

## Teaching Portfolio

<b>1</b>	<b>Biographical Sketch</b>	<b>2</b>
<b>2</b>	<b>Teaching Philosophy</b>	<b>2</b>
<b>3</b>	<b>Teaching Responsibilities</b>	<b>4</b>
3.1	Course Instruction . . . . .	4
3.2	Instructional Apprenticeships . . . . .	4
3.3	Teaching Assistantships . . . . .	5
3.4	Curriculum Development . . . . .	5
3.5	Relevant Committee Service . . . . .	5
<b>4</b>	<b>Evidence of Teaching Effectiveness</b>	<b>6</b>
4.1	Summary . . . . .	6
4.2	Honours and Awards . . . . .	6
4.3	Representative Feedback from Students . . . . .	6
4.4	Representative Feedback from Colleagues . . . . .	7
<b>5</b>	<b>Professional Development</b>	<b>7</b>
5.1	Professional Development in University Teaching and Learning Program, Queen's University	7
5.2	Fundamentals of University Teaching Program, University of Waterloo . . . . .	7
<b>A</b>	<b>Course Descriptions</b>	<b>9</b>
<b>B</b>	<b>Course Evaluations (Past 12 Months)</b>	<b>11</b>
<b>C</b>	<b>Recent Course Outline</b>	<b>17</b>
<b>D</b>	<b>Sample Course Materials</b>	<b>20</b>
<b>E</b>	<b>Sample Scholarship Materials</b>	<b>29</b>

## 1 Biographical Sketch

I am an assistant professor in the Department of Computer Science at St. Francis Xavier University. Generally, I teach and develop courses in theoretical computer science covering subjects such as formal languages, automata theory, computability and complexity theory, formal logic, and algorithm analysis/design. I have also taught related subjects, such as discrete mathematics and matrix computation.

As the instructor of undergraduate and graduate courses in each of the above topics, I have had opportunities not only to impart my own style onto regular course offerings, but also to completely revise and refresh infrequently-offered courses and, in some cases, to develop entirely new courses from the ground up. I have produced hundreds of pages of open-access curriculum materials covering the spectrum of theoretical computer science, and my materials are used at multiple universities. In addition to the standard curriculum, I always welcome opportunities to guide independent study/reading courses for motivated students.

Throughout my career, my teaching abilities have been recognized both by students and by the broader university community. My courses are regularly at or near registration capacity, which is a trend that runs counter to typical levels of enrolment in theoretical courses. I have received teaching evaluation scores consistently higher than both the department and university averages, and I have earned multiple teaching awards, including the university-wide Queen's Teaching Assistant/Teaching Fellow Excellence Award.

## 2 Teaching Philosophy

...the best theory is inspired by practice and the best practice is inspired by theory.

— DONALD KNUTH  
*Theory and Practice* (1991)

Teaching theoretical computer science presents a unique challenge: typically, at the level of instruction when most students are introduced to theory, all of their previous courses have been application-focused and, hence, directly applicable to what they expect to see when they enter the workplace. In a theoretical computer science course, then, where the concepts taught cannot be immediately connected to any real-world application, it is not unusual to hear the question “why do we care about this?” Indeed, having taught and assisted with courses that span the spectrum of theoretical computer science—algorithms, computability and complexity, data structures, discrete mathematics, formal languages and automata—this one question has been a universal constant. Students want to know why they are investing their time and energy into learning something that is too often presented in the abstract.

My attempt to guide students toward finding an answer to this question is best summarized by the above quote by Donald Knuth: I use practice to inspire theory and, in turn, I strive to have students' future practices inspired by this theory. As hard as some theorists may work to keep their field “pure”, it is an undeniable truth that theory and practice are inextricably intertwined components of computer science. Rather than resisting this association, I propose that we embrace it: teach students from the very beginning about the deep and meaningful connections between theoretical computer science and other areas, so that they have a better understanding not only of theory, but of the field as a whole. My goal as an educator is to bring meaning to theory, and I make progress toward that goal by drawing three connections: between theory and application, between theory and the bigger picture, and between students and their education.

**Connecting Theory to Application.** One pitfall of testing a student's understanding of theory lies in asking “so what?” questions. These questions are often designed to test rote knowledge of definitions or theorems, and while answering such questions provides an extrinsic motivation for, say, students preparing to write an exam, there is no intrinsic motivation for those same students to care about the solution they obtain. I aim to design problems that require students to apply concepts taught in the classroom to a situation found in the real world. Often, I select applications I have not discussed in the classroom, leaving students to discover the link on their own and solidifying in their minds both the idea itself and how they might use the idea.

In terms of course design, I tend to allocate a higher percentage of marks to assignments and projects rather than to time-constrained assessments such as exams. I believe exams have their merits, but the fact that most exams are held at a point in the term when students no longer have an opportunity to improve their understanding severely limits the use of an exam as a learning tool. Placing a greater weight on assignments and projects encourages students to put more work into crafting a submission they can be proud of while, in the process, gaining a better understanding of how the material they learned can be applied to novel problems.

**Connecting Theory to the Bigger Picture.** When I teach concepts in theoretical computer science, I make an effort to connect the concept to its broader context. I often do so by outlining the history and development of the concept, or by highlighting a few areas of application where this concept may appear. For example, when I introduce students to Greibach normal form, I do not restrict the lesson simply to talking about context-free grammars. Rather, I introduce students to Sheila Greibach, how she developed her idea, and why compiler designers care about transforming grammars into the form bearing her name.

I include this broader context within my lectures not only to pique interest in otherwise dry topics, but also to imbue within students the sense that these topics exist outside of the realm of mathematics. Framing the introduction of a lesson through the lens of the broader arts and sciences—through biography, history, chemistry, or otherwise—both provides for a gentler introduction to an abstract concept and allows students to emerge from a course as more well-rounded academics.

Providing a historical perspective within lectures also comes with the added benefit of introducing students to the people behind the ideas, thus revealing that these ideas were not simply gifted unto us centuries ago by brilliant minds from atop the ivory tower. I feel that this approach encourages students to attempt their own discoveries, as the barrier to entry vanishes once students realize that the people who developed the very ideas being taught today were once just like them. This approach was influenced by texts such as Rosen's *Discrete Mathematics and Its Applications*, which includes biographical sidebars of mathematical figures.

**Connecting Students to Their Education.** When I was an undergraduate student, the theoretical computer science course in which I was enrolled was taught through PowerPoint. While this may work for some subjects, I staunchly believe that PowerPoint (and, indeed, any static presentation format) has no place in theoretical computer science. At its core, theoretical computer science is applied mathematics, and so much like a course in mathematics, instructors should employ a fluid, discovery-based approach to teaching theory.

To promote interest in a course, I invite students to participate in the development and presentation of course material during each lecture, rather than flicking through slides and beaming contextless information at the class. I write lecture material on the chalkboard, giving me the ability to amend and add to my planned notes as students ask questions, extend ideas, and even offer corrections—after all, no instructor is infallible! For the same reason, I engage students in worked examples, solving problems as a group in real time and highlighting potential mistakes as they arise. To accommodate different learning styles, I use props in the classroom. For example, I have shared a collection of different-sided dice prior to a lecture on sample spaces and outcomes—students especially enjoyed rolling the d120—and solicited volunteers to flip coins in order to build a skip list interactively as a class. The guided discovery process facilitated by students handling props establishes a foundation upon which I can build a memorable lecture. My approach incorporates a dimension of flexibility into lectures, allowing me to mould the content to the students rather than the other way around.

I believe my approaches have a positive effect on student learning, as evidenced by both the numeric ratings and the student comments I have received. Such feedback provides reassurance that I am achieving my goal of bringing meaning to theory. That being said, I still seek to improve various aspects of my teaching style. Currently, my priority is to identify areas where I can increase the use of technology in my lectures to facilitate active learning, including using software tools to illustrate concepts (e.g., constructing automata with JFLAP) or leading live-coding sessions during algorithm-focused lectures. In past courses, I have introduced students to the  $\text{\LaTeX}$  document preparation system via an interactive workshop, and this was generally received well. I also plan to incorporate more small group activities and student-led lessons in my lectures, taking advantage of the often-smaller class sizes in theory courses to build a sense of community.

### 3 Teaching Responsibilities

The following lists outline all courses for which I had some kind of teaching responsibility, including as an instructor, as an instructional apprentice (IA) or teaching assistant (TA), or as a curriculum developer. Appendix A contains course descriptions for all courses listed here. Appendices C and D contain examples of materials from a recent course I have taught.

All courses from Fall 2021 to present were held at St. Francis Xavier University.

All courses from Fall 2017 to Winter 2021 were held at Queen's University.

All courses from Fall 2015 to Spring 2017 were held at the University of Waterloo.

#### 3.1 Course Instruction

Primary responsibilities include planning and delivering lectures, designing in-class activities, writing lecture notes, creating assignments and exams, coordinating teaching assistants, holding office hours, and handling administrative matters.

<i>Fall 2022</i>	CSCI 356: Theory of Computing 7 students, 1 instructor, 3 lecture hours per week
<i>Fall 2022</i>	CSCI 541: Theory of Computing 34 students, 1 instructor, 1 TA, 3 lecture hours per week
<i>Fall 2022</i>	CSCI 550: Approximation Algorithms 32 students, 1 instructor, 1 TA, 3 lecture hours per week
<i>Winter 2022</i>	CSCI 355: Algorithm Design and Analysis 22 students, 1 instructor, 1 TA, 3 lecture hours per week
<i>Winter 2022</i>	CSCI 544: Computational Logic 30 students, 1 instructor, 1 TA, 3 lecture hours per week.
<i>Winter 2022</i>	CSCI 554: Matrix Computation 27 students, 1 instructor, 1 TA, 3 lecture hours per week
<i>Fall 2021</i>	CSCI 356: Theory of Computing 35 students, 1 instructor, 1 TA, 3 lecture hours per week
<i>Fall 2021</i>	CSCI 541: Theory of Computing 18 students, 1 instructor, 1 TA, 3 lecture hours per week
<i>Fall 2021</i>	CSCI 550: Approximation Algorithms (online) 22 students, 1 instructor, 1 TA, 3 synchronous meeting hours per week
<i>Winter 2019</i>	CISC 203: Discrete Mathematics for Computing II 49 students, 1 instructor, 2 TAs, 3 lecture hours per week
<i>Spring 2017</i>	CS 240: Data Structures and Data Management 340 students, 3 instructors, 1 IA, 7 TAs, 3 lecture hours per week

#### 3.2 Instructional Apprenticeships

At the University of Waterloo, the role of instructional apprentice recognizes the advanced skills of some graduate students. IAs are selected by instructors on the basis of TA excellence. Primary responsibilities include leading tutorials, coordinating teaching assistants, and creating solution sets and marking schemes.

<i>Fall 2016</i>	CS 234: Data Types and Structures
<i>Spring 2016</i>	CS 240: Data Structures and Data Management 121 students, 2 sections, 1 tutorial hour per week per section

### 3.3 Teaching Assistantships

Primary responsibilities include holding office hours, marking assignments and exams, and proctoring assessments. In recent terms, responsibilities have also included coordinating junior teaching assistants, creating solution sets and marking schemes, and adapting content for an online environment.

<i>Winter 2021</i>	CISC/CMPE 223: Software Specifications (online)
<i>Fall 2020</i>	CISC 203: Discrete Mathematics for Computing II (online)
<i>Winter 2020</i>	CISC/CMPE 223: Software Specifications (partially online)
<i>Fall 2019</i>	CISC 203: Discrete Mathematics for Computing II
<i>Fall 2018</i>	CISC 462: Computability and Complexity
<i>Winter 2018</i>	CISC/CMPE 223: Software Specifications
<i>Fall 2017</i>	CISC 462: Computability and Complexity
<i>Winter 2017</i>	CS 462/662: Formal Languages and Parsing
<i>Winter 2016</i>	CS 240: Data Structures and Data Management
<i>Fall 2015</i>	CS 234: Data Types and Structures

### 3.4 Curriculum Development

<i>Computational Logic</i>	– Wrote ~75 pages of lecture notes, structured with a roughly even split between propositional and predicate logic. Both propositional and predicate logic topics follow the same progression: syntax/semantics, semantic tableaux, deductive systems, and resolution. This intentional mirroring allows students in the second half of the course to build on what they learn in the first half, with a view to future logical extensions.
<i>Discrete Mathematics</i>	– Wrote ~110 pages of lecture notes covering standard undergraduate topics including combinatorics, discrete probability, recurrence relations, graph theory, and trees. My notes contain illustrations, complete proofs, and fully worked examples. The notes continue to be used by faculty at Queen's University in current offerings of the course.
<i>Theory of Computation</i>	– Produced ~110 pages of comprehensive, detailed, and heavily illustrated lecture notes suitable for both undergraduate and graduate audiences. The undergraduate version of the notes discusses standard material on regular/context-free languages, Turing machines, decidability/reducibility, and time/space complexity. The expanded graduate version of the notes leaves elementary proofs as exercises and redirects the focus to advanced material on lesser-known theoretical topics. My notes are currently being revised to include additional graduate material, to connect existing content to applications, and to incorporate activities involving automata-theoretic software tools.
<i>Other Topics</i>	– Developed ~75 pages of lecture notes for a graduate course on approximation algorithms. – Developed ~65 pages of lecture notes with self-contained, commented code examples indexed to the notes for a graduate course on matrix computation.

### 3.5 Relevant Committee Service

<i>2016 – 2017</i>	Graduate Studies Committee, Faculty of Mathematics, University of Waterloo – Primarily tasked with reviewing/approving new courses and changes to graduate programs offered by the Departments of Pure Mathematics, Applied Mathematics, Statistics, Combinatorics and Optimization, and the School of Computer Science.
<i>2014 – 2015</i>	Curriculum Committee, Department of Computer Science, University of Western Ontario – Primarily tasked with undertaking a curriculum mapping process to align undergraduate course learning outcomes with CIPS CSAC graduate attributes.

## 4 Evidence of Teaching Effectiveness

### 4.1 Summary

The following table summarizes my “effectiveness rating” as an instructor for past courses. Department means are provided when available. Additional data from the past 12 months are provided in Appendix B.

Term	Course	Effectiveness Rating	Department Mean
Winter 2022	CSCI 355	4.50/5.00	4.30/5.00
Winter 2022	CSCI 544	4.83/5.00	4.30/5.00
Winter 2022	CSCI 554	4.87/5.00	4.30/5.00
Fall 2021	CSCI 356	4.38/5.00	4.15/5.00
Fall 2021	CSCI 541	5.00/5.00	4.15/5.00
Fall 2021	CSCI 550	4.40/5.00	4.15/5.00
Winter 2019	CISC 203	4.7/5.0	3.7/5.0
Spring 2017	CS 240	4.3/5.0	—

### 4.2 Honours and Awards

- 2020** TA/TF Excellence Award, Queen’s Society of Graduate & Professional Students (SGPS)
- University-level award given to recognize the outstanding contributions of a teaching assistant/teaching fellow to the SGPS and Queen’s community. One recipient per year.
  - First student from the School of Computing to receive this award.
- 2018** Excellence in Teaching Assistance Award, School of Computing, Queen’s University
- Department-level award given to a teaching assistant in a computing course who went above and beyond in their duties. One recipient per year.

### 4.3 Representative Feedback from Students

I would like to thank Dr. Smith for showing great motivation and dedication to teaching. From the lecture and the lecture notes, you can clearly tell that he is going over and beyond and spending a lot of outside time so that we can have a good term. Seeing instructors like Dr. Smith put this effort increases the students learning appetite. Thank you Dr. Smith, you made my last term in StFX truly useful.

*(Graduate student, Winter 2022)*

[Prof. Smith] is highly knowledgeable in the subject area. He is clearly passionate about this subject and it shows in his lectures. He put in a lot of hard work to prepare notes for our use. [...] I usually don’t find theory classes interesting but ended up enjoying this one and I feel the teachers enthusiasm for the subject helped a lot.

*(Third year undergraduate students, Fall 2021)*

My name is [...]. I was a Discrete Mathematics II student of yours this past semester. Your enthusiasm for the subject matter was infectious, and inspired me to consider Post-Graduate research in Pure Mathematics. I am reaching out to thank you, and wish you the best of luck, whatever your future may hold.

*(Email from former undergraduate student, Winter 2019)*

I liked how genuinely interested Prof. Smith was in teaching the course. His enthusiasm makes coming to this class the highlight of my day. His lecture notes both in-person and online are in-depth and I know that if I have to miss a class, the online notes will prepare me for the next assignment.

*(Second year undergraduate student, Winter 2019)*

I was originally skeptical about having a grad student as an instructor, but Taylor ended up being one of the better instructors I've had. It may be a symptom of him sympathizing more with us undergrads than some tenured profs do, but he targeted his explanations at precisely the right level, and actually answered low-level questions when students were having a tough time understanding the content, never blowing off questions with the equivalent of "I shouldn't have to explain that, you should be smart enough to figure that out," which I've seen from several professors before.

*(Second year undergraduate student, Spring 2017)*

#### 4.4 Representative Feedback from Colleagues

Taylor was my choice for head TA for the CISC 223 course. I made this choice when I heard that he has had stellar feedback and standing ovations from students in CISC 203 in Winter 2019. I run my courses by democratic council to avoid power dynamics as well as a redundancy layer for important choices. Taylor has been vital in this process. He has provided great insight for some choices, including midterm and final exam design. He is available for discussion, and I have heard great feedback from students that go to office hours. He also volunteered to do a test run of the midterm; it was a welcome surprise.

*(Course instructor, Winter 2020)*

### 5 Professional Development

#### 5.1 Professional Development in University Teaching and Learning Program, Queen's University

The Queen's University Professional Development in University Teaching and Learning program is intended for students and fellows who have an interest in developing themselves in areas of university teaching and learning. The program consists of five components that are specific to each of the areas of university teaching and learning.

As a participant in this program, I gained an understanding of foundational issues in university teaching and used my teaching experience to articulate my beliefs and attitudes toward these issues. I also became acquainted with the scholarship of teaching and learning within the context of my subject area. Finally, I applied Universal Design for Learning strategies to course material I created in the past to make it more accessible and inclusive.

Appendix E contains a research prospectus and annotated bibliography I developed as part of the "Scholarship in Teaching and Learning" component of this program.

<i>Workshops completed</i>	Foundations in Teaching and Learning
	Practical Experience
	Educational Leadership
	Scholarship in Teaching and Learning
	Accessible Teaching and Learning

#### 5.2 Fundamentals of University Teaching Program, University of Waterloo

The University of Waterloo Fundamentals of University Teaching program supports graduate students in developing their knowledge and skills as university TAs and instructors. Participants attend a minimum of six teaching workshops and lead three small-group microteaching sessions.

As a participant in this program, I gained experience in teaching topics from my subject area to a small non-expert audience, developed techniques for organizing and structuring lectures according to pre-established learning objectives, and experimented with a variety of active learning and student-centred teaching methods. I also completed two self-directed modules on the theory of online learning, where I developed strategies for facilitating online learning effectively in both fully-online and hybrid models.

<i>Workshops</i>	Effective Lesson Plans
<i>completed</i>	Giving and Receiving Feedback
	Teaching Online – Basic Skills
	Teaching Online – Advanced Skills
	Classroom Delivery Skills
	Teaching Methods
	Teaching STEM Tutorials



## A Course Descriptions

### St. Francis Xavier University

#### **CSCI 355: Algorithm Design and Analysis**

The development of provably-correct algorithms to solve problems and their analyses. Topics include basic algorithm design techniques such as greedy, divide- and-conquer, and dynamic programming, and network flows. Intractability and NP-completeness.

#### **CSCI 356: Theory of Computing**

An introduction to the theoretical foundations of computer science, examining finite automata, context-free grammars, Turing machines, undecidability, and NP-completeness. Abstract models are employed to help categorize problems as undecidable, intractable, tractable, and efficient.

#### **CSCI 541: Theory of Computing**

This course focuses on three areas central to the theory of computation: automata, computability and complexity, to investigate the question: What are the fundamental capabilities and limitations of computers? We study automata (models of computation) e.g., finite state machines, pushdown automata and Turing machines and the languages recognized by them. We investigate complexity theory, to classify problems as easy or hard and computability theory to classify problems as solvable or not.

#### **CSCI 544: Computational Logic**

This course focuses on automated theorem proving. We start with a rigorous treatment of propositional and first order calculus (with equality) and the method of natural deduction, giving a thorough investigation of the soundness and completeness proofs and decidability. Then we compare and contrast several automated theorem proving methods such as tableau, resolution, sequent style calculus and rewrite systems. Extensions to other logics will be discussed.

#### **CSCI 550: Approximation Algorithms**

Approximation algorithms are efficient algorithms that are guaranteed to compute solutions such that the value of the solution is provably close to the optimum. This course provides an introduction at the graduate level to the area of approximation algorithms, highlighting key algorithm design techniques for approximation algorithms and the complementary study of hardness of approximation for hard optimization problems.

#### **CSCI 554: Matrix Computation**

Through the use of lectures, discussions, the text, assignments, and labs, this course will familiarize students with the advanced knowledge of triangular systems, positive definite systems, banded systems, sparse positive definite systems, general systems; Sensitivity of linear systems; orthogonal matrices and least squares; singular value decomposition; eigenvalues and eigenvectors; and QR algorithm with their applications.

### Queen's University

#### **CISC 203: Discrete Mathematics for Computing II**

Proof methods. Combinatorics: permutations and combinations, discrete probability, recurrence relations. Graphs and trees. Boolean and abstract algebra.

**CISC/CMPE 223: Software Specifications**

Introduction to techniques for specifying the behaviour of software, with applications of these techniques to design, verification and construction of software. Logic-based techniques such as loop invariants and class invariants. Automata and grammar-based techniques, with applications to scanners, parsers, user-interface dialogs and embedded systems. Computability issues in software specifications.

**CISC 462: Computability and Complexity**

Turing machines and other models of computability such as  $\lambda$ -recursive functions and random-access machines. Undecidability. Recursive and recursively enumerable sets. Church-Turing thesis. Resource-bounded complexity. Complexity comparisons among computational models. Reductions. Complete problems for complexity classes.

**University of Waterloo****CS 234: Data Types and Structures**

Top-down design of data structures. Using representation-independent data types. Introduction to commonly used data types, including lists, sets, mappings, and trees. Selection of data representation.

**CS 240: Data Structures and Data Management**

Introduction to widely used and effective methods of data organization, focusing on data structures, their algorithms, and the performance of these algorithms. Specific topics include priority queues, sorting, dictionaries, data structures for text processing.

**CS 462/662: Formal Languages and Parsing**

Languages and their representations. Grammars –Chomsky hierarchy. Regular sets and sequential machines. Context-free grammars –normal forms, basic properties. Pushdown automata and transducers. Operations on languages. Undecidable problems in language theory. Applications to the design of programming languages and compiler construction.

## B Course Evaluations (Past 12 Months)

### Winter 2022 – CSCI 355: Algorithm Design and Analysis

Response rate: 6 / 22 (27.2%)

**Instructor is effective in organizing/  
presenting course materials:**



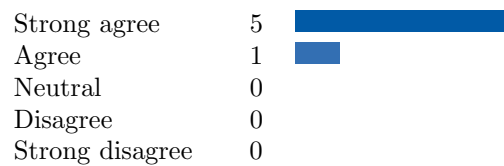
**Instructor shows interest/enthusiasm in  
teaching this course:**



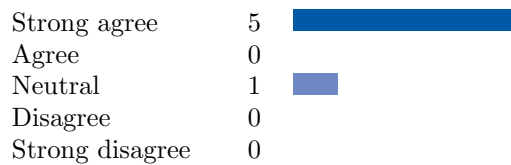
**Instructor stimulated my interest in this  
subject area:**



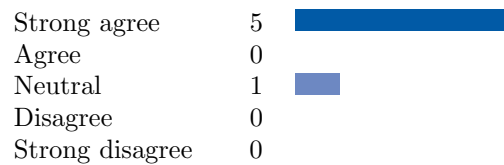
**Instructor makes students feel free to ask  
questions and express ideas:**



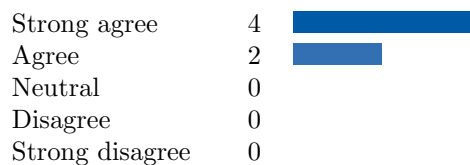
**Instructor's communication skills are appropriate  
for the course:**



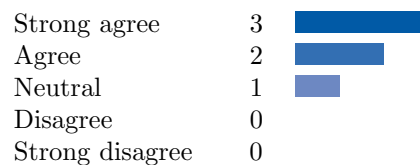
**Instructor is available for consultation outside  
of the class:**



**Course material is well organized:**





**I would recommend this course to fellow students:**





## Winter 2022 – CSCI 544: Computational Logic

Response rate: 18 / 30 (60.0%)



**Instructor is effective in organizing/  
presenting course materials:**

<b>Strong agree</b>	<b>15</b>	
<b>Agree</b>	<b>3</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor shows interest/enthusiasm in  
teaching this course:**

<b>Strong agree</b>	<b>17</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor stimulated my interest in this  
subject area:**

<b>Strong agree</b>	<b>16</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor makes students feel free to ask  
questions and express ideas:**

<b>Strong agree</b>	<b>17</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor's communication skills are appropriate  
for the course:**

<b>Strong agree</b>	<b>17</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor is available for consultation outside  
of the class:**

<b>Strong agree</b>	<b>16</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

**Course material is well organized:**

<b>Strong agree</b>	<b>15</b>	
<b>Agree</b>	<b>3</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**I would recommend this course to fellow students:**

<b>Strong agree</b>	<b>16</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



## Winter 2022 – CSCI 554: Matrix Computation

Response rate: 15 / 27 (55.5%)



**Instructor is effective in organizing/  
presenting course materials:**

<b>Strong agree</b>	<b>13</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor shows interest/enthusiasm in  
teaching this course:**

<b>Strong agree</b>	<b>13</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor stimulated my interest in this  
subject area:**

<b>Strong agree</b>	<b>13</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor makes students feel free to ask  
questions and express ideas:**

<b>Strong agree</b>	<b>13</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor's communication skills are appropriate  
for the course:**

<b>Strong agree</b>	<b>14</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor is available for consultation outside  
of the class:**

<b>Strong agree</b>	<b>13</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

**Course material is well organized:**

<b>Strong agree</b>	<b>12</b>	
<b>Agree</b>	<b>3</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

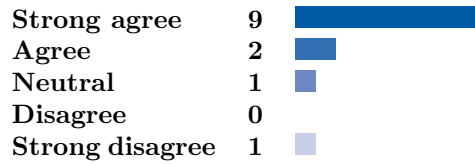
**I would recommend this course to fellow students:**

<b>Strong agree</b>	<b>13</b>	
<b>Agree</b>	<b>2</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

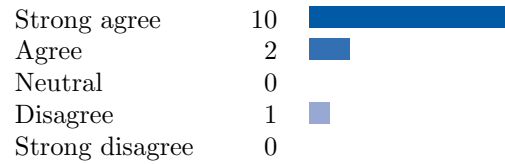
## Fall 2021 – CSCI 356: Theory of Computing

Response rate: 13 / 35 (37.1%)

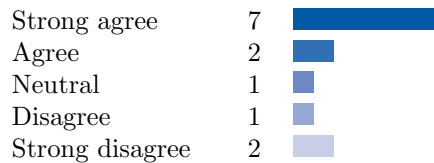
**Instructor is effective in organizing/  
presenting course materials:**



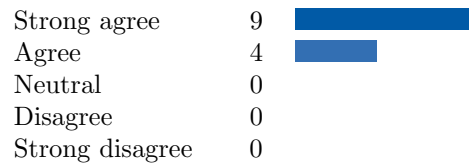
**Instructor shows interest/enthusiasm in  
teaching this course:**



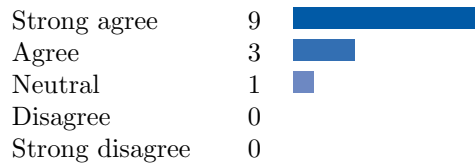
**Instructor stimulated my interest in this  
subject area:**



**Instructor makes students feel free to ask  
questions and express ideas:**



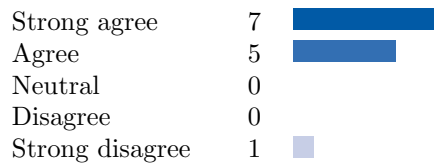
**Instructor's communication skills are appropriate  
for the course:**



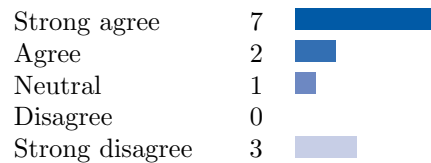
**Instructor is available for consultation outside  
of the class:**



**Course material is well organized:**




**I would recommend this course to fellow students:**




## Fall 2021 – CSCI 541: Theory of Computing

Response rate: 10 / 18 (55.5%)



**Instructor is effective in organizing/  
presenting course materials:**

<b>Strong agree</b>	<b>10</b>	
<b>Agree</b>	<b>0</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	


**Instructor shows interest/enthusiasm in  
teaching this course:**

<b>Strong agree</b>	<b>10</b>	
<b>Agree</b>	<b>0</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	


**Instructor stimulated my interest in this  
subject area:**

<b>Strong agree</b>	<b>9</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor makes students feel free to ask  
questions and express ideas:**

<b>Strong agree</b>	<b>10</b>	
<b>Agree</b>	<b>0</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	


**Instructor's communication skills are appropriate  
for the course:**

<b>Strong agree</b>	<b>10</b>	
<b>Agree</b>	<b>0</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	



**Instructor is available for consultation outside  
of the class:**

<b>Strong agree</b>	<b>9</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

**Course material is well organized:**

<b>Strong agree</b>	<b>10</b>	
<b>Agree</b>	<b>0</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

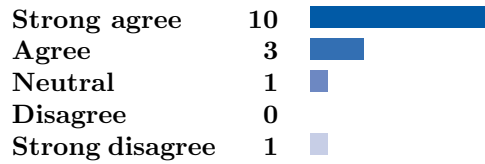
**I would recommend this course to fellow students:**

<b>Strong agree</b>	<b>9</b>	
<b>Agree</b>	<b>1</b>	
<b>Neutral</b>	<b>0</b>	
<b>Disagree</b>	<b>0</b>	
<b>Strong disagree</b>	<b>0</b>	

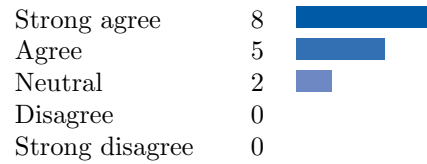
## Fall 2021 – CSCI 550: Approximation Algorithms

Response rate: 15 / 22 (68.1%)

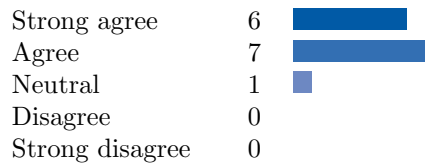
**Instructor is effective in organizing/  
presenting course materials:**



**Instructor shows interest/enthusiasm in  
teaching this course:**



**Instructor stimulated my interest in this  
subject area:**



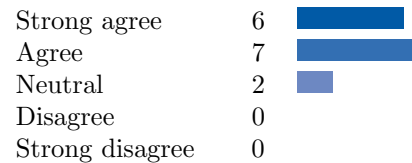
**Instructor makes students feel free to ask  
questions and express ideas:**



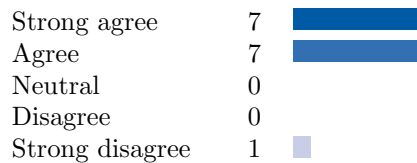
**Instructor's communication skills are appropriate  
for the course:**



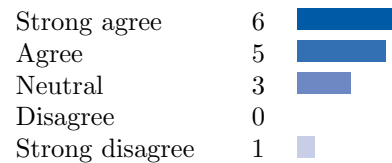
**Instructor is available for consultation outside  
of the class:**



**Course material is well organized:**



**I would recommend this course to fellow students:**





## C Recent Course Outline

**St. Francis Xavier University  
Department of Computer Science  
CSCI 541: Theory of Computing  
Course Outline  
Fall 2021**

### 1 Course Overview

This course focuses on three areas central to the theory of computation: automata, computability and complexity, to investigate the question: What are the fundamental capabilities and limitations of computers? We study automata (models of computation) e.g., finite state machines, pushdown automata and Turing machines and the languages recognized by them. We investigate complexity theory, to classify problems as easy or hard and computability theory to classify problems as solvable or not.

### 2 Learning Objectives

By the end of this course, you will be able to:

- Determine a language's place in the Chomsky hierarchy.
- Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs.
- Explain the Church-Turing thesis and its significance.
- Explain Rice's Theorem and its significance.
- Provide examples of uncomputable functions.
- Prove that a problem is uncomputable by reducing a classic known uncomputable problem to it.

Objectives from *CS2013: Curriculum Guidelines for Undergraduate Programs in Computer Science*, ACM/IEEE.

### 3 Instructor

Taylor J. Smith

- Email: [tjsmith@stfx.ca](mailto:tjsmith@stfx.ca)
- Office location: Annex, Room 9A
- Student hours: Monday, 2:15pm–3:15pm; Tuesdays, 9:15am–10:15am

### 4 Class Time and Location

- Tuesday, 8:15am–9:05am
- Wednesday, 10:15am–11:05am
- Friday, 9:15am–10:05am

All lectures are held in Mulroney Hall, Room 4032.

## 5 Evaluations

Your final grade will be based on the following components:

- Two assignments (15% each, total 30%)
- Two quizzes (12.5% each, total 25%)
- Written report (total 40%): a topic proposal document (10%) and the report itself (30%)
- Participation in lectures (5%)

You must complete both the topic proposal and the written report in order to pass the course, even if the weighted sum of your other submissions is at least 50%.

Your mid-term grade will be communicated to you by the deadline specified in the university's Academic Regulations. Your mid-term grade will consist of the weighted sum of the grades of your first assignment and your first quiz.

## 6 Method of Instruction

This course will be delivered face-to-face (i.e., all contact between instructor and students is in a physical classroom on campus). Course materials will be posted to the instructor's website.

## 7 Tentative Course Schedule

Week/Date	Topic	Due Dates
Week 1	Introduction to course, mathematical preliminaries	
Week 2	Regular langs.: finite automata, nondeterminism, nonregularity	
Week 3	Context-free langs.: pushdown automata, grammars, unary CFLs	
Week 4	Context-free langs.: ambiguous grammars, non-CFness	
Week 5	Deterministic context-free langs., parsing	<b>Quiz 1</b> (Oct. 6)
Week 6	Parsing, state complexity	<b>Assn. 1</b> (Oct. 15)
Week 7	Beyond context-free: Turing machines, variants of the model	
Week 8	Decidability, undecidability, mapping reducibility	<b>Proposal</b> (Oct. 29)
Week 9	Mapping reducibility, Turing reducibility, Kolmogorov complexity	<b>Quiz 2</b> (Nov. 3)
Week 10	Complexity theory basics: algorithm analysis, P, NP	
Week 11	NP-completeness, reducibility, space complexity	<b>Assn. 2</b> (Nov. 26)
Week 12	PSPACE, L, NL, completeness, course review	<b>Report</b> (Dec. 7)

## 8 Course Materials and Resources

Course notes will be provided for each lecture. The course textbook will be used as an optional supplement.

**Required Text.** None.

**Recommended Text.** M. Sipser, *Introduction to the Theory of Computation*. Cengage, 3rd edition, 2012.

## 9 Method of Evaluation

**Assignments.** This component will give you an opportunity to both demonstrate your understanding of course material and apply your understanding to a variety of problems. Each of the two assignments will consist of questions relating to material covered in the course between the assignment being issued and the due date. Assignments may be completed either individually or in groups of up to four people; however, if an assignment is completed as a group, each member of the group will receive the same grade for that assignment.

**Quizzes.** This component will serve as a diagnostic to gauge your individual understanding of course material. Each quiz will consist of questions relating to material covered in the course up to the date of that quiz, and will generally be shorter than an assignment. Quizzes will be distributed at the end of Tuesday's lecture and will be due at the beginning of Wednesday's lecture (i.e., 25 hours later). Quizzes must be completed individually; collaboration or group work is not permitted.

**Report.** This component consists of two submissions: a topic proposal document and the report itself.

- The topic proposal is a one- to two-page document meant to serve as a summary of your chosen topic and what you plan to discuss in your report.
- The report itself will be a survey-style paper introducing your chosen topic and reviewing the big results in the area. It is expected to be about 10 to 20 pages, including full bibliographic references.

The written report component will be an individual submission. Further details will be distributed later in the term.

**Participation.** This component is designed to encourage active engagement with the course material during lectures; for example, by asking questions or by involving yourself in discussions. As long as you consistently attend lectures and engage with the material, you will receive the full 5% for this component.

## D Sample Course Materials

### Lecture Notes: Theory of Computing

St. Francis Xavier University  
Department of Computer Science  
CSCI 356: Theory of Computing  
Lecture 1: Regular Languages  
Fall 2021

#### 1 Regular Operations and Regular Languages

In this lecture, we will begin our exploration into the theory of computation by investigating a rather simple model of computation and determining the kinds of languages this model can recognize. Before we get to defining our model, though, we will take a look at the languages themselves.

##### 1.1 Regular Operations

Recall that, if we're given two sets  $A$  and  $B$ , we can apply certain operations to produce new sets. The set operations we're most familiar with are those of union, intersection, complement, and difference. Since languages are essentially sets, we can similarly apply certain operations to languages in order to produce new languages. Three operations in particular are so important that we give them a special name: the *regular operations*.

**Definition 1** (Regular operations). Let  $L$ ,  $L_1$ , and  $L_2$  be languages. Then the regular operations of union, concatenation, and Kleene star are defined as follows:

- Union:  $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$ ;
- Concatenation:  $L_1 L_2 = \{wv \mid w \in L_1 \text{ and } v \in L_2\}$ ; and
- Kleene star:  $L^* = \bigcup_{i \geq 0} L^i$ , where  $L^0 = \{\epsilon\}$ ,  $L^1 = L$ , and  $L^i = \{wv \mid w \in L^{i-1} \text{ and } v \in L\}$ .

The union operation, naturally, works in exactly the same way for languages as it does for sets. The other two operations, on the other hand, don't have an exact match to a set operation, but we can reason about them by drawing analogies to other operations we've seen.

The concatenation operation is most similar to our notion of tuples, if we stripped away all of the sequence-y notation; concatenation takes two words and “connects” the end of the first word to the beginning of the second word.

Lastly, the Kleene star operation is somewhat similar to taking a repeated Cartesian product, if we “connect” our elements (words) via concatenation rather than in a tuple. Note that, since the Kleene star allows us to take zero copies of a word, the empty word  $\epsilon$  is always included in the resulting language.

**Example 2.** Let  $L_1 = \{a, b\}$  and  $L_2 = \{d, e\}$ . Then  $L_1 \cup L_2 = \{a, b, d, e\}$ ,  $L_1 L_2 = \{ad, ae, bd, be\}$ ,  $L_1^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$ , and  $L_2^* = \{\epsilon, d, e, dd, de, ed, ee, ddd, dde, \dots\}$

##### 1.2 Regular Languages

So, what makes these particular operations so special, and why do we refer to them as the regular operations? As it turns out, taking just these three operations is sufficient to allow us to define the smallest class of languages that is “interesting enough” to study<sup>1</sup>: the *regular languages*.

<sup>1</sup>There is a smaller class called the class of *finite languages*. However, it's not too interesting: it consists only of languages with a finite number of words. Introducing the Kleene star allows us to produce infinite-size languages.

**Definition 3** (Regular languages—language-theoretic def'n). Let  $\Sigma$  be an alphabet. The class of regular languages is defined inductively as follows:

1. The empty language,  $\emptyset$ , is regular.
2. For each  $a \in \Sigma$ , the language  $\{a\}$  is regular.
3. If  $L_1$  and  $L_2$  are regular, then  $L_1 \cup L_2$  is regular.
4. If  $L_1$  and  $L_2$  are regular, then  $L_1 L_2$  is regular.
5. If  $L_1$  is regular, then  $L_1^*$  is regular.

At this point, you might be asking yourself: why do we call these operations and languages “regular”? Stephen Kleene introduced the notion of a regular language in the 1950s, but his justification for the terminology was basically that he couldn’t come up with any better name:

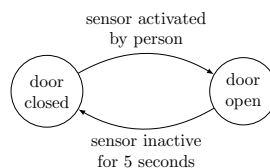
“We would welcome any suggestions as to a more descriptive term.”  
 — Stephen Kleene, *Representation of Events in Nerve Nets and Finite Automata*  
 RAND Corporation Research Memorandum RM-704, 1951.

Keep the definition of the class of regular languages in mind as we go forward. It will reappear once we introduce our chosen model of computation in this lecture.

## 2 Finite Automata

The entire point of studying computer science, some might argue, is to determine exactly what computers are capable of. Indeed, humans created computers so that we could pass off boring or repetitive work onto a machine and give our brains a break! However, considering a full computer in the very beginning of our studies is kind of like learning to swim by jumping into the deep end of a pool. In order to learn without getting overwhelmed, we will begin by considering a very simple model of computation that gives us just enough power to actually perform an elementary computation.

If you’ve ever used a vending machine, or waited in a car at a traffic light, or walked through an automatic door, then you’re already familiar with the notion of a *finite automaton*. Consider, for example, how an automatic door works:



The door transitions between two states—closed and open—depending on what the sensor is reporting. The states (circles) represent the door’s current status, and the transitions (arrows) correspond to an input given to the door. Note that the door has no way of knowing or remembering that it’s closed or open apart from being in a state; it responds solely based on the input it receives from the sensor. This is a finite automaton: an automaton in the sense that it’s a machine that performs an action based on predetermined conditions or instructions, and finite in the sense that there’s a finite number of possible states the machine can be in at a given time.

### 2.1 Definition

We can use finite automata to model simple computations that take some input word and don’t require memory or storage. In a computation, the states of the finite automaton correspond to our current step of the computation. For example, did we just begin the computation, or are we midway through reading some

input, or something else? The transitions of the finite automaton take us between states, depending on the label of the transition. If we have, say, a binary word as the input to our finite automaton, then we can transition to a different state depending on whether the next symbol in the word is a 0 or a 1.

Formally speaking, a finite automaton is just a 5-tuple.

**Definition 4** (Finite automaton). A finite automaton is a tuple  $(Q, \Sigma, \delta, q_0, F)$ , where

- $Q$  is a finite set of *states*;
- $\Sigma$  is an *alphabet*;
- $\delta: Q \times \Sigma \rightarrow Q$  is the *transition function*;
- $q_0 \in Q$  is the *initial* or *start state*; and
- $F \subseteq Q$  is the set of *final* or *accepting states*.

We're already familiar with states and alphabets, and we know a little bit about transitions from our example. The transition function  $\delta$  is the mathematical formalization of the arrows in our diagram. Given an ordered pair of state and symbol being read, the transition function tells us which state to go to next. For example, if we had a very simple finite automaton like



then the single transition would be represented by the function  $\delta(q_0, a) = q_1$ . If a given finite automaton has a large number of transitions, then we can represent each transition concisely in a table format rather than writing each transition out individually.

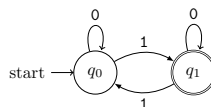
Note that, since we're dealing with a transition *function*, any pair of state and symbol can map to *at most* one state. This condition ensures that we always make the same transition on the same state/symbol pair.

You may have also noticed that the states in our very simple finite automaton had some special flair added to them. The state  $q_0$  has an arrow labelled "start" pointing to it, and the state  $q_1$  has two circles instead of one. This is how we denote initial and final states in our diagram. Initial states have an incoming transition arrow pointing at the state, while final states are double-circled. We typically have just one initial state in a finite automaton, but it's possible to have more than one. On the other hand, we can have as many or as few final states as we want.

**Example 5.** Consider the finite automaton  $\mathcal{M}_1 = (Q, \Sigma, \delta, q_0, F)$  where  $Q = \{q_0, q_1\}$ ,  $\Sigma = \{0, 1\}$ ,  $q_0$  is the initial state,  $F = \{q_1\}$ , and  $\delta$  is defined as follows:

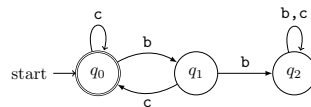
	0	1
$q_0$	$q_0$	$q_1$
$q_1$	$q_1$	$q_0$

We can draw this finite automaton diagrammatically:



This finite automaton checks whether a binary word has odd parity; that is, whether it contains an odd number of 1s.

**Example 6.** Consider the following diagram of a finite automaton:



This finite automaton checks whether every occurrence of **b** in an input word is immediately followed by an occurrence of **c**.

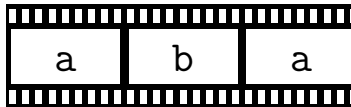
Based on this diagram, we can establish that  $Q = \{q_0, q_1, q_2\}$ ,  $\Sigma = \{b, c\}$ ,  $q_0$  is the initial state,  $F = \{q_0\}$ , and  $\delta$  is defined as follows:

	b	c
$q_0$	$q_1$	$q_0$
$q_1$	$q_2$	$q_0$
$q_2$	$q_2$	$q_2$

## 2.2 Computations: Inputs, Acceptance, and Rejection

Now that we know how to define a finite automaton, what can we do with it? Observe that, in our definition, we took care to specify the alphabet  $\Sigma$ . This alphabet gives us information about the kinds of *input words* we can give to a finite automaton. Giving an input word to a finite automaton is much like typing `input()` in a Python program or `scanf()` in a C program; it gives the computer something to read and work with.

When a finite automaton is given an input word, we can imagine the word is written on a reel of film where each symbol in the word has its own frame.



Now, imagine the finite automaton is a film projector, but the rewind button is broken. When we play the film reel starting at the first frame, the projector can only show one frame at a time, and once it moves to the next frame it can never return to the previous one. This is essentially how a finite automaton processes its input: starting with the first symbol of the input word, the finite automaton reads the symbol, transitions to a state, and then moves to the next symbol.

Once the finite automaton reaches the end of its input word, it must make a decision to either *accept* or *reject* the word. Whether or not the finite automaton accepts the input word depends entirely on the state the finite automaton is in at the moment it reaches the end of the word. If the finite automaton is in a final state and it has no more symbols left to read, then it accepts the word. Otherwise, the finite automaton must be in a non-final state, and it therefore rejects the word.

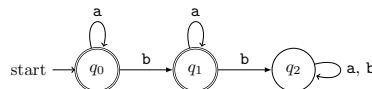
The set of all input words that a finite automaton  $\mathcal{M}$  accepts is called the *language* of the finite automaton, denoted  $L(\mathcal{M})$ , and it's just like any other language: it consists of words over the alphabet  $\Sigma$ . If a finite automaton  $\mathcal{M}$  accepts (or *recognizes*<sup>2</sup>) a language  $A$ , then  $L(\mathcal{M}) = A$ . Note that, even though a finite automaton can accept many input words, it can only recognize *one* language.

<sup>2</sup>For clarity's sake, I will try to use the word "accept" only when referring to input words given to a finite automaton, and I will use "recognize" when referring to the language of a finite automaton.

**Example 7.** Let  $\Sigma = \{a, b\}$ , and consider the language

$$L_{|w|_b \leq 1} = \{w \mid w \text{ contains at most one occurrence of the symbol } b\}.$$

This language can be recognized by the following automaton:



If the input word  $w$  contains zero  $b$ s, then the finite automaton will remain in the final state  $q_0$ . Likewise, if  $w$  contains one  $b$ , then the finite automaton will enter and remain in the final state  $q_1$ . Only if  $w$  contains two or more  $b$ s does the finite automaton enter the state  $q_2$ , where it becomes “stuck” and can no longer accept the input word.

**Example 8.** A finite automaton with no final states is still able to recognize one language: the empty language,  $\emptyset$ . This is because the language of input words accepted by the finite automaton is empty.

As a matter of notation, we will refer to the class of languages recognized by *some* finite automaton by the abbreviation DFA. (What does the D mean? We’ll find out in the next section. . . )

We wrap up this section by precisely defining what it means for a finite automaton to accept an input word; that is, by formalizing the notion of an *accepting computation*. We don’t need anything new to do this; we already have all the machinery we need.

**Definition 9** (Accepting computation of a finite automaton). Let  $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$  be a finite automaton, and let  $w = w_0w_1 \dots w_{n-1}$  be an input word of length  $n$  where  $w_0, w_1, \dots, w_{n-1} \in \Sigma$ . The finite automaton  $\mathcal{M}$  accepts the input word  $w$  if there exists a sequence of states  $r_0, r_1, \dots, r_n \in Q$  satisfying the following conditions:

1.  $r_0 = q_0$ ;
2.  $\delta(r_i, w_i) = r_{i+1}$  for all  $0 \leq i \leq (n-1)$ ; and
3.  $r_n \in F$ .

Now that we have the formal notions of a finite automaton and an accepting computation, we can provide an alternative definition of what it means for a language to be regular.

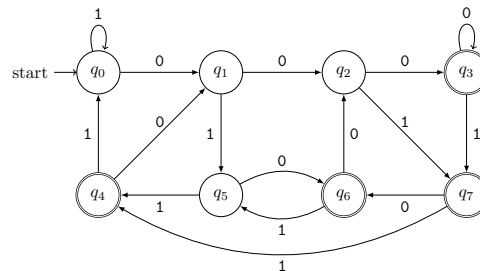
**Definition 10** (Regular languages—automata-theoretic def’n). If some finite automaton  $\mathcal{M}$  recognizes a language  $L$ , then  $L$  is regular.

### 2.3 Nondeterminism

Remember how, when we were discussing the transition function earlier, we mandated a condition that any pair of state and symbol must map to *at most* one state? This condition ensured that if we gave the same input word to the same finite automaton, we would end up with the same result. This is known as *deterministic* computation. (And now you know what the D in DFA stands for!)

While determinism isn’t inherently a bad thing, it can unfortunately make our job harder if we’re trying to construct a finite automaton that recognizes certain “difficult” languages. For example, suppose we wanted to construct a deterministic finite automaton that recognizes the language of words over the alphabet  $\Sigma = \{0, 1\}$  where the third-from-last symbol is 0. This finite automaton should accept input words like 011, 10010, and 1010001010011000, but it should reject input words like 110 or 01. Sounds easy to do, right? After all, we really just need to check one symbol: the symbol in the third-from-last position. As it turns out, however, this is the deterministic finite automaton in question:





Keep in mind also that this deterministic finite automaton *only* works for input words where the third-from-last symbol is 0. If we wanted to, say, check the fourth-from-last symbol, we would need to construct a whole new finite automaton—and this one would have *twice as many* states as our previous one!

So, how do we make our job easier and our finite automata smaller? We get rid of the determinism condition. Specifically, we allow for state/symbol pairs to map to one *or more* states. (We're able to preserve the "function" part of our transition function by mapping each state/symbol pair not to multiple different states, but rather to an element of the power set of states. We'll clarify this in the definition.)

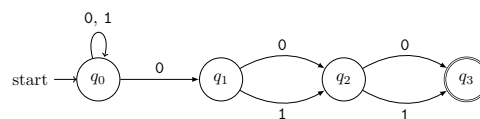
If we get rid of the determinism condition, then the finite automaton can, in a sense, "guess" which step to take at certain points in the computation. If, in a given state, there is more than one transition out of that state on the same symbol, then the finite automaton has multiple options for which transition it can take. As you might have guessed, this is called *nondeterminism*, and the definition of a nondeterministic finite automaton is nearly identical to our earlier definition of a deterministic finite automaton.

**Definition 11** (Nondeterministic finite automaton). A nondeterministic finite automaton is a tuple  $(Q, \Sigma, \delta, q_0, F)$ , where

- $Q$  is a finite set of states;
- $\Sigma$  is an alphabet;
- $\delta: Q \times \Sigma \rightarrow \mathcal{P}(Q)$  is the transition function;
- $q_0 \in Q$  is the initial or start state; and
- $F \subseteq Q$  is the set of final or accepting states.

As you can see, the only change we had to make to the definition is in the transition function, where we now map to the power set  $\mathcal{P}(Q)$  instead of the state set  $Q$ . The element of the power set being mapped to is exactly the subset of states that the nondeterministic finite automaton can transition to from its current state and on its current symbol.

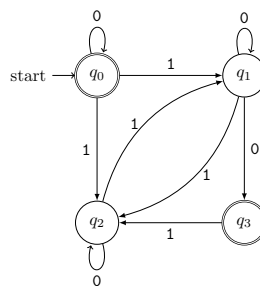
As an illustration of how nondeterminism can simplify the finite automata we construct, think back to our example of the language of words whose third-from-last symbol is 0. Here is the nondeterministic version of the finite automaton recognizing this language:



Here, the state  $q_0$  is doing double duty: not only is it reading all of the symbols in the input word up to the third-from-last symbol, but it's also checking that the third-from-last symbol is in fact 0. If it is, then we transition from state  $q_0$  to state  $q_1$ , and the remaining states simply read the last two symbols (whatever they may be).

The nondeterminism in this machine is limited to state  $q_0$ , where we have two outgoing transitions on the same symbol 0: one transition loops back to the same state  $q_0$ , while the other transition takes us to state  $q_1$ . We can represent this with the transition function by writing  $\delta(q_0, 0) = \{q_0, q_1\}$ , and this abides by our definition since  $\{q_0, q_1\} \in \mathcal{P}(Q)$ .

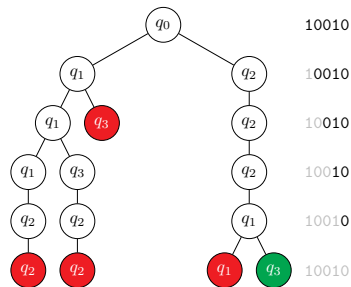
**Example 12.** The following finite automaton is nondeterministic, because some states have multiple outgoing transitions on the same symbol:



A nondeterministic finite automaton accepts an input word in exactly the same way as a deterministic finite automaton: if the finite automaton is in a final state and there are no more symbols of the input word left to read, then the input word is accepted. If not, then the input word is rejected. We will refer to the class of languages recognized by *some* nondeterministic finite automaton by the abbreviation **NFA**.

The computation of a nondeterministic finite automaton, however, is slightly different than in the deterministic case. Since the finite automaton can take potentially many transitions from one state/symbol pair, at such a point in the computation, the finite automaton “splits up” and runs multiple copies of itself in parallel. If we were to visualize such a computation, we would obtain a diagram that resembles a tree. (In fact, such a visualization is called a *computation tree*.) In each branch of the computation, the corresponding copy of the finite automaton continues its computation until it either reaches the end of the input word or finds itself with no more transitions to follow (which could happen if the finite automaton reads a symbol in a state with no outgoing transition on that symbol). In the latter case, that branch dies while the remaining branches continue with their computations. Similarly, if there are no more symbols to read in the input word and that copy of the finite automaton isn’t in a final state, that branch dies. A computation of a nondeterministic finite automaton is accepting only if there exists at least one branch of the computation where the finite automaton is in a final state after reading every symbol of the input word.

**Example 13.** Recall the nondeterministic finite automaton from the previous example. Does this automaton accept the input word 10010? Let’s check by drawing the computation tree. Each vertex indicates the current state of the finite automaton at that point in the computation, and the symbols remaining in the input word are listed on the right.



Since there exists at least one branch of the computation tree where the finite automaton is in a final state after reading the entire input word, the finite automaton accepts the word 10010.

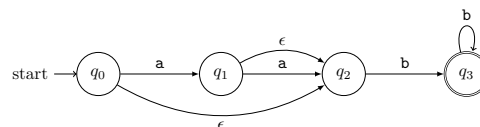
We can formalize the notion of an accepting computation once again for nondeterministic finite automata; the only change we need to make is in the second condition.

**Definition 14** (Accepting computation of a nondeterministic finite automaton). Let  $\mathcal{M} = (Q, \Sigma, \delta, q_0, F)$  be a nondeterministic finite automaton, and let  $w = w_0w_1 \dots w_{n-1}$  be an input word of length  $n$  where  $w_0, w_1, \dots, w_{n-1} \in \Sigma$ . The finite automaton  $\mathcal{M}$  accepts the input word  $w$  if there exists a sequence of states  $r_0, r_1, \dots, r_n \in Q$  satisfying the following conditions:

1.  $r_0 = q_0$ ;
2.  $r_{i+1} \in \delta(r_i, w_i)$  for all  $0 \leq i \leq (n-1)$ ; and
3.  $r_n \in F$ .

Going one step further, we can take a nondeterministic finite automaton and modify it so that it can transition not just after reading a symbol, but *whenever it wants*. If a certain special transition called an *epsilon transition* exists between two states  $q_i$  and  $q_j$ , a finite automaton in state  $q_i$  can immediately transition to state  $q_j$  without reading the next symbol of the input word. We call such a model a nondeterministic finite automaton *with epsilon transitions*, and the class of languages recognized by this model is denoted by  $\epsilon$ -NFA.

**Example 15.** The following nondeterministic finite automaton uses epsilon transitions:



This finite automaton accepts all input words starting with zero, one, or two as followed by at least one b.

## Assessment: Algorithm Analysis and Design

4. A group of ornithology researchers has asked you to help them track the spread of a novel crow virus called Corvid. The researchers have tagged  $n$  birds in their system and named them  $B_1$  through  $B_n$ . They give you a set of  $m$  data tuples indicating when pairs of birds were detected together: this data is of the form  $(B_i, B_j, t_k)$ , indicating that birds  $B_i$  and  $B_j$  were together at time  $t_k$ .

If one infected bird  $B_i$  is detected with an uninfected bird  $B_j$  at time  $t_k$ , then bird  $B_j$  becomes infected from time  $t_k$  onward. This infection is modelled by the existence of either of the tuples  $(B_i, B_j, t_k)$  or  $(B_j, B_i, t_k)$ . The spread of the virus can then be modelled by a sequence of tuples: if bird  $B_i$  is infected by time  $t_k$ , and there are tuples  $(B_i, B_j, t_k)$  and  $(B_j, B_m, t_\ell)$  where  $t_k \leq t_\ell$ , then bird  $B_m$  will be infected via bird  $B_j$ .

Design an algorithm that answers the following question: if bird  $B_a$  was infected by the virus at time  $t_x$ , could it have infected bird  $B_b$  by time  $t_y$ ? Your input is the set of  $m$  tuples defined earlier, as well as  $B_a, B_b, t_x$ , and  $t_y$ . You can assume that the tuples are sorted by time, that each pair of birds is found together at most once, and that a bird remains forever infectious once it is infected.

You do not need to establish the correctness or running time of your algorithm, but it may help to know that the official solution runs in linear time.

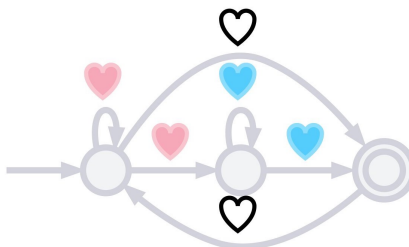
## Assessment: Discrete Mathematics

2. UTF-8 is part of the Unicode character encoding standard. It is the most common standard for encoding text on the web, in email, and in documents.

- UTF-8 is designed to be backwards-compatible with the ASCII encoding, which contains English alphabet characters and punctuation symbols. Characters in these alphabets have UTF-8 encodings of the form  $0xxxxxxx$ , where  $x$  is one bit (0 or 1). How many possible characters can be represented with this encoding?
- Other alphabets, like Greek and Russian, have UTF-8 encodings of the form  $110xxxxx10xxxxxx$ , where  $x$  is defined as before. How many possible characters can be represented with this encoding?
- The English alphabet has 26 uppercase letters, the Greek alphabet has 24 uppercase letters, and the Russian alphabet has 33 uppercase letters. Some of these alphabets share letters that look identical, so they don't need to be encoded multiple times. English and Greek share 14 letters, English and Russian share 12 letters, Greek and Russian share 14 letters, and all three languages share 11 letters. How many encodings in total do these three languages require, if shared letters also share encodings?

## Assessment: Theory of Computing

4. The Twitter account @happyautomata (<https://twitter.com/happyautomata>) automatically generates examples of finite automata over both the English alphabet and the “emoji alphabet”. An example of an “emoji automaton” is shown below:



Visit the Twitter account, choose your favourite “emoji automaton” with at least three states, and convert the finite automaton to an equivalent regular expression. Show all your work in addition to giving the regular expression.

If you don't use Twitter, then you are welcome to use the above “emoji automaton” to form your answer.

If you don't wish to use emoji in your regular expression (or you can't get it to work nicely), you can convert emoji to letters; for example, = b, = p, and = w.

## E Sample Scholarship Materials

### Scholarship of Teaching and Learning Research Prospectus Taylor J. Smith

Courses in theoretical computer science are unlike most traditional computer science courses at the university level; they are, in fact, more akin to a course in pure or applied mathematics. In a theoretical computer science lecture, both students and instructors are unlikely to do any coding, and neither party may even be required to use a computer at all! Unfortunately, the inclination of the typical computer science student is to avoid courses that are math-heavy, and as a result, theory courses are often considered by students to be the least popular courses offered by a department.

As an undergraduate student, my first course in theoretical computer science was led by an instructor who relied on PowerPoint slides made by the author of the course textbook. As a result, I felt that neither I nor the instructor had any “real” connection to the material being taught in the course, and my attention waned during each lecture. Prior to beginning my doctoral studies, I taught a second-year computer science course that had a set of pre-made slides compiled by past instructors. Here, I was exposed to the other side of the experience: if I just read off of slides, then what incentive do students have to come to class versus reading the slides from home? Could student interest in the course increase if they physically engaged with the material, say, through writing notes by hand or participating in live classroom activities?

While I opted during that course to deliver material using a combination of slides and chalkboard work, the question of slides and incentives never left my mind. As I began to hone my teaching skills, I observed various instructors and compared courses that were delivered primarily via slides to courses that were delivered primarily through other means, such as chalkboard work or seminar-style lectures. While I was aware of studies investigating the effects of technology in the university classroom, I had never heard of any study investigating technology specifically in a theoretical computer science course. Can the type of abstract material covered in such a course be delivered effectively using technology, or (like mathematics) are students better off learning such material “by hand”, through writing notes and working offline? With more and newer classroom technology being introduced by the month, and especially with courses moving online in response to this year’s pandemic, can—or should—courses in theoretical computer science change accordingly, and how will students be impacted?

Based on the existing literature, it seems that prior work can be distilled into three general statements:

1. PowerPoint slides should be used to communicate difficult or complex topics, especially if they can be presented in a visual manner;
2. PowerPoint slides do not have a noticeable effect on visual information retention, but may affect other forms of information retention; and
3. Students in engineering disciplines appreciate the use of PowerPoint slides for purposes such as note-taking, but do not have a strong preference for lectures that use PowerPoint slides.

However, I found it notable that I could not locate any research on the use of PowerPoint specifically in computer science lectures. Since computer science is such a vast discipline, it may be the case that some topics of study lend themselves more readily to PowerPoint-based lectures or other educational uses of technology. Thus, it is necessary for me to restrict the scope of my research question to the specific subdiscipline of theoretical computer science (and perhaps even to *introductory* theory courses, to ensure all students have the same baseline knowledge). Thus, I propose to investigate the following question:

*“Does the use of presentation technology (e.g., PowerPoint slides) have a positive effect on student understanding in an undergraduate theoretical computer science course, compared to the use of a traditional (e.g., chalkboard) lecture style?”*

### Annotated Bibliography

**1. Y. Inoue-Smith. College-based case studies in using PowerPoint effectively. *Cogent Education*, 3:1-15, 2016.**

This paper examines the potential of PowerPoint as a tool to enhance traditional pedagogical techniques. Namely, the pedagogical technique under consideration in this paper is active learning, or the practice of engaging students during a lecture and allowing students to become involved with their own education. The paper takes a case study approach by considering the experiences of seven faculty members teaching across a variety of departments at an American university. All of the professors taught courses which are typically considered to belong to the general area of “liberal arts”.

The paper focuses on a comparison between use of PowerPoint slides in a university lecture and students’ perceptions of PowerPoint slide use. The author observes that three of the seven lectures were taught in a large lecture hall, where it is difficult to engage with students using a traditional lecture style (i.e., no PowerPoint). The author posits that using PowerPoint in a large lecture can both grab students’ attention and help to structure lecture content, but slides should be written with the student in mind; teacher-centered slides do not give students an idea of how to approach the material on their own. Participants in the study claimed that using PowerPoint helps with lecture organization, but the responsibility for student learning remains with the professor. Before a professor creates slides for a course, they must take care to set specific objectives to measure a student’s mastery of the material in the course.

The author specifically highlights in the conclusion of the paper that “Teaching mathematics with PowerPoint is not common; instead, most professors still use chalkboards. Chalk allows the mathematics professor to unveil each equation or theorem step by step.” The author goes on to write that “PowerPoint is good for visually enriching the content and illustrating complex concepts”, but it is easy to fall into the trap of overloading students with information or rushing too quickly through a presentation. Given that many theoretical computer science courses involve the use and construction of diagrams (e.g., for drawing finite automata), PowerPoint may be a good tool to enhance the visual aspect of a lecture.

**2. C. Swati, T. Suresh, D. Sachin. Student assessment on learning based on PowerPoint versus chalkboard. *International Journal of Recent Trends in Science and Technology*, 13:347-351, 2014.**

This paper investigates the impact of PowerPoint slide use versus chalkboard use on student learning, and the lecture delivery preferences of students in a microbiology department. It is a cross-sectional, interventional study: data was collected over a period of one term, and two groups of students received two lectures on topics in microbiology. The first topic was considered to be “simple”, while the second topic was considered to be “complex”. The content of each lecture was the same across both groups, and the instructors of each group had the same teaching experience. For the first lecture, one group was taught via chalkboard while the other was taught via PowerPoint slides, and for the second lecture, the teaching methods for each group were swapped. Students in both groups completed a pre-test before each lecture and a post-test following each lecture to evaluate their understanding of the material, with both tests containing the same set of questions.

The authors performed a statistical analysis on the pre-test and post-test performance data of both groups (hereafter referred to as “Group A” and “Group B”, as in the paper). For the first lecture, Group A was taught via PowerPoint while Group B was taught via chalkboard. The pre-test mean score difference between Group A and Group B was not statistically significant, as was the difference in mean score change from pre-test to post-test between Group A and Group B. The proportion of passing students on the post-test versus on the pre-test was significant for both groups. For the second lecture, the pre-test mean score difference was again not significant, but the difference in mean score change from pre-test to post-test between both groups was significant. The proportion of passing students on the post-test versus on the pre-test was more significant for Group A (chalkboard) than for Group B (PowerPoint slides).

The authors conclude that traditional lecture delivery is more effective than PowerPoint slide lecture delivery in terms of raw scoring and student pass results. The authors note that chalkboard use resulted in a stronger focus and better lecture structure than PowerPoint slide use for complex topics. Since many computer science students consider theory to be a “complex topic”, this study suggests that it may be a subject better suited for traditional lecture delivery.

**3. A. Savoy, R. W. Proctor, G. Salvendy. Information retention from PowerPoint and traditional lectures. *Computers and Education*, 52: 858-867, 2009.**

This paper investigates the effects of PowerPoint slide use in lecture on three aspects of student outcomes on examinations: auditory scores, graphic scores, and alphanumeric scores. The paper studies students in a course that is cross-listed between departments of psychology and industrial engineering. The course was designed for engineering majors with little background in psychology. The study ran over a period of one term, and consisted of two lectures on differing topics. Both lectures were presented using a combination of traditional delivery and PowerPoint slides. Students were assessed on lecture content using a quiz consisting of multiple-choice questions; questions about lecture content were divided into questions about information presented auditorily by the instructor, information presented auditorily with visual support, information presented graphically, and alphanumeric information (i.e., text).

The authors conjectured that auditory information is difficult to recall when presented in conjunction with PowerPoint slides, and their findings showed that auditory scores with the traditional delivery style were 15% higher than with the PowerPoint delivery style. In terms of graphic scores, the authors observed no notable gain when using PowerPoint slides versus traditional lecturing, but the scores of students who attended class were higher than the scores for students who did not attend class, suggesting that attending class offers a tangible benefit over simply reading PowerPoint slides or texts from home. Combining both audio and visual content, the authors found no significant difference in scores between PowerPoint lectures and traditional lectures, suggesting that there is no benefit to using PowerPoint when lectures consist mostly of text and simple graphics. The authors finally conjectured that students would prefer PowerPoint slides over traditional lectures; subjective data collected by the authors showed that students had no strong preference for either lecture style.

This paper relates to Reference 2 in that both study a cross-section of students in a real-world lecture environment. However, given that the students in this study are engineering majors, their experience and personal inclinations may align more closely to the typical computer science student (i.e., they may be more technology-savvy). However, the fact that students in this study seemed not to have a strong opinion on PowerPoint lectures might suggest that technological ability does not translate to pedagogical preference.

**4. A. O'Dwyer. Responses of engineering students to lectures using PowerPoint. In *Proceedings of the International Symposium for Engineering Education (ISEE 2008)*, pages 219-226, Dublin, 2008.**

This paper, much like Reference 3, investigates the responses of engineering students to PowerPoint-based lectures versus traditional lectures. The authors survey three cohorts of engineering students across three different levels of study at one university. Between 65% and 75% of lecture time during the term involved use of PowerPoint across all three cohorts. Students were given a questionnaire at the end of the term where they were asked to compare lectures delivered using PowerPoint to lectures delivered in the traditional style (i.e., using chalkboards or overhead projector slides).

The authors found that students across all cohorts identified the value of using PowerPoint slides, and indicated that PowerPoint lectures were both more interesting and facilitated greater learning. Students also indicated a preference for having lecture material online in the form of PowerPoint slides, as they can add notes to printed copies of slides as a form of active learning. Lastly, students claimed to have more motivation to attend PowerPoint-based lectures, but claimed to feel less bad about missing a PowerPoint-based lecture over a traditional lecture.