# *Algorithm Analysis*

Krisana Chinnasarn, Ph.D.
Assistant Professor,
Reading: Baase Ch 1



## *Outline*

1. Introduction
2. Analyzing Algorithms and Problems
3. Asymptotic Growth Rate



## *1. Introduction*


- **Definition**: (*Algorithm*)
  An algorithm is a finite sequence of instructions, if followed, to accomplish a particular task
- Five requirements for an algorithm
  *Input*: Zero or more quantities (data) are supplied externally
  *Output*: At least one quantity (data) is produced
  *Definiteness*: Each instruction must be clear and unambiguous
  *Finiteness*: The algorithm is required to terminate after a finite number of steps
  *Effectiveness*: Every instruction must be sufficiently basic so that anyone can follow

## Preliminaries

- Algorithm description languages
  - pidgin ALGOL
  - FORGOL
  - .....
  - Modula-2 + Pascal
  - Java
- Unambiguous
- Independent of computers
- Easy to translate into a computer language

4

## What this course is about ?

- Data structures: conceptual and concrete ways to organize data for efficient storage and efficient manipulation
- Employment of this data structures in the design of efficient algorithms

5

## Why do we need them ?

- Computers take on more and more complex tasks
  - Imagine: index of 8 billion pages ! (Google)
- Software implementation and maintenance is difficult.
- Clean conceptual framework allows for more efficient and more correct code

6

## Why do we need them

- Requirements for a good software:
  - Clean Design
  - Easy maintenance
  - Reliable (no core dumps)
  - Easy to use
  - Fast algorithms

→**Efficient data structures**
→**Efficient algorithms**

7

## Example

- A collection of 3,000 texts with avg. of 20 lines each, with avg. 10 words / line
  - → 600,000 words
- Find all occurrences of the word "*happy*"
- Suppose it takes 1 sec. to check a word for correct matching
- What to do?

8

## Example (cont'd)

- What to do?

Sol. 1 Sequential matching: 1 sec. x 600,000 words = 166 hours

Sol. 2 Binary searching:

- order the words
- search only half at a time

Ex. Search 25 in 5 8 12 15 15 17 23 25 27

25 ? 15 15 17 23 25 27

25 ? 23 23 25 27

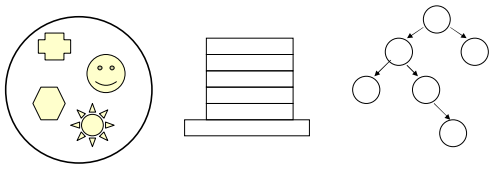
25 ? 25

How many steps?

9

## Some example data structures

- $\log_2 600000 = 19$ sec. vs .166 hours!



Set Stack Tree

Data structure = representation and operations associated with a data type

10

## What will you learn?

- What are some of the common data structures
- What are some ways to implement them
- How to analyze their efficiency
- How to use them to solve practical problems

11

## What you need

- Programming experience with C / C++
  - Some Java experience may help as well (but not required)
- Textbook
  - Computer Algorithms: Introduction to Design and Analysis, 2nd edition
  - Sara Baase

12

## Mathematics Reviews: Exponents

$$X^A X^B = X^{A+B}$$

$$\frac{X^A}{X^B} = X^{A-B}$$

$$(X^A)^B = X^{AB}$$

$$X^N + X^N = 2X^N \neq X^{2N}$$

$$2^N + 2^N = 2^{N+1}$$

13

## Mathematics Reviews: Logarithms

All logarithms are to the base 2.

$X^A = B \quad iff \quad \log_X B = A$

$$\log_A B = \frac{\log_C B}{\log_C A}; \quad A, B, C > 0, A \neq 1$$

$$\log AB = \log A + \log B; \quad A, B > 0$$

$$\log A / B = \log A - \log B$$

$$\log(A^B) = B \log A$$

$$\log X < X \quad for\ all\ X > 0$$

$$\log 1 = 0; \quad \log 2 = 1, \quad \log 1024 = 10, \log 1048576 = 20$$

14

## Mathematics Reviews: Series

$$\sum_{i=0}^{N} 2^i = 2^{N+1}$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=0}^{N} A^i \leq \frac{1}{1 - A}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

$$\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}; \quad k \neq 1$$

15

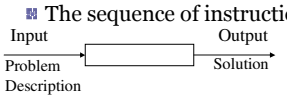
# 2. Analyzing Algorithms and Problems

- Criteria for choosing an algorithm
  - Correctness
  - Amount of work done (time complexities)
  - Amount of space used (space complexities)
  - Simplicity (clarity)
  - Optimality

16

# Correctness

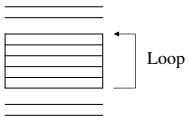
- A precise statement:
  - What inputs to work on
  - What results to produce
- Solution method
- Implementation
  - The sequence of instructions for carrying it out




17

# How to Prove Correctness

- Loop invariants
  - Program structure




- Definition: (Loop invariant)
  - Loop invariants are conditions and relationships that are satisfied by the variables and data structures at the end of each iteration of the loop.
- Establish *loop invariants* by mathematical induction !!

18

# Example

- Given an array $L$ containing n elements ($n \geq 0$) and a number $x$, find the index of the first occurrence of $x$ in $L$, if $x$ is in $L$, and report 0 otherwise.
- What is input? $L$ and x
- What is output ? If $x \in L$, the index of the first occurrence of x
  Otherwise, 0
- Algorithm
  1. *index := 1*;
  2. while *index* $\leq n$ and $L(index) \neq x$ do
  3. *index := index + 1*
  4. end {while}
  5. if *index* $> n$, then *index := 0* end;

Sequential search !!!



- Loop invariants
  For $1 \leq k \leq n+1$, whenever control reaches in line 2 for the $k^{th}$ time the following conditions are satisfied:
  *index* $= k$ and $L(i) \neq x$ for $1 \leq i < k$

1. *index := 1*;
2. while *index* $\leq n$ and $L(index) \neq x$ do
3. *index := index + 1*
4. end {while}
5. if *index* $> n$, then *index := 0* end;

- [Proof] By induction
  (*index* = 1) Obvious
  (*index* = $k$) Suppose that *index* $= k$ and $L(i) \neq x$ for $1 \leq i < k$
  when line 2 is executed $k$ times for some $1 \leq k < n+1$
  (*index* = $k+1$) From line 2, $L(k) \neq x$. By line3, *index* $= k+1$. By the induction hypothesis, $L(i) \neq x$, $1 \leq i < k$.
  $\therefore L(i) \neq x$, $1 \leq i < k+1$
- Now, the loop body cannot be performed more than $n+1$ times.
- At line 5, there are two cases
  (case 1) *index* $> n \rightarrow$ *index* $= 0$
  (case 2) *index* $\leq n \rightarrow$ *index* $= i$ such that $L(i) = x$



# Observation

- Proving the correctness of an algorithm is tedious even for a trivial one such as the sequential search.
- *What would a proof of correctness of a full-sized program with complex data and control structure be like ?*
- Any way to crack it?
  *"Divide and Conquer"*
  (1) Break a large program down into smaller modules.
  (2) Show that, if all of the smaller modules do their jobs properly, then the whole program is correct.
  (3) Prove that each of modules is correct.
  *"Modular Programming"*

# Amount of Work Done

- How to measure the amount of work done by an algorithm ?
  - Well, .....
  - However, a measure of work should be *both precise and general enough to develop a rich theory that is useful for many algorithms and applications.*
- Time Complexity: $T(n)$
- Space Complexity: $S(n)$

$T(n) = f_M(n)$ , where

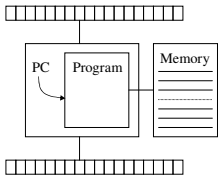
$M$ : the machine for executing an algorithm.

$n$ : the size of the problem to be solved.



# Machines

- Type of machines
  - Turing Machine
  - Von Neumann
  - .....




+,-,*,/,**,<,>,=, ≤, ≥,&,|
I/O instructions
Some functions (log, mod, …)

- Von Neumann (Real RAM)
  - Unit cost/operation
  - A real number/memory cell
  - A program cannot modify itself



- Size of the Problem
  - $T(n) = f_M(n)$
    - $M$ → ?
    - $n$ → real RAM
  - n: input size
  - The input size for an instance of the problem is said to be the number of symbols in the description of the problem instance under a *reasonable* encoding scheme.
- Binary encoding scheme
  - $k$.....$\lfloor \log_2 k \rfloor$ bits
  - Under this assumption, the magnitude of a number is also an important factor for determining an input size.
- String encoding scheme
  - # of characters

.....

**Observation 1**

1 word = 32 bits

1 byte = 8 bits

| | | $T(n)$ | | |
|---|---|---|---|---|
| input size | $O(n)$ | $O(n^2)$ | $O(n^p)$ | |
| bits | $n$ | $c_1 n$ | $c_2 n^2$ | $c_3 n^p$ |
| bytes | $n/8$ | $(c_1/8)n$ | $(c_2/64)n^2$ | $(c_3/8^p)n^p$ |
| words | $n/32$ | $(c_1/32)n$ | $(c_2/32^2)n^2$ | $(c_3/32^p)n^p$ |

As long as $P$ is a constant, encoding schemes do not affect time complexity.

- **Observation 2**
  - Find $x$ in a list of names — # of names
  - Multiply two matrices — dimensions of matrices
  - Sort a list of numbers — # of entries in the list
  - Traverse a binary tree — # of nodes
  - ..... — .....

There usually is a reasonable measure for the size of a problem.

25

- It is assumed that there exists a reasonable measure for the size of a problem
- In the real RAM, *the number of real numbers* is reasonable for measuring the size of a problem !!!

26

$T(n) = f_M(n)$ , where

$M$ : real RAM

$n$ : the size of the input under a reasonable encoding scheme, i.e., the number of real numbers

By the unit cost assumption,

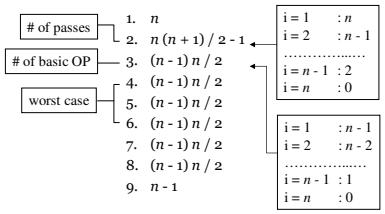
$T(n)$ : the number of instructions executed by the real RAM, which is expressed in terms of the input size n.

27

# Example

Given a set of n real number, sort them in the ascending order.

```
1.  for i:=1 to n-1 do
2.    for j:=n downto i+1 do
3.      if A[j-1] > A[j] then
          { swap A[j-1] and A[j]}
4.        temp := A[j-1];
5.        A[j-1] := A[j];
6.        A[j] := temp;
7.      end {if}
8.    end{for}
9.  end{for}
```




$O(n^2)$

$$n(n+1)/2 - 1 + 3(n-1)n + 2n - 1 = 7/2\, n^2 - n/2 - 2$$

28

# Observation

- T(n): total number of operations $\propto$ # of passes of a loop # of basic operations
- # of passes of a loop (which loop ?)
  Depending on the control structure of an algorithm.
  → Well, ...
- # of basic operations
  A particular operation fundamental to the problem under study.
- *Problem* — *Basic operations*
  Find x in a list of names — Comparison
  Multiply two matrices — Multiplication
  Sort a list of numbers — Comparison
  Traversing a binary tree — Handling pointer

Hammer
- Drive in a nail with a hammer
- Strike a nail with a hammer

29

# Example

- Sort a set $S$ of n integers
  $S = \{1, 2, 3, ..., n\}$
- Basic operation: comparison
- Bubble Sort:

$$T(n) = c \cdot (\#\text{ of comparison})$$
$$= c \cdot n^2 = O(n^2)$$

- How about bucket sort ?

```
for i :=1 to n do
      N(A(i)) := A(i)
end
```

*Well, no comparison !!!*

30

- $A$: Given $n$ numbers, sort them
- $B$: Given a set $S$ of $n$ integers, sort them, where $S$={1, 2, 3, ..., $n$}

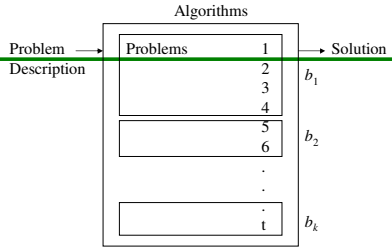
- Is $A = B$ ? $B \subset A$, i.e., $B$ is a special instance of $A$.

  No !!

  Why ?

  $\therefore A \neq B$






$B = \{b_1, b_2, ..., b_k \mid b_i,\ 1 \le i \le k$ is a basic operation$\}$

- For each $b_i$, $1 \le i \le k$, there is a class of problems for which $b_i$ is appropriate.
- However, a basic operation should be chosen so that most of problems can be in the same class.



# *Example: Sequential Search*

- Basic operation: comparison
- $W(n) = n$
- $A(n)$ ?
- Assumption 1

  (1) $x \in L$

  (2) $x$ is equally likely to be in any particular position of $L$ .

$I_i = \{I \mid x$ is in the $i^{th}$ position of $L$ and $I \in D_n\}$

Clearly, $D_n = \cup_i I_i$, $I_i \cap I_j = \varnothing$ for all $i \neq j$

$\therefore P(I_i)$ is well - defined.

$$P(I_i) = \sum_{I \in I_i} P(I) = \frac{1}{n} \quad \forall i$$

$$\therefore A(n) = \sum_{i=1}^{n} P(I_i) \cdot t(I_i) = \sum_{i=1}^{n} \frac{1}{n} \cdot i$$

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

$$= \frac{1+n}{2}$$

$$\boxed{A(n) = \frac{1+n}{2}}$$

# Amount of Space Used

Space complexity: $S(n)$

- Space complexity can be analyzed in the almost similar way to analyzing time complexity.
- $S(n)$ = space for input & program + extra space
- If the amount of extra space is constant regardless of the input size, then the algorithm is said to work *in place*.

34

# 3. Time Complexity

- Often more important than space complexity
  - space available (for computer programs!) tends to be larger and larger
  - time is still a problem for all of us
- 3-4GHz processors on the market
  - still …
  - researchers estimate that the computation of various transformations for 1 single DNA chain for one single protein on 1 TerraHZ computer would take about 1 year to run to completion
- Algorithms running time is an important issue

35

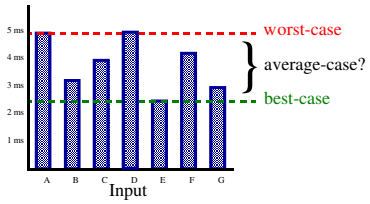
# Running Time

- Problem: prefix averages
  - Given an array X
  - Compute the array A such that A[i] is the average of elements X[0] … X[i], for i=0..n-1
- Sol 1
  - At each step i, compute the element X[i] by traversing the array A and determining the *sum* of its elements, respectively the average
- Sol 2
  - At each step i update a *sum* of the elements in the array A
  - Compute the element X[i] as *sum/I*

**Big question: Which solution to choose?**

36

# Running time




Suppose the program includes an *if-then* statement that may execute or not: → variable running time
Typically algorithms are measured by their ***worst case***

37

# Experimental Approach

- Write a program that implements the algorithm
- Run the program with data sets of varying size.
- Determine the actual running time using a system call to measure time (e.g. *system (date)* );
- Problems?

38

# Experimental Approach

- It is necessary to implement and test the algorithm in order to determine its running time.
- Experiments can be done only on a limited set of inputs, and may not be indicative of the running time for other inputs.
- The same hardware and software should be used in order to compare two algorithms. – condition very hard to achieve!

39

## Use a Theoretical Approach

- Based on high-level description of the algorithms, rather than language dependent implementations
- Makes possible an evaluation of the algorithms that is independent of the hardware and software environments

  → **Generality**

40

## 4. Algorithm Description

- How to describe algorithms independent of a programming language
- Pseudo-Code = a description of an algorithm that is
  - more structured than usual prose but
  - less formal than a programming language
- (Or diagrams)
- Example: find the maximum element of an array.

**Algorithm** arrayMax(A, *n*):
*Input:* An array A storing n integers.
*Output:* The maximum element in A.
currentMax ← A[0]

41

## Pseudo Code

- Expressions: use standard mathematical symbols
  - use ← for assignment ( ? in C/C++)
  - use = for the equality relationship (? in C/C++)
- Method Declarations: -**Algorithm** name(***param1, param2***)
- Programming Constructs:
  - decision structures: **if … then … [else ..]**
  - while-loops **while … do**
  - repeat-loops: **repeat … until …**
  - for-loop: **for … do**
  - array indexing: **A[i]**
- Methods
  - calls: object method(args)
  - returns: **return** value
- Use comments

42

## Low Level Algorithm Analysis

- Based on primitive operations (low-level computations independent from the programming language)
- E.g.:
  - Make an addition = 1 operation
  - Calling a method or returning from a method = 1 operation
  - Index in an array = 1 operation
  - Comparison = 1 operation etc.
- Method: Inspect the pseudo-code and count the number of primitive operations executed by the algorithm

43

## Example

**Algorithm** arrayMax(A, n):
*Input:* An array A storing n integers.
*Output*: The maximum element in A.

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

How many operations ?

44

## 5. Asymptotic Growth Rate

How good is our measure of work done to compare algorithms ?

How precisely can we compare two algorithms using our measure of work ?

Measure of work
- # of passes of a loop
- # of basic operations

Time complexity
- $c \cdot$ (measure of work)

45

For solving a problem $P$, suppose that two algorithms $A_1$ and $A_2$ need $10^6n$ and $5n$ basic operations, respectively, i.e.

| | basic operations |
|---|---|
| $A_1$ | $10^6n$ |
| $A_2$ | $5n$ |

Which one is better ?

What are their time complexities ?

| | | |
|---|---|---|
| $A_1$ | $10^6n$ | $O(n)$ |
| $A_2$ | $5n$ | $O(n)$ |

Does this mean that $A_1(A_2)$ is as good as $A_2(A_1)$ ?

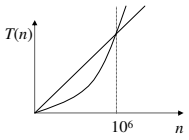
Well, .....

$\therefore$ Both # of basic operations (loops) and time complexity are imprecise !!!



Now, suppose that algorithms $A_1$ and $A_2$ need the following amount of time:

| | | time complexity |
|---|---|---|
| $A_1$ | $10^6n$ | $O(n)$ |
| $A_2$ | $n^2$ | $O(n^2)$ |




Which one is better ?

$A_1$ is better if $n > 10^6$

$A_2$ is better if $n < 10^6$

Then, why time complexity ?

Suppose that $n \to \infty$

Then, $n^2$ grows much faster than $10^6n$. i.e.,

$$\lim_{n\to\infty} \frac{n^2}{10^6 n} \to \infty$$

Under the assumption that $n \to \infty$

$A_1$ is better than $A_2$

$\therefore$ Asymptotic growth rate !!!

$\therefore$ Time complexity (measure of work) compares and classifies algorithms by the asymptotic growth rate.



$\mathbf{N}$ = {0, 1, 2, ...}

$\mathbf{N}^+$= {1, 2, 3, ...}

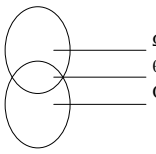
$\mathbf{R}$ = the set of real numbers

$\mathbf{R}^+$= the set of positive real numbers

$\mathbf{R}^*$ = $\mathbf{R}^+ \cup$ {0}

$f$: $\mathbf{N} \to \mathbf{R}^*$ and $g$: $\mathbf{N} \to \mathbf{R}^*$

$g$ is:



| | |
|---|---|
| $\Omega(f)$: | $g$ grows at least as fast as $f$. |
| $\theta(f)$: | $g$ grows as the same rate as $f$. |
| $O(f)$: | $g$ grows no faster than $f$. |

**Definition**: Let $f$: $\mathbf{N} \to \mathbf{R}^*$. $O(f)$ is the set of functions, $g$: $\mathbf{N} \to \mathbf{R}^*$ such that for some $c \in \mathbf{R}^+$ and some $n_o \in \mathbf{N}$, $g(n) \le c \cdot f(n)$ for all $n \ge n_o$.

$O(f)$ is usually called
"big oh of $f$", "oh of $f$", or "order of $f$".

Note: In other books,
$g(n) = O(f(n))$ if and only if there exist two positive constants $c$ and $n_o$ such that $|g(n)| \le c \cdot |f(n)|$ for all $n \ge n_o$

Under the assumption that $f$: $\mathbf{N} \to \mathbf{R}^*$ and g: $\mathbf{N} \to \mathbf{R}^*$, two definitions have a minor difference.

How to check (What is it ?)

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c, c \in \mathbf{R}^* \Rightarrow g \in O(f)$$

note : By L' Hopital' s rule

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

$n^2$, $10^5n^2 - n$, $n^2 + 10^{10}$, $10^3n^2 + n - 1 \in O(n^2)$

Is $10^{10}n \in O(n^2)$ ?



**Definition**: Let $f$: $\mathbf{N} \to \mathbf{R}^*$. $\Omega(f)$ is the set of functions, $g$: $\mathbf{N} \to \mathbf{R}^*$ such that for some $c \in \mathbf{R}^+$ and some $n_0 \in \mathbf{N}$, $g(n) \ge c \cdot f(n)$ for all $n \ge n_0$.

$\Omega(f)$ is usually called
"big omega of $f$" or "omega of $f$".

Note: In other books,
$g(n) = \Omega(f(n))$ if and only if there exist two positive constants $c$ and $n_o$ such that $|g(n)| \ge c \cdot |f(n)|$ for all $n \ge n_o$

How to check

$$\left[ \begin{array}{l} \lim_{n\to\infty} \frac{g(n)}{f(n)} \to c > 0 \quad \text{or} \\ \lim_{n\to\infty} \frac{g(n)}{f(n)} \to \infty \end{array} \right] \Rightarrow g \in \Omega(f)$$

$10^5n$, $n^2$, $n^2 + 10n + 10^6 \in \Omega(n)$.

Are they also $\Omega(\log n)$ ?



**Definition**: Let $f$: $\mathbf{N} \to \mathbf{R}^*$. $\theta(f) = O(f) \cap \Omega(f)$.

$\theta(f)$ is usually called
"theta of $f$" or "order of $f$".

Note: (Alternative definition of $\theta(f)$)
$g(n) = \theta(f(n))$ if and only if there exist two positive constants $c_1$ and $c_2$ and $n_o$ such that $c_1 \cdot |f(n)| \le |g(n)| \le c_2 \cdot |f(n)|$ for all $n \ge n_o$

How to check

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} = c, \ c \in \mathbf{R}^+ \Rightarrow g \in \theta(f)$$

$n^2$, $10^{10}n^2$, $n^2 - 5n + 10^7 \in \theta(n^2)$

**Definition**: Let $f$: $\mathbf{N} \to \mathbf{R}^*$. $o(f) = O(f) - \theta(f)$.
$o(f)$ is usually called "little oh of $f$".

How to check :

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \to 0 \Rightarrow g(n) = o(f(n))$$

$n - 5$, $n$, $n^2$, $10^{10}n^2 + 10^5 n + 10^9$, $n^2 - 9 \in o(n^3)$



**Definition**: Let $f$: $\mathbf{N} \to \mathbf{R}^*$. $\omega(f) = \Omega(f) - \theta(f)$.
$\omega(f)$ is usually called "little omega of f"

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

How to check :

$$n^2, 10^{10}n^2 + 10^5 n + 10^9, n^3 - 9 \in .(n)$$

Note:

$$g(n) = o(f(n)) \Leftrightarrow f(n) = \quad .(g(n))$$



## *How important is time complexity ?*

| Algorithm | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Time function (microsec.) | $33n$ | $46n \lg n$ | $13n^2$ | $3.4n^3$ | $2^n$ |

| Input size ($n$) | *Solution time* | | | | |
|---|---|---|---|---|---|
| 10 | .00033 sec. | .0015 sec. | .0013 sec. | .0034 sec. | .001 sec. |
| 100 | .0033 sec. | .03 sec. | .13 sec. | 3.4 sec. | $4 \cdot 10^{16}$ yr. |
| 1,000 | .033 sec. | .45 sec. | 13 sec. | .94 hr. | |
| 10,000 | .33 sec. | 6.1 sec. | 22 min. | 39 days | |
| 100,000 | 3.3 sec. | 1.3 min. | 1.5 days | 108 yr. | |

| Time allowed | *Maximum solvable input size (approx.)* | | | | |
|---|---|---|---|---|---|
| 1 second | 30,000 | 2,000 | 280 | 67 | 20 |
| 1 minute | 1,800,000 | 82,000 | 2,200 | 260 | 26 |

| $n$ | Cray-1 Fortran [a] $3n^3$ nanoseconds | TRS-80 Basic [b] 19,500,000$n$ nanoseconds |
|---|---|---|
| 10 | 3 microseconds | .2 seconds |
| 100 | 3 milliseconds | 2.0 seconds |
| 1,000 | 3 seconds | 20.0 seconds |
| 2,500 | 50 seconds | 50.0 seconds |
| 10,000 | 49 minutes | 3.2 minutes |
| 1,000,000 | 95 years | 5.4 hours |

[a] Cray-1 is a trademark of Cray Research, Inc.
[b] TRS-80 is a trademark of Tandy Corporation.

55

**Properties** of O, Ω, θ

Let $f, g, h$: $\mathbf{N} \rightarrow \mathbf{R}^*$. Then,

- P1: (Transitivity)
  $f \in O(g)$ and $g \in O(h) \Rightarrow f \in O(h)$
  How about Ω, θ, o, ω ?
- P2: $f \in O(g) \Leftrightarrow g \in \Omega(f)$
  $f \in o(g) \Leftrightarrow g \in \omega(f)$
  Duality
- P3: $f \in \theta(g) \Rightarrow g \in \theta(f)$
- P4: θ is an equivalence relation
- P5: $O(f+g) = O(\max\{f, g\})$
  How about Ω, θ, o, ω ?

  [Proof] Exercise. (Homework)

**Theorem**: $\log n$ is in $o(n^\alpha)$ for any $\alpha > 0$. $n^k$ is in $o(2^n)$ for any $k > 0$.

[Proof] Exercise. (Homework)

56

# *Back to the original question*

- Which solution would you choose?
  - $O(n^2)$ vs. $O(n)$
- Some math …
  - properties of logarithms:

$\log_b(xy) = \log_b x + \log_b y$
$\log_b (x/y) = \log_b x - \log_b y$
$\log_b xa = a\log_b x$
$\log_b a = \log_x a / \log_x b$

  - properties of exponentials:

$a^{(b+c)} = a^b a^c$
$a^{bc} = (a^b)^c$
$a^b / a^c = a^{(b-c)}$
$b = a^{\log_a b}$
$b^c = a^{c*\log_a b}$

57

# Important Series

$$S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = N(1+N)/2$$

- Sum of squares: $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N
- Sum of exponents: $\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|}$ for large N and k ≠ -1
- Geometric series: $\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$
  - Special case when A = 2
    - $2^0 + 2^1 + 2^2 + \ldots + 2^N = 2^{N+1} - 1$

58