times)	
if *A*[*i*] > *currentMax* **then**	2(*n* – 1)
currentMax ← *A*[*i*]	2(*n* – 1)
return *currentMax*	1
Total	6*n* –1

Estimating Running Time



- Algorithm ***arrayMax*** executes $6n - 1$ primitive operations in the worst case.

- Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation

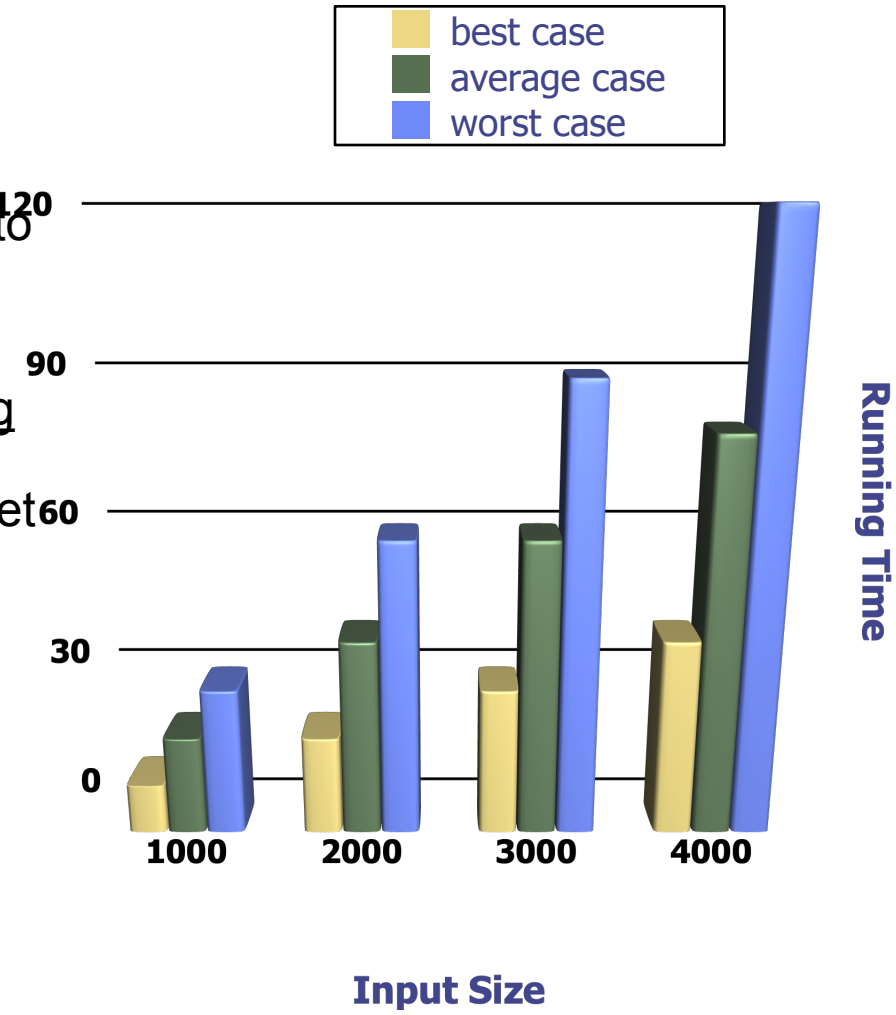
- Let $T(n)$ be worst-case time of ***arrayMax.*** Then

$$a(6n - 1) \leq T(n) \leq b(6n - 1)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Running Time

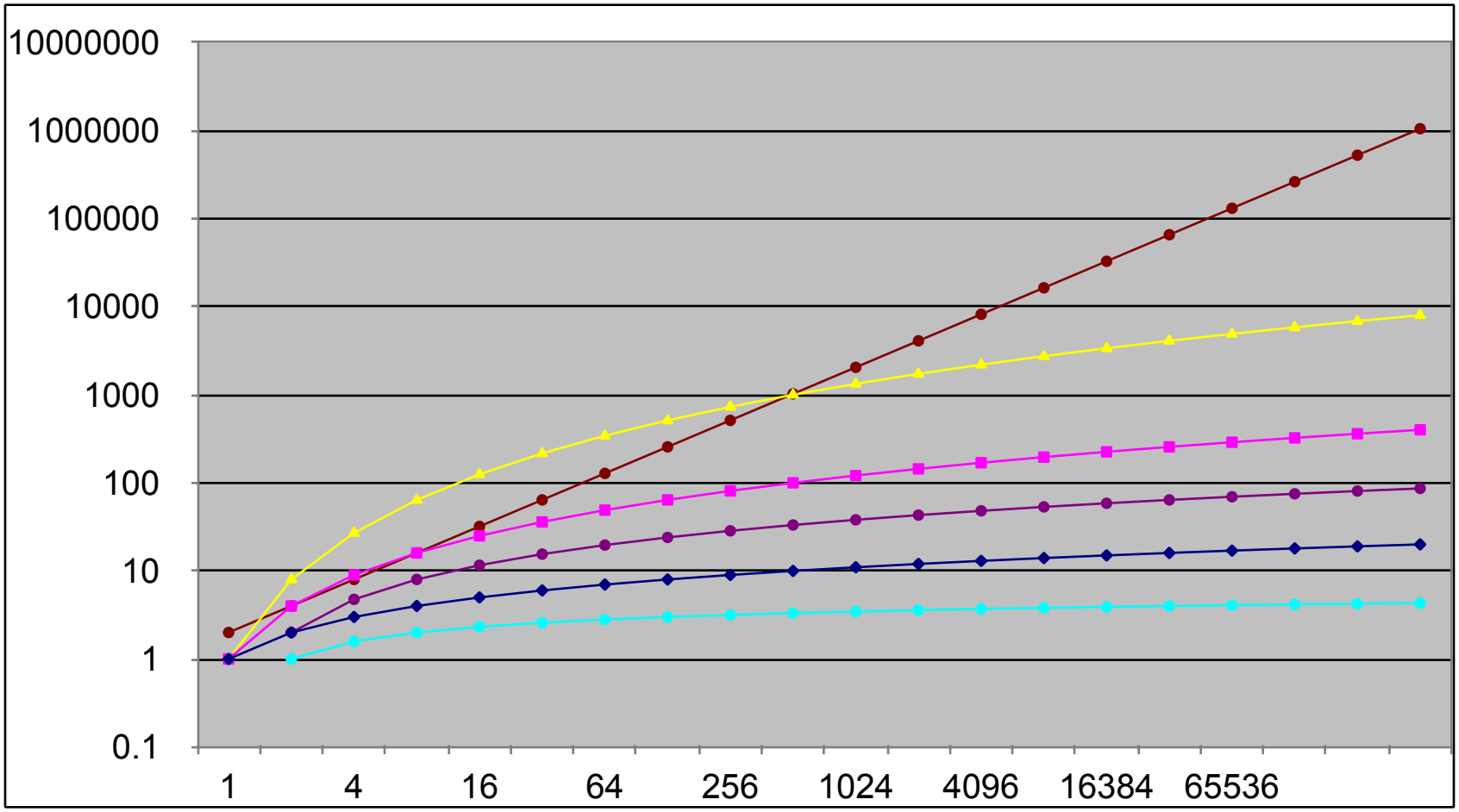
- What is best, average, worst case running of a problem?
- Average case time is often difficult to determine – why?
- We focus on the worst case running time.
  - Easier to analyze and best to bet
  - Crucial to applications such as games, finance and robotics



The Growth Rate of the Six Popular functions

n	$\log n$	n	$n\log n$	n^2	n^3	2^n
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4,096	65,536
32	5	32	160	1,024	32,768	4,294,967,296
64	6	64	384	4,094	262,144	$1.84 * 10^{19}$
128	7	128	896	16,384	2,097,152	$3.40 * 10^{38}$
256	8	256	2,048	65,536	16,777,216	$1.15 * 10^{77}$
512	9	512	4,608	262,144	134,217,728	$1.34 * 10^{154}$
1024	10	1,024	10,240	1,048,576	1,073,741,824	$1.79 * 10^{308}$

Practical Complexity



Asymptotic Dominance in Action

n $f(n)$	$\lg n$	n	$n \lg n$	n^2	2^n	$n!$
10	0.003 μs	0.01 μs	0.033 μs	0.1 μs	1 μs	3.63 ms
20	0.004 μs	0.02 μs	0.086 μs	0.4 μs	1 ms	77.1 years
30	0.005 μs	0.03 μs	0.147 μs	0.9 μs	1 sec	8.4×10^{15} yrs
40	0.005 μs	0.04 μs	0.213 μs	1.6 μs	18.3 min	
50	0.006 μs	0.05 μs	0.282 μs	2.5 μs	13 days	
100	0.007 μs	0.1 μs	0.644 μs	10 μs	4×10^{13} yrs	
1,000	0.010 μs	1.00 μs	9.966 μs	1 ms		
10,000	0.013 μs	10 μs	130 μs	100 ms		
100,000	0.017 μs	0.10 ms	1.67 ms	10 sec		
1,000,000	0.020 μs	1 ms	19.93 ms	16.7 min		
10,000,000	0.023 μs	0.01 sec	0.23 sec	1.16 days		
100,000,000	0.027 μs	0.10 sec	2.66 sec	115.7 days		
1,000,000,000	0.030 μs	1 sec	29.90 sec	31.7 years		

Implications of Dominance

- Exponential algorithms get hopeless fast.
- Quadratic algorithms get hopeless at or before 1,000,000.
- $O(n \log n)$ is possible to about one billion.
- $O(\log n)$ never sweats.

Asymptotic Dominance in Action

n	*n*	*nlogn*	n^2	n^3	n^4	n^{10}	2^n
1000	1mic	10mic	1milli	1sec	17min	3.2×10^{13} years	3.2×10^{283} years
10000	10mic	130mic	100milli	17min	116 days	???	???
10^6	1milli	20milli	17min	32years	3×10^7 years	??????	??????

10^9 instructions/second

Faster Computer Vs Better Algorithm

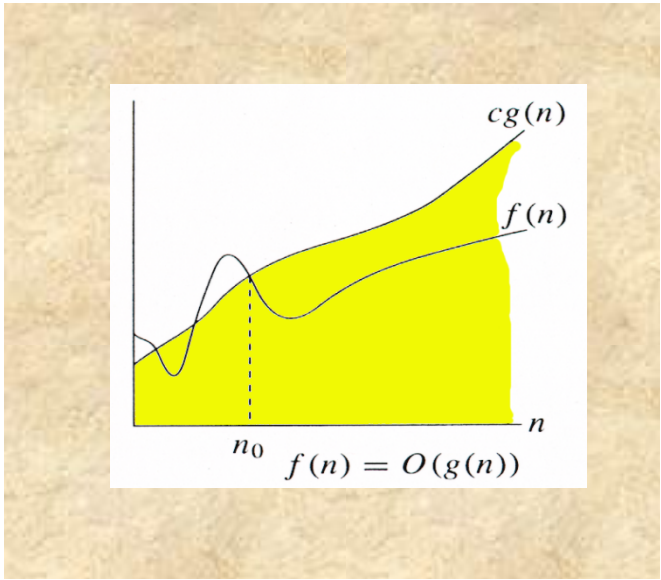
Algorithmic improvement more useful than hardware improvement.

E.g. 2^n to n^3

Big-Oh Notation

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

$f(n) \leq cg(n)$ for $n \geq n_0$



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

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

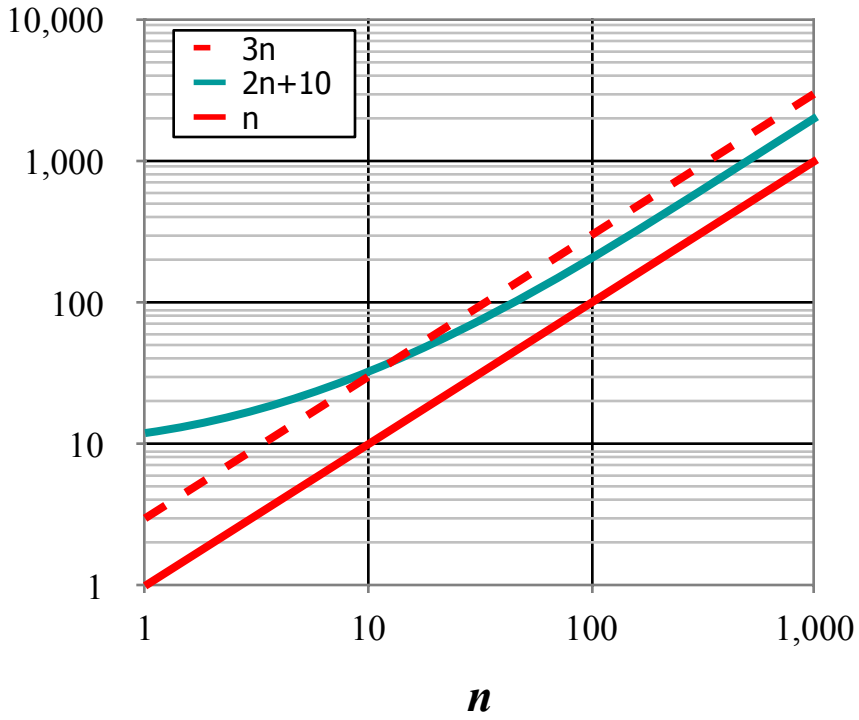
Example

$O(n^2) = \{2n^2 + 10n + 2000, 125n^2 - 233n - 250, 10n + 25, 150, \ldots\}$

Big-Oh Notation

Example: $2n + 10$ is $O(n)$

- $2n + 10 \leq cn$
- $(c - 2)\, n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$



To simplify the running time estimation, for a function *f(n),*

we ignore the constants and lower order terms.

Example: $10n^3+4n^2-4n+5$ is $O(n^3)$.

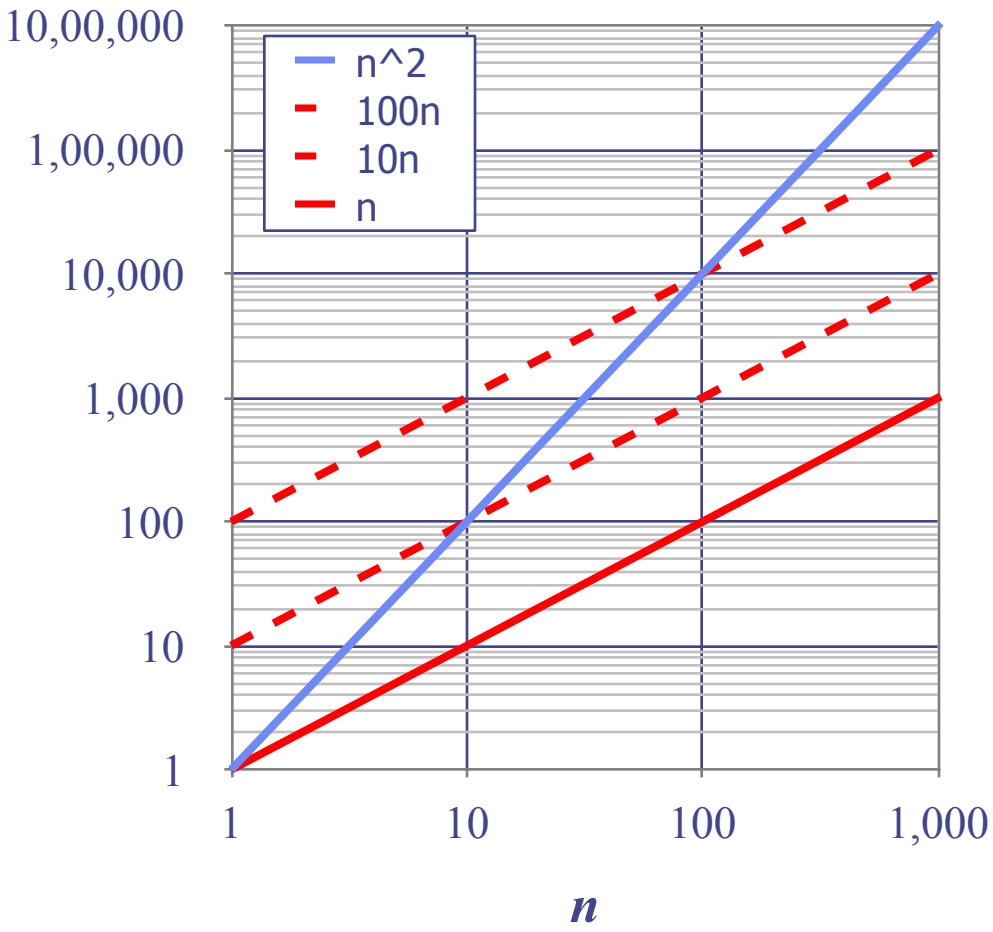
Big-Oh Example

Example: the function n^2 is not $O(n)$

$$n^2 \leq cn$$

$$n \leq c$$

The above inequality cannot be satisfied since c must be a constant and hence n^2 is $O(n^2)$.



More Big-Oh Examples

- 7n-2

  7n-2 is O(n)
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \bullet n$ 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 $3n^3 + 20n^2 + 5 \leq c \bullet n^3$ 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 $3 \log n + 5 \leq c \bullet \log n$ for $n \geq n_0$
  this is true for $c = 8$ and $n_0 = 2$

Big-Oh Rules

- ✓ If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  - ➢ Drop lower-order terms
  - ➢ Drop constant factors

- ✓ Use the smallest possible class of functions
  - ➢ Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"

- ✓ Use the simplest expression of the class
  - ➢ Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Relatives of Big-Oh

$O(n)$

$o(n)$

$\Omega(n)$

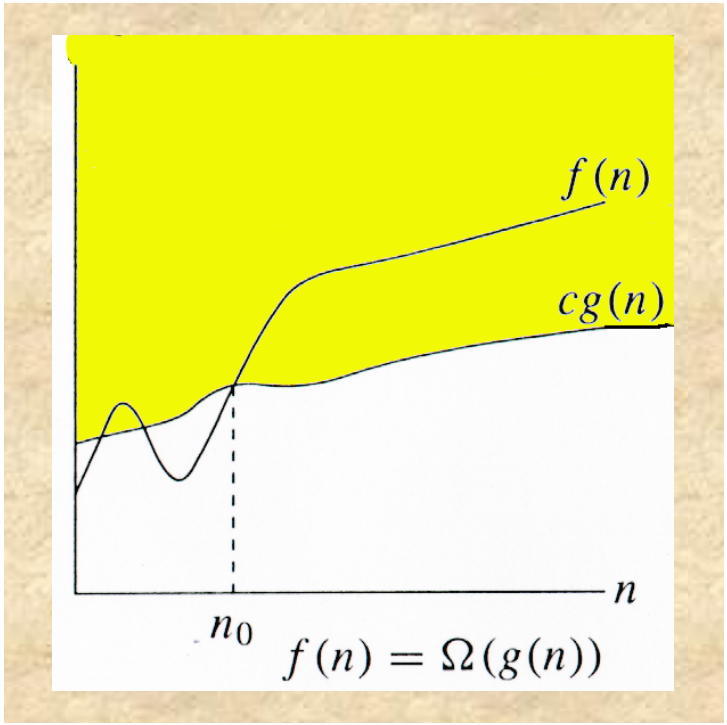
$\omega(n)$

$\Theta(n)$

Relatives of Big-Oh (Ω(n))

- **big-Omega**

  f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that f(n) ≥ c•g(n) for n ≥ n_0



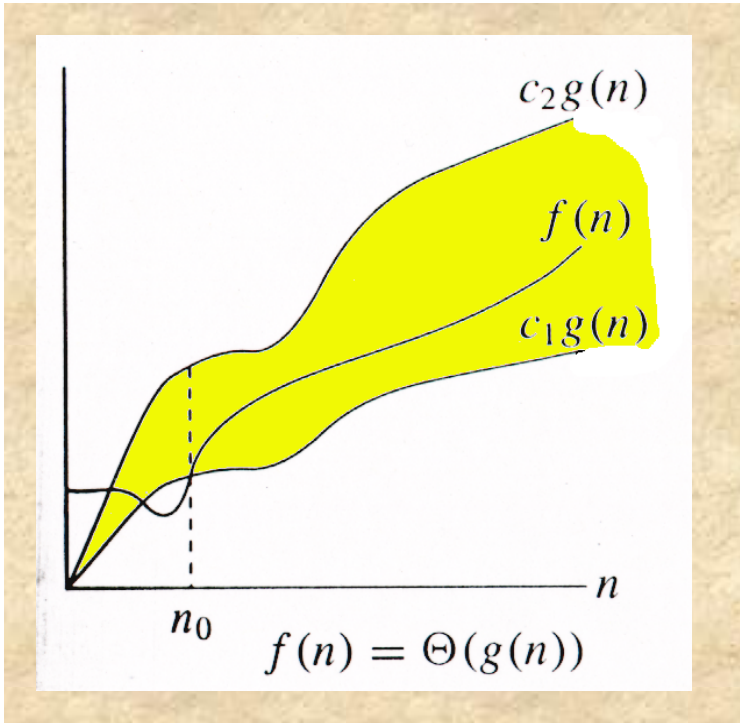
For function *g*(*n*), we define Ω(*g*(*n*)), big-Omega of *n*, as the set:

Ω(*g*(*n*)) = {*f*(*n*) : ∃ positive constants *c* and n_0, such that 0 ≤ *cg*(*n*) ≤ *f*(*n*) ∀*n* ≥ *n*0}

Relatives of Big-Oh (Θ(n))

- **big-Theta**
  - f(n) is Θ(g(n)) if there are constants c' > 0 and c'' > 0 and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$

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 $f \in \Theta(g)$.



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

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

Relatives of Big-Oh (small oh – o(n))

Example
$O(n^2)$ = {$2n^2 + 10n + 2000$, $125n^2 – 233n - 250$, $10n + 25$, 150, …..}

What is the difference between

n^2 being the asymptotic upper bound ***$2n^2 + 10n + 2000$***

And

n^2 being the asymptotic upper bound of $10n + 25$?

n^2 is **Asymptotically Tight Upper Bound** of ***$2n^2 + 10n + 2000$***

Relatives of Big-Oh (small o – o(n))

For a given function $g(n)$,

$o(g(n)) = \{f(n): \forall\ c > 0, \exists\ n_0 > 0$ such that $\forall\ n \geq n_0$, we have $0 \leq f(n) < cg(n)\}$.

$\forall c > 0, \exists\ n_0 > 0$ such that $0 \leq f(n) < cg(n)\ \forall\ n \geq n_0$

$\Rightarrow \forall\ c > 0, \exists\ n_0 > 0$ such that $0 \leq f(n) / g(n) < c\ \forall\ n \geq n_0$

$\Rightarrow \lim_{n \to \infty} [f(n) / g(n)] = 0$

Alternatively

$o(g(n)) = \{f(n): \lim_{n \to \infty} [f(n) / g(n)] = 0\}$

Example

$o(n^2) = \{10n + 25, 150, ...\}$

Relatives of Big-Oh (ω(n))

For a given function $g(n)$,

$\omega(g(n))$ = {$f(n)$: $\forall$ $c > 0$, $\exists$ $n_0 > 0$ such that
$\forall$ $n \geq n_0$, we have $0 \leq cg(n) < f(n)$}.

$g(n)$ is a ***lower bound*** for $f(n)$ that is not asymptotically tight.

Alternatively
$w(g(n))$ = {$f(n)$: $\lim_{n \to \infty} [f(n) / g(n)] = \infty$}

Example
$w(n^2)$ = {$8n^3 + 25, 150, \ldots..$}

Example Uses of the Relatives of Big-Oh

- **3logn+loglogn is Ω(logn)**

  f(n) is Ω(g(n)) if there is a constant c > 0 and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  let c = 3 and $n_0 = 2$

- **3logn+loglogn is Θ(logn)**
  - **f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)**
- **$12n^2 + 6n$ is o(n^3)**
  - **f(n) is o(g(n)) if f(n) is asymptotically strictly less than g(n)**
- **$12n^2 + 6n$ is ω(n)**
  - **f(n) is ω(g(n)) if f(n) is asymptotically strictly greater than g(n)**

Some more Examples

- **$3n + 2$ is $\Omega(n)$**

  As $3n + 2 >= 3n$ for $n>=1$, though the inequality holds for $n>=0$, for Ω we need $n_0>0$

- **$3n + 2$ is $\Theta(n)$**
  - **As $3n+2 >=3n$ for all $n>=2$ and $3n+2 <=4n$ for all $n>=2$, $c1=3$, $c2=4$ and $n0=2$**
- **$3n + 2$ is $o(n^2)$**

Useful Facts about Big O

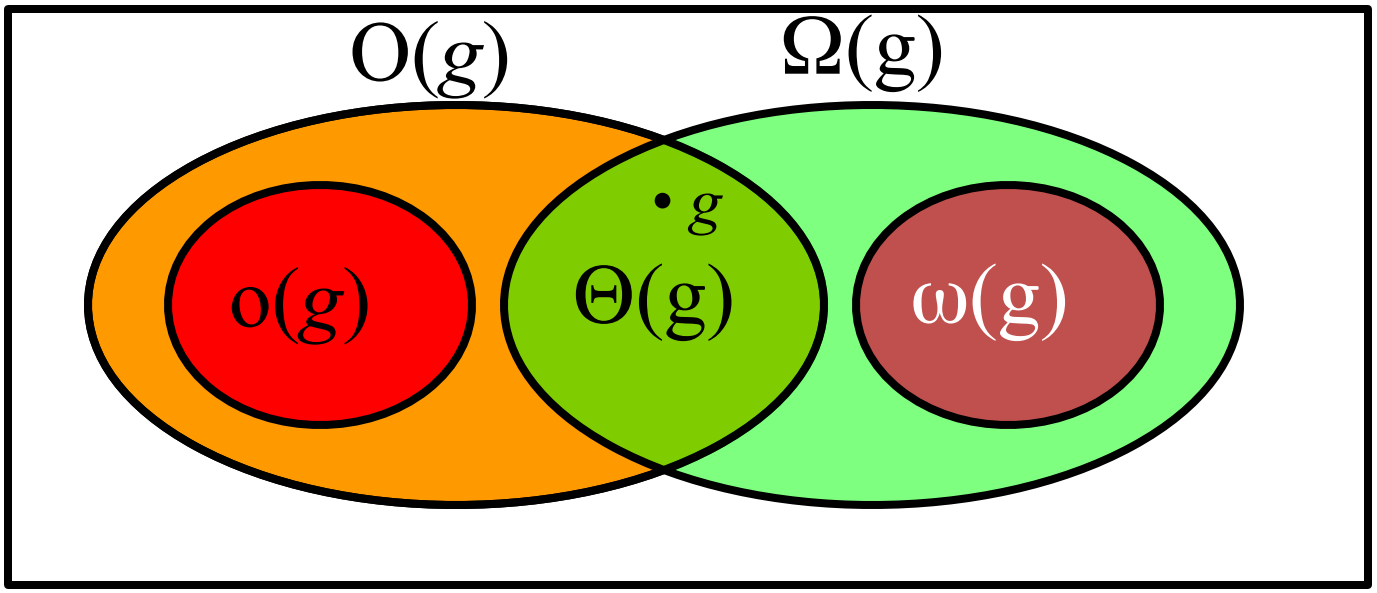
- Big O, as a relation, is transitive:
  $f \in O(g) \wedge g \in O(h) \rightarrow f \in O(h)$

- Sums of functions:
  If f∈O(g) and $h \in O(g)$, then f+$h \in O(g)$.

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

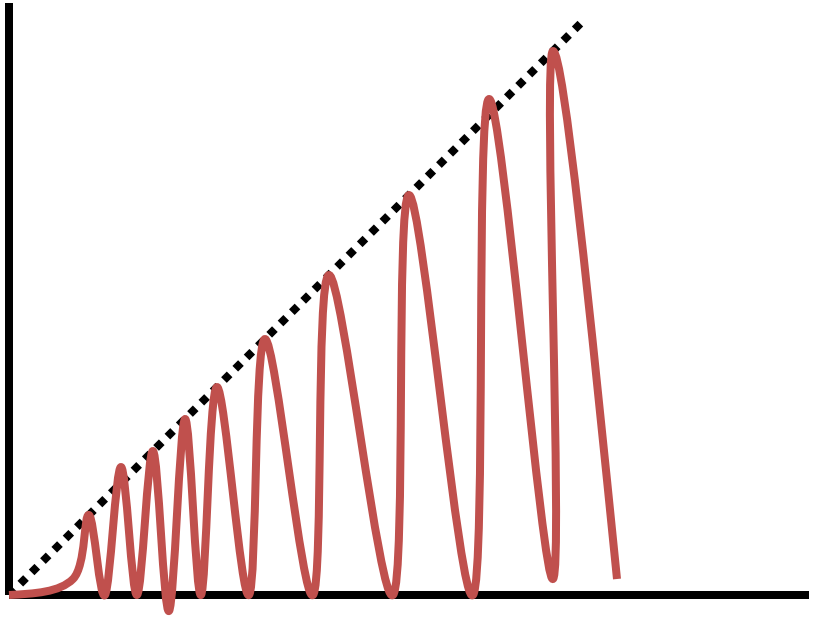
- $f1 \in O(g1) \wedge f2 \in O(g2) \rightarrow$ $f1\ f2 \in O(g1g2)$
  $f1+f2 \in O(g1+g2)$

Subset relations between order-of-growth sets



Why o(f)⊂O(x)–Θ(x)

A function that is O(x), but neither o(x) nor Θ(x):



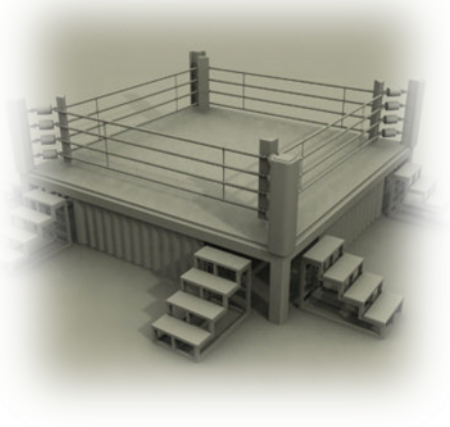
Other Order-of-Growth Relations

- ✓ $\Omega(g) = \{f \mid g \in O(f)\}$
  "The functions that are *at least order g*."

- ✓ $o(g) = \{f \mid \forall c>0\ \exists k\ \forall x>k : 0 \leq f(n) < cg(n)\}$
  "The functions that are *strictly lower order than g*."
  $o(g) \subset O(g) - \Theta(g)$.

- ✓ $\omega(g) = \{f \mid \forall c>0\ \exists k\ \forall x>k : 0 \leq cg(x) < f(x)\}$
  "The functions that are *strictly higher order than g*."
  $\omega(g) \subset \Omega(g) - \Theta(g)$.

Champion Problem



1. int *i, j*; — $O(1)$ time
2. *j* = 1; — $O(1)$ time
3. for (*i* = 2; *i* <= *n*; *i*++) — $O(n)$ iterations
4. if (*A*[*i*] > *A*[*j*]) — In fact n -1 iterations; O(1) time for operations like i = 2, i <= n and i++); $O(1)$ time
5. *j* = *i*; — $O(1)$ time
6. return *j*; — $O(1)$ time

Adding everything together
yields
an upper bound on the worst-case time complexity.

Complexity is O(1) + O(1) + O(n) {O(1) + O(1) + O(1)} + O(1) = O(n)

Practice exercises – Find sum of 'n' integers

Devise an algorithm that finds the sum of all the integers in a list.

procedure *sum*(a_1, a_2, …, a_n: integers)

    s := 0 {sum of elems so far}

    for i := 1 **to** n {go thru all elements}

        s := s + a_i {add current item}

        {at this point s is the sum of all items}

    return s

Practice exercises

What is the complexity of the following algorithm?

```
Algorithm Awesome(A: list [1,2,..n])
                    var int i, j, k, sum
                    for i from 1 to 100
                              for j from 1 to 1000
                                        for k from 1 to 50
                                                  sum = sum + i + j + k
```

Practice exercises

Work out the computational complexity of the following piece of code:

```
for( int i = n;  i > 0;  i /= 2 ) {
  for( int j = 1;  j < n;  j *= 2 ) {
    for( int k = 0;  k < n;  k += 2 ) {
          ... // constant number of operations
    }
  }
}
```

In the outer `for`-loop, the variable `i` keeps halving so it goes round $\log_2 n$ times. For each `i`, next loop goes round also $\log_2 n$ times, because of doubling the variable `j`. The innermost loop by `k` goes round $\frac{n}{2}$ times. Loops are nested, so the bounds may be multiplied to give that the algorithm is $O\left(n(\log n)^2\right)$.

Practice exercises

Work out the computational complexity of the following piece of code assuming that $n = 2^m$:

```
for( int i = n; i > 0; i-- ) {
        for( int j = 1; j < n; j *= 2 ) {
                for( int k = 0; k < j; k++ ) {
                        for(int m = 0; m < 10000; m++)
                                sum = sum + m;
                                        }
                                }
                        }
```

Practice exercises

```
for( int i = n; i > 0; i-- ) {                    -------L1
                for( int j = 1; j < n; j *= 2 ) {    --------L2
                        for( int k = 0; k < j; k++ ) {    --------L3
                           for(int m = 0; m < 10000; m++)     --------L4
                                   sum = sum + m;
                                          }
                                   }
                           }
```

The outer most for-loop, i.e., L1, runs for O(n) times.
For each j in L2 loop, the L3 loop runs for j times, so that the two inner loops, L2 and L3
together go round $1 + 2 + 4 + \ldots + 2^{m-1} = 2^m - 1 = n - 1 = O(n)$ times.
The inner-most loop, i.e., L4, is of O(1) complexity.
Loops are nested, so the bounds may be multiplied to give that the algorithm is $O(n^2)$.

Matrix Multiplication

Input: two $n \times n$ matrices A and B.

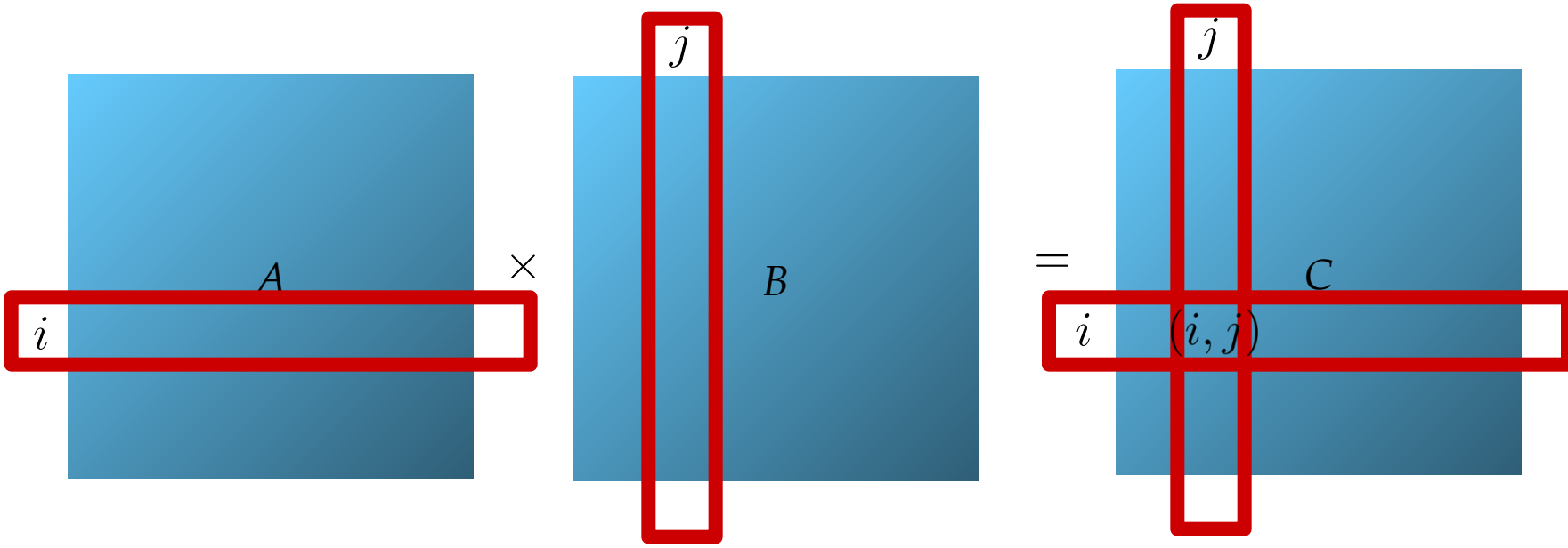
Output: the product matrix $C = A \times B$

Naïve algorithm

---

for $i = 1$ to n
  for $j = 1$ to n
    Let $C(i, j)$ be the inner product of
      the i-th row of A and the j-th column of B.

Since each inner product of two n-element vectors takes $\Theta(n)$ time, the time complexity of the naive algorithm is $\Theta(n^3)$.


A
i
×
j
B
=
j
C
i
(i, j)

Is $o(n^3)$ time possible?

Search Algorithm #1: Linear Search

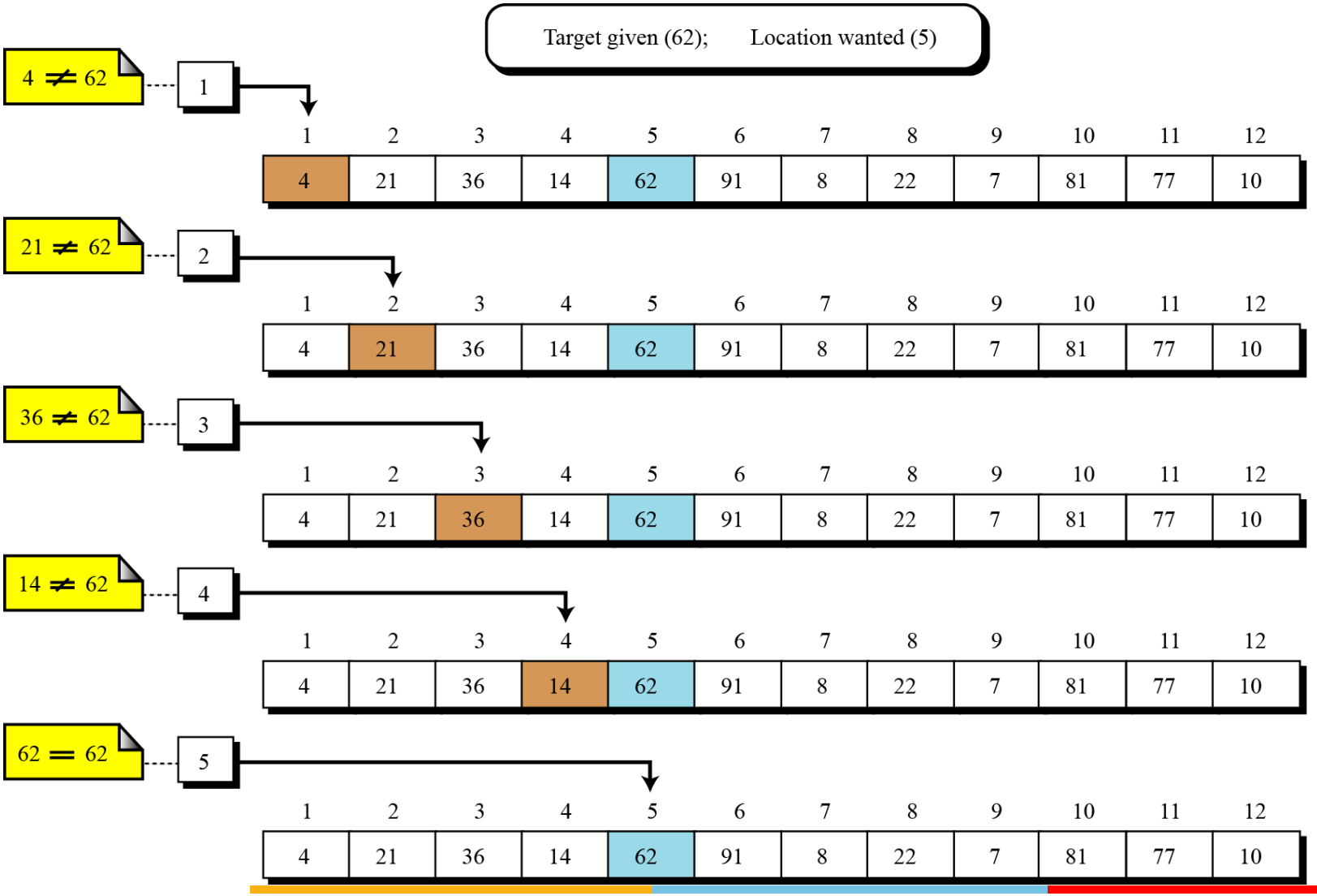
procedure *linear search*
(*x*: integer, a_1, a_2, …, a_n: distinct integers)
i := 1 {start at beginning of list}

while ($i \leq n$ and $x \neq a_i$) {not done, not found}
i := *i* + 1 {go to the next position}

if $i \leq n$ **then** *location* := *i* {it was found}
else *location* := 0 {it wasn't found}

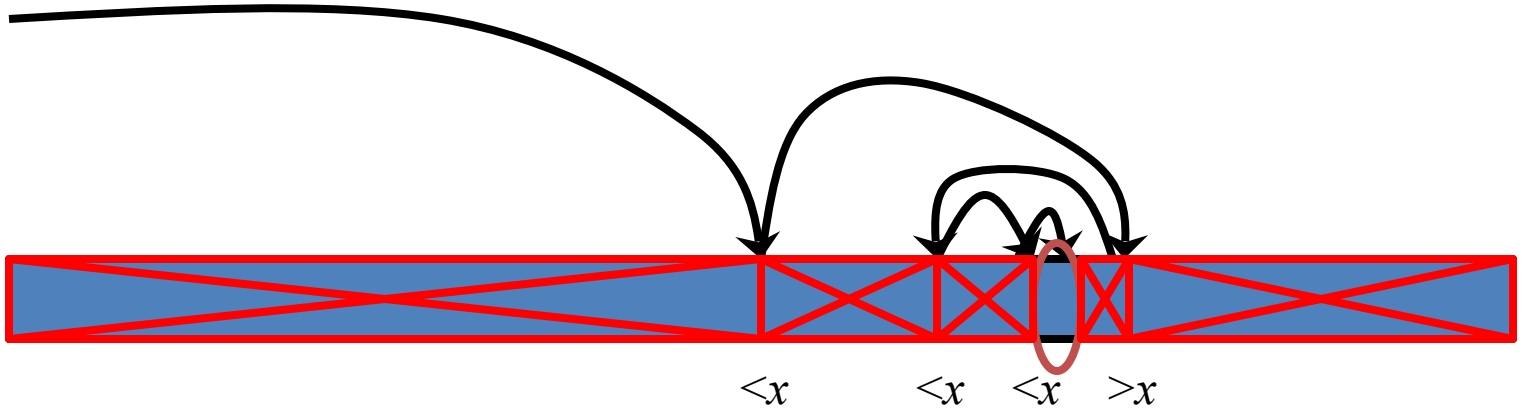
return *location* {index or 0 if not found}

Search Algorithm #1: Linear Search



Search Algorithm #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 Algorithm #2: Binary Search

procedure *binary search*
(*x*:integer, a_1, a_2, …, 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*

Thank You!!