COMP 2140 - Data Structures

Shahin Kamali

Topic 1 - Introductions
University of Manitoba

Picture is from the cover of the textbook CLRS.
Introduction
In a Glance . . .

- Data structures are
  - building blocks for designing algorithms
  - practical and diverse
  - fun (really!)

- Let’s ‘learn & play’ data structures & algorithms and enjoy
Shahin Kamali

- B.Sc. in Computer Science (2002-2006)
  - University of Tehran (Iran)
- M.Sc. in Computer Science (2006-2008)
  - Concordia University (Canada)
- PhD in Computer Science (2008-2014)
  - University of Waterloo (Canada)
  - LIAFA, Paris (France)
- Postdoctoral Fellow, Associate (2015-2017)
  - MIT (USA)
- Assistant professor (since 2017)
  - University of Manitoba
Formalities
Logistics

- Lecture: Mondays, Wednesdays, and Fridays, 9:30-10:20am
  EITC E2 **Rm: 110** (Jan. 07, 2018 - Apr. 09, 2018)

- Webpage:
  http://www.cs.umanitoba.ca/~kamalis/winter19/comp2140.html

- Piazza: piazza.com/umanitoba.ca/winter2019/comp2140

- Office hours: 1:00 to 2:00 pm on Mondays, 10:30 to 11:30 am
  Tuesdays, Location: E2 586 or by appointment

  - You can post your questions (if you prefer anonymously) on Piazza
    so that all your classmates see the solution
  - You should have received an email with respect to Piazza
Textbook

- All material (including the slides) will be posted online
- Slides are the main source for assignments and exams.
- The following free book can be used as a reference:
  - Open Data Structures (in Java), Pat Morin
Programming

- For programming, you use Java
  - You can use any operating system
- Post your questions on Piazza
  - Students who actively respond to questions on Piazza will get bonus marks.
- Help center: located in E2-422A
- Email precise and specific questions to the instructor
Grading

There will be:

- Five assignments (20 percent of the final mark)
- Two quizzes (10 percent)
- Labs (10 percent)
- A midterm exam (20 percent)
- A final exam (40 percent)

Theorem

The focus of this course is on learning, practising, and discovering.

Corollary

Having fun in the process is important.
Grading (cntd.)

- Five assignments:
  - 20 percent of the final mark
  - 5 to 10 percent extra for bonus questions.
  - Submit only pdf files

- Quizzes, Midterm & Final exams:
  - Respectively 10, 20 & 40 percent of the final mark.
  - There will be extra for bonus questions in midterm and final.
  - All are closed-book.
  - Sample exams will be provided for practice for the final.

- Labs:
  - 10 percent of the final mark
  - There will be a lab roughly in each two weeks.
  - Two motivated graduate students will help you.
  - You will implement data structures and algorithms reviewed in the class using Java.
Important Dates (tentative)

Important Dates.

January 7 first class
January 23 assignment 1 due
January 30 quiz
February 8 assignment 2 due
February 18 assignment 3 due
February 19–22 midterm break - no class
March 2 midterm exam

March 14 Pi (π or pie) day
March 20 voluntary withdrawal deadline
March 22 assignment 4 due
March 29 quiz
April 5 assignment 5 due
April 9 last class
April 9–23 exam period

More information on ROASS.
Prerequisites

What I have learned from previous courses?

- Object Oriented Programming
- Java syntax
- Basic sorting (bubble-sort, insertion-sort, selection-sort)
- Recursion (base-case, recursive-case)
- Basic data structures (arrays, linked-lists)
- Basic analysis of algorithms (e.g., big O notation)
Introduction to Algorithms & Data Structures
What is an algorithm?

**Definition**

An algorithm is a computational procedure formed by a sequence of instructions (steps) to solve a problem.

- The problem has an **input** and requires an **output**.
- Solving the problem requires the algorithm to **terminate**.
Angle bisection

Example

**input:** an angle defined by three points \( \angle aob \).

**output:** a point c such that \( \angle aoc = \angle boc \).

**Algorithms:**

1. Draw a circle \( C \) centred at \( o \).
2. Label the intersection of \( C \) and \( ao \) as \( a' \).
3. Label the intersection of \( C \) and \( bo \) as \( b' \).
4. Draw circles \( Ca \) and \( Cb \) centred at \( a' \) and \( b' \) that pass through \( o \).
5. Label the intersection of \( Ca \) and \( Cb \) as \( c \).
6. Return \( c \).
Algorithm definition

Why that was an algorithm for angle bisection?

- It has an input (the angle) and an output (the point $C$)
- It terminates (there is no loop in this algorithm)
- It is correct: one can prove that the reported point is always on the bisection.
  - This step often require theoretical analysis
Algorithm Description

- How to describe an algorithm?

**Example**

**input:** a set of integers  
**output:** the largest number in the set

- An algorithm can be described in a number of ways:
  - English prose
  - pseudo-code
  - actual source code in a programming language (e.g., Java or C++)
  - pseudo-code
  - assembler language
  - machine language (executable)
Finding the Maximum: English Prose

Example

input: a set of integers
output: the largest number in the set

- Initialize a variable $maxMark$ to the value of the first element in the list.
- For each subsequent element in the list, if that element is larger than $maxMark$, then replace $maxMark$ with the value of that element.
- After examining all elements, return $maxMark$. 
## Finding the Maximum: Pseudocode

A more precise description of an algorithm (than English prose) which is closer to an actual implementation.

### Example

**input:** a set of integers marksList  
**output:** the largest number in the set

```plaintext
n ← |marksList|  
maxMark ← marksList[0]

for i = 0 to n-1
    if marksList[i] > maxMark then
        maxMark ← marksList[i]

return maxMark
```
Finding the Maximum: Java Code

The actual source code using a programming language.

Example

**input:** a set of integers marksList

**output:** the largest number in the set

```java
int findHighestMark(int [] marksList) {
    int n = marksList.length;
    int maxMark = marksList[0];

    for (int i = 0 ; i < n ; i++) {
        if (marksList[i] > maxMark)
            maxMark = marksList[i];
    }

    return maxMark;
}
```
Finding the Maximum: Assembly

- Each CPU has a low-level set of instructions to perform basic operations on data stored in registers and in memory.

**Example**

**input:** a set of integers marksList  
**output:** the largest number in the set

```
MOVE Index, Temp
ADD 10, Temp
PUSH Max
STORE Temp, Index
HALT
```

Index 0  
Max 0  
Temp 0  
Array 12  
50  
33
Finding the Maximum: Machine language

- A binary sequence that is specific to the CPU for which it was compiled.

**Example**

**input:** a set of integers marksList  
**output:** the largest number in the set

```
11010111 11010001 01110010 10100011
01010110 11011000 10010010 00111101
11010111 01010101 00110101 10101010
01011101 11001011 00101011 11110010
```

""
Types of Problems We Will Solve Using Algorithms

Example

**input:** a list of integers/real-values  
**output:** the same list in sorted order

Example

**input:** a database of records and a query key  
**output:** determine whether the key exists is the database

Example

**input:** the address of two houses in Winnipeg  
**output:** the shortest route between these two houses
Types of Algorithms We Will Study

We will examine algorithms for:

- sorting a list
- inserting/deleting elements into/from a database
- querying a database
- merging databases
- scheduling prioritized events
- updating the priority of an event
- finding shortest routes

We will consider multiple algorithms for solving each of these problems, and compare these various algorithms against each other.
Comparing Algorithms

- Alice and Bob implemented two algorithms for the same problem X.
  - **Alice:** When I input an array of 200 integers, my algorithm takes 910 milliseconds on a 3.3 GHz quadcore Intel i7 with 8Gb of RAM running Windows 7. It’s only 80 lines of C code. I optimized the compilation but the executable doesn’t run on anyone else’s computer except mine.
  - **Bob:** When I input an array of 2000 floating point numbers, my algorithm 12.59 seconds on a 2.8 GHz AMD Athlon with 512Mb of RAM running Linux. It’s only 122 lines of Java code. I used the default compiler settings and it can be interpreted by any version of Java 2.0 or higher.

Note they used different inputs, different hardware, different operating systems, different programming language, and different compiler optimization.

- Which algorithm is better?
Comparing Algorithms

If two algorithms A and B both solve problem P, what makes one algorithm better than the other?

possible criteria:
  - runtime taken
  - amount of memory storage required
  - number of data movements
  - amount of network traffic generated
  - time required to implement
  - ease of debugging, modifying, and updating
  - cost of hiring specialized programmers (U of M graduates)

Runtime and memory requirements will be of particular interest in this course.

We will discuss tools for comparing algorithms (e.g., big Oh notation).
Determining the Size of the Intersection

Example

**input:** a list of names of students taking COMP 2140 and a list of names of students taking MATH 1500

**assumption:** no two students have the same name

**output:** the number of students taking both courses
Determining the Size of the Intersection: solution I

- Compare every name from list $A$ against every name from list $B$ and count the number of matches.
- Assume list $A$ is stored in an array of length $n$ and list $B$ is stored in an array of length $m$.

```python
common ← 0

for i = 0 to n - 1
    for j = 0 to m - 1
        if A[i] = B[j] then common ← common + 1

return common
```

- access to arrays $A$ or $B$: $f(n, m) =$  ?
Determining the Size of the Intersection: solution II

- Merge the two lists into a single list, sort it, and count the number of adjacent values that match.

  \[
  \text{for } i = 0 \text{ to } n + m - 1
  \]
  
  \[
  \begin{align*}
  &\text{if } i < n \text{ then } C[i] \leftarrow A[i] \\
  &\text{else } C[i] \leftarrow B[i-n]
  \end{align*}
  \]

  \[
  \text{sort}(C) \\
  \text{common} \leftarrow 0
  \]

  \[
  \text{for } i = 0 \text{ to } n + m - 2
  \]
  
  \[
  \begin{align*}
  &\text{if } C[i] = C[i+1] \text{ then } \text{common} \leftarrow \text{common} + 1
  \end{align*}
  \]

- access to arrays A or B: \( g(n, m) = \quad ? \)
Determining the Size of the Intersection: comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>( f(n, m) )</th>
<th>( g(n, m) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 10^3 )</td>
<td>( 10^3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 10^6 )</td>
<td>( 10^6 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- To compare algorithms we often care about larger inputs. Why?
Determining the Size of the Intersection

Other possible solutions:

- **Solution 3**
  - Sort array \( B \).
  - For each \( i = 0 \ldots n - 1 \), search for \( A[i] \) in array \( B \) using binary search. Count the number of matches.

  access to arrays \( A \) or \( B \) (worst case): ?

- **Solution 4**
  - Sort array \( A \).
  - Sort array \( B \).
  - Initialize \( i \) and \( j \) to 0.
  - Check whether \( A[i] = B[j] \); if so, add one to the number of matches, and increment \( i \). If \( A[i] > B[j] \) then increment \( j \). Otherwise, if \( A[i] < B[j] \) then increment \( i \). Repeat until \( i = n \) or \( j = m \).

  access to arrays \( A \) or \( B \) (worst case)?
Comparing Algorithms

Conclusion

There are many ways - algorithms - to solve a given problem. To compare different algorithms, we often measure the time complexity. In doing so, we consider worst-case scenarios and assume large input sizes.

- A good implementation (programming language, compiler, good programming techniques, etc.) can make a code tens, possibly hundreds of times faster.
- A good design, that is a good algorithm/data-structure, can make a solution hundreds of thousands, possibly millions of times faster.
In order to access & manipulate data (e.g., a list of integers, a database of records, a map of the subway) we need a representation of these data that is stored in memory.

**Definition**

A **data structure** is a systematic way of organizing and accessing data.

- In COMP 2140, we analyze different data structures for storing various types of data.
- Depending on the intended application and the algorithms involved, different data structures might be used to store the same data.
Examples of Data Structures

Some of the data structures we will discuss include:

- array
- linked list
- doubly-linked list
- binary search tree
- binary heap
- hash table
- adjacency list
- adjacency matrix
Examples of Data Structures

The following are graphical representations of data structures that store the integers 30, 61, 12, 23, 77, and 52.
Abstract Data Type

What is an Abstract Data Type (ADT)

Definition
An abstract data type is formed by I) a set of values (data items) and II) a set of operations allowed on these items

- Stack is an ADT. Data items can be anything and operations are push and pop
- An ADT is abstract way of looking at data (no implementation is prescribed)
- An ADT is the way data ‘looks’ from the view point of user
ADTs vs Data Structures

- ADTs: stacks, queues, priority queues, dictionaries
- Data structures array, linked-list, binary-search-tree, binary-heap
  hash-table-using-probing, hash-table-using-chaining, adjacency list, adjacency matrix, etc.
Data Abstraction

- In designing algorithmic solutions, we often separate implementation from data and its operations.

**Design Phase**
- design algorithmic solution
- select corresponding ADTs
- analyze memory and running time requirements

**Implementation Phase**
- plan and organize code using the object-oriented programming paradigm: classes, methods, and instance variables
- use good coding style
- testing