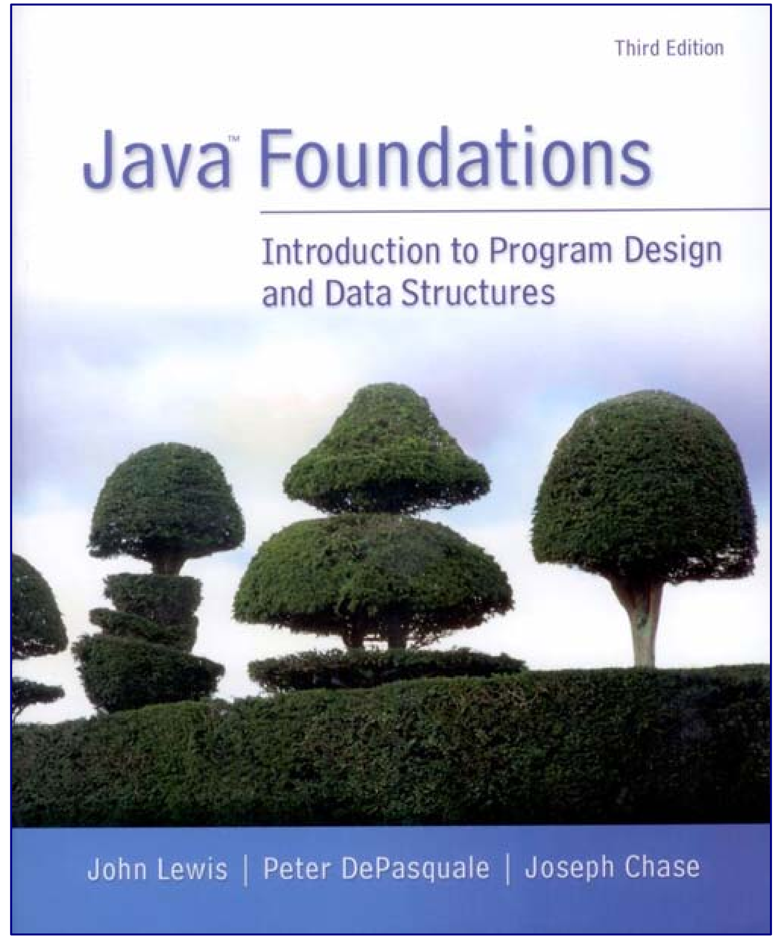
Analysis of Algorithms
Chapter 11

Instructor: Scott Kristjanson

CMPT 125/125

SFU Burnaby, Fall 2013



Scope

Analysis of Algorithms:

- Efficiency goals
- The concept of algorithm analysis
- Big-Oh notation
- The concept of asymptotic complexity
- Comparing various growth functions

Algorithm Efficiency

The efficiency of an algorithm is usually expressed in terms of its use of CPU time

The analysis of algorithms involves categorizing an algorithm in terms of efficiency

An everyday example: washing dishes

- Suppose washing a dish takes 30 seconds and drying a dish takes an additional 30 seconds
- Therefore, n dishes require n minutes to wash and dry

Algorithm Efficiency

Now consider a less efficient approach that requires us to dry all previously washed dishes each time we wash another one

Each dish takes 30 seconds to wash

But because we get the dishes wet while washing,

- must dry the last dish once, the second last twice, etc.
- Dry time = 30 + 2*30 + 3* 30 + … + (n-1)*30 + n*30
- = 30 * (1 + 2 + 3 + … + (n-1) + n)

$$= n * (30 \text{ seconds wash time}) + \sum_{\text{i}=1}^{\text{n}} (i * 30)$$

$$\text{time } (n \text{ dishes}) = 30n + \frac{30n(n+1)}{2}$$

$$= 15n^2 + 45n \text{ seconds}$$

Problem Size

For every algorithm we want to analyze, we need to define the size of the problem

The dishwashing problem has a size *n*

n = number of dishes to be washed/dried

For a search algorithm, the size of the problem is the size of the search pool

For a sorting algorithm, the size of the program is the number of elements to be sorted

Growth Functions

We must also decide *what* we are trying to efficiently optimize

- *time complexity* – CPU time
- *space complexity* – memory space

CPU time is generally the focus

A growth function shows the relationship between the size of the problem (n) and the value optimized (time)

Asymptotic Complexity

The growth function of the second dishwashing algorithm is

$$t(n) = 15n^2 + 45n$$

It is not typically necessary to know the exact growth function for an algorithm

We are mainly interested in the ***asymptotic complexity*** of an algorithm – the general nature of the algorithm as n increases

Asymptotic Complexity

Asymptotic complexity is based on the ***dominant term*** of the growth function – the term that increases most quickly as n increases

The dominant term for the second dishwashing algorithm is n^2:

Number of dishes (n)	$15n^2$	$45n$	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

Big-Oh Notation

The coefficients and the lower order terms become increasingly less relevant as n increases

So we say that the algorithm is *order* n^2, which is written $O(n^2)$

This is called *Big-Oh notation*

There are various Big-Oh categories

Two algorithms in the same category are generally considered to have the same efficiency, but that doesn't mean they have equal growth functions or behave exactly the same for all values of n

Big-Oh Categories

Some sample growth functions and their Big-Oh categories:

Growth Function	Order	Label
$t(n) = 17$	$O(1)$	constant
$t(n) = 3\log n$	$O(\log n)$	logarithmic
$t(n) = 20n - 4$	$O(n)$	linear
$t(n) = 12n \log n + 100n$	$O(n \log n)$	n log n
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	quadratic
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	cubic
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponential

Comparing Growth Functions

You might think that faster processors would make efficient algorithms less important

A faster CPU helps, but not relative to the dominant term.

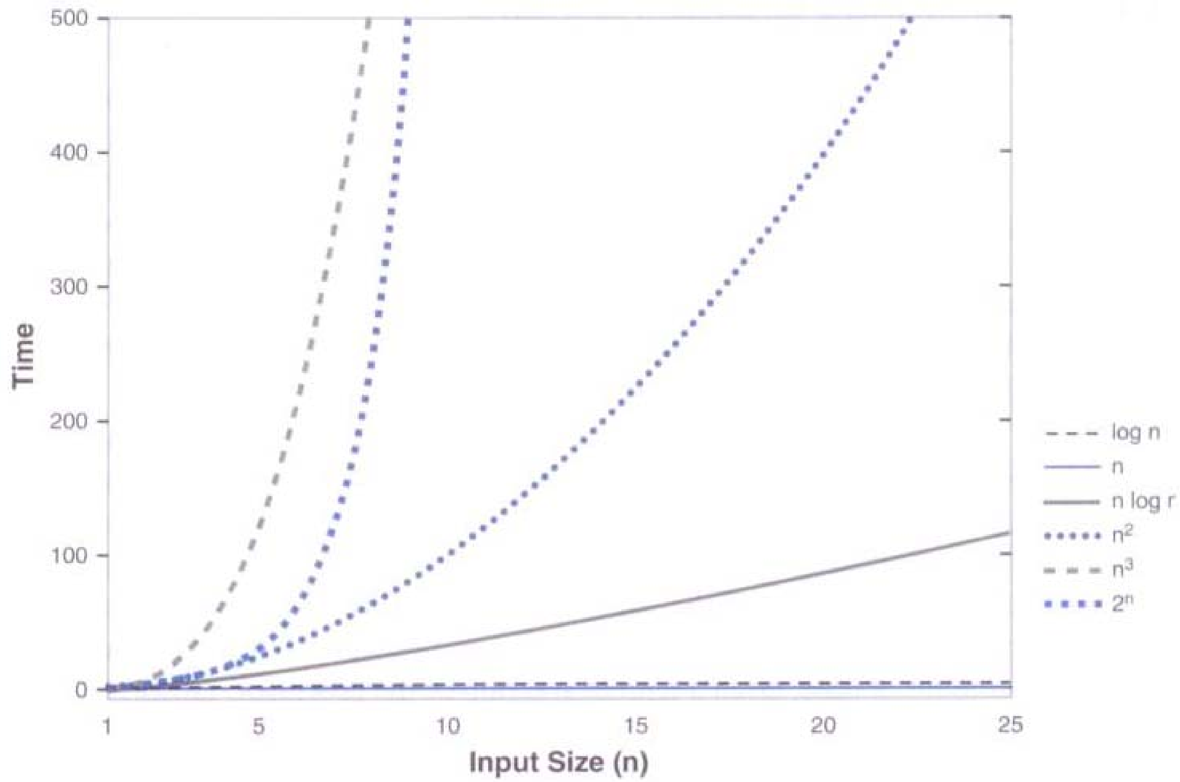
What happens if we increase our CPU speed by 10 times?

Algorithm	Time Complexity	Max Problem Size Before Speedup	Max Problem Size After Speedup
A	n	s_1	$10s_1$
B	n^2	s_2	$3.16s_2$
C	n^3	s_3	$2.15s_3$
D	2^n	s_4	$s_4 + 3.3$

Comparing Growth Functions

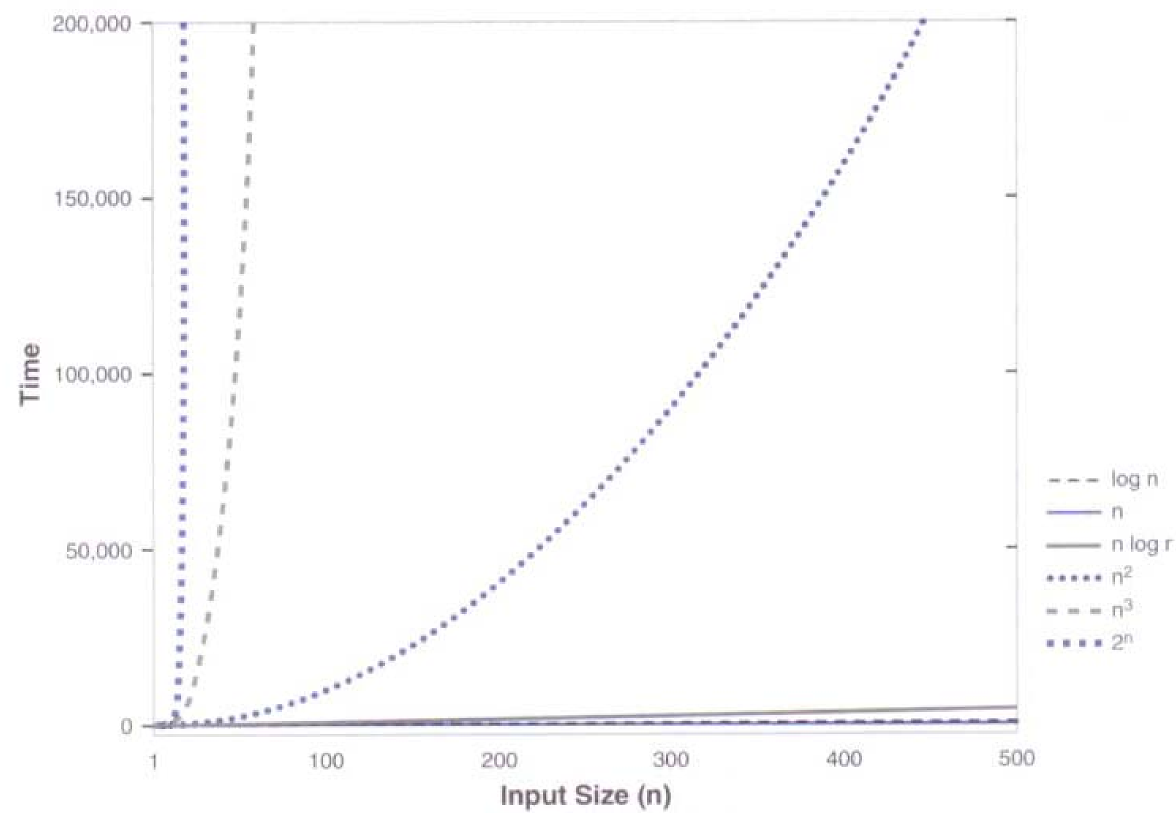
12

As n increases, the various growth functions diverge dramatically:



Comparing Growth Functions

For large values of n, the difference is even more pronounced:



Analyzing Loop Execution

First determine the order of the body of the loop, then multiply that by the number of times the loop will execute

```
for (int count = 0; count < n; count++)
    // some sequence of O(1) steps
```

N loop executions times O(1) operations results in a O(n) efficiency

Can write:

- CPU-time Complexity = n * O(1)
- = O(n*1)
- = O(n)

Analyzing Loop Execution

Consider the following loop:

```
count = 1;
while (count < n)
{
    count *= 2;
    // some sequence of O(1) steps
}
```

How often is the loop executed given the value of n?

The loop is executed $\log_2 n$ times, so the loop is O(log n)

CPU-Time Efficiency = log n * O(1) = O(log n)

Analyzing Nested Loops

When loops are nested, we multiply the complexity of the outer loop by the complexity of the inner loop

```
for (int count = 0; count < n; count++)
    for (int count2 = 0; count2 < n; count2++)
    {
        // some sequence of O(1) steps
    }
```

Both the inner and outer loops have complexity of O(n)

For Body has complexity of O(1)

CPU-Time Complexity $= O(n)*(O(n) * O(1))$

$= O(n) * (O(n * 1))$

$= O(n) * O(n)$

$= O(n*n) = O(n^2)$

The overall efficiency is $O(n^2)$

Analyzing Method Calls

- The body of a loop may contain a call to a method
- To determine the order of the loop body, the order of the method must be taken into account
- The overhead of the method call itself is generally ignored

Interesting Problem from Microbiology

Predicting RNA Secondary Structure

- using Minimum Free Energy (MFE) Models

Problem Statement:

Given:

- an ordered sequence of RNA bases S = (s1, s2, …, sn)
- where si is over the alphabet {A, C, G, U}
- and s1 denotes the first base on the 5’ end, s2 the second, etc.,

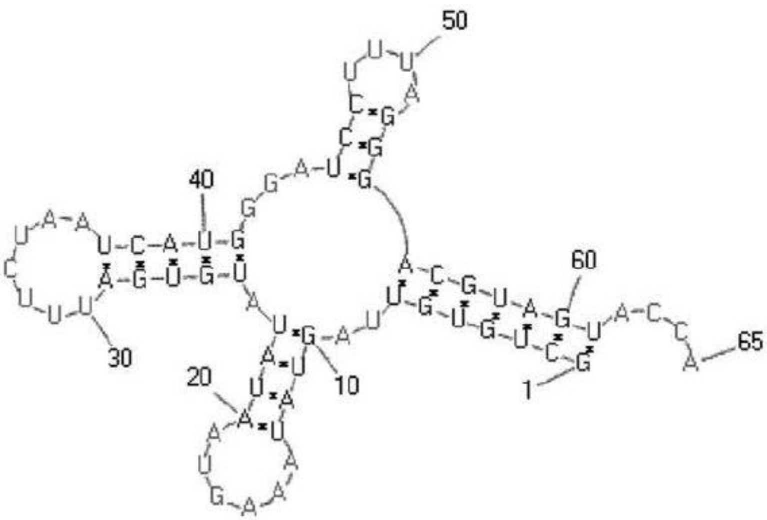
Using Watson-Crick pairings: A-U, C-G, and wobble pair G-U

Find Secondary Structure R such that:

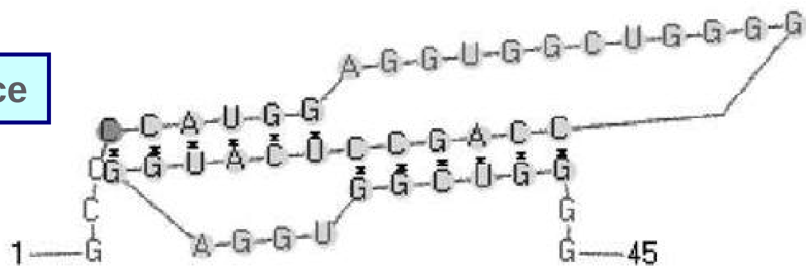
- R described by the set of pairs i,j with $1 \leq i < j \leq n$
- The pair i.j denotes that the base indexed i is paired with base indexed j
- For all indexes from 1 to n, no index occurs in more than one pair
- Structure R has minimum free energy (MFE) for all such structures
- MFE estimated as sum energies of the various loops and sub-structures

Example RNA Structures and their Complexity

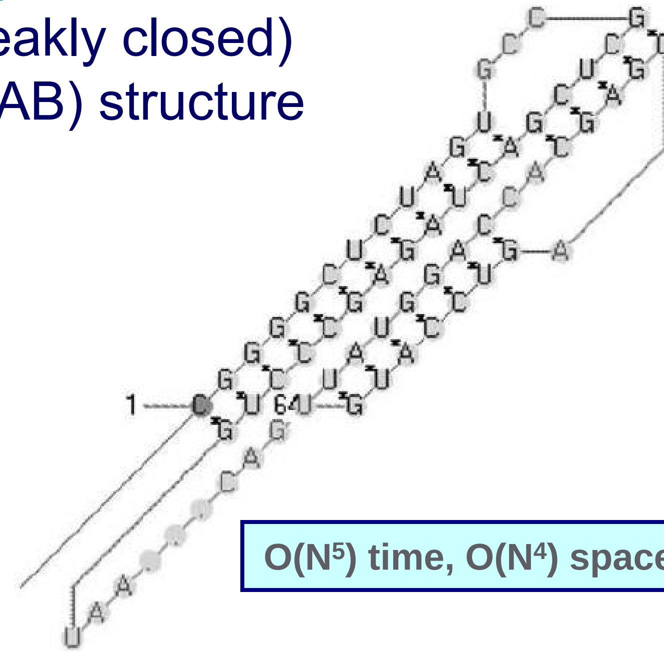
Left - a pseudoknot-fee structure (weakly closed)
Center - an H-Type pseudoknotted (ABAB) structure
Right - a kissing hairpin (ABACBC)



O(N^3) time, O(N^2) space



O(N^4) time, O(N^2) space



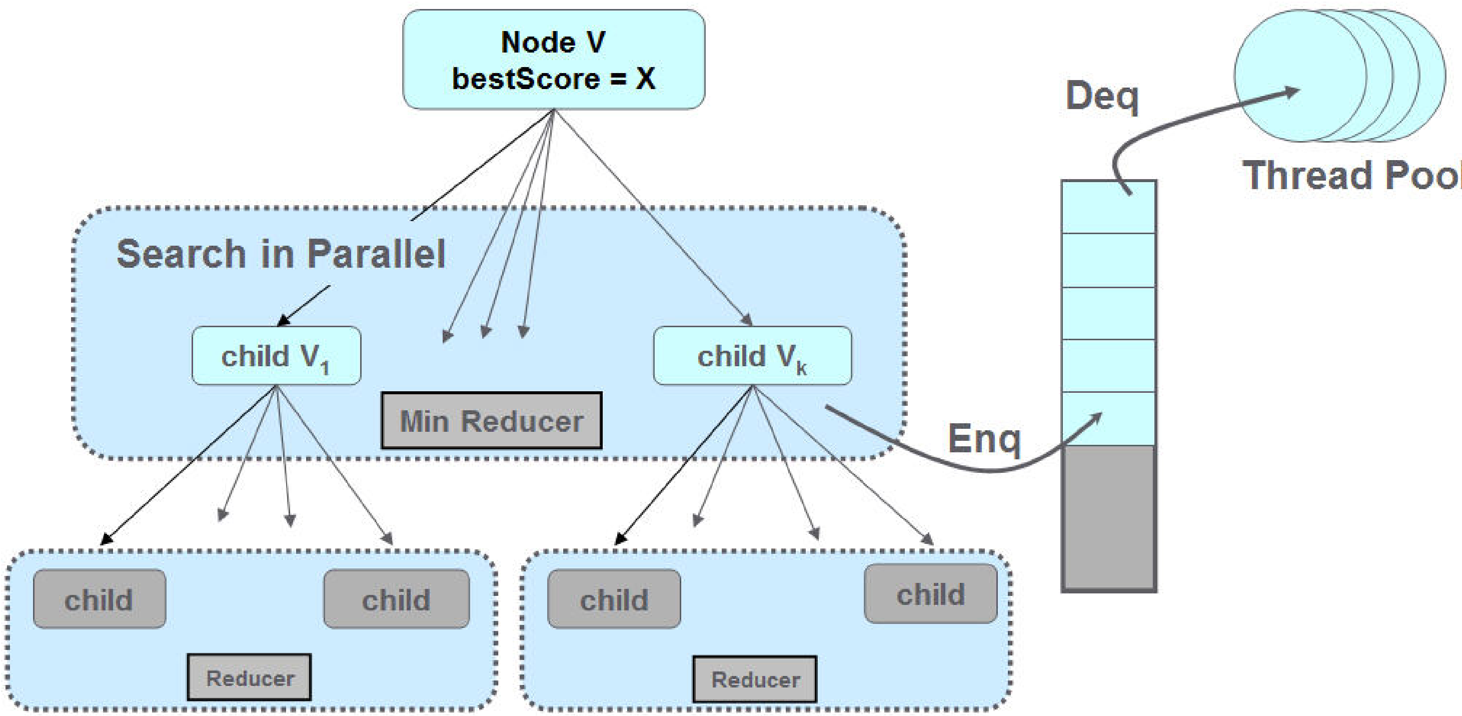
O(N^5) time, O(N^4) space

Solve the Problem in Parallel

Search the various possible RNA foldings using search trees

Use Branch and Bound to cut off bad choices

Use Parallelism to search multiple branches at the same time on different CPUs



Key Things to take away:

Algorithm Analysis:

- Software must make efficient use of resources such as CPU and memory
- Algorithm Analysis is an important fundamental computer science topic
- The order of an algorithm is found be eliminating constants and all but the dominant term in the algorithm’s growth function
- When an algorithm is inefficient, a faster processor will not help
- Analyzing algorithms often focuses on analyzing loops
- Time complexity of a loop is found by multiplying the complexity of the loop body times the number of times the loop is executed.
- Time complexity for nested loops must multiply the inner loop complexity with the number of times through the outer loop

References:

1. J. Lewis, P. DePasquale, and J. Chase., *Java Foundations: Introduction to Program Design & Data Structures*. Addison-Wesley, Boston, Massachusetts, 3rd edition, 2014, ISBN 978-0-13-337046-1