CSC 263H Summer 2016

Instructor: Tyrone Strangway

Website: www.cs.toronto.edu/~tyrone/teaching/CSC263_Summer_2016.html

Email: tyrone@cs.toronto.edu
I’m not the instructor

• The instructor will be back next week (he is in Singapore for a conference right now)
  • He is likely asleep (it is 6 AM in Singapore)
The Instructor

Tyrone Strangway (1st Year Ph.D. student with the theory group)

• Previously an undergraduate (took the course with Sam) and M.Sc. Student here.
• Contact: tyrone@cs.toronto.edu
• Office hours: Wed 1-3 PM BA3219 (or by appointment)
The TAs

- Yingzhou Wu
- Jai (That’s me)
- Sasa Milic
- Yulia Rubanova

All very experienced with this material.
Today’s plan

- Administrative stuff (see syllabus for more details)
- What is the goal of this course?
- Why take this course?
- Review of asymptotic notation (this stuff is really important)
- Probability review (for average case runtime analysis).
Administrative Stuff
(See the Syllabus)
Website


- All information (assignments / grades / lectures / ...) can be found here

- Consult it for announcements.
Textbook: “CLRS”

• We will post readings each week.
• Available online at UofT library.
• Second or Third editions are fine.
Lectures

• Weekly from 6-8 PM Wed. (In this room)
• Cover new material and concepts.
Tutorials

• 8-9PM after lecture.
• Starts next week.
• Go to tutorial, we will cover lecture material and problems in more detail.
• Good prep for the assignments and tests.
• Divided by last name (see website for details).
Discussion Board

• We will use Piazza.

• piazza.com/utoronto.ca/summer2016/csc263h1

• For discussion amongst students. The instructor will also be answering questions.

• Don’t use the board to share answers or details of answers.
Office Hours

• TBA
• Why pay a tutor when the instructor is already here?
  • Also have access to the help centre.
• Can also email instructor to schedule alternate meeting time.
Marking Scheme

• 3 Assignments (10% each)
• 1 Midterm (30%) 1.5 hours, held during the F course exam period
• 1 Final (40%)
  • Must get at least 35% to pass course.
Assignments

• Must be PDFs submitted to Markus
• You have late tokens for the course
• You may do so in groups of 2
• You should not collaborate with those outside your group.
• See syllabus for details
• (Some of) Assignment 1 will be out this weekend. Due Week 3 (Check website for exact date).
Tests

• No aids permitted.
• Most of the marks come from these.
• The assignments are only worth 30% in total.
  • This is just enough so you will not ignore them
  • This is not enough so that you will achieve a high mark solely from them
  • They prepare you for the tests (don’t rely on others to do the assignment work).
Feedback

• You can email the instructor with feedback
• We have an anonymous feedback tool on the website.
Prerequisites

Course and GPA requirements are strictly enforced, there will be no exceptions.

- See Calendar for exact details, it is your job to confirm this.
- You may be enrolled now even if you don’t meet the requirements, you will be removed when the university checks if the requirements are met.
What is CSC 263?
Study of two concepts

- Data Structures
- Analysis
Data Structures

Actually two related concepts:

• Abstract Data Types (ADT)
  • What data is stored
  • What operations are provided

• Data Structures
  • How the data is stored
  • How the operations are implemented
Why two concepts?

• **ADTs** model problems we need to solve
  • They provide an interface to build more complex systems
  • Often in software engineering you don’t care how a class is implemented, just that it functions correctly.

• **Data Structures** are how we implement **ADTs**
  • They provide a way to store, access and modify the underlying data
  • The underlying code in a class

**ADTs** can be implemented with different **Data Structures** which can in turn implement different **ADTs**. How you choose to do so is dependent on several factors.
An Example

**ADT:** Stack, a **FILO** waiting list.
- **Push(x):** Insert element $x$ on top of stack.
- **Pop():** Return and remove element on top of stack.

**Data Structure:** Linked List
- **Push(x):** Insert at the head of the list.
- **Pop():** Return and remove element from the head of the list.
Analysis

Using math to:

• Show our programs do what we want them to (Proof of correctness)
• Show our programs don’t require too many resources (Upper Bounds)
• Show our programs require some resources (Lower Bounds)
• Show no method can solve a problem (more general Lower Bounds)

If ADTs are what we want to implement and Data Structures are how we implement them, Analysis is why it works and how well it works.
Why take CSC263?
ADT/Data Structures are useful

• Dictionaries:
  • In CSC443 (Database System Technology) you will see how many modern Database Management Systems implement basic operations with them.

• Priority Queues:
  • In CSC369 (Operating Systems) they are used to make sure important tasks are done immediately.

• Graphs:
  • Widely used in networks for routing CSC458(Networks) and economic modeling CSC200(Social Networks).
Analysis and Algorithms are (also) useful

- Data structures wouldn’t be that useful if we couldn’t show that they work.
Job Interviews

Taken from a software engineering interview:

Given an unsorted list $X$ of numbers (say positive integers), identify and return a list of all numbers that appear multiple times.

Say $X = [3, 2, 1, 4, 9, 3, 1]$
A non CSC263 attempt

Compare each pair of numbers in $X$, whenever you see a pair contain the same number return that number.

$$X = [3, 2, 1, 4, 9, 3, 1]$$

Compares 3 to 2 and to 1 and to 4 and to 9...
Then compares 2 to 1 and to 3 and to 9...
...

• It works.
• It is not very efficient, it makes around $|X|^2$ comparisons.
  • For large lists this can be really slow.
A more elegant solution

Using **Heap Sort (week 2)** sort the list in place. After compare all pairs of numbers that appear next to each other.

\[ X = [3, 2, 1, 4, 9, 3, 1] \]

Do some Heap Sort magic ...

\[ X = [1, 1, 2, 3, 3, 4, 9] \]

Compare \((1, 1), (1, 2), (2, 3), (3, 3), (3, 4), (4, 9)\).

• Much more efficient:
  • Final step makes \(|X| − 1\) comparisons
  • In week two we will see heap sort makes roughly \(|X|\log(|X|)\) comparisons.
  • This is much more efficient than comparing all pairs.
A possibly better solution

Using a **Hash Table (we will see this later)** we can map the integers in an online fashion to several buckets. When we detect a collision we can quickly see if we have a duplicate number.

- Don’t worry about the details we will revisit this later.
- Can sometimes be better can sometimes be worse than sorting, but in **expectation (under certain assumptions)** it is better.
  - Works really well if **RAM** is limited.
Review of Asymptotic Notations
What you should know.

You should be comfortable with:

• Induction
• Recursion (and the Master Theorem)
• Asymptotic notation
• Calculating Probabilities and Expectation
• Random variables and probability distributions
Time Complexity

The amount of **time** required to perform the task as a function of the **input size**.

- **Time:**
  - since this is theory we don’t actually use a clock
  - Often we use a proxy such as the number of arithmetic operations performed or the number of comparisons made.

- **Input Size:**
  - The physical size of the input
  - For lists, the number of elements
  - For trees / graphs, the number of nodes and edges
  - For an integer $n$ the input size is $\log(n)$ (why?)
A Note

We can count how many times all lines are executed (see the text for this analysis) if we know some lines are more expensive we can assign a higher cost to them.
Running Time

• $t_A(x)$ is the running time of algorithm $A$ on input $x$.
• For the rest of these slides we let $A$ be insertion sort and omit it.
• The worst case running time $T(n)$ is:
  \[ T_{\text{max}}(n) = \max_x \{t(x) | x \text{ is an input of size } n\} \]
• The best case running time is:
  \[ T_{\text{min}}(n) = \min_x \{t(x) | x \text{ is an input of size } n\} \]
Running Time

We rarely care about the **best case running time**, instead we will usually focus on the **worst case running time**.
Average Case Running Time

We can similarly define the **average case running time** to be the expected running time over all inputs. We will do this later in the course.
Asymptotic Notation

You should have seen these in CSC165 and CSC236

• Big-Oh (the upper-bound)
• Big-Omega (the lower-bound)
• Big-Theta (the tight-bound)
Asymptotic Notation

$O(f)$: the set of functions that grow no faster than $f$.
- $g \in O(f)$ if $f(n)$ grows at least as fast as $g(n)$ with respect to $n$.
- $\exists n_0, c > 0$ where $\forall n \geq n_0 \ g(n) \leq c \cdot f(n)$
- $f(n)$ upper bounds $g(n)$

$\Omega(f)$: the set of functions that grow no slower than $f$.
- $g \in \Omega(f)$ if $g(n)$ grows at least as fast as $f(n)$ with respect to $n$.
- $\exists n_0, c > 0$ where $\forall n \geq n_0 \ g(n) \geq c \cdot f(n)$
- $f(n)$ lower bounds $g(n)$
Asymptotic Notation, the details

• Only the rate of growth matters, we can ignore constant factors.
  • $100 \, n^3$ and $3000 \, n^3$ both grow at the same rate. They are both $O(n^3)$.

• We can ignore low order terms.
  • $n^3 + 232n^2 - 4000000\log(n)$ is eventually dominated by the high order term.
  • This is also $O(n^3)$.
The most important takeaway (for now)

• Big-Oh **does not** describe the worst-case running time
• Big-Omega **does not** describe the best-case running time

• Both $O$ (and $\Omega$) can be used to upper-bound (and lower-bound) the best-case and worst-case running times (or any function of $n$).
Example: Insertion Sort

**Insertion-Sort**(A)

1. for \( j = 2 \) to \( A.length \)
2. \( \text{key} = A[j] \)
3. // Insert \( A[j] \) into the sorted sequence \( A[1...j-1] \).
4. \( i = j - 1 \)
5. while \( i > 0 \) and \( A[i] > \text{key} \)
7. \( i = i - 1 \)
8. \( A[i+1] = \text{key} \)
Insertion Sort Running Time

• Each line can be done in constant time (can’t always assume this)
• For each iteration of the outer loop the inner will make at least one comparison.
  • So it will be sufficient to count the number of times the inner loop executes
  • This is equivalent to asking how many comparisons are made
How to show $T_{max}(n)$ is $O(f(n))$

Recall, $T_{max}(n) = \max_x \{ t(x) \mid x \text{ is an input of size } n \}$

Formally, we need constants $n_0, c > 0$ where:

For all $n \geq n_0$, $T_{max}(n) \leq c \cdot f(n)$

For every, input $x$ of size $n$, the algorithms running time on input $x$ is no larger than $c \cdot f(n)$.

There is no input for which the algorithm is slower than $c \cdot f(n)$. 
Insertion Sort running time (upper bound)

**Worst-Case upper-bound:**
- The outer loop goes over all $n$ elements.
- For each iteration of the outer loop we may compare that element at most once to all previously sorted elements (at most $n$ comparisons).
- Thus we get at most $n^2$ comparisons, and since each comparison takes $O(1)$ time, we get $T_{\text{max}}(n)$ is $O(n^2)$
How to show $T_{max}(n)$ is $\Omega(f(n))$

Recall, $T_{max}(n) = \max_x \{t(x) \mid x \text{ is an input of size } n\}$

Formally, we need constants $n_0, c > 0$ where:

For all $n \geq n_0$, $T_{max}(n) \geq c \cdot f(n)$

There exists, input $x$ of size $n$, the algorithms running time on input $x$ is no smaller than $c \cdot f(n)$.

There is an input that causes the algorithm to run no faster than $c \cdot f(n)$. 
Insertion Sort running time (lower bound)

**Worst-Case lower-bound**

- Consider a list in reverse sorted order.
- The first element gets compared 0 times, the second 1 time, the third 2 times,..., the $i^{th}$ $i-1$ times,...
- So we get $1 + 2 + 3 + ... + (n - 1)$ comparisons
- This sum is equal to $\frac{n(n-1)}{2}$.
- Thus we get at least $0.5 \cdot n^2 - 0.5 \cdot n$ comparisons, that is $T_{max}(n)$ is $\Omega(n^2)$
Big-Theta

If a function is both $\Omega(f(n))$ and $O(f(n))$ it has both an upper-bound and lower-bound of $f(n)$. It is essentially entirely described by $f(n)$, we call such a situation Big-Theta.

If $g(n) \in O(f(n))$ and $g(n) \in \Omega(f(n))$ we say $g(n) \in \Theta(f(n))$

We showed for insertion sort $T_{max}(n) \in \Theta(n^2)$
How to show $T_{min}(n)$ is $O(f(n))$

Recall, $T_{min}(n) = \min_{x} \{ t(x) \mid x \text{ is an input of size } n \}$

Formally, we need constants $n_0, c > 0$ where:

For all $n \geq n_0$, $T_{min}(n) \leq c \cdot f(n)$

There exists no larger, input $x$ of size $n$, the algorithms running time on input $x$ is no larger than $c \cdot f(n)$.

There is an input for which the algorithm is at least as fast as $c \cdot f(n)$. 
Insertion Sort Best Case Running Time

• **Best-Case Upper-Bound:**
  • Consider a sorted list
  • The inner loop does one comparisons per element since each element is in place.
  • Thus we only do $n$ comparisons, $T_{min}(n) \in O(n)$
How to show $T_{\text{min}}(n)$ is $\Omega(f(n))$

Recall, $T_{\text{min}}(n) = \min_{x} \{t(x) | x \text{ is an input of size } n\}$

Formally, we need constants $n_0, c > 0$ where:

For all $n \geq n_0$, $T_{\text{min}}(n) \geq c \cdot f(n)$

For all, inputs $x$ of size $n$, the algorithms running time on input $x$ is no smaller than $c \cdot f(n)$.

There is no input that causes the algorithm to run faster than $c \cdot f(n)$. 
Insertion Sort Best Case Running Time

- **Best-Case Lower-Bound:**
  - The outer loop looks at all $n$ elements and must make at least one comparison in the inner loop.
  - Thus we always do at least $n$ comparisons, $T_{\text{min}}(n) \in \Omega(n)$
  - Thus $T_{\text{min}}(n) \in \Theta(n)$
Why Study Insertion Sort

• We will see many asymptotically (in the worst case) superior sorting algorithms.
• Insertion sort is conceptually simple and lends itself to easy analysis.
• The behavior in the best case is near optimal.
  • No sorting in any case can do better than linear time.
• Because of the good best case performance many sorting algorithms use insertion sort when the input is nearly sorted.
A quick review on Run-Times

From fastest to slowest run time:
- $O(1)$ (constant time)
- $O(\log(n))$
- $O(n^{1/2})$
- $O(n)$
- $O(n \cdot \log(n))$
- $O(n^2)$
- $O(n^3)$
- $O(n^c), c > 3$
- $O(2^n)$ (at $n = 300$ more than the number of atoms in the universe)
- $O(n^n)$
A Warning about Asymptotic notation

• Because we ignore high order terms and constants information about the running time is lost.
  • While it is true that eventually high order terms dominate this may require a very large input size, larger than you would ever see in practice.

• We are ignoring hardware and software details.
  • Real computers need to deal with issues like caching (CSC369), arithmetic errors (CSC336) and conserving page loads (CSC443).
  • The concepts here serve as building blocks, what you learn here helps you understand the material in more advanced courses.
Review of Basic Probability Concepts
Sample Space and events

- **Sample space** is the set of all possible outcomes of a (random) Experiment.
- An **Event** is a set of outcomes of the experiment (a subset of sample space)
- A **Probability Rule** assigns a probability to each event.
Independent Events, Mutually Exclusive Events and Conditional Probability

• In general, \( \Pr(A \cap B) = \Pr(A|B) \Pr(B) \) where \( \Pr(A|B) \) is the **conditional probability** that event \( A \) occurs given \( B \) occurs.

• If \( A, B \) are **independent** events, \( \Pr(A \cap B) = \Pr(A)(B) \)

• In general, \( \Pr(A \cup B) = \Pr(A) + \Pr(B) - \Pr(A \cap B) \)

• If \( A, B \) are **mutually exclusive** events, \( \Pr(A \cup B) = \Pr(A) + \Pr(B) \)
Examples

• Consider a deck of 52 cards and the experiment of picking a card (uniformly) at random from the deck.
  • $\Pr(\text{Drawing a number}) = ?$
  • $\Pr(\text{Drawing a Hearts | Drawn Red}) = ?$
  • $\Pr(\text{Drawing Red or a 7}) = ?$

• Consider the experiment of drawing two cards at random (without replacement) from a deck of 52 cards
  • Probability you get a spade and a heart?

• Consider the experiment of drawing 10 cards at random (without replacement) from a deck of 52 cards
  • Probability you get 6 spades, 2 hearts and 2 diamonds?
Distribution and Random Variables

• We only worry about discrete distributions with a finite sample space.
• A **Probability Distribution** assigns a probability to each possible outcome (characterized by a **Probability Mass Function**).
  • We will mostly consider the **Uniform Distribution**.
• A **Random Variable** is a variable whose possible values are numerical values corresponding to different outcomes of a random experiment.
• For the problem of sorting n values (assume distinct),
  • What is the sample space of inputs?
Expectation

- Expectation of a discrete random variable is a probability-weighted average of all possible values (that the random variable can take).
  - If a random variable \( X \) takes values \( \{x_1, \ldots, x_n\} \) with probabilities \( \{p_1, \ldots, p_n\} \) respectively, then \( E(X) = \sum x_i p_i \)
  - Consider a deck of 52 cards and the experiment of picking a card (uniformly) at random from the deck.
    - You win $1 if you draw a number, lose $2 if you draw a face.
    - What is your expected winnings?

- Linearity of expectation.
  - If \( X_1, X_2, \ldots, X_n \) are discrete random variables (not necessarily independent), then \( E(X_1 + X_2 + \ldots + X_n) = E(X_1) + E(X_2) + \ldots + E(X_n) \)