Complexity Analysis

2501ICT/7421ICT Nathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

Overview

- Efficiency of Algorithms
- Time and Space
- Measuring Efficiency
- Big-O Analysis
- Examples and Case Studies
- Search Algorithms
- Sort Algorithms

Why Analyse Complexity?

- Improve Algorithm Efficiency
  - need to assess efficiency
  - $\rightarrow$ can Make a Big Difference
- Large data sets
- Computationally intense problems
- But: Correctness is most important
  - $\rightarrow$ make correct algorithm efficient, not vice versa!

Algorithm Efficiency

- Algorithms Consume Resources
  - processing time
  - memory space
- Compromise
  - speed up at expense of memory
  - slower, but more memory efficient algorithms

  $\rightarrow$ application dependent!

Problem Size

- Problem Size is the data set size an algorithm works on

Example (Sum Integers in an Array)

12	4	17	$\cdots$	29	18
1	2	3	$\cdots$	n − 1	n

- *n* Integers:
  - $\rightarrow$ problem size is *n*

Measuring Efficiency

- Time a program using the computer clock
  - → `time` *`command`* on the command line
  - takes too long for large data sets
    - → may not even complete!
  - system dependent (compiler, hardware, . . .)
- Count block or instruction *iterations*
  - better overall indicator
  - works with abstract representations
  - → constant and variable figures (loops)

Execution Time

- AAAA: Abstractly assess an algorithm
  $\rightarrow$ without executing an actual program

Example (How many times is S_1 executed?)

```
for (i = 0; i < 10; i++)
for (i = 0; i < N; i++)
    S1;
```

- S_1 is the Privileged Instruction
  - if S_1 takes a constant amount of time k_1, how long will the loop take to execute?
    $\Rightarrow$ execution time $t = N \cdot k_1$

Execution Time (2)

Example (Adding a constant Overhead)

S_0;
```
for (i = 0; i < N; i++)
```
    S_1;
S_2;

- Assuming S_0 and S_2 together take a constant time k_2:
  $\Rightarrow$ total execution time $t = N \cdot k_1 + k_2$

Complexity and Problem Size

- Small n
  - time differences are small
  - most algorithms perform similarly
- Large n
  - huge difference
    - $\rightarrow$ several seconds vs. thousands of years or more!
- Complexity figure is significant for large values of n

Large Values of n

Example (comparing different efficiencies)

n	$n \log_{10} n$	n^2	2^n
5	3.49	25	32
10	10	100	1,024
100	200	10,000	10^{30}
1,000	3,000	1,000,000	10^{301}
10,000	40,000	100,000,000	10^{3010}

Influence of Lesser Terms

Example (Lesser Terms become Insignificant)

n	n^2	$n^2 + 14n + 26$	Influence
10	100	266	166 %
100	10,000	11,426	14.3 %
1,000	1,000,000	1,014,026	1.4 %
10,000	100,000,000	100,140,026	0.14 %

- n^2 has the biggest effect
  - $\Rightarrow$ lesser terms become insignificant!

Big-O Notation

- Less Significant Terms are Ignored
  - $\Rightarrow$ $n^2 + 14n + 26$ becomes $O(n^2)$
  - $\rightarrow$ $O(n^2)$ is read "Order n Squared"
- General Case
  - factor with highest significance determines Order of Magnitude
    - $\rightarrow$ $O(n^k)$: $a + bn + cn^2 + \cdots + n^k$
    - $\rightarrow$ $O(k^n)$: $a + bn + cn^2 + \cdots + n^k + k^n$

Magnitude of Complexity

Example (most significant term matters)

$$
\begin{aligned}
2n^2 + 6 &= O(n^2) \\
n^3 + 3000n^2 + 6 &= O(n^3) \\
2^n + 4n^100 + 5n &= O(2^n)
\end{aligned}
$$

- Placement of data may influence algorithm complexity
  - Best-Case, Average, and Worst-Case
  - → Significant: Average and Worst-Case

Big-O Limitations

- Less Significant Terms
  - → can be quite significant for small and medium size data sets!
- Constant of Proportionality
  - → $800n^2$
    - almost 3 orders of magnitude more complex than n^2

Common Orders of Magnitude

Example (Common Orders of Magnitude)

$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	Log Linear
$O(n^2)$	n-Squared
$O(2^n)$	Exponential

Common Cases

- Single loop processing n items
  - $O(n)$
- Two nested loops of n items
  - $O(n^2)$
- Binary Search
  - $O(\log_2 n)$
- Efficient Sort
  - $O(n \log_2 n)$

Logarithms

- $log_2 n$ complexity is very common!
  $\rightarrow$ $\log n$
- Examples:
  - Repeated doubling
    - Start at 1, double until size reaches n
  - Repeated halving
    - Start at n, half data set until 1 is reached
  - Binary searches, efficient sorting, ...