# CSE 143 Java

Program Efficiency &
Introduction to Complexity Theory



# GREAT IDEAS IN COMPUTER SCIENCE

ANALYSIS OF ALGORITHMIC COMPLEXITY



## Overview

- Measuring time and space used by algorithms
- Machine-independent measurements
- Costs of operations
- Asymptotic complexity – O( ) notation and complexity classes
- Comparing algorithms
- Performance tuning



## Comparing Algorithms

- Example: We'll see two different list implementations
  - Dynamic expanding array
  - Linked list
- We'll see multiple ways of implementing other kinds of collections
- Which implementations are "better"?
- How do we measure?
  - Stopwatch? Why or why not?



## Program Efficiency & Resources

- Goal: Find way to measure "resource" usage in a way that is independent of particular machines or implementations
- Resources
  - Execution time
  - Execution space
  - Network or disk bandwidth
  - others
- We will focus on execution time
  - Techniques/vocabulary apply to other resource measures



## Example

- What is the running time of the following method?

```
// Return the sum of the elements in array.
double sum(double[ ] data) {
    double ans = 0.0;
    for (int k = 0; k < data.length; k++) {
        ans = ans + data[k];
    }
    return ans;
}
```

- How do we analyze this?
- What does the question even mean?

## Analysis of Execution Time

1. First: describe the *size* of the problem in terms of one or more parameters
   - For the sum method, the size of the data array makes sense
   - Often size of data structure, but can be magnitude of some numeric parameter, etc.
2. Then, count the number of *steps* needed *as a function of the problem size*

- Need to define what a "step" is
  - First approximation: one simple statement
  - More complex statements will be multiple steps



## Cost of operations: Constant Time Ops

- Constant-time operations: each take one abstract time "step"
  - Simple variable declaration/initialization (double sum = 0.0;)
  - Assignment of numeric or reference values (var = value;)
  - Arithmetic operation (+, -, *, /, %)
  - Array subscripting (a[index])
  - Simple conditional tests (x < y, p != null)
  - Operator new itself (not including constructor cost)

    Note: new takes significantly longer than simple arithmetic or assignment, but its cost is independent of the problem we're trying to analyze
- Watch out for things like method calls or constructor invocations that look simple, but can be expensive



## Cost of operations: Zero-time Ops

- Can sometimes perform operations at compile time
  - Nothing left to do at runtime
- Variable declarations without initialization

  double[ ] overdrafts;
- Variable declarations with compile-time constant initializers

  static final int maxButtons = 3;
- Some casts (but not those that need a runtime check)

  int code = (int) '?';
- These are generally either ignored or treated as constant-time



## Sequences of Statements

- Cost of

  S1; S2; …; Sn

  is sum of the costs of S1 + S2 + … + Sn



## Conditional Statement

- We're generally trying to figure out how long it *might* take to execute a statement (*worst case*), so the cost of

```
if (condition) {
    S1;
} else {
    S2;
}
```

is usually the max cost of S1 or S2 plus cost of the condition

- Other possibilities (less common)
  - *Best case* – use the min cost of S1 or S2
  - *Expected (average) case* – probabilistic analysis needed



## Analyzing Loops

- Basic analysis
  1. Calculate cost of each iteration
  2. Calculate number of iterations
  3. Total cost is the product of these

     Caution -- sometimes need to add up the costs differently if cost of each iteration is not roughly the same
- Nested loops
  - Total cost is number of iterations of the outer loop times the cost of the inner loop
  - same caution as above

## Method Calls

- Cost for calling a function is cost of...
  - cost of evaluating the arguments (constant or non-constant)
  - \+ cost of actually calling the function (constant overhead)
  - \+ cost of passing each parameter (normally constant time in Java for both numeric and reference values)
  - \+ cost of executing the function body (constant or non-constant?)

```
System.out.print(lineNumber);
System.out.println("Answer is " + calculateResult(x, y*y+42.0));
```

- Note that "evaluating" and "passing" an argument are two different things



## Exercise

- Analyze the running time of printMultTable
  - Pick the problem size
  - Count the number of steps

```
// print multiplication table with
// n rows and columns
void printMultTable(int n) {
    for (int k=1; k <= n; k++) {
        printRow(k, n);
    }
}
```

```
// print row r with length n of a
// multiplication table
void printRow(int r, int n) {
    for (int k = 1; k <= n; k++) {
        System.out.print(r*k + " ");
    }
    System.out.println( );
}
```



## Analysis



## Comparing Algorithms

- Suppose we analyze two algorithms and get these times (numbers of steps):
  - Algorithm 1: $37n + 2n^2 + 120$
  - Algorithm 2: $50n + 42$

How do we compare these? What really matters?

- Answer: In the long run, the thing that is most interesting is the cost as the problem size n gets large
  - What are the costs for n=10, n=100; n=1,000; n=1,000,000?
  - Mainstream computers are so fast these days that time needed to solve small problems is rarely of interest
    - Not necessarily so for slow, low-power, or embedded systems



## Orders of Growth

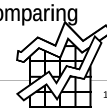
- What happens as the problem size doubles?

| N | $\log_2 N$ | 5N | $N \log_2 N$ | $N^2$ | $2^N$ |
|---|---|---|---|---|---|
| 8 | 3 | 40 | 24 | 64 | 256 |
| 16 | 4 | 80 | 64 | 256 | 65536 |
| 32 | 5 | 160 | 160 | 1024 | $\sim 10^9$ |
| 64 | 6 | 320 | 384 | 4096 | $\sim 10^{19}$ |
| 128 | 7 | 640 | 896 | 16384 | $\sim 10^{38}$ |
| 256 | 8 | 1280 | 2048 | 65536 | $\sim 10^{76}$ |
| 10000 | 13 | 50000 | $10^5$ | $10^8$ | $\sim 10^{3010}$ |



## Asymptotic Complexity

- Asymptotic: Behavior of complexity function as problem size gets large
  - Only thing that really matters is higher-order term
  - Can drop low order terms and constants
- The asymptotic complexity gives us a (partial) way to answer "which algorithm is more efficient"
  - Algorithm 1: $37n + 2n^2 + 120$ is proportional to $n^2$
  - Algorithm 2: $50n + 42$ is proportional to n
- Graphs of functions are handy tool for comparing asymptotic behavior



## Big-O Notation

- Definition: If f(n) and g(n) are two complexity functions, we say that

  f(n) = O(g(n))  ( pronounced f(n) is O(g(n)) or is order g(n) )

  if there is a constant c such that

  $f(n) \le c \cdot g(n)$

  for all sufficiently large n

5/14/2004 (c) 2001-4, University of Washington 16-19

## Exercise 1

- Prove that 5n+3 is O(n)

5/14/2004 (c) 2001-4, University of Washington 16-20

## Exercise 2

- Prove that $5n^2 + 42n + 17$ is $O(n^2)$

5/14/2004 (c) 2001-4, University of Washington 16-21

## Implications

- The notation f(n) = O(g(n)) is *not* an equality

  (yet another abuse of the = sign; c.f., assignment operator)
- Think of it as shorthand for
  - "f(n) grows at most like g(n)" or
  - "f grows no faster than g" or
  - "f is bounded by g"
- O( ) notation is a *worst-case* analysis
  - Generally useful in practice
  - Sometimes want *average-case* or *expected-time* analysis if worst-case behavior is not typical (but often harder to analyze)

5/14/2004 (c) 2001-4, University of Washington 16-22

## Complexity Classes

- Several common complexity classes (problem size n)
  - Constant time: O(k) or O(1)
  - Logarithmic time: O(log n) [Base doesn't matter. Why?]
  - Linear time: O(n)
  - "n log n" time: O(n log n)
  - Quadratic time: $O(n^2)$
  - Cubic time: $O(n^3)$

    …
  - Exponential time: $O(k^n)$
- $O(n^k)$ is often called *polynomial time*

5/14/2004 (c) 2001-4, University of Washington 16-23

## Big-O Arithmetic

- For most common functions, comparison can be enormously simplified with a few simple rules of thumb
- Memorize complexity classes in order from smallest to largest: O(1), O(log n), O(n), O(n log n), $O(n^2)$, etc.
- Ignore constant factors

  $300n + 5n^4 + 6 + 2^n = O(n + n^4 + 2^n)$
- Ignore all but highest order term

  $O(n + n^4 + 2^n) = O(2^n)$

5/14/2004 (c) 2001-4, University of Washington 16-24

## Rule of Thumb

- If the algorithm has polynomial time or better: practical
  - typical pattern: examining all data, a fixed number of times
- If the algorithm has exponential time: impractical
  - typical pattern: examine *all combinations* of data
- What to do if the algorithm is exponential?
  - Try to find a different algorithm
  - Some problems can be proved not to have a polynomial solution
  - Other problems don't have known polynomial solutions, despite years of study and effort
  - Sometimes you settle for an approximation

    The correct answer most of the time, or an almost-correct answer all of the time

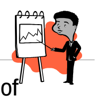


## Computer Science Note

- Algorithmic complexity theory is one of the key intellectual contributions of Computer Science
- Typical problems
  - What is the worst/average/best-case performance of an algorithm?
  - What is the best complexity bound for all algorithms that solve a particular problem? (i.e., how intrinsically difficult is the problem – regardless of how clever a programmer you are?)
- Interesting and (in many cases) complex, sophisticated math
  - Probabilistic and statistical as well as discrete
- Still some key open problems
  - Most notorious: P ?= NP



## Analyzing List Operations (1)



- We can use O( ) notation to compare the costs of different list implementations
- Operation Dynamic Array Linked List
  - Construct empty list
  - Size of the list
  - isEmpty
  - clear



## Analyzing List Operations (2)

- Operation Dynamic Array Linked List
  - Add item to end of list
  - Locate item (contains, indexOf)
  - Add or remove item once it has been located



## Wait! Isn't this totally bogus??

- Write better code!!
  - More clever hacking in the inner loops

    (assembly language, special-purpose hardware in extreme cases)
- Moore's law: Speeds double every 18 months
  - Wait and buy a faster computer in a year or two!

- But ...



## How long is a Computer-Day?

- If a program needs f(n) microseconds to solve some problem, how big a problem can it solve in a day?
  - One day = 1,000,000*24*60*60 = $9*10^{10}$ (aprox)

| f(n) | n such that f(n) = one day |
|---|---|
| n | $9 * 10^{10}$ |
| 5n | $2 * 10^{10}$ |
| $n \log_2 n$ | $3 * 10^9$ |
| $n^2$ | $3 * 10^5$ |
| $n^3$ | $4 * 10^3$ |
| $2^n$ | 36 |

## Speed Up The Computer by 1,000,000

- Suppose technology advances so that a future computer is 1,000,000 fast than today's

(Or you discover a clever hack that gives a 1,000,000 speedup)

| f(n) | original n | speedup on future machine |
|---|---|---|
| n | $9 * 10^{10}$ | million times |
| 5n | $2 * 10^{10}$ | million times |
| $n \log_2 n$ | $3 * 10^9$ | 60,000 times |
| $n^2$ | $3 * 10^5$ | 1,000 times |
| $n^3$ | $4 * 10^3$ | 100 times |
| $2^n$ | 36 | +20 |



## Practical Advice For Speed Lovers

- First pick the right algorithm and data structure
  - Implement it carefully, insuring correctness
- Then optimize for speed – but only where it matters
  - Constants do matter in the real world
  - Clever coding can speed things up, but the result is likely to be harder to read, modify
  - Use tools to find hotspots – concentrate on these

**"Premature optimization is the root of all evil"**
– Donald Knuth



## More Advice...

**"It is easier to make a correct program efficient than to make an efficient program correct"**

**-- Edsgar Dijkstra**



## Summary

- Analyze algorithm sufficiently to determine complexity
- Compare algorithms by comparing asymptotic complexity
- For large problems, an asymptotically faster algorithm will always trump clever coding tricks
- Optimize/tune only things that actually matter, once you've picked the best algorithm