Computer Science 1000: Part #2

Algorithms

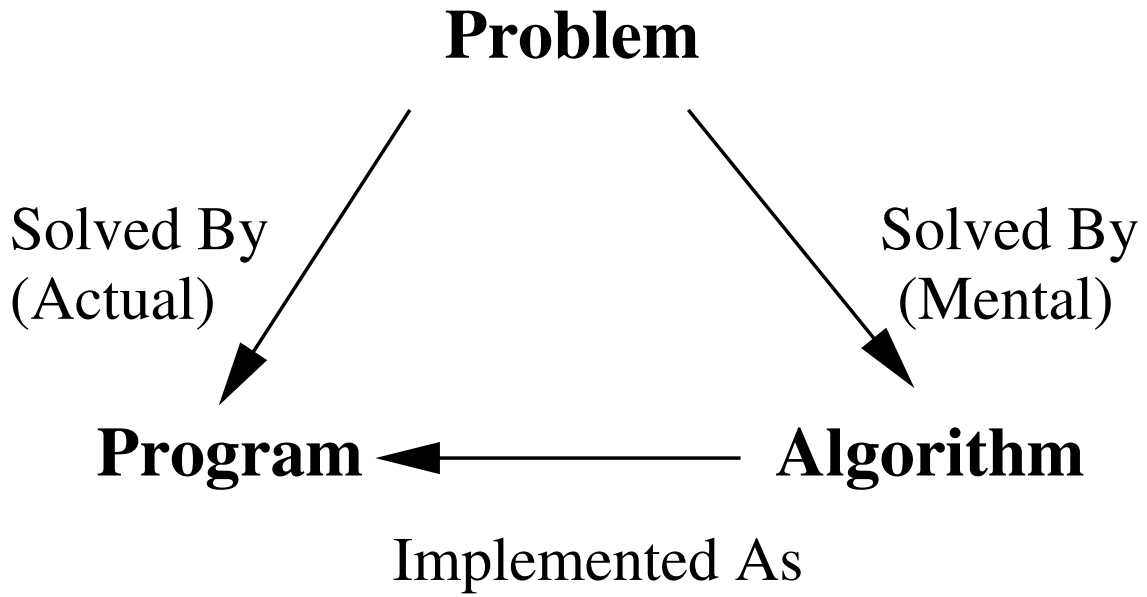
- Problems, Algorithms, and Programs
- Representing Algorithms as Pseudocode
- Example Algorithms
- Assessing Algorithm Efficiency

. . . To Recap . . .

- The fundamental task of Computer Science is the design and development of algorithms for solving important problems.
- An **algorithm** is a well-ordered sequence of unambiguous and effectively computable operations that produces a result and halts in a finite amount of time.

IF WE CAN SPECIFY AN ALGORITHM
TO SOLVE A PROBLEM,
WE CAN AUTOMATE ITS SOLUTION!!!

Problems, Algorithms, and Programs



Problem: A set of inputs and their associated outputs.

Algorithm: A sequence of instructions that solves a problem, *i.e.*, computes the output for a given input.

Program: A sequence of instructions *in some computer language* that solves a problem.

Example: Finding the Area of a Circle

Problem:

Input: A radius r.
Output: The area of a circle with radius r.

Algorithm:

Get the value of radius r
Set the value of area to $\pi \times r^2$
print area

Program:

```
import sys
r = sys.argv[1]
area = 3.14159 * r * r
print area
```

Example: Summing a List

Problem:

Input: A list L of n numbers.
Output: The sum of the numbers in L.

Algorithm:

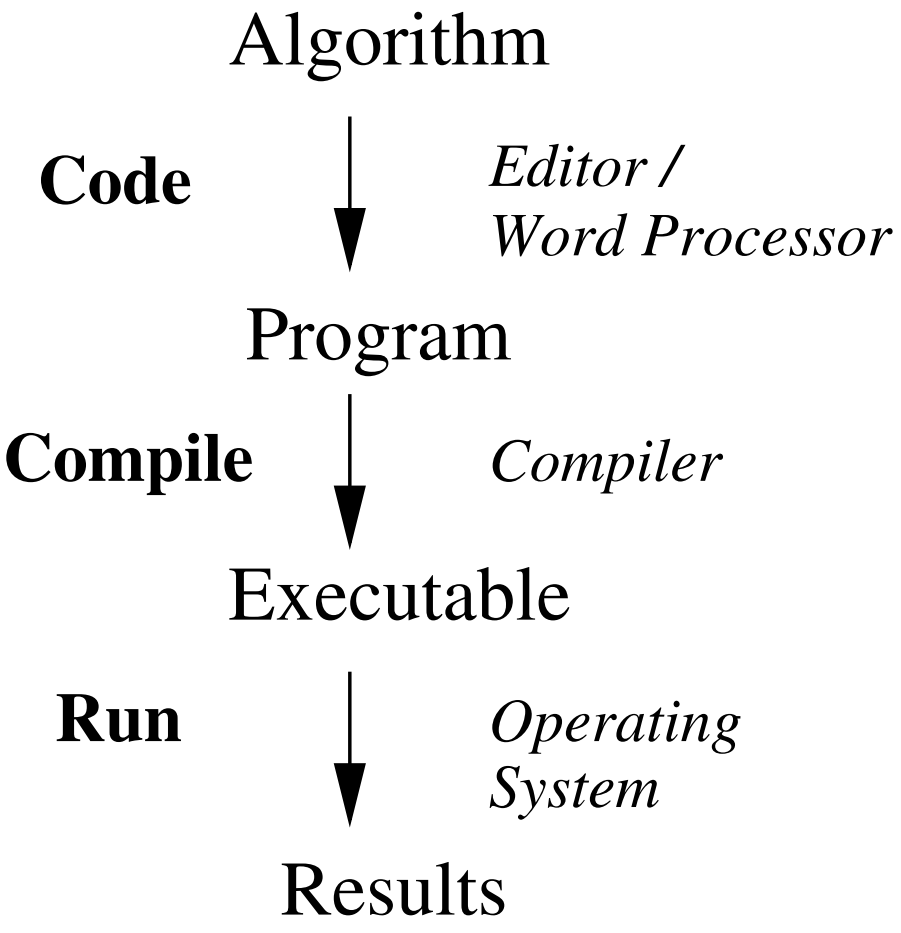
Get the values for list L and n
Set the values of IND to 1 and SUM to 0
While (IND $\leq$ n) do
    Set the value of SUM to SUM + L_{IND}
    Set the value of IND to IND + 1
print the value of SUM

Example: Summing a List (Cont'd)

Program:

```
import sys
sum = 0
n = int(sys.argv[1])
for i in range(1, n + 1):
    curL = int(sys.argv[i + 1])
    sum = sum + curL
print sum
```

Solving Problems with Algorithms: The Big Picture



Representing Algorithms as Pseudocode

- Natural language not precise enough to express algorithms as it often relies on context and background knowledge; programming languages too precise for abstract thinking needed during algorithm development.
- **Pseudocode** = programming-language-like statement of algorithm that does not run on a computer.
- No standard way of writing pseudocode – anything goes, as long as the result satisfies the definition of an algorithm.
- Three basic types of pseudocode statements:
  1. Sequential operations
  2. Conditional operations (IF-THEN-ELSE)
  3. Iterative operations (loops)

Representing Algorithms as Pseudocode: Sequential Operations

Computation: Set the value of a variable to something, e.g.,

Set the value of IND to IND + 1
Add 1 to IND
IND = IND + 1

Input: Get variable values from the outside world, e.g.,

Get the values for X and list L
Read in the values for X and list L

Output: Send some message (such as computed variable values) to the outside world, e.g.,

Print the value of SUM
Print the message "not found"
Return the value of SUM

Representing Algorithms as Pseudocode: Conditional Operations

IF-THEN: Do something if some condition is true, e.g.,

```
if (L_IND is X) then
    Set the value of FOUND to YES
    Print the message "found"
```

IF-THEN-ELSE: Do first thing if some condition is true and if not do second thing, e.g.,

```
if (L_IND is X) then
    Set the value of FOUND to YES
    Print the message "found"
else
    Set the value of IND to IND + 1
```

Representing Algorithms as Pseudocode: Conditional Operations (Cont'd)

Multiple IF-THEN-ELSE: Do something depending on which of a set of conditions is true and if none of the conditions are true do something else, e.g.,

```
if (L_IND is "apple" or "orange") then
    NUMFRUIT = NUMFRUIT + 1
else if (L_IND is "potato") then
    NUMVEG = NUMVEG + 1
else if (L_IND is "thyme") then
    NUMHERB = NUMHERB + 1
else
    print "unrecognized produce item"
```

Representing Algorithms as Pseudocode: Iterative Operations

Conditional Iteration: Repeat something as long as a condition is true, e.g.,

Set the values of IND to 1 and SUM to 0
while (IND $\leq$ n) do
    Set the value of SUM to SUM + L_{IND}
    Set the value of IND to IND + 1

Counted Iteration: Repeat something as long as a condition is true, e.g.,

Set the value of SUM to 0
for IND = 1 to n do
    Set the value of SUM to SUM + L_{IND}

Example Algorithms: Overview

- Every algorithm has at least one underlying intuition which is subsequently refined into an algorithm proper.
- Let's look at how some classic algorithms are derived from their underlying intuitions.
- These algorithms are for the following problems:
  1. List Search
  2. List Maximum Value
  3. List Sorting
  4. Bin Packing

The List Search Problem

LIST SEARCH
Input: A list L with n elements and a value X.
Output: The position of the element with value X in L if such an element exists and -1 otherwise.

- Has many applications, e.g., looking up a person's telephone number, charging some amount to a credit card, finding out if anyone won the Lotto Max jackpot this week.
- For simplicity, assume X and all list-elements are numbers; however, our algorithms work for any values that can be ordered, e.g., words, names.

Sequential List Search: Intuition

"Well, if I don't know anything else about the list except that it has n elements, I suppose I'll have to look at each element in the list and see if it is equal to the target-value. If I find such an element, I can stop and save that element's position; otherwise, I return -1 after I've look at all elements in the list. Sounds like a lot of work. Bummer."

Sequential List Search: Algorithm (Version 0)

Get values for X, list L, and n
Set the value of IND to 1 and FOUND to NO
While (FOUND = NO) and (IND $\leq$ n) do
    If L_{IND} is X then
        Set the value of FOUND to YES
        Print the message “found”
    Else
        Set the value of IND to IND + 1
If (FOUND = NO) then
    Print the message “not found”

Sequential List Search: Algorithm (Version 1)

Get values for X, list L, and n
Set the value of FPOS to -1 and IND to 1
while (FPOS = -1) and (IND $\leq$ n) do
    if (L_{IND} = X) then
        Set the value of FPOS to IND
    else
        Set the value of IND to IND + 1
return FPOS

Sequential List Search: Algorithm (Version 2)

Get values for X, list L, and n
FPOS = -1
IND = 1
while (FPOS = -1) and (IND $\leq$ n) do
    if (L_{IND} = X) then
        FPOS = IND
    else
        IND = IND + 1
return FPOS

Binary List Search: Intuition

"Hmmm ... Suppose this time I know L is sorted. Whenever I look at L_{IND} where IND is the middle of the list and L_{IND} is not equal to the target-value, as L is sorted, I know that the target-value must be either above or below IND in the list (depending on whether the target-value is greater or less than L_{IND}). I can keep repeating this in a loop until I either find the target-value or run out of list to search. This should finish way faster because each time I halve the size of the list I'm looking at. Cool!"

Binary List Search: Algorithm (Version 0)

```
Get values for X, list L, and n
Set the current list to all of L
while we haven't found X in list L and
  there's still a current list to search do
      if X isn't the middle element of the current list then
            if X > middle element then
                  set current list to upper
                  part of current list
            else
                  set current list to lower
                  part of current list
```

Binary List Search: Algorithm (Version 1)

Get values for X, list L, and n
FPOS = -1
LEFT = 1
RIGHT = n
while (FPOS = -1) and (LEFT $\leq$ RIGHT) do
    FPOS = (LEFT + RIGHT) / 2
    if (L_{FPOS} is not equal to X) then
        if (X > L_{FPOS}) then
            LEFT = FPOS + 1
        else
            RIGHT = FPOS - 1
        FPOS = -1
return FPOS

The List Maximum Value Problem

LIST MAXIMUM VALUE
Input: A list L with n elements.
Output: The position of the largest-valued element in L.

- Has many applications, e.g., looking for the employer that pays the highest salary for a particular job; is also a useful building block in more complex algorithms, e.g., list sorting.
- For simplicity, assume all list-elements are numbers; however, our algorithm work for any values that can be ordered, e.g., words, names.
- Can be readily adapted to find smallest list values.

List Maximum Search: Intuition

"Well, if I don't know anything else about the list except that it has n elements, I suppose I'll have to look at each element in the list and keep track as I go of which element is the largest I've found so far. After I've gone through this list, the largest I found by then is the largest in the list. Again, sounds like a lot of work. Bummer."

List Maximum Search: Algorithm (Version 0)

Get values for list L and n
Set the values of LARGEST to L_1, FPOS to 1, and
  IND to 2
While (IND $\leq$ n) do
    If $L_{IND} >$ LARGEST then
        Set the value of LARGEST to L_{IND}
        Set the value of FPOS to IND
    Set the value of IND to IND + 1
Print the values of LARGEST and FPOS

List Maximum Search: Algorithm (Version 1)

Get values for list L and n
FPOS = 1
IND = 2
while (IND $\leq$ n) do
    if $L_{IND} > L_{FPOS}$ then
        FPOS = IND
    IND = IND + 1
return FPOS

List Maximum Search: Algorithm (Version 2)

Get values for list L and n
FPOS = 1
for IND = 2 to n do
    If $L_{IND} > L_{FPOS}$ then
        FPOS = IND
return FPOS

The List Sorting Problem

LIST SORTING
Input: A list L with n elements.
Output: The version of LL sorted in ascending value order.

- Has many applications, e.g., generating lists of employees by name or salary; also enables algorithms that require sorted list, e,g„ binary list search.
- For simplicity, assume all list-elements are numbers; however, our algorithms work for any values that can be ordered, e.g., words, names.
- Can be readily adapted to sort in descending value order.

Selection Sort: Intuition

"The last element in a sorted list is the largest in the list, the second-last element is the largest among the remaining elements in the list, and so on. Perhaps we could use a find-list-maximum algorithm in a loop!"

Selection Sort: Algorithm (Version 0)

Get values for list L and n
Set the marker for the unsorted section at the end of L
While the unsorted section is not empty do
    Find largest element in unsorted section of list
    Swap this largest element with the last element in
      the unsorted part of the list
    Move the marker for the unsorted section left
      one position

Selection Sort: Algorithm (Version 1)

Get values for list L and n
ENDUNSORT = n
While (ENDUNSORT $>$ 1) do
    FPOS = 1
    for IND = 2 to ENDUNSORT do
        If $L_{IND} > L_{FPOS}$ then
            FPOS = IND
    TMP = $L_{ENDUNSORT}$
    $L_{ENDUNSORT}$ = L_{FPOS}
    L_{FPOS} = TMP
    ENDUNSORT = ENDUNSORT - 1

Bubble Sort: Intuition

"In order for a list to be unsorted, there must be at least one pair of adjacent list-elements L_{IND} and L_{IND+1} such that $L_{IND} > L_{IND+1}$. Suppose we kept going through the list, swapping bad adjacent list-element pairs until there weren't any more such pairs?"

Bubble Sort: Algorithm (Version 0)

```
Get values of list L
while list is not sorted do
    traverse L, swapping bad adjacent list-element pairs
      as necessary
    if no swaps occurred then
        the list is sorted
```

Bubble Sort: Algorithm (Version 1)

Get values of list L and n
SORTED = NO
while (SORTED = NO) do
    NUMSWAP = 0
    for IND = 2 to n do
        if ($L_{IND-1} > L_{IND}$) then
            TMP = L_{IND-1}
            L_{IND-1} = L_{IND}
            L_{IND} = TMP
            NUMSWAP = NUMSWAP + 1
        if (NUMSWAP = 0) then
            SORTED = YES

Bubble Sort: Algorithm (Version 2)

Get values of list L and n
SORTED = NO
while (SORTED = NO) do
    SORTED = YES
    for IND = 2 to n do
        if ($L_{IND-1} > L_{IND}$) then
            TMP = L_{IND-1}
            L_{IND-1} = L_{IND}
            L_{IND} = TMP
            SORTED = NO

The Bin Packing Problem

BIN PACKING
Input: A bin size B and a list L of n item sizes, each $\leq B$.
Output: The smallest number of bins of size B that can hold all of the items in L.

- Has many applications, e.g., minimizing order packaging for online retailers.
- Has a set of candidate solutions (packings of the items of L into bins), each with their own cost (number of bins in a packing), and requires a candidate solution that optimizes that cost; hence, this is an **optimization problem**.

Bin Packing: Intuitions

Intuition #1: "If I have at most n items, I'll need at most n bins. How about I try all possible ways of dividing the items in L among n or less bins, and then check each packing to make sure that no bin has items that are too big for their bin?"

Intuition #2: "Intuition #1 sounds way too hard. How about I just do it like Doug at Sobey's – take each item in L in turn and add it to the current bin, and if that item is too large, make a new bin and add it to that one?"

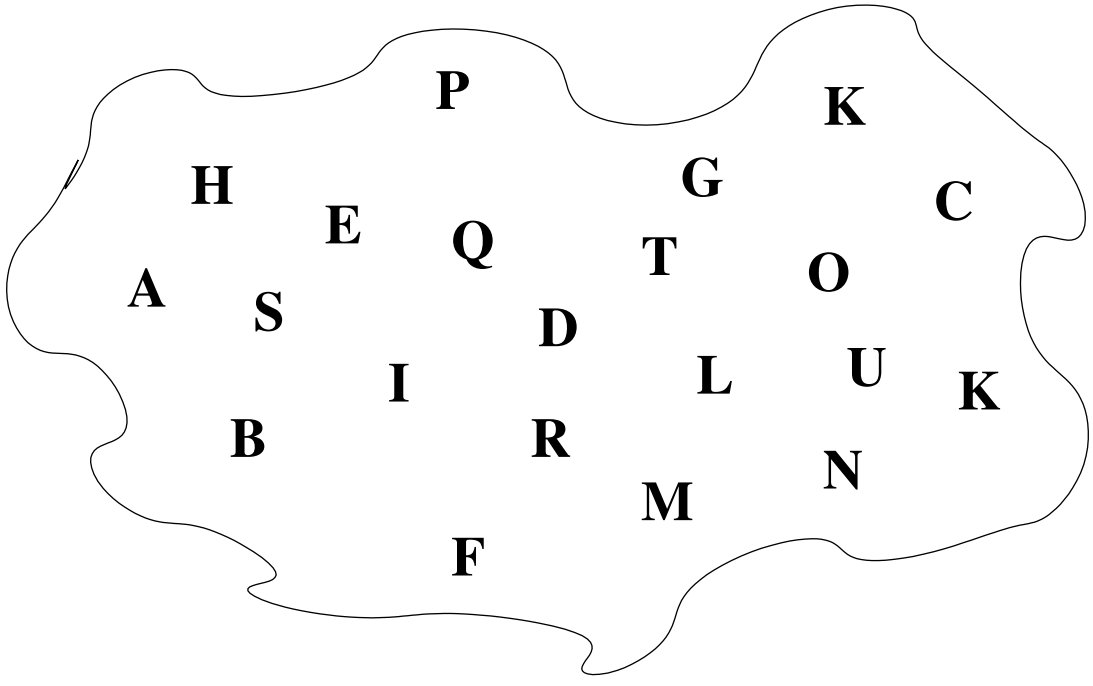
Problems, Algorithms, and Programs Redux

- A problem typically has many algorithms – which one should we implement in a program?
- Attributes of algorithms:
  - **Correctness**, i.e., does the algorithm produce the requested outputs for all inputs?
  - **Comprehensibility**, i.e., is the algorithm easy to understand?
  - **Elegance**, i.e., is the algorithm compact and/or mathematically beautiful?
  - **Efficiency**, i.e., is the runtime and/or memory used by the algorithm practical as input sizes increase?
- There may not be one algorithm for a problem that has all four attributes above. However, in general, correctness and efficiency are primary in real-world applications.

Assessing Algorithm Efficiency

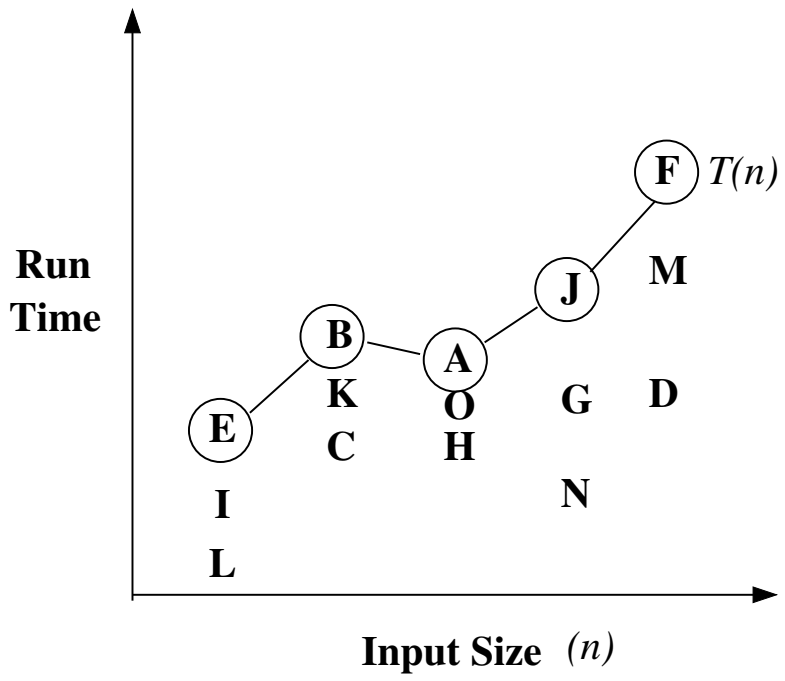
- In general, the best algorithm is the one with the lowest running time.
- Comparing algorithms by raw running time problematic:
  - Raw running times machine / language / OS dependent.
  - Raw running times input dependent.
  - Algorithm may not be implemented in a program.
- Need an abstract mathematical conception of algorithm efficiency, phrased in terms of a function of input size n, which is easily usable and comprehensible.
- The three abstractions involved in this conception sketched here can be viewed as necessary lies.

Necessary Lie #1: Focus on Important Instructions



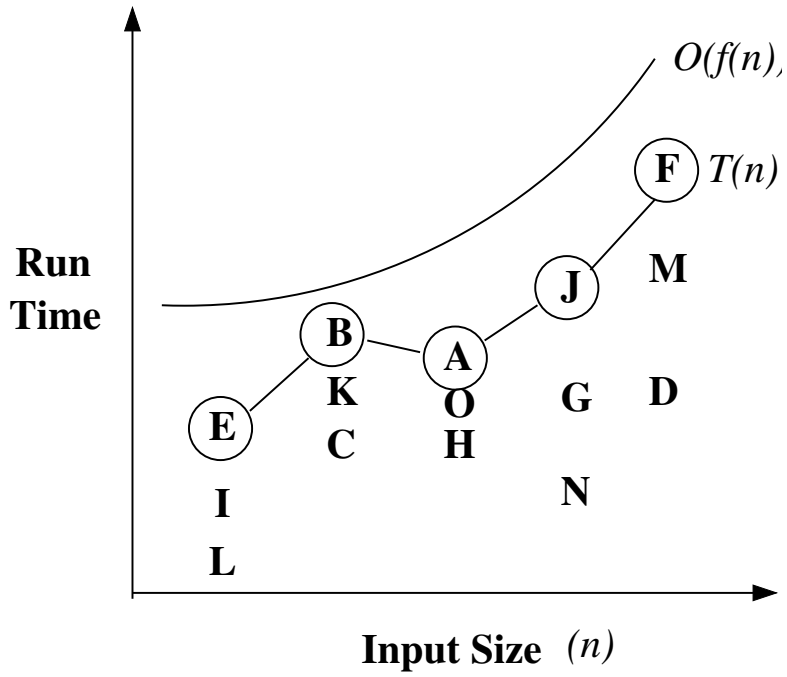
- Compute runtime on an input by counting the number of important instructions that are executed.
- Is machine-independent (raw abstract runtime).

Necessary Lie #2: Worst-Case Runtime Summary



- Group inputs by input size; summarize each size by largest runtime for that size.
- Is input-independent (worst-case abstract runtime).

Necessary Lie #3: Asymptotic Smoothing



- Reduce worst-case abstract runtime function to largest term.
- Is simple (asymptotic worst-case abstract runtime, i.e., worst-case time complexity).

Deriving Worst-Case Time Complexities

If already have worst-case abstract runtime function, select largest term, e.g.,

$$2\log n + 4 \quad \Rightarrow \quad O(\log n)$$

$$3n^2 + 1000n + 13 \quad \Rightarrow \quad O(n^2)$$

$$12n^4 + 5n^2 + 900 \quad \Rightarrow \quad O(n^4)$$

$$(3 \times 2^n) + 900n^{50} + 57 \quad \Rightarrow \quad O(2^n)$$

Deriving Worst-Case Time Complexities (Cont'd)

Otherwise, multiply out "deepest" loop-chain in algorithm, *e.g.*, $n \times n = O(n^2)$ time for Selection Sort.

```
Get values for list L and n
ENDUNSORT = n
While (ENDUNSORT > 1) do
    FPOS = 1
    for IND = 2 to ENDUNSORT do
        If L_IND > L_FPOS then
            FPOS = IND
    TMP = L_ENDUNSORT
    L_ENDUNSORT = L_FPOS
    L_FPOS = TMP
    ENDUNSORT = ENDUNSORT - 1
```

. . . And This All Matters Because? . . .

- Considering all instructions rather than just important ones only increases raw abstract runtime and hence worst-case abstract runtimes by constant multiplicative factors. But this is not the problem that it initially seems (see below).
- Could consider best- or average-case time complexity; however, worst-case best when trying to plan using algorithm-produced results in the real world, e.g., aircraft collision avoidance.
- Asymptotic smoothing gives much simpler functions than worst-case abstract runtimes, which eases algorithm comparison.

. . . And This All Matters Because? . . . (Cont'd)

- We typically want to solve larger and larger inputs (**living in Asymptopia**); moreover, functions with leading terms often differ regardless of constant multiplicative factors for sufficiently large inputs, i.e.,

$$\begin{aligned} c_1 n^2 &> c_2 n \\ c_1 n &> c_2 \\ n &> \frac{c_2}{c_1} \end{aligned}$$

This is the ultimate justification for asymptotic smoothing and our focus on orders of magnitude in assessing algorithm efficiency.

Time Complexity Orders of Magnitude

$O(\log n)$	Logarithmic Time	(Binary Search)
$O(n)$	Linear Time	(Sequential Search)
$O(n^2)$	Quadratic Time	(List Sort)
$O(2^n)$	Exponential Time	(Bin Packing)

Polynomial Time $= O(n^c)$ time for constant c

Table of Doom (1 Gigaflop/s Version)

Input Size (n)	Time Complexity				
	B-Search ($\log_2 n$)	S-Search (n)	Sort (n^2)	MST (n^3)	BP-E (2^n)
10	< 1 second	< 1 second	< 1 second	< 1 second	< 1 second
50	< 1 second	< 1 second	< 1 second	< 1 second	13 days
100	< 1 second	< 1 second	< 1 second	< 1 second	4×10^{13} years
1000	< 1 second	< 1 second	< 1 second	1 second	4×10^{284} years
one million	< 1 second	< 1 second	2 minutes	30 years	–
300 million	< 1 second	< 1 second	10 days	9×10^5 years	–
five billion	< 1 second	5 seconds	8 centuries	4×10^{12} years	–

Algorithms: Some Final Thoughts

- The conception of algorithm efficiency sketched here makes various assumptions that need not always be relevant, e.g., input sizes may always be small.
- It can be proven that certain problems do not have algorithms that are fast and correct – hence, fast algorithms that produce approximate solutions may need to be invoked, e.g., Bin Packing.
- There is typically no best algorithm for a problem – rather, there are algorithmic options that are more or less appropriate.

. . . Much work remains to be done . . .

. . . And If You Liked This . . .

- MUN Computer Science courses on this area:
  - COMP 2002: Data Structures and Algorithms
  - COMP 4740: Design and Analysis of Algorithms
  - COMP 4741: Formal Languages and Computability
- MUN Computer Science professors teaching courses / doing research in in this area:
  - Donna Batten
  - Rod Byrne
  - Antonina Kolokolova
  - Manrique Mata-Montero
  - Todd Wareham