Reclaiming Computer Science with Stroustrup's *Programming Practices and Principles in C++*

Geoff Hagopian

College of the Desert

ghagopian@collegeofthedesert.edu

for CMC3 43rd Annual Conference

December 12, 2015

Overview

1. History of Computer Science at COD.

Overview

1. History of Computer Science at COD.
2. Stroustrup's Philosophy

Overview

1. History of Computer Science at COD.

2. Stroustrup's Philosophy

3. Computer Science and its Relation to Mathematics
   (why it's important to love the monster)

Overview

1. History of Computer Science at COD.
2. Stroustrup's Philosophy
3. Computer Science and its Relation to Mathematics (why it's important to love the monster)
   - Is Computer Science Part of Mathematics?

Overview

1. History of Computer Science at COD.
2. Stroustrup's Philosophy
3. Computer Science and its Relation to Mathematics (why it's important to love the monster)
   - Is Computer Science Part of Mathematics?
4. Stroustrup's Book

Overview

1. History of Computer Science at COD.

2. Stroustrup's Philosophy

3. Computer Science and its Relation to Mathematics (why it's important to love the monster)
   - Is Computer Science Part of Mathematics?

4. Stroustrup's Book

5. Using Programming Projects to Learn Math.

Overview

1. History of Computer Science at COD.

2. Stroustrup's Philosophy

3. Computer Science and its Relation to Mathematics (why it's important to love the monster)
   - Is Computer Science Part of Mathematics?

4. Stroustrup's Book

5. Using Programming Projects to Learn Math.
   - Shorter coding assignments that reinforce math students are learning.

Overview

1. History of Computer Science at COD.
2. Stroustrup's Philosophy
3. Computer Science and its Relation to Mathematics (why it's important to love the monster)
    - Is Computer Science Part of Mathematics?
4. Stroustrup's Book
5. Using Programming Projects to Learn Math.
    - Shorter coding assignments that reinforce math students are learning.
    - Larger coding projects that develop new ideas.

Brief History of CS at COD

- Hired as a math teacher, I started teaching Physics 5 (Scientific Computing) in 1997.

Brief History of CS at COD

- Hired as a math teacher, I started teaching Physics 5 (Scientific Computing) in 1997.
- With physics teacher, Doug Macintire, received CS equivalence and created new Computer Science program in 2012.

"qualifications that are at least equivalent to the min. quals specified"

Brief History of CS at COD

- Hired as a math teacher, I started teaching Physics 5 (Scientific Computing) in 1997.
- With physics teacher, Doug Macintire, received CS equivalence and created new Computer Science program in 2012.

"qualifications that are at least equivalent to the min. quals specified"

- "Master's in computer science or computer engineering OR Bachelor's in either of the above AND Master's in mathematics, cybernetics, business administration, accounting or engineering OR Bachelor's in engineering AND Master's in cybernetics, engineering, mathematics, or business administration OR Bachelor's in mathematics AND Master's in cybernetics, engineering[,] mathematics, or business administration OR Bachelor's degree in any of the above AND a Master's degree in information science, computer information systems, or information systems

Brief History of CS at COD

- Hired as a math teacher, I started teaching Physics 5 (Scientific Computing) in 1997.
- With physics teacher, Doug Macintire, received CS equivalence and created new Computer Science program in 2012.

"qualifications that are at least equivalent to the min. quals specified"

- "Master's in computer science or computer engineering OR Bachelor's in either of the above AND Master's in mathematics, cybernetics, business administration, accounting or engineering OR Bachelor's in engineering AND Master's in cybernetics, engineering, mathematics, or business administration OR Bachelor's in mathematics AND Master's in cybernetics, engineering[,] mathematics, or business administration OR Bachelor's degree in any of the above AND a Master's degree in information science, computer information systems, or information systems

- **COMPUTER SCIENCE A. S. DEGREE**
  and transfer preparation

Dept.	No.	Title	Units
Required Courses:			
CS	7A	Computer Science I	4
CS	7B	Computer Science II	4
CS	8	Computer Architecture & Org.	4
MATH	1A	Calculus	5
MATH	1B	Calculus	5
MATH	15	Discrete Math for Computers	4
PH	3A	Engineering Physics	4
PH	3B	Engineering Physics	4
Electives - *A minimum of 2 courses to be chosen from the following (8-10 units):*			
CS	9	Data Structures & Algorithms	4
CH	1A	General Chemistry I	5
MATH	2A	Multivariate Calculus	5
MATH	2B	Linear Algebra	4
MATH	2C	Ordinary Differential Equations	4
PH	3C	Engineering Physics	4
PH	6A	Electric Circuits for Engr & Science	4

Brief History of CS at COD

- Hired as a math teacher, I started teaching Physics 5 (Scientific Computing) in 1997.
- With physics teacher, Doug Macintire, received CS equivalence and created new Computer Science program in 2012.

"qualifications that are at least equivalent to the min. quals specified"

- "Master's in computer science or computer engineering OR Bachelor's in either of the above AND Master's in mathematics, cybernetics, business administration, accounting or engineering OR Bachelor's in engineering AND Master's in cybernetics, engineering, mathematics, or business administration OR Bachelor's in mathematics AND Master's in cybernetics, engineering[,] mathematics, or business administration OR Bachelor's degree in any of the above AND a Master's degree in information science, computer information systems, or information systems

- **COMPUTER SCIENCE A. S. DEGREE**
  and transfer preparation

Dept.	No.	Title	Units
Required Courses:			
CS	7A	Computer Science I	4
CS	7B	Computer Science II	4
CS	8	Computer Architecture & Org.	4
MATH	1A	Calculus	5
MATH	1B	Calculus	5
MATH	15	Discrete Math for Computers	4
PH	3A	Engineering Physics	4
PH	3B	Engineering Physics	4
Electives - *A minimum of 2 courses to be chosen from the following (8-10 units):*			
CS	9	Data Structures & Algorithms	4
CH	1A	General Chemistry I	5
MATH	2A	Multivariate Calculus	5
MATH	2B	Linear Algebra	4
MATH	2C	Ordinary Differential Equations	4
PH	3C	Engineering Physics	4
PH	6A	Electric Circuits for Engr & Science	4

Stroustrup's Philosophy

(A summary of S's article, *Programming in an undergraduate CS curriculum*)

- Programming is a means of making ideas into reality using computers.

Stroustrup's Philosophy

(A summary of S's article, *Programming in an undergraduate CS curriculum*)

- Programming is a means of making ideas into reality using computers.
- What universities produce $\neq$ what industry needs.

Stroustrup's Philosophy

(A summary of S's article, *Programming in an undergraduate CS curriculum*)

- Programming is a means of making ideas into reality using computers.
- What universities produce $\neq$ what industry needs.
- CS must emphasize software development (even at the expense of algorithmic complexity, data structures and...subsurface luminosity).

Stroustrup's Philosophy

(A summary of S's article, *Programming in an undergraduate CS curriculum*)

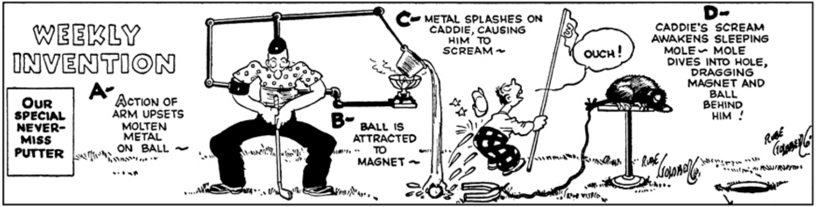
- Programming is a means of making ideas into reality using computers.
- What universities produce $\neq$ what industry needs.
- CS must emphasize software development (even at the expense of algorithmic complexity, data structures and...subsurface luminosity).
- Fashions come and go so rapidly that only a solid grasp of the fundamentals of CS and software development has lasting value. Industry wants software "developers" more than computer scientists and engineers.

Stroustrup's Philosophy

(A summary of S's article, *Programming in an undergraduate CS curriculum*)

- Programming is a means of making ideas into reality using computers.
- What universities produce $\neq$ what industry needs.
- CS must emphasize software development (even at the expense of algorithmic complexity, data structures and…subsurface luminosity).
- Fashions come and go so rapidly that only a solid grasp of the fundamentals of CS and software development has lasting value. Industry wants software "developers" more than computer scientists and engineers.
- Preferably, an understanding of programming extends to several kinds of languages (declarative, scripting, machine level) and applications (embedded systems, text manipulation, small commercial application, scientific computation); language bigots do not make good professionals.

Avoid Unprincipled Hacking!



For many, "programming" has become a strange combination of unprincipled hacking and invoking other people's libraries (with only the vaguest idea of what's going on). The notions of "maintenance" and "code quality" are at best purely academic. Consequently, many in industry despair over the difficulty of finding graduates who understand "systems" and "can architect software."

kluge – The OED Definition

kludge *slang* (orig. *U.S.*).

(klu:dʒ)

Also **kluge**.

[J. W. Granholm's jocular invention: see first quot.; cf. also BODGE *v.*, FUDGE *v.*]

‘An ill-assorted collection of poorly-matching parts, forming a distressing whole’ (Granholm); esp. in *Computing*, a machine, system, or program that has been improvised or ‘bodged’ together; a hastily improvised and poorly thought-out solution to a fault or ‘bug’.

1962 J. W. GRANHOLM in *Datamation* Feb. 30/1 The word ‘kludge’ is..derived from the same root as the German *Kluge*.., originally meaning ‘smart’ or ‘witty’... ‘Kludge’ eventually came to mean ‘not so smart’ or ‘pretty ridiculous’. *Ibid.* 30/2 The building of a Kludge..is not work for amateurs. There is a certain, indefinable, masochistic finesse that must go into true Kludge building. **1966** *New Scientist* 22 Dec. 699/1 Kludges are conceived of man's natural fallibility, nourished by his loyalty to erroneous opinion, and perfected by the human capacity to apply maximum effort only when proceeding in the wrong direction. **1976** *Electronic Design* 5 Jan. 120 The technique uses some kluge

Stroustrup's Three Modes of Exposition:

- Philisophy, "Blue: concepts and techniques" We characterize our approach as "depth-first." It is also "concrete-first" and "concept-based." First, we quickly (well, relatively quickly, Chapters 1–11) assemble a set of skills needed for writing small practical programs. In doing so, we present a lot of tools and techniques in minimal detail. We focus on simple concrete code examples because people grasp the concrete faster than the abstract. That's simply the way most humans learn. At this initial stage, you should not expect to understand every little detail. In particular, you'll find that trying something slightly different from what just worked can have "mysterious" effects. Do try, though! And please do the drills and exercises we provide. Just remember that early on you just don't have the concepts and skills to accurately estimate what's simple and what's complicated; expect surprises and learn from them.

Stroustrup's Three Modes of Exposition:

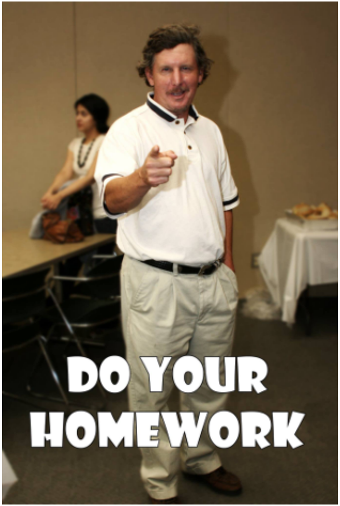
- Philisophy, "Blue: concepts and techniques" We characterize our approach as "depth-first." It is also "concrete-first" and "concept-based." First, we quickly (well, relatively quickly, Chapters 1–11) assemble a set of skills needed for writing small practical programs. In doing so, we present a lot of tools and techniques in minimal detail. We focus on simple concrete code examples because people grasp the concrete faster than the abstract. That's simply the way most humans learn. At this initial stage, you should not expect to understand every little detail. In particular, you'll find that trying something slightly different from what just worked can have "mysterious" effects. Do try, though! And please do the drills and exercises we provide. Just remember that early on you just don't have the concepts and skills to accurately estimate what's simple and what's complicated; expect surprises and learn from them.

- Practical perspective, "Green: advice" At the end of this book, will you be an expert at programming and at C++? Of course not! When done well, programming is a subtle, deep, and highly skilled art building on a variety of technical skills. You should no more expect to be an expert at programming in four months than you should expect to be an expert in biology, in math, in a natural language (such as Chinese, English, or Danish), or at playing the violin in four months — or in half a year, or a year. What you should hope for, and what you can expect if you approach this book seriously, is to have a really good start that allows you to write relatively simple useful programs, to be able to read more complex programs, and to have a good conceptual and practical background for further work.

Stroustrup's Three Modes of Exposition:

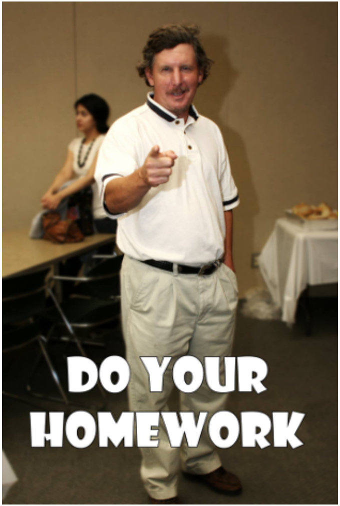
- Philisophy, "Blue: concepts and techniques" We characterize our approach as "depth-first." It is also "concrete-first" and "concept-based." First, we quickly (well, relatively quickly, Chapters 1–11) assemble a set of skills needed for writing small practical programs. In doing so, we present a lot of tools and techniques in minimal detail. We focus on simple concrete code examples because people grasp the concrete faster than the abstract. That's simply the way most humans learn. At this initial stage, you should not expect to understand every little detail. In particular, you'll find that trying something slightly different from what just worked can have "mysterious" effects. Do try, though! And please do the drills and exercises we provide. Just remember that early on you just don't have the concepts and skills to accurately estimate what's simple and what's complicated; expect surprises and learn from them.
- Practical perspective, "Green: advice" At the end of this book, will you be an expert at programming and at C++? Of course not! When done well, programming is a subtle, deep, and highly skilled art building on a variety of technical skills. You should no more expect to be an expert at programming in four months than you should expect to be an expert in biology, in math, in a natural language (such as Chinese, English, or Danish), or at playing the violin in four months — or in half a year, or a year. What you should hope for, and what you can expect if you approach this book seriously, is to have a really good start that allows you to write relatively simple useful programs, to be able to read more complex programs, and to have a good conceptual and practical background for further work.
- Cautionary tales, " Red: warning " [N]ever skip the drills, no matter how tempted you are; they are essential to the learning process. Just start with the first step and proceed, testing each step as you go to make sure you are doing it right.

- `do { yourHomework();`



`//Dr. Doug Macintire delivers the admonishment`

```
do { yourHomework();
```



//Dr. Doug Macintire delivers the admonishment

```
} while(!orElse);
```

Computer Science vis-à-vis Mathematics

- What would The Donald say?

Computer Science vis-à-vis Mathematics

- What would The Donald say?
- 

Donald Knuth, the “father of the analysis of algorithms”, is a computer scientist, mathematician, and professor emeritus at Stanford University who wrote the multi-volume work *The Art of Computer Programming*. The following quotes are from

Computer Science vis-à-vis Mathematics

- What would The Donald say?
- 

  Donald Knuth, the "father of the analysis of algorithms", is a computer scientist, mathematician, and professor emeritus at Stanford University who wrote the multi-volume work *The Art of Computer Programming*. The following quotes are from
- *Computer Science and Mathematics Newsletter*
  ACM SIGCSE Bulletin Homepage archive Volume 2 Issue 4, September-October 1970 Pages 19-29 ACM New York, NY, USA

Definition of Computer Science

- "My favorite way to describe computer science is to say that it is *the study of algorithms**."" cf Stroustrup's "making ideas into reality"
  - **Algorithm*: "a precisely-defined sequence of rules to produce specified output from given input in finite steps" or "1. a precise rule (or set of rules) specifying how to solve some problem."

Definition of Computer Science

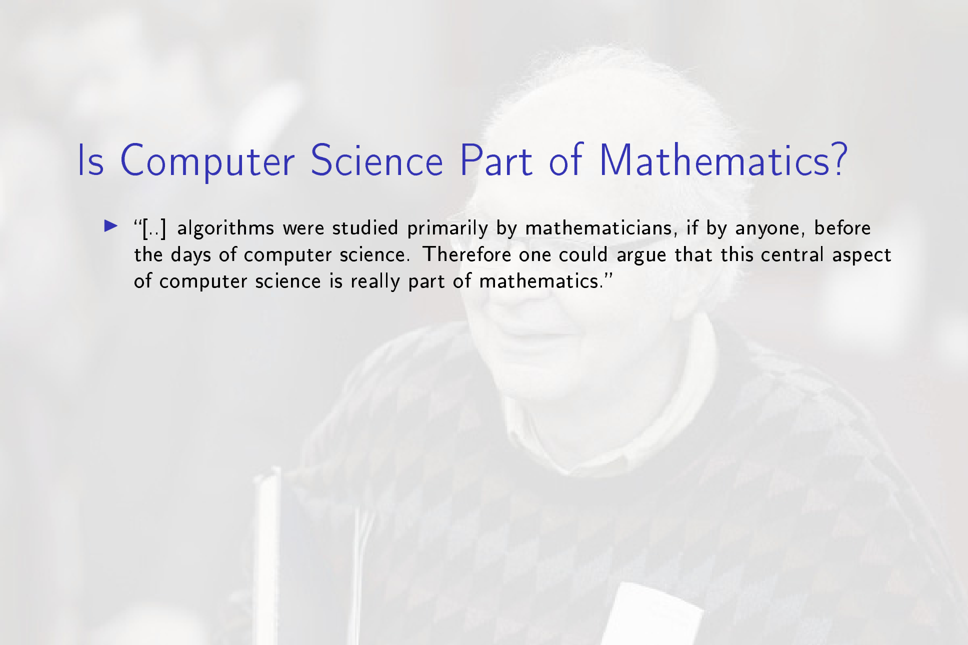
- "My favorite way to describe computer science is to say that it is *the study of algorithms**."" cf Stroustrup's "making ideas into reality"
  - **Algorithm*: "a precisely-defined sequence of rules to produce specified output from given input in finite steps" or "1. a precise rule (or set of rules) specifying how to solve some problem."
  - "A particular representation of an algorithm is called a program."
- *Algorithms* "are extraordinarily rich in interesting properties; and furthermore, an algorithmic point of view is a useful way to organize knowledge in general."

Definition of Computer Science

- “My favorite way to describe computer science is to say that it is _the study of algorithms*_."” cf Stroustrup’s “making ideas into reality”
  - *_Algorithm_: “a precisely-defined sequence of rules to produce specified output from given input in finite steps” or “1. a precise rule (or set of rules) specifying how to solve some problem.”
  - “A particular representation of an algorithm is called a program.”
- _Algorithms_ “are extraordinarily rich in interesting properties; and furthermore, an algorithmic point of view is a useful way to organize knowledge in general.”
- Forsythe: “the question ’What can be automated?’ is one of the most inspiring philosophical and practical questions of contemporary civilization”

Definition of Computer Science

- "My favorite way to describe computer science is to say that it is *the study of algorithms**."" cf Stroustrup's "making ideas into reality"
  - **Algorithm*: "a precisely-defined sequence of rules to produce specified output from given input in finite steps" or "1. a precise rule (or set of rules) specifying how to solve some problem."
  - "A particular representation of an algorithm is called a program."
- *Algorithms* "are extraordinarily rich in interesting properties; and furthermore, an algorithmic point of view is a useful way to organize knowledge in general."
- Forsythe: "the question 'What can be automated?' is one of the most inspiring philosophical and practical questions of contemporary civilization"
- "Computers are really necessary before we can learn much about the general properties of algorithms; human beings are not precise enough nor fast enough to carry out any but the simplest procedures."



Is Computer Science Part of Mathematics?

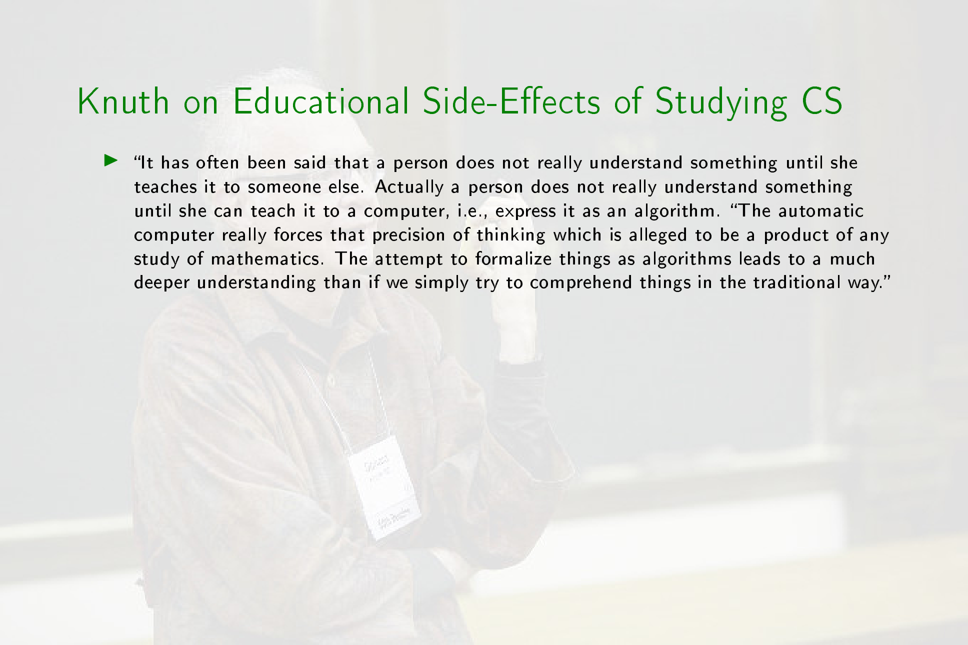
- "[..] algorithms were studied primarily by mathematicians, if by anyone, before the days of computer science. Therefore one could argue that this central aspect of computer science is really part of mathematics."

Is Computer Science Part of Mathematics?

- "[..] algorithms were studied primarily by mathematicians, if by anyone, before the days of computer science. Therefore one could argue that this central aspect of computer science is really part of mathematics."
- "Like mathematics, computer science will be somewhat different from the other sciences, in that it deals with man-made laws which can be proved, instead of natural laws which are never known with certainty [..]— mathematics dealing more or less with theorems, infinite processes, static relationships, and computer science dealing more or less with algorithms, finitary constructions, dynamic relationships."

Knuth on Educational Side-Effects of Studying CS

- “It has often been said that a person does not really understand something until she teaches it to someone else. Actually a person does not really understand something until she can teach it to a computer, i.e., express it as an algorithm. “The automatic computer really forces that precision of thinking which is alleged to be a product of any study of mathematics. The attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way.”



Knuth on Educational Side-Effects of Studying CS

- "It has often been said that a person does not really understand something until she teaches it to someone else. Actually a person does not really understand something until she can teach it to a computer, i.e., express it as an algorithm. "The automatic computer really forces that precision of thinking which is alleged to be a product of any study of mathematics. The attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way."
- "..the pedagogic value of an algorithmic approach [.] aids in the understanding of concepts of all kinds. A student who is properly trained in computer science is learning something which will implicitly help her cope with many other subjects; and therefore there will soon be good reason for saying that undergraduate computer science majors have received a good general education, just as we now believe this of undergraduate math majors. On the other hand, the present-day undergraduate courses in computer science are not yet fulfilling this goal; at least, I find that many beginning graduate students with an undergraduate degree in computer science have been more narrowly educated than I would like."

Strstr's Problem Solving Principles/Practices

- As you work on a problem you repeatedly go through these stages:
  1. Analysis: Figure out what should be done and write a description of your (current) understanding of that. Draw diagrams, solve simpler problems, develop invariants and write pseudocode and create structures needed to solve the problem.

Strstr's Problem Solving Principles/Practices

- As you work on a problem you repeatedly go through these stages:
  1. Analysis: Figure out what should be done and write a description of your (current) understanding of that. Draw diagrams, solve simpler problems, develop invariants and write pseudocode and create structures needed to solve the problem.
  2. Design: Create an overall structure for the system, deciding which parts the implementation should have and how those parts should communicate. As part of the design consider which tools – such as libraries – can help you structure the program.

- Compare with Polya's problem solving method:
  1. Understand the problem.
  2. Make a plan.

Strstr's Problem Solving Principles/Practices

- As you work on a problem you repeatedly go through these stages:
  1. Analysis: Figure out what should be done and write a description of your (current) understanding of that. Draw diagrams, solve simpler problems, develop invariants and write pseudocode and create structures needed to solve the problem.
  2. Design: Create an overall structure for the system, deciding which parts the implementation should have and how those parts should communicate. As part of the design consider which tools – such as libraries – can help you structure the program.
  3. Implementation: Write the code, debug it, and test that it actually does what it is supposed to do.
- Compare with Polya's problem solving method:
  1. Understand the problem.
  2. Make a plan.
  3. Carry out the plan.

Strstr's Problem Solving Principles/Practices

- As you work on a problem you repeatedly go through these stages:
  1. Analysis: Figure out what should be done and write a description of your (current) understanding of that. Draw diagrams, solve simpler problems, develop invariants and write pseudocode and create structures needed to solve the problem.
  2. Design: Create an overall structure for the system, deciding which parts the implementation should have and how those parts should communicate. As part of the design consider which tools – such as libraries – can help you structure the program.
  3. Implementation: Write the code, debug it, and test that it actually does what it is supposed to do.
- Compare with Polya's problem solving method:
  1. Understand the problem.
  2. Make a plan.
  3. Carry out the plan.
  4. L∞k back.

The Collatz ($3x + 1$) Problem at Nexus of Math & CS

- Richard Guy: "The $3x + 1$ sequences take a positive integer and iteratively apply the following rule: If a number is odd, triple it and add one; if even, halve it. The sequences produced by this rule always appear to reach an infinite string of $4, 2, 1, 4, 2, 1$, etc., and the problem is whether all sequences reach this *cycle*; that is, whether for all t_0, there is some n where $t_n = 1$. Here are some examples:

The Collatz $(3x+1)$ Problem at Nexus of Math & CS

- Richard Guy: "The $3x + 1$ sequences take a positive integer and iteratively apply the following rule: If a number is odd, triple it and add one; if even, halve it. The sequences produced by this rule always appear to reach an infinite string of $4, 2, 1, 4, 2, 1$, etc., and the problem is whether all sequences reach this *cycle*; that is, whether for all t_0, there is some n where $t_n = 1$. Here are some examples:
- $6, 3, 10, 5, 16, 8, 4, 2, 1, 4, \dots$

The Collatz $(3x + 1)$ Problem at Nexus of Math & CS

- Richard Guy: "The $3x + 1$ sequences take a positive integer and iteratively apply the following rule: If a number is odd, triple it and add one; if even, halve it. The sequences produced by this rule always appear to reach an infinite string of $4, 2, 1, 4, 2, 1$, etc., and the problem is whether all sequences reach this *cycle*; that is, whether for all t_0, there is some n where $t_n = 1$. Here are some examples:
- $6, 3, 10, 5, 16, 8, 4, 2, 1, 4, \dots$
- $17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \dots$

The Collatz $(3x+1)$ Problem at Nexus of Math & CS

- Richard Guy: "The $3x+1$ sequences take a positive integer and iteratively apply the following rule: If a number is odd, triple it and add one; if even, halve it. The sequences produced by this rule always appear to reach an infinite string of $4, 2, 1, 4, 2, 1$, etc., and the problem is whether all sequences reach this *cycle*; that is, whether for all t_0, there is some n where $t_n = 1$. Here are some examples:
- $6, 3, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- $17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- $30, 15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$

The Collatz $(3x + 1)$ Problem at Nexus of Math & CS

- Richard Guy: "The $3x + 1$ sequences take a positive integer and iteratively apply the following rule: If a number is odd, triple it and add one; if even, halve it. The sequences produced by this rule always appear to reach an infinite string of $4, 2, 1, 4, 2, 1$, etc., and the problem is whether all sequences reach this *cycle*; that is, whether for all t_0, there is some n where $t_n = 1$. Here are some examples:
- $6, 3, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- $17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- $30, 15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- For $c_0 \in \mathbb{N}$, iterate

$$c_{n+1} = \begin{cases} \dfrac{c_n}{2} & \text{if } c_n \text{ is even} \\ \dfrac{3 \cdot c_n + 1}{2} & \text{if } c_n \text{ is odd} \end{cases}$$

The Collatz $(3x + 1)$ Problem at Nexus of Math & CS

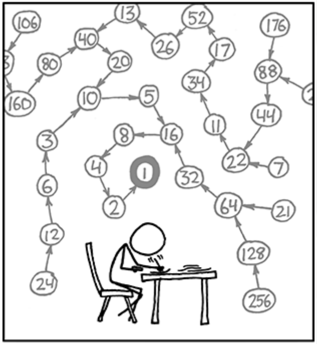
- Richard Guy: "The $3x + 1$ sequences take a positive integer and iteratively apply the following rule: If a number is odd, triple it and add one; if even, halve it. The sequences produced by this rule always appear to reach an infinite string of $4, 2, 1, 4, 2, 1$, etc., and the problem is whether all sequences reach this *cycle*; that is, whether for all t_0, there is some n where $t_n = 1$. Here are some examples:
- $6, 3, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- $17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- $30, 15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, \ldots$
- For $c_0 \in \mathbb{N}$, iterate

$$c_{n+1} = \begin{cases} \dfrac{c_n}{2} & \text{if } c_n \text{ is even} \\ \dfrac{3 \cdot c_n + 1}{2} & \text{if } c_n \text{ is odd} \end{cases}$$

- Erdős: "Mathematics is not yet ripe for such problems."

The Collatz Problem is an Unproven Conjecture!

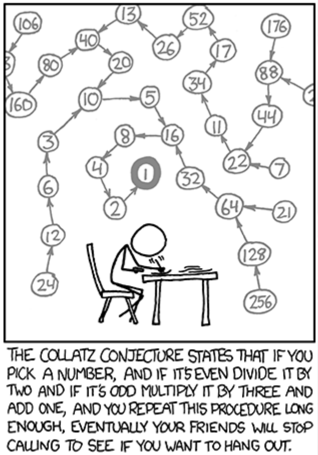
- Guy again: "Despite the simple rule, the paths of the sequences are rather unpredictable. Starting with 33 takes 26 steps and climbs to 100 before reaching 1, while 27 takes 111 steps and climbs to over 9000 before reaching 1. Such behavior has made this and other similar problems seem intractable; we cannot even show that such sequences could not go to infinity! As Lagarias introduces the problem in his $3x + 1$ compendium, he states that it touches number theory, ergodic theory, stochastic processes, and more, while not lying squarely in any of their domains."



THE COLLATZ CONJECTURE STATES THAT IF YOU PICK A NUMBER, AND IF IT'S EVEN DIVIDE IT BY TWO AND IF IT'S ODD MULTIPLY IT BY THREE AND ADD ONE, AND YOU REPEAT THIS PROCEDURE LONG ENOUGH, EVENTUALLY YOUR FRIENDS WILL STOP CALLING TO SEE IF YOU WANT TO HANG OUT.

The Collatz Problem is an Unproven Conjecture!

- Guy again: "Despite the simple rule, the paths of the sequences are rather unpredictable. Starting with 33 takes 26 steps and climbs to 100 before reaching 1, while 27 takes 111 steps and climbs to over 9000 before reaching 1. Such behavior has made this and other similar problems seem intractable; we cannot even show that such sequences could not go to infinity! As Lagarias introduces the problem in his $3x + 1$ compendium, he states that it touches number theory, ergodic theory, stochastic processes, and more, while not lying squarely in any of their domains."
- Note that the Collatz Conjecture is equivalent to the claim that, working backwards from 1, we get a binary tree spanning all of $\mathbb{N}$.



▶ The Collatz Conjecture

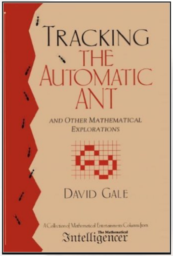
```cpp
/// Collatz Conjecture generalization
#include <iostream>
using namespace std;

long countIterations(long c) {
    long count=0;
    while (c!=1) {
        c = (c%2==0)? c/2 : (3*c+1)/2;   //ternary op
        ++count;
    }
    return count; //the number of iteration to 1
}

int main() {
    long maxIters{0}, maxIterSeed{0}, count{0}, sumIters{0};
    for(long c=2; c<=1000; c++) {
        count = countIterations(c);
        sumIters += count;
        if(maxIters<count) {
            maxIters = count;
            maxIterSeed = c;
        }
    }
    cout << endl << maxIterSeed << " produced a maximum of "
         << maxIters << " iterations.";
    cout << "\nThe average number of iterations is " << sumIters/1000;
}
```

```
871 produced a maximum of 113 iterations.
The average number of iterations is 39
Process returned 0 (0x0)   execution time : 0.006 s
```

Generalizing Collatz to Guy

- Guy sequences are a variation on Collatz sequences, as described in the chapter, Historic Conjectures: More Sequence Mysteries in the book, *Tracking the Automatic Ant, and Other Mathematical Explorations*, by David Gale. A Guy sequence here is defined as a sequence which uses the iterative function:

$$G_{n+1} = \begin{cases} \dfrac{3 \cdot G_n}{2} & \text{if } G_n \mod 2 == 0 \\ \dfrac{3 \cdot G_n + 1}{4} & \text{if } G_n \mod 4 == 1 \\ \dfrac{3 \cdot G_n - 1}{4} & \text{if } G_n \mod 4 == 3 \end{cases}$$



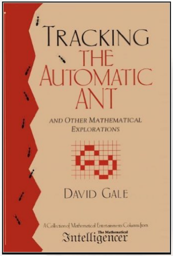
Questions:
1. What happens when the first value is 8? 2. What other first values lead to the same fate? 3. We have found at least three cycles, are there more? How many?

Generalizing Collatz to Guy

- Guy sequences are a variation on Collatz sequences, as described in the chapter, Historic Conjectures: More Sequence Mysteries in the book, *Tracking the Automatic Ant, and Other Mathematical Explorations*, by David Gale. A Guy sequence here is defined as a sequence which uses the iterative function:

$$G_{n+1} = \begin{cases} \frac{3 \cdot G_n}{2} & \text{if } G_n \mod 2 == 0 \\ \frac{3 \cdot G_n + 1}{4} & \text{if } G_n \mod 4 == 1 \\ \frac{3 \cdot G_n - 1}{4} & \text{if } G_n \mod 4 == 3 \end{cases}$$

- This leads to a very different dynamic. Using the following code we find sequences



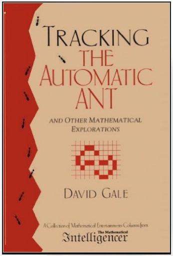
Questions:
1. What happens when the first value is 8? 2. What other first values lead to the same fate? 3. We have found at least three cycles, are there more? How many?

Generalizing Collatz to Guy

- Guy sequences are a variation on Collatz sequences, as described in the chapter, Historic Conjectures: More Sequence Mysteries in the book, *Tracking the Automatic Ant, and Other Mathematical Explorations*, by David Gale. A Guy sequence here is defined as a sequence which uses the iterative function:

$$G_{n+1} = \begin{cases} \dfrac{3 \cdot G_n}{2} & \text{if } G_n \mod 2 == 0 \\ \dfrac{3 \cdot G_n + 1}{4} & \text{if } G_n \mod 4 == 1 \\ \dfrac{3 \cdot G_n - 1}{4} & \text{if } G_n \mod 4 == 3 \end{cases}$$

- This leads to a very different dynamic. Using the following code we find sequences
- $2, 3, 2, 3, 2, 3, 2, 3, \ldots$



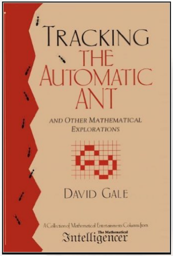
Questions: 1. What happens when the first value is 8? 2. What other first values lead to the same fate? 3. We have found at least three cycles, are there more? How many?

Generalizing Collatz to Guy

- Guy sequences are a variation on Collatz sequences, as described in the chapter, Historic Conjectures: More Sequence Mysteries in the book, *Tracking the Automatic Ant, and Other Mathematical Explorations*, by David Gale. A Guy sequence here is defined as a sequence which uses the iterative function:

$$G_{n+1} = \begin{cases} \dfrac{3 \cdot G_n}{2} & \text{if } G_n \mod 2 == 0 \\ \dfrac{3 \cdot G_n + 1}{4} & \text{if } G_n \mod 4 == 1 \\ \dfrac{3 \cdot G_n - 1}{4} & \text{if } G_n \mod 4 == 3 \end{cases}$$

- This leads to a very different dynamic. Using the following code we find sequences
- $2, 3, 2, 3, 2, 3, 2, 3, \ldots$
- $4, 6, 9, 7, 5, 4, 6, 9, 7, 5, 4, \cdots$



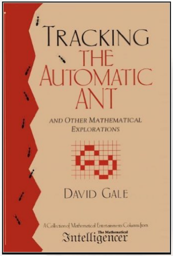
Questions: 1. What happens when the first value is 8? 2. What other first values lead to the same fate? 3. We have found at least three cycles, are there more? How many?

Generalizing Collatz to Guy

- Guy sequences are a variation on Collatz sequences, as described in the chapter, Historic Conjectures: More Sequence Mysteries in the book, *Tracking the Automatic Ant, and Other Mathematical Explorations*, by David Gale. A Guy sequence here is defined as a sequence which uses the iterative function:

$$G_{n+1} = \begin{cases} \frac{3 \cdot G_n}{2} & \text{if } G_n \mod 2 == 0 \\ \frac{3 \cdot G_n + 1}{4} & \text{if } G_n \mod 4 == 1 \\ \frac{3 \cdot G_n - 1}{4} & \text{if } G_n \mod 4 == 3 \end{cases}$$

- This leads to a very different dynamic. Using the following code we find sequences
- $2, 3, 2, 3, 2, 3, 2, 3, \ldots$
- $4, 6, 9, 7, 5, 4, 6, 9, 7, 5, 4, \cdots$
- $44, 66, 99, 74, 111, 83, 62, 93, 70, 105, 79, 59, 44, \cdots$



Questions: 1. What happens when the first value is 8? 2. What other first values lead to the same fate? 3. We have found at least three cycles, are there more? How many?

Guy Sequence Code

```
// The iterative function
long long nextGuy(long long G_n) {
    if(G_n%2==0) return 3*G_n/2;
    if(G_n%4==1) return (3*G_n+1)/4;
    else return (3*G_n-1)/4;
}
// generate the sequence
void gen(vector<long long>& seq, long long x) {
    seq.push_back(x);
    long long next = nextGuy(x);
    while(find(seq.begin(), seq.end(), next) == seq.end()) {
        seq.push_back(next);
        next = nextGuy(next);
    }
}
void print(vector<long long> v) {
    for(const long long &i : v) // access by const reference
        std::cout << i << ' ';
}
int main() {
    long long start{0};  // long long is a 64-bit int
    vector<long long> seq;
    cout << "\nEnter a starting value: ";
    while(cin >> start) {
        seq.clear();
        gen(seq, start);
        print(seq);
        cout << "\nEnter a starting value: ";
    }
}
```

Building a Mean Value Theorem Function - Part I

- A good project for students who are currently enrolled in precalculus or above is to experiment with the Mean Value Theorem components.

```
int main() {
    double x;
    vector<double> coeff = getPoly();
    cout << "\nInput x =  ";
    while(cin >> x) {
        cout << "\np(" << x << ") = " << poly(coeff,x);
        cout << "\nx = ";
    }
}
```

Building a Mean Value Theorem Function - Part I

- A good project for students who are currently enrolled in precalculus or above is to experiment with the Mean Value Theorem components.
- The first assignment could be as follows. Write a program that will prompt the user to enter the degree of a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots a_0$$

  and then the polynomial's coefficients. Divide the task into two functions: getPoly() which prompts the user for polynomial's degree and coefficients and returns the coefficients as a vector<double>. Then write a function poly(vector<double> coeff, double x which takes the coeff vector, an input x and returns the value of the polynomial at x.

```
int main() {
    double x;
    vector<double> coeff = getPoly();
    cout << "\nInput x =  ";
    while(cin >> x) {
        cout << "\np(" << x << ") = " << poly(coeff,x);
        cout << "\nx = ";
    }
}
```

Building a Mean Value Theorem Function - Part I

- A good project for students who are currently enrolled in precalculus or above is to experiment with the Mean Value Theorem components.
- The first assignment could be as follows. Write a program that will prompt the user to enter the degree of a polynomial

  $$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots a_0$$

  and then the polynomial's coefficients. Divide the task into two functions: getPoly() which prompts the user for polynomial's degree and coefficients and returns the coefficients as a vector<double>. Then write a function poly(vector<double> coeff, double x which takes the coeff vector, an input x and returns the value of the polynomial at x.
- Test that the functions work with a main() function like so:

```cpp
int main() {
    double x;
    vector<double> coeff = getPoly();
    cout << "\nInput x =  ";
    while(cin >> x) {
        cout << "\np(" << x << ") = " << poly(coeff,x);
        cout << "\nx = ";
    }
}
```

Building a Mean Value Theorem Function - Part II

- The solution might look like this:

```
vector<double> getPoly() {
    double deg{0};
    cout << "\nWhat is the degree of your polynomial? ";
    do {
        cin >> deg;
    } while(deg <= 0 || int(deg) != deg);
    vector<double> coeff(int(deg)+1);
    cout << "\nEnter the coefficients in ascending order: ";
    for(int i = 0; i < deg+1; ++i) {
        cout << "\nThe coefficient of x^" << i << " = ";
        cin >> coeff[i];
    }
    return coeff;
}
//evaluate poly in Horner's form:
double poly(vector<double> coeff, double x) {
    double value = coeff[coeff.size()-1];
    for(int i = coeff.size()-1; i > 0; --i) {
        value *= x;
        value += coeff[i-1];
    }
    return value;
} /*typical output
What is the degree of your polynomial? 2
Enter the coefficients in ascending order:
The coefficient of x^0 = 3
The coefficient of x^1 = 2
The coefficient of x^2 = 1
Input x =  1
p(1) = 6
x = 2
p(2) = 11 */
```

Building a Mean Value Theorem Function - Part III

- Next, you may want a function which gives the slope of the secant line between two points, and another function which approximates the slope of the tangent line.

```
int main() {
    double x;
    vector<double> coeff = getPoly();
    cout << "\nInput x =  ";
    while(cin >> x) {
        cout << "\np(" << x << ") = " << poly(coeff,x);
        cout << "\nx = ";
    }
}
```

Building a Mean Value Theorem Function - Part III

- Next, you may want a function which gives the slope of the secant line between two points, and another function which approximates the slope of the tangent line.
- Write a program that will prompt the user to enter two points, a and b, and returns the slope of the secant line connecting those points for your polynomial function.

$$m_{sec} = \frac{p(b) - p(a)}{b - a}$$

secant() computes the slope of line from the point on the polynomial where $x = a$ to the point where $x = b$. poly(vector<double> coeff, double x which takes the coeff vector, an input x and returns the value of the polynomial at x.

```
int main() {
    double x;
    vector<double> coeff = getPoly();
    cout << "\nInput x =  ";
    while(cin >> x) {
        cout << "\np(" << x << ") = " << poly(coeff,x);
        cout << "\nx = ";
    }
}
```

Building a Mean Value Theorem Function - Part III

- Next, you may want a function which gives the slope of the secant line between two points, and another function which approximates the slope of the tangent line.
- Write a program that will prompt the user to enter two points, a and b, and returns the slope of the secant line connecting those points for your polynomial function.

  $$m_{sec} = \frac{p(b) - p(a)}{b - a}$$

  `secant()` computes the slope of line from the point on the polynomial where $x = a$ to the point where $x = b$. `poly(vector<double> coeff, double x` which takes the `coeff` `vector`, an input `x` and returns the value of the polynomial at `x`.
- Test that the functions work with a `main()` function like so:

```
int main() {
    double x;
    vector<double> coeff = getPoly();
    cout << "\nInput x =  ";
    while(cin >> x) {
        cout << "\np(" << x << ") = " << poly(coeff,x);
        cout << "\nx = ";
    }
}
```

Building a Mean Value Theorem Function - Part IV

- The tangent function and a first shot at the Mean Value Theorem point function (`mvtPoint()`) are shown below. The strategy for `mvtPoint()` is walk through a sequence of secant lines of points relatively close together to see where the product `(m-secant(coeff,x,x+eps))*(m-secant(coeff,x+eps,x+2*eps))` changes sign.

```
double tangent(vector<double> coeff, double x, double epsilon) {
   return secant(coeff,x-epsilon/2, x+epsilon/2);
}
double mvtPoint(vector<double> coeff, double m, double a, double b) {
   double eps = (b-a)/1.e2;
   for(double x = a; x < b-2*eps; x += eps) {
      if((m-secant(coeff,x,x+eps))*(m-secant(coeff,x+eps,x+2*eps))<0) return x;
   }
}
int main() {
   double a, b, m, mvt, epsilon;
   vector<double> coeff = getPoly();
   cout << "\nInput two points for the slope of the secant line:";
   while(cin >> a >> b) {
      m = secant(coeff,a,b);
      cout << "\nThe secant from (" << a << ", " << poly(coeff,a)
           << ") to (" << b << ", " << poly(coeff,b) << ") = "
           << "\n(p(" << b << ") - P(" << a <<"))/(" << b << " - "
           << a << ") = " << m;
      mvt = mvtPoint(coeff,m,a,b);
      epsilon = (b-a)/1.e10;
      cout << "\nA point guaranteed by the MVT is near where x = " << mvt
           << "\nThe slope of the tangent line at x = " << mvt << " is "
           << tangent(coeff,mvt,epsilon);
       cout << "\nInput two points for the slope of the secant line:";
    }
}
```

Building a Mean Value Theorem Function - Part V

- Of course, if the student knows the power rule they could compute the derivative of the polynomial function that way, but the approach here is more general and doesn't rely on derivative rule shortcuts. The method demonstrated by the code here is not terrible accurate and could be improved in many ways...it may also fail, depending on the behavior of the polynomial, but it is a good starting point.

Building a Mean Value Theorem Function - Part V

- Of course, if the student knows the power rule they could compute the derivative of the polynomial function that way, but the approach here is more general and doesn't rely on derivative rule shortcuts. The method demonstrated by the code here is not terrible accurate and could be improved in many ways...it may also fail, depending on the behavior of the polynomial, but it is a good starting point.
- Suppose the student is investigating the MVT for $f(x) = x^3 - x^2$ on the intervals $[0, 1], [0, 2]$, and $[1, 2]$. The output would then be as shown below:

Building a Mean Value Theorem Function - Part V

- Of course, if the student knows the power rule they could compute the derivative of the polynomial function that way, but the approach here is more general and doesn't rely on derivative rule shortcuts. The method demonstrated by the code here is not terrible accurate and could be improved in many ways...it may also fail, depending on the behavior of the polynomial, but it is a good starting point.
- Suppose the student is investigating the MVT for $f(x) = x^3 - x^2$ on the intervals $[0, 1], [0, 2]$, and $[1, 2]$. The output would then be as shown below:

```
I:\Documents\cbprojects\meanValueTheorem\bin\Debug\meanValueTheorem.exe

What is the degree of your polynomial? 3

Enter the coefficients in ascending order:
The coefficient of x^0 = 0
The coefficient of x^1 = 0
The coefficient of x^2 = -1
The coefficient of x^3 = 1

Input two points for the slope of the secant line:0 1

The secant from (0, 0) to (1, 0) =
(p(1) - P(0))/(1 - 0) = 0
A point guaranteed by the mean value theorem is near where x = 0.66
The slope of the tangent line at x = 0.66 is -0.0131997
Input two points for the slope of the secant line:0 2

The secant from (0, 0) to (2, 4) =
(p(2) - P(0))/(2 - 0) = 2
A point guaranteed by the mean value theorem is near where x = 1.2
The slope of the tangent line at x = 1.2 is 1.92
Input two points for the slope of the secant line:1 2

The secant from (1, 0) to (2, 4) =
(p(2) - P(1))/(2 - 1) = 4
A point guaranteed by the mean value theorem is near where x = 1.53
The slope of the tangent line at x = 1.53 is 3.9627
Input two points for the slope of the secant line:
```

Stroustrup's Central Problem: A Calculator

- The first hint of the calculator appears in chapter 5, exercise 4:

5. Write a program that performs as a very simple calculator. Your calculator should be able to handle the four basic math operations — add, subtract, multiply, and divide — on two inpu values. Your program should prompt the user to enter three arguments: two **double** values a character to represent an operation. If the entry arguments are **35.6**, **24.1**, and **'+'**, the pro output should be **The sum of 35.6 and 24.1 is 59.7**. In Chapter 6 we look at a much more sophisticated simple calculator.

Stroustrup's Central Problem: A Calculator

- The first hint of the calculator appears in chapter 5, exercise 4:

5. Write a program that performs as a very simple calculator. Your calculator should be able to handle the four basic math operations — add, subtract, multiply, and divide — on two inpu values. Your program should prompt the user to enter three arguments: two **double** values a character to represent an operation. If the entry arguments are **35.6**, **24.1**, and **'+'**, the pro output should be **The sum of 35.6 and 24.1 is 59.7**. In Chapter 6 we look at a much more sophisticated simple calculator.

- The approach is "depth first" in the sense that it quickly moves through a series of basic techniques, concepts, and language supports before broadening out for a more complete understanding. The first 10 chapters (which Stroustrup does in about 6 weeks–but I took 15) cover objects, types and values, computation, debugging, error handling, the development of a "significant program" (a desk calculator).

Stroustrup's Central Problem: A Calculator

- The first hint of the calculator appears in chapter 5, exercise 4:

5. Write a program that performs as a very simple calculator. Your calculator should be able to handle the four basic math operations — add, subtract, multiply, and divide — on two inpu values. Your program should prompt the user to enter three arguments: two **double** values a character to represent an operation. If the entry arguments are **35.6**, **24.1**, and **'+'**, the pro output should be **The sum of 35.6 and 24.1 is 59.7**. In Chapter 6 we look at a much more sophisticated simple calculator.

- The approach is "depth first" in the sense that it quickly moves through a series of basic techniques, concepts, and language supports before broadening out for a more complete understanding. The first 10 chapters (which Stroustrup does in about 6 weeks–but I took 15) cover objects, types and values, computation, debugging, error handling, the development of a "significant program" (a desk calculator).
- The development of the calculator through redesign, extension of functionality, serves as a model of what it means to create a large, complex program.

Stroustrup's Central Problem: A Calculator

- **6.3.1 First attempt**
  "At this point, we are not really ready to write the calculator program. We simply haven't thought hard enough, but thinking is hard work and – like most programmers – we are anxious to write some code. So let's take a chance, write a simple calculator, and see where it leads us. The first idea is something like

Stroustrup's Central Problem: A Calculator

- **6.3.1 First attempt**
  "At this point, we are not really ready to write the calculator program. We simply haven't thought hard enough, but thinking is hard work and – like most programmers – we are anxious to write some code. So let's take a chance, write a simple calculator, and see where it leads us. The first idea is something like

```
int main()
{
    cout << "Please enter expression (we can handle + and -): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin>>lval>>op>>rval;      // read something like  1 + 3

    if (op=='+')
        res = lval + rval;    // addition
    else if (op=='-')
        res = lval - rval;    // subtraction

    cout << "Result: " << res << '\n';
    keep_window_open();
    return 0;
}
```

-

Stroustrup's Central Problem: A Calculator

- **6.3.1 First attempt**
  "At this point, we are not really ready to write the calculator program. We simply haven't thought hard enough, but thinking is hard work and – like most programmers – we are anxious to write some code. So let's take a chance, write a simple calculator, and see where it leads us. The first idea is something like

-
```
int main()
{
    cout << "Please enter expression (we can handle + and -): ";
    int lval = 0;
    int rval;
    char op;
    int res;
    cin>>lval>>op>>rval;      // read something like  1 + 3

    if (op=='+')
        res = lval + rval;    // addition
    else if (op=='-')
        res = lval - rval;    // subtraction

    cout << "Result: " << res << '\n';
    keep_window_open();
    return 0;
}
```

- To do:
  1. Clean up the code a bit
  2. Add multiplication and division (e.g., 2*3)
  3. Add the ability to handle more than one operand (e.g., 1+2+3)

Stroustrup's Central Problem: A Calculator

- After a few false starts and after correcting a few syntax and logic errors, we arrive at code at right:

```
int main()
try
{
    cout << "Enter expression (we can handle +, -, *, and /)
    int lval = 0;
    int rval;
    char op;
    cin>>lval;              // read leftmost operand
    if (!cin) error("no first operand");
    while (cin>>op) {  // read operator and right-hand op re
        cin>>rval;
        if (!cin) error("no second operand");
        switch(op) {
        case '+':
            lval += rval;    // add: lval = lval + rval
            break;
        case '-':
            lval -= rval;    // subtract: lval = lval - rval
            break;
        case '*':
            lval *= rval;    // multiply: lval = lval * rval
            break;
        case '/':
            lval /= rval;    // divide: lval = lval / rval
            break;
        default:       // not another operator: print result
            cout << "Result: " << lval << '\n';
            keep_window_open();
            return 0;
        }
    }
    error("bad expression");
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    return 2;
}
```

Stroustrup's Central Problem: A Calculator

- After a few false starts and after correcting a few syntax and logic errors, we arrive at code at right:
- This isn't bad, but then we try 1+2*3 and see that the result is 9 and not the 7 our arithmetic teachers told us was the right answer. Similarly, 1–2*3 gives –3 rather than the –5 we expected. We are doing the operations in the wrong order: 1+2*3 is calculated as (1+2)*3 rather than as the conventional 1+(2*3). Similarly, 1–2*3 is calculated as (1–2)*3 rather than as the conventional 1–(2*3). Bummer!

```
int main()
try
{
    cout << "Enter expression (we can handle +, -, *, and /)
    int lval = 0;
    int rval;
    char op;
    cin>>lval;              // read leftmost operand
    if (!cin) error("no first operand");
    while (cin>>op) {  // read operator and right-hand op re
        cin>>rval;
        if (!cin) error("no second operand");
        switch(op) {
        case '+':
            lval += rval;    // add: lval = lval + rval
            break;
        case '-':
            lval -= rval;    // subtract: lval = lval - rval
            break;
        case '*':
            lval *= rval;    // multiply: lval = lval * rval
            break;
        case '/':
            lval /= rval;    // divide: lval = lval / rval
            break;
        default:       // not another operator: print result
            cout << "Result: " << lval << '\n';
            keep_window_open();
            return 0;
        }
    }
    error("bad expression");
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    return 2;
}
```

Stroustrup's Central Problem: A Calculator

- After a few false starts and after correcting a few syntax and logic errors, we arrive at code at right:
- This isn't bad, but then we try 1+2*3 and see that the result is 9 and not the 7 our arithmetic teachers told us was the right answer. Similarly, 1–2*3 gives –3 rather than the –5 we expected. We are doing the operations in the wrong order: 1+2*3 is calculated as (1+2)*3 rather than as the conventional 1+(2*3). Similarly, 1–2*3 is calculated as (1–2)*3 rather than as the conventional 1–(2*3). Bummer!
- So (somehow), we have to "look ahead" on the line to see if there is a * (or a /). If so, we have to (somehow) adjust the evaluation order from the simple and obvious left-to-right order. Unfortunately, trying to barge ahead here, we immediately hit a couple of snags.

```
int main()
try
{
    cout << "Enter expression (we can handle +, -, *, and /)
    int lval = 0;
    int rval;
    char op;
    cin>>lval;              // read leftmost operand
    if (!cin) error("no first operand");
    while (cin>>op) {   // read operator and right-hand op re
        cin>>rval;
        if (!cin) error("no second operand");
        switch(op) {
        case '+':
            lval += rval;    // add: lval = lval + rval
            break;
        case '-':
            lval -= rval;    // subtract: lval = lval - rval
            break;
        case '*':
            lval *= rval;    // multiply: lval = lval * rval
            break;
        case '/':
            lval /= rval;    // divide: lval = lval / rval
            break;
        default:       // not another operator: print result
            cout << "Result: " << lval << '\n';
            keep_window_open();
            return 0;
        }
    }
    error("bad expression");
}
catch (exception& e) {
    cerr << "error: " << e.what() << '\n';
    return 1;
}
catch (...) {
    cerr << "Oops: unknown exception!\n";
    return 2;
}
```

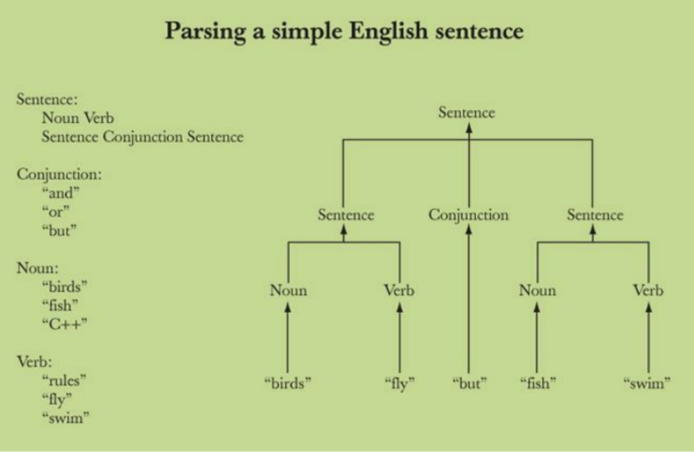
Parsing Tokens

- In *linguistics* a token is an individual occurrence of a linguistic unit in speech or writing, like a particular noun or verb.

Parsing Tokens

- In *linguistics* a token is an individual occurrence of a linguistic unit in speech or writing, like a particular noun or verb.
- In *computing* a token is the smallest meaningful unit of information in a sequence of data for a compiler, like a number or an operation.

Parsing Tokens

- In *linguistics* a token is an individual occurrence of a linguistic unit in speech or writing, like a particular noun or verb.
- In *computing* a token is the smallest meaningful unit of information in a sequence of data for a compiler, like a number or an operation.
- 

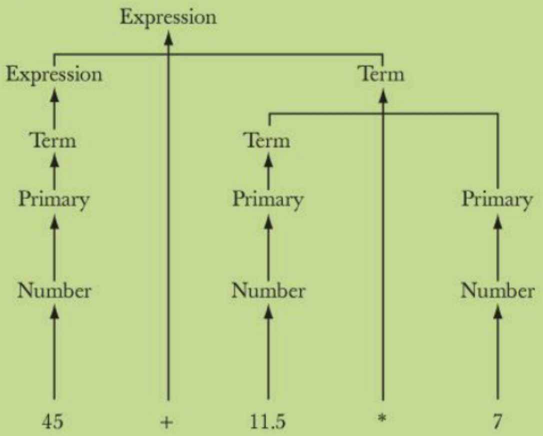
Parsing the expression 45 + 11.5 * 7

Expression:
- Term
- Expression "+" Term
- Expression "–" Term

Term:
- Primary
- Term "*" Primary
- Term "/" Primary
- Term "%" Primary

Primary:
- Number
- "(" Expression ")"

Number:
- floating-point-literal



Try This

- **Try This**

  This first version of the calculator program (including **get_token()**) is available as file **calculator00.cpp**. Get it to run and try it out.

Try This

- **Try This**

  This first version of the calculator program (including **get_token()**) is available as file **calculator00.cpp**. Get it to run and try it out.

- Unsurprisingly, this first version of the calculator doesn't work quite as we expected. So we shrug and ask, "Why not?" or rather, "So, why does it work the way it does?" and "What does it do?" Type a 2 followed by a newline. No response. Try another newline to see if it's asleep. Still no response. Type a 3 followed by a newline. It answers 2!

Try This

- **Try This**

  This first version of the calculator program (including **get_token()**) is available as file **calculator00.cpp**. Get it to run and try it out.

- Unsurprisingly, this first version of the calculator doesn't work quite as we expected. So we shrug and ask, "Why not?" or rather, "So, why does it work the way it does?" and "What does it do?" Type a 2 followed by a newline. No response. Try another newline to see if it's asleep. Still no response. Type a 3 followed by a newline. It answers 2!

```
class Token_stream {  //an
public:
    Token_stream();   // make a Token_stream to reads from cin
    Token get();     //get Token (get() is defined elsewhere)
    void putback(Token t);     // put a Token back
    void ignore(char c);    //discard tokens up a c
private:
    bool full;          // is there a Token in the buffer?
    Token buffer;    // we keep a Token using putback()
};
```

Program Structure

- Sometimes, the proverb says, it's hard to see the forest for the trees. Similarly, it is easy to lose sight of a program when looking at all its functions, classes, etc. So, let's have a look at the program with its details omitted:

Program Structure

- Sometimes, the proverb says, it's hard to see the forest for the trees. Similarly, it is easy to lose sight of a program when looking at all its functions, classes, etc. So, let's have a look at the program with its details omitted:

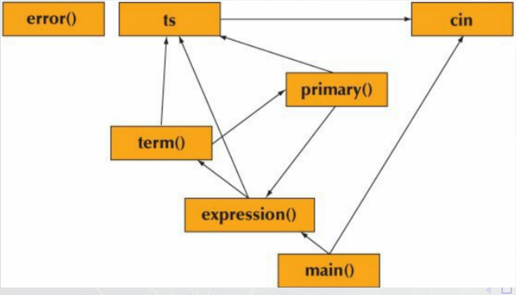
```
#include "std_lib_facilities.h"
class Token { /* . . . */ };
class Token_stream { /* . . . */ };
void Token_stream::putback(Token t) { /* . . . */ }
Token Token_stream::get() { /* . . . */ }

Token_stream ts;              // provides get() and putback()
double expression()           // declaration so that primary() can call expression()

double primary() { /* . . . */ }       // deal with numbers and parentheses
double term() { /* . . . */ }          // deal with * and /
double expression() { /* . . . */ }    // deal with + and –
```

-
```
int main() { /* . . . */ }    // main loop and deal with errors
```

Program Structure

- Sometimes, the proverb says, it's hard to see the forest for the trees. Similarly, it is easy to lose sight of a program when looking at all its functions, classes, etc. So, let's have a look at the program with its details omitted:

```
#include "std_lib_facilities.h"
class Token { /* . . . */ };
class Token_stream { /* . . . */ };
void Token_stream::putback(Token t) { /* . . . */ }
Token Token_stream::get() { /* . . . */ }

Token_stream ts;            // provides get() and putback()
double expression()         // declaration so that primary() can call expression()

double primary() { /* . . . */ }       // deal with numbers and parentheses
double term() { /* . . . */ }          // deal with * and /
double expression() { /* . . . */ }    // deal with + and –

int main() { /* . . . */ }     // main loop and deal with errors
```

- 

-

Cellular Automata and Conway's "Life"

1. **Survivals.** Every counter with two or three neighboring counters survives for the next generation.

Cellular Automata and Conway's "Life"

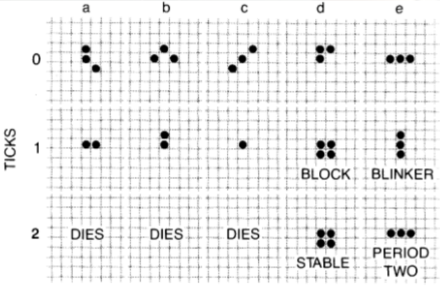
1. **Survivals.** Every counter with two or three neighboring counters survives for the next generation.
2. **Deaths.** Each counter with four or more neighbors dies (is removed) from overcrowding. Every counter with one neighbor or none dies from isolation.

Cellular Automata and Conway's "Life"

1. **Survivals.** Every counter with two or three neighboring counters survives for the next generation.
2. **Deaths.** Each counter with four or more neighbors dies (is removed) from overcrowding. Every counter with one neighbor or none dies from isolation.
3. **Births.** Each empty cell adjacent to exactly three neighbors-no more, no fewer-is a birth cell.

Cellular Automata and Conway's "Life"

1. **Survivals.** Every counter with two or three neighboring counters survives for the next generation.
2. **Deaths.** Each counter with four or more neighbors dies (is removed) from overcrowding. Every counter with one neighbor or none dies from isolation.
3. **Births.** Each empty cell adjacent to exactly three neighbors-no more, no fewer-is a birth cell.



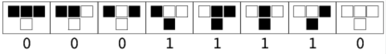
4. The fate of five triplets in "life"

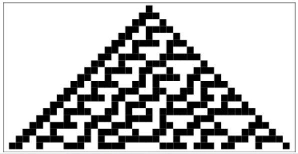
Cellular Automata and Conway's "Life"

- The most immediate practical application of cellular automata theory, Banks believes, is likely to be the design of circuits capable of self-repair or the wiring of any specified type of new circuit. No one can say how significant the theory may eventually become for the physical and biological sciences. It may have important bearings on cell growth in embryos, the replication of DNA molecules, the operation of nerve nets, genetic changes in evolving populations and so on. Analogies with life processes are impossible to resist. If a primordial broth of amino acids is large enough, and there is sufficient time, self-replicating, moving automata may result from complex transition rules built into the structure of matter and the laws of nature. There is even the possibility that space-time itself is granular, composed of discrete units, and that the universe, as Fredkin and others have suggested, is a vast cellular automaton run by an enormous computer. If so, what we call motion may be only simulated motion. A moving spaceship, on the ultimate microlevel, may be essentially the same as one of Conway's spaceships, appearing to move on the macrolevel whereas actually there is only an alteration of states of basic space-time cells in obedience to transition rules that have not yet been discovered.

Cellular Automata and Conway's "Life"

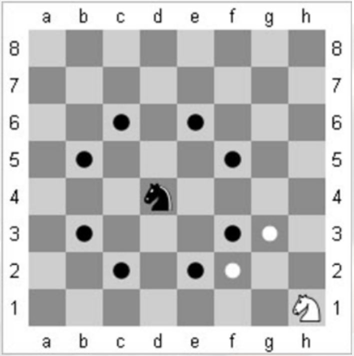
- The most immediate practical application of cellular automata theory, Banks believes, is likely to be the design of circuits capable of self-repair or the wiring of any specified type of new circuit. No one can say how significant the theory may eventually become for the physical and biological sciences. It may have important bearings on cell growth in embryos, the replication of DNA molecules, the operation of nerve nets, genetic changes in evolving populations and so on. Analogies with life processes are impossible to resist. If a primordial broth of amino acids is large enough, and there is sufficient time, self-replicating, moving automata may result from complex transition rules built into the structure of matter and the laws of nature. There is even the possibility that space-time itself is granular, composed of discrete units, and that the universe, as Fredkin and others have suggested, is a vast cellular automaton run by an enormous computer. If so, what we call motion may be only simulated motion. A moving spaceship, on the ultimate microlevel, may be essentially the same as one of Conway's spaceships, appearing to move on the macrolevel whereas actually there is only an alteration of states of basic space-time cells in obedience to transition rules that have not yet been discovered.
- Wolfram's cellular automata:





Knight's Tours

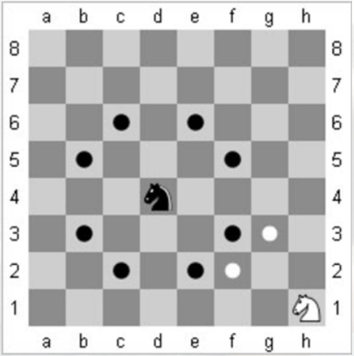
- A knight's path is the path a knight takes in moving around the chess board. In general, a knight is known to move from its current position on a chess board to a new position by either going up or down 1 or 2 and then going left or right 2 or 1, making an "L" shape which is 1 square in one direction and 2 squares in the other direction. So a black knight on an a standard chess board at column d and row 4 (as shown) can move to 8 positions (black circles), while the white knight in the corner at h1 has only two moves (white circles)



1. a knight's tour visits each square exactly once and

Knight's Tours

- A knight's path is the path a knight takes in moving around the chess board. In general, a knight is known to move from its current position on a chess board to a new position by either going up or down 1 or 2 and then going left or right 2 or 1, making an "L" shape which is 1 square in one direction and 2 squares in the other direction. So a black knight on an a standard chess board at column d and row 4 (as shown) can move to 8 positions (black circles), while the white knight in the corner at h1 has only two moves (white circles)



- Special knight's tours:

1. a knight's tour visits each square exactly once and
2. a knight's circuit visits every square exactly once and then can return to the original square on the last move

Knight's Tours

- The first project is to create an interface to
  1. Get the dimensions of the board from the user.

Knight's Tours

- The first project is to create an interface to
  1. Get the dimensions of the board from the user.
  2. Get the initial position of the knight.

- What is the probability that a knight making random moves will complete tour?

Knight's Tours

- The first project is to create an interface to

  1. Get the dimensions of the board from the user.
  2. Get the initial position of the knight.
  3. Display the board as a rectangular array showing '0' for unvisited squares and the number of the move on the visited squares: '1','2',...

- What is the probability that a knight making random moves will complete tour?
- How many different tours are there?

Knight's Tours

- The first project is to create an interface to
  1. Get the dimensions of the board from the user.
  2. Get the initial position of the knight.
  3. Display the board as a rectangular array showing '0' for unvisited squares and the number of the move on the visited squares: '1','2',...
  4. Get a legal move.

- What is the probability that a knight making random moves will complete tour?
- How many different tours are there?
- Modify the code to move on a toroidal chess board.

Knight's Tours

- The first project is to create an interface to
  1. Get the dimensions of the board from the user.
  2. Get the initial position of the knight.
  3. Display the board as a rectangular array showing '0' for unvisited squares and the number of the move on the visited squares: '1','2',...
  4. Get a legal move.
  5. Update the board and got to step 3.

- What is the probability that a knight making random moves will complete tour?
- How many different tours are there?
- Modify the code to move on a toroidal chess board.

Babylonian Basins of Attraction for $z^n = 1$

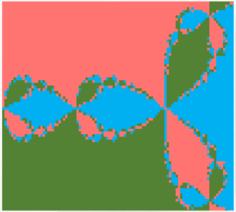
- We have seen that the Babylonian algorithm iterates

$$x_{n+1} \leftarrow \frac{x_n + A/x_n}{2}$$

for convergence to $\sqrt{A}$. This can be generalized to cube roots and so on using the iterative formula

$$x_{n+1} \leftarrow \frac{(k-1)x_n + A/x_n^{k-1}}{k}$$

for convergence to a kth root of a complex number, A.



Babylonian Basins of Attraction for $z^n = 1$

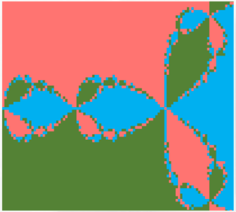
- We have seen that the Babylonian algorithm iterates

$$x_{n+1} \leftarrow \frac{x_n + A/x_n}{2}$$

for convergence to $\sqrt{A}$. This can be generalized to cube roots and so on using the iterative formula

$$x_{n+1} \leftarrow \frac{(k-1)x_n + A/x_n^{k-1}}{k}$$

for convergence to a kth root of a complex number, A.



- In the complex plane, the kth roots unity ($\sqrt[k]{1}$) are evenly distributed around the unit circle. For example, if z is a complex number then $z^3 = 1$ has three different solutions which can be found algebraically by solving

$$z^3 = 1$$
$$z^3 - 1 = 0$$
$$(z-1)(z^2+z+1) = 0$$

Mastermind

Implement a little guessing game called (for some obscure reason) “Bulls and Cows.” The program has a vector of four different integers in the range 0 to 9 (e.g., 1234 but not 1122) and it is the user’s task to discover those numbers by repeated guesses. Say the number to be guessed is 1234 and the user guesses 1359; the response should be “1 bull and 1 cow” because the user got one digit (1) right and in the right position (a bull) and one digit (3) right but in the wrong position (a cow). The guessing continues until the user gets four bulls, that is, has the four digits correct and in the correct order.

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour
4. Sam Lloyd's Fifteen Puzzle

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour
4. Sam Lloyd's Fifteen Puzzle
5. Beginning AI in Solving the Fifteen Puzzle

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour
4. Sam Lloyd's Fifteen Puzzle
5. Beginning AI in Solving the Fifteen Puzzle
6. Word Ladders (doublets)

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour
4. Sam Lloyd's Fifteen Puzzle
5. Beginning AI in Solving the Fifteen Puzzle
6. Word Ladders (doublets)
7. Word Squares

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour
4. Sam Lloyd's Fifteen Puzzle
5. Beginning AI in Solving the Fifteen Puzzle
6. Word Ladders (doublets)
7. Word Squares
8. Happy and Sad Numbers

Directed Student Projects

1. Generalized Collatz Conjecture: Guy Sequences
2. Ramsey Theory with Vectors
3. Queen's Tour
4. Sam Lloyd's Fifteen Puzzle
5. Beginning AI in Solving the Fifteen Puzzle
6. Word Ladders (doublets)
7. Word Squares
8. Happy and Sad Numbers
9. Bitwise Calculator

!(The End)