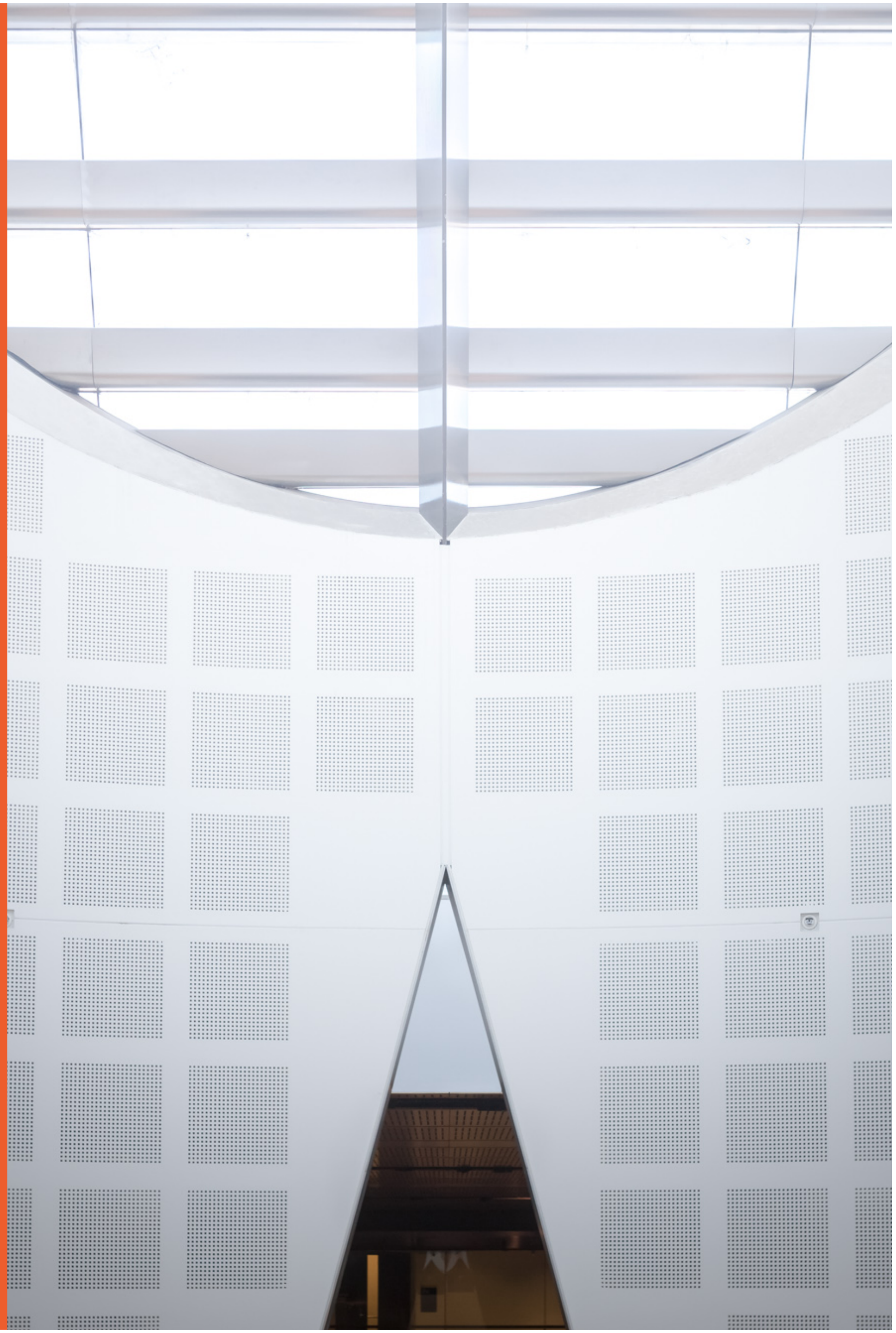
INFO1105/1905/9105
Data Structures
Week 3b: Scalability
see textbook sect 4.1, 4.2, 4.3

Professor Alan Fekete
Professor Seokhee Hong
School of Information Technologies

using material from Dr Taso Viglas,
A/Prof Kalina Yacef,
A/Prof Michael Charleston, and others

THE UNIVERSITY OF SYDNEY



Copyright warning

Concepts

- Measuring time and space
- Big "Oh" notation
- Analysing & comparing complexity
- Examples

Computer science is lucky

- Unlike other disciplines, we have a good mathematical measure of how ‘hard’ problems are.
- This can tell us how long it takes to solve them
- We will talk about how to *determine and compare* the complexity of algorithms and methods in your programs.

Different tasks, different techniques

- Suppose you want to
- You might

find someone's address, given their name and a phone book	go to the middle: found it? is it to the right or left? continue with a *smaller problem*.
sort 1000 names in a list	make piles of A's, B's, etc., then make piles of AA's, AB's, etc.
find someone's name, given their phone number, and a phone book	look at the first number: is this it? if not, continue...

Sorting

– If I deal you 5 cards, how do you sort them? Take a moment and think about it... (*NB: you have to sort them in place!*)

– Now, what if you have 10 cards: will you do it the same way? What if you have 100 cards, or 1000? (and very big hands)

– *Sorting can be fast or slow: it depends on the method you use.*

Finding things

- If you have an idea where to start looking, then finding things can be very fast.
- In the phone book example all the names are *ordered* so finding a name is fast…
- but the numbers in the phone book are not in order: searching for a particular phone number might mean *traversing the entire list.*
- *Access can be fast or slow: it depends on the data structure you use.*

Time and Space

- Complexity can be measured both in terms of time and of space.
  - on *average*, in the *best* case, or the *worst* case
- Quite often we have loads of storage capacity available, so the more immediate quantity is time: how long will my program take to run?
- but *don't forget about space*: in some situations space is severely limited (like on your mobile phone)!

Which is better?

```
public class Student {

  private int marks = 0;

  public Student(int m) {
    marks = m;
  }

  public int getMarks( ) {
    return marks;
  }
}
```

```
// summing over an array:
int arrayTotal = 0;
for (int i = 0; i < max; i++) {
    arrayTotal += s_array[i].getMarks( );
}
System.out.println("answer = " + arrayTotal);
```

```
// summing over an ArrayList:
int alTotal = 0;
for (int i = 0; i < max; i++) {
    s = (Student) s_al.get(i);
    alTotal += s.getMarks( );
}
System.out.println("answer = " + alTotal);
```

```
// summing over a LinkedList:
int llTotal = 0;
for (int i = 0; i < max; i++) {
    s = (Student) s_ll.get(i);
    llTotal += s.getMarks( );
}
System.out.println("answer = " + lTotal);
```

how long do these take?

So, what *is* scalability?

- Scalability refers to how well a system copes with increasing load/demand/data
- Scalability is described in terms of the increase in resources required:
  - CPU
  - Disk space
  - Memory
  - Network ports

  space

  - Run time cost

  time

Complexity and Scalability

- *Scalability* is the concept of how well a program performs, when the size of the input increases.
- This is closely linked with *complexity*, which describes how algorithms and methods behave when the problem size increases.

high complexity ⇔ poor scalability

low complexity ⇔ good scalability

Why it matters

- Sometimes, resources are constrained
  - processing todays sales needs to be done before tomorrow!
  - computer has a limited amount of memory
- Sometimes, one can get more resources, but they cost money
  - eg pay for computation
- Sometimes, there is “opportunity cost”
  - we could spend more time on this task, but that would mean we are not doing something else worthwhile

- We usually want to use as little time as possible (and sometimes, use little space too)

An alternative view

- Sometimes, we want to process as much input as possible in a given amount of time or space
  - especially with machine learning, the more data we process the better our understanding/predictions
- Scalability influences how big the input can be, that we can handle within the resources available

How do we measure this?

- Two main ways to analyze the efficiency of programs and algorithms:
  - Empirical / Real Timing: measure time taken by running the program
  - Analytical: analyze the running time theoretically

Real Timing/Empirical Approach

- Run a program with some input and use a clock to time it.
- Pros
  - Extremely precise, and it's a real cost.
  - Can show costs for memory allocation and indexing, which cannot be seen in the form of algorithm
  - Easy to do!
- Cons
  - It depends on the environment: the
    - compiler
    - hardware
  - Timing may depend on loads (what else is running?).
  - Can only be done after a program is written

Analytical Approach

- Examine an algorithm / a program and determine how long it will take.
- Pros
  - Able to carry out *before* the program is written up
  - Independent of the hardware/compiler.
- Cons
  - Can be difficult and subtle: Different Java instructions and data structures can give different performance (e.g., accessing a LinkedList using an index).

First Taste of Analysis

How many steps are taken in the execution of these codes?

```
int temp = x;
x = y;
y = temp;
```

Number of steps = 1 + 1 + 1 = 3

First Taste of Analysis

How many steps are taken in the execution of these codes?

```
if(x < y) {
   int temp = x;
   x = y;
   y = temp;
}
```

It depends (more if x<y, less otherwise)!
The worst case (when x<y): Num of steps = 1 + 1 + 1 + 1 = 4

First Taste of Analysis

How many steps are taken in the execution of this loop?

```
for (i = 0; i <  10  ; i++) {
    ar[i] = i;
}
```

1

1 * 10 + 1

1 * 10

1 * 10

Num of steps = 1 + (1* 10 + 1) + (1* 10) + (1 * 10) = 32

First Taste of Analysis (Generalized)

How many steps are taken in the execution of this loop?

```
for (i = 0; i < n  ; i++) {
    ar[i] = i;
}
```

1

$1 * n + 1$

$1 * n$

$1 * n$

- Number of steps:
  - $= 1 + (1 * n + 1) + (1 * n) + (1 * n)$
  - $= 1 + (n + 1) + (n) + (n)$
  - $= 3n + 2$
  - $= O(n)$

Scalability of run-time

- The key to good estimation of run-time is knowing the impact of *growth* in the *size of the input*
- Untrained people often assume that run-time grows proportionately with the input:
  - they assume that if you double the size of a list, the run-time (of accessing it, sorting it) doubles.
  - This is *not* necessarily true!
  - Some code slows down much *more* or *less* than this.

Terminology

- The commonest term seen in this area is big 'Oh' notation*.
- We say a function is O(g(n)) ("order g of n") if it grows no faster than g does, when n increases.
  - n^2 is O(n^2), 25n is O(n)

  (we don't care about the 25 above)
  - n is also O(n^2) because it grows *no faster than* n^2

  We focus on cases where the function g(n) is fairly simple, eg $g(n) = n^2$
- We are concerned with the *asymptotic growth* of the functions.

*there are others, for growing *at least as fast as*, or at the *same* rate...

Asymptotic growth

- We treat run-time cost as a *function* of the size of the input.
  - Measure an appropriate input size for the project
    - e.g., total number of objects in the system, number of elements in a list
    - Input size is usually called N (or n)
  - For each input size, consider the worst (or average, or best) case possible:
    - Focus on inputs of each size that make a program take the longest time
    - The Big-'Oh' notation, i.e., O(g(n))

Simplifying complexity

- We don't care about the details, just the shape of the function.
- Different functions can be transformed to their general shape, e.g.,
  - $f(n) = 4n + 3$: main term is n (linear) $\Rightarrow$ f is in $O(n)$
  - $f(n) = 2(n^2) + n$: main term is n^2 (quadratic) $\Rightarrow$ f is in $O(n^2)$
  - $f(n) = 2^n + \log(n^2)$: main term is 2^n (exponential) $\Rightarrow$ f is in $O(2^n)$

Some functions to remember

1	constant	access an array
$\log(n)$	log(-arithmic)	binary search
n	linear	traverse a list
n^2	quadratic	bubble sort
2^n	exponential	?

Combining complexity

- There are some simple rules to combine complexity of functions in big Oh:
  - ignore constants
  - ignore slower-growing terms
  - nested loops and methods *multiply*
  - non-nested loops and methods *add*
  - $O(1) << O(\log(n)) << O(n) << O(n^2) << O(a^n)$

Combining functions

- When you add functions, the order of the sum is the same as the order of the larger addend
  - e.g., $O(n^a) + O(n^b)$ is $O(n^a)$ if $a \geq b$
  - e.g., $O(n^2) + O(n \log n)$ is $O(n^2)$
- When you multiply functions, the order of the product is the product of the orders
  - e.g., $O(n^2) \times O(n)$ is $O(n^3)$

Combining complexity examples

- $O(n) + O(n) = ?$
- $O(3n) + O(n^2) = ?$
- $O(\log(n) + n) = ?$
- $O(x^4 + 3x^2 + \log(x)) = ?$
- $O(n!) = ?$

- $O(n)$
- $O(n^2)$
- $O(n)$
- $O(x^4)$
- $O(n!)$

Some functions we use a lot in big-Oh analysis

- The growth of functions, in **ascending order**, as n (i.e., the size of the data) becomes large:
  - O (1) – the growth is bounded
  - O (log n) – it grows, but slower and slower as n increases
  - O (n) – linear growth
  - O (n log n) – very common in tree-like data structures, grows faster than linear but not as fast as quadratic
  - O (n^2) – quadratic growth
  - O (n^3)
  - O (2^n)

A sublety

- The exact definition used by mathematics for big-Oh, has an unexpected consequence
- f is in O(g) means that f grows *no faster* than a constant multiple of g
  - but this includes the case where f actually grows slower than g
- So in mathematics, the following statements are TRUE
  - $3n + 2$ is in $O(n^2)$
  - $4 n \log n + 5n$ is in $O(n^2)$
  - $3 \log n + 7$ is in $O(n)$

- Usually, our focus is on finding the smallest simple function g for a given f

Summary - Comparing Algorithms

- There are many dimensions along which we might compare. One of measures we often care about is *running time*.
- Running time depends on input. So, we decide that what really matters is running time as a function of *input size*.
- This can be hard to characterize. So, we may decide that what really matters is *worst-case* running time as a function of input size.
- One algorithm might be better on small inputs and the other on large inputs. So, we decide that what really matters is worst-case running time as a function of input size for *large inputs*.
- This can still be quite hard to determine precisely. So, we decide that what really matters is worst-case running time as a function of input size for large inputs, *ignoring constant factors*.

Acknowledgement - this excellent overview was taken from : http://www.cs.duke.edu/education/courses/cps130/fall98/lectures/lect02/ Comparing Algorithms

Key skills

- Know relative growth of simple functions, e.g.,
  - n^2 grows faster than n
  - n grows faster than $\log n$
- Find dominant term in complicated function
  - e.g., $3n^2 + n \log n + 2$ is $O(n^2)$
- Find order of growth for code
  - based on structure of the code
    - where are the loops?
    - where are method calls?

General guidelines for Big-O calculations on code

- The following steps are *guidelines* and should not be applied blindly:
  - Calculate the cost of the parts
    - Don't forget: a part can be a method call with its own cost!
  - For successive / iterated parts, *add* together
  - For nested parts, *multiply* together
  - Ignore constant factors and lower order terms

Examples of O(1)

Growth is bounded (constant, not dependant on n)

```
public Dog fight(Dog enemy) {
    System.out.println("Dog fight !!!");
    this.setHungry();
    enemy.setHungry();
    if (length > enemy.getLength()) {
       System.out.print("The enemy runs away ");
       System.out.println("as " + name + " triumphs!");
       return this;
    }
    System.out.println("Uh-oh " + name + " runs away");
    return enemy;
}
```

```
public void swap(int i, int j, int[] numbers) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
```

Examples of O(n)

Growth is linear (eg each element is visited once, in the worst case)

```
public void checkKennel() {
  for (int i = 0; i < dogs.length); i++)
    System.out.println(dogs[i].getName());
}
```

```
public int findMax(int[] numbers) throws Exception {
  int max;
  if (numbers.length == 0)
    throw new Exception("Collection must have at least one element");
  max = numbers[0];
  for (int i = 1; i < numbers.length; i++)
    if (numbers[i] > max) max = numbers[i];
      return max;
}
```

Example of $O(n^2)$

Growth is quadratic (eg for each element visited, each element is visited again, in the worst case)

Use *n* to denote the length of the array passed as parameter

n

worst-case *n*

```
public void sort(int[] numbers) {
    for (int i = 0; i < numbers.length; i++) {
        for (int j = i; j < numbers.length; j++) {
            if (numbers[i] > numbers[j]) swap(i, j, numbers);
        }
    }
}
```

1

Textbook references

– Sections 4.1, 4.2, 4.3