What is computational thinking?
Computational thinking is a high level thought process that considers the world in computational terms. It begins with learning to see opportunities to compute something, and it develops to include such considerations as computational complexity, utility of approximate solutions, computational resource implications of different algorithms, selection of appropriate data structures and the ease of coding, maintaining, and using the resulting program. Computational thinking is applicable across disciplinary domains because it takes place at a level of abstraction where similarities and differences can be seen in terms of the computational strategies available. A person skilled in computational thinking is able to harness the power of computing to gain insights. At its best, computational thinking is multi-disciplinary and cross-disciplinary thinking with an emphasis on the benefits of computational strategies to augment human insights. Computational thinking is a way of looking at the world in terms of how information can be generated, related, analyzed, represented, and shared.

The International Society for Technology in Education (ISTE) and the Computer Science Teachers Association (CSTA) have collaborated with leaders from higher education, industry, and K–12 education to develop an operational definition of computational thinking.

Computational thinking (CT) is a problem-solving process that includes (but is not limited to) the following characteristics:

• Formulating problems in a way that enables us to use a computer and other tools to help solve them.
• Logically organizing and analyzing data
• Representing data through abstractions such as models and simulations
• Automating solutions through algorithmic thinking (a series of ordered steps)
• Identifying, analyzing, and implementing possible solutions with the goal of achieving the most efficient and effective combination of steps and resources
• Generalizing and transferring this problem solving process to a wide variety of problems

These skills are supported and enhanced by a number of dispositions or attitudes that are essential dimensions of CT. These dispositions or attitudes include:
• Confidence in dealing with complexity
• Persistence in working with difficult problems
• Tolerance for ambiguity
• The ability to deal with open ended problems
• The ability to communicate and work with others to achieve a common goal or solution.

**Computational thinking in this module:**

• Describe the problem you want to solve;
• Logically organize;
• Analyze the problem, think of different ways to solve it, and think of ways to use computers to help solve it - think outside the box;
• Use abstraction, estimates, and simplifying assumptions to formulate a model to help solve the problem;
• Use spreadsheets as tools;
• Implement various solutions with the goal of achieving the most effective, and (sometimes) most efficient solution;
• Construct ways to handle uncertainty in the solution to increase your confidence in the solution.

**Types of classes where this module can be used:**

This module could be used in a number of different classes in grades 10-12, many of which may have the flexibility in curriculum to incorporate non-standard materials. Examples of potential target classrooms include those that discuss uses of technology.
These include computer science, mathematics, any science or social science class, and business classes.

**Prerequisites** Students should have had algebra and geometry, simple exponentials.

**About the VCTAL project and its computational thinking modules**

The VCTAL Project is a collaboration among the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) at Rutgers University, the Consortium for Mathematics and its Applications (COMAP), Colorado State University, Hobart and William Smith Colleges, the Computer Science Teachers Association (CSTA), and a number of school districts around the country. The project is funded by the National Science Foundation (NSF) to develop a set of modules and mini-modules (or teasers) for use in high school classrooms to help cultivate a facility with computational thinking in students across different grade levels and subject areas. The modules are intended to provide 4-6 days of classroom activities on a variety of topics that bring in computing and computational thinking but are drawn from applications in everyday life. Our goal is to show broad applicability of computational thinking and uses if it that extend well beyond programming. Each module will have one or more associated mini-modules or “teaser” modules. The mini-modules are intended to give a one- or two-day introduction to a topic that is either covered in or related to the larger module. Mini-modules give teachers greater ability to include computational thinking in an already-packed curriculum and allow them to experiment in a limited way.

**Recursion** can be used in many areas, from modeling population growth or the spread of disease to determining how much money you will have in an account at retirement. The purpose of the module is to provide a general background on the process of recursion, a method for solving problems where the solution to a problem depends on solutions to smaller instances of the same problem. A recursive process is one in which objects (a whole system) are defined in terms of other objects (stages of the system) of the same type. The whole system can be built knowing only a few initial values or stage of the system and applying a small set of rules. Computers routinely use recursion in performing many standard operations or processes.
Teacher notes are all in red.

**Section 1: Introduction to Recursion**

**Driving Question:** How does one think recursively?

This section introduces recursion with a number of specific examples. By the end of the section you should be able to trace the recursive steps used to solve the problems, begin to think recursively, and understand basic recursion.

**Initial Question:**
Do you know what recursion is?

Some students who have had algebra 2 may have a vague idea and give an example of a sequence.

Are there sports examples? Running races, savings accounts, things that grow at a certain rate etc.

**Activity 1.1:**
Consider having students work on this problem in groups.

Your grandfather started a college savings plan for you when you were born in 2000. He put $5,000.00 in the account to start it in 2000, year 0. The account promised a return of 7% annually. How much is in the account in 2001, on your birthday? How about when you start grade one in 2006? And in 2014 when you start high school? How much is in the account in 2018 when you start college? Explain how you arrived at your answers.

Let students pose various ways of solving the problem. Eventually move them to considering a relationship that expresses one year’s amount on the preceding year.

A useful approach to getting students there might be to have them fill in a table with the account balance at the end of each year from 2000 to 2014.

In 2000 you have $5,000.00.

Let that be denoted \( C(0) \) cash at time 0 (in 2000). So \( C(0) = 5000 \)

In 2001 you have \( C(1) = 5,000 + 0.07C(0) = 5,000 + 0.07(5,000.00) = 5,000 + 400 = 5,400.00 \)

So after 1 year we can denote the amount by \( C(1) \). Thus \( C(1) = 5,400.00 \)

Note this is the same as \( C(1) = 1.07(5000) = 5,400 \)
C(2) = C(1) + 0.07C(1) = 5,400 + 0.07(5400) = 5,400 + 432 = 5,832.
But this could be written as C(2) = 1.07C(1) = 1.07(5,400) = 1.07(1.07C(0)) = 1.07^2(5000)

If your students have not had much work with exponents, skip the conversion to exponents and simple write one amount in terms of the preceding amount.

Continue to compute the amount in year 2003, 2004, etc. and discover that
C(3) = 1.07(C(2)) = 1.07^3(5000). Do this until 2018, as tedious as it might be, where
C(18) = 1.07C(17) = 1.07^{18}C(0) = 1.07^{18}(5000).

Let C(t) denote the amount in year t where t is years after 2000. Thus
C(0) = 5000
C(1) = 1.07C(0) = 1.07(5000)
C(2) = 1.07C(1) = 1.07(1.07)5000 = 1.07^2(5000)
C(3) = 1.07C(2) = 1.07(1.07(C(1))) = 1.07(1.07(1.07(5000)))) = 1.07^3(5000)
C(t) = 1.07C(t-1) = 1.07(1.07C(t-2))) = 1.07^2C(t-2)

Note that we can also write the above as
Do more of these until the student gets the idea and can come up with the expression in terms of C(0) and ultimately get C(t) = 1.07^tC(0) = 1.07^t(5000)
C(18) = 1.07^{18}C(0) = 1.07^{18}(5000)

Activity 1.2
Your grandmother is even more generous and decides to put $5000 in another college savings account at the start of each year until you start college at 18. This account also earns 7% interest each year. How much is in the account in 2001, on your birthday? How about when you start grade one in 2006? And in 2014 when you start high school? How much is in the account in 2018 when you start college? Explain how you arrived at your answers.

Write an expression for the amount in terms of the amount in 2017. Write an expression for the amount in terms of the amount in 2000.

Let students pose various ways of solving the problem.

In 2000 you have an amount of $5,000.00. Let that be denoted A(0), the amount at time 0 (in 2000). So A(0) = 5000.
In 2001 you have \( A(1) = C(0) + 0.07C(0) + 5000 = 5,000 + 0.07(5,000.00) + \) an additional 5000

\[ = 5,000 + 400 + 5,000.00 = 10,400 \]

or

\[ A(1) = 1.07A(0) + 5000 = 5,400 + 5000 = 10,400 \]

Skip the part with exponents if you wish.

\[ A(2) = 1.07A(1) + 5000 = 1.07(10,400) + 5000 \]

or

\[ A(2) = A(1) + 0.07A(1) + 5000 = 10,400 + 0.07(10,400) + 5000 = 1.07(10,400) + 5000 = 11,232 + 5000 = 16,232. \]

But this could be written as

\[ A(2) = 1.07(1.07(5000) + 5000) + 5000 = 1.07^2(5000) + 1.07^1(5000) + 1.07^0(5000) \]

Continue to compute the amount in year 2003, 2004, etc. and discover that

\[ A(3) = 1.07 A(2) + 5000 \]

Do this until 2018.

\[ A(18) = 1.07 A(17) + 5000 \]

Let \( A(t) \) denote the amount in year \( t \) where \( t \) is years after 2000. Thus

\[ A(0) = 5000 \]

\[ A(1) = 1.07 A(0) + 5000 \]

\[ A(2) = 1.07 A(1) + 5000 \]

\[ A(3) = 1.07 A(2) + 5000 \]

\[ A(t) = 1.07 A(t-1) + 5000 \]

A possible approach to aiding student understanding is to use a spreadsheet program to do the calculations for each year, perhaps after they have worked through the first two problems on their own.

**Activity 1.3:**
Groupon’s business model has been to offer reduced prices on items when they get enough people to sign up to buy them. This model became so popular that companies like Living Social and Amazon tried their hand at it. Groupon started 2012 with a cash account of $1.2 billion dollars. Cash was depleted 61% during 2012. (1) How much cash remains after 10 years (in 2022), assuming it is depleted at the same rate each year? At that rate will they ever use up all of their available cash?

They also report 23% annual growth in income over the same period. (2) Assuming they started 2012 with $1,600,000,000 and they sustain this growth, what is the income in year 2022?

Let students pose various ways of solving the first problem of how much cash is on hand in 10 years. Eventually move them to considering a recursive relationship. Point out, if needed, that if cash is depleted by 61% then the cash on hand is 39% of the initial amount, or .39 C(0)

\[ C(0) = 1,200,000,000, \text{ the amount at the beginning of 2012} \]
\[ C(1) = 1,200,000,000 - .61(1,200,000,000) = (1-.61)(1,200,000,000) = .39(1,200,000,000), \text{ the amount at the beginning of 2013}. \]

(1) Let \( C(t) \) denote the cash in year \( t \) where \( t \) is years after 2012.

\[ C(0) = 1,200,000,000 \text{ at the outset in 2012}. \]
\[ C(1) = 1,200,000,000 - .61(1,200,000,000) = .39(1,200,000,000) = .39C(0) \]
\[ = .39(1,200,000,000), \text{ the amount at the beginning of 2013}. \]

They will start 2014 with
\[ C(2) = 46,800,000 - .61(46,800,000) = .39(46,800,000) = 18,252,000 = .39C(1), \]
and 2015 with
\[ C(3) = 18,252,000 - .61(18,252,000) = .39(18,252,000) = 7,118,280 = .39C(2) \text{ etc.} \]
Continue to compute \( C(t) \) for \( t = 4,5,6,7,8,9,10 \) and observe that
\[ C(t) = C(t-1) - .61C(t-1) = .39C(t-1) \text{ is an easy way to compute } C \text{ for each additional year.} \]
\[ C(t) = .39C(t-1) \text{ is a recursive relationship used to compute } C \text{ for the } t\text{-th year, } C(t), \]
based on the preceding year, the \((t-1)\text{st year, } C(t-1).\]
Do you think that they ever completely run out of money?
Some students will guess yes, others will guess no. The discussion is important to have. You might have the students with substantial mathematics background try to write an equation for $C(t)$ in terms of $C(0)$. $C(t) = (.39)^{t-1}(1,200,000,000)$ which means they never completely run out of money since $C(t)$ will never be exactly 0. It can, however, could come arbitrarily close to 0.

(2) Similarly income can be expressed as Inc as follows:
\[
\text{Inc}(0) = 1,600,000,000 \\
\text{Inc}(1) = 1,600,000,000 + .23(1,600,000,000) = 1.23(1,600,000,000) = 1,968,000,000 \\
\text{Inc}(2) = 1,968,000,000 + .23(1,968,000,000) = 1.23(1,968,000,000) = 2,420,640,000 \\
\text{Inc}(n) = \text{Inc}(n-1)+.23\text{Inc}(n-1) = 1.23(\text{Inc}(n-1)).
\]
\[
\text{Inc}(n) = 1.23\text{Inc}(n-1)
\]
is a recursive relationship used to compute Inc in year $n$ based on income the preceding year.

Thus, it appears there is sufficient income, which can be used to replenish the cash on hand.

A **recurrence relation** for a sequence of numbers is a formula which gives each number in the sequence in terms of the preceding number(s).

A **recursive process** is a set of steps used to solve a problem, where one step is based on the results of the preceding step(s).

We have seen some examples of thinking recursively. Let’s look at some more!

**Activity 1.4:** How would you find the largest number in a sizeable set of numbers, say among 30 numbers?

Brainstorm ways of finding the largest number. One approach is to make this a whole-class exercise: hand out slips of paper with distinct numbers on them to the students in the class, one for each student. Now ask them to try out their possible methods for
finding a solution with their set of student numbers and see which one works best, however you define “best”.  

Another approach is to have students work in smaller groups. To make sure the problem is not so small that students can immediately determine the answer, give each group a large enough set of numbers (e.g. between 3 and 5 numbers for each group member). To make the exercise especially helpful ensure that some groups get an even quantity of numbers in their set some get an odd quantity. Have each group write an algorithm for finding the largest number. Students should try their algorithm in their group before trying it with the whole class. Then try a few groups’ algorithms with the whole class where each class member has a number.

Did it make a difference if there was an even number or an odd number of elements in the set? 

Can you find the smallest number in a similar way? Explain.

One way to solve the problem is “pair up the numbers in the set” and find the largest of each pair of elements, two by two until you exhaust the set. If you have one number left over, keep it. You now have half as many numbers as when you started, each larger than one of the numbers left out, plus possibly one more number. This new set of numbers is approximately half the size of the original set. Repeat the process with this new set of numbers; in other words, with the half that are left, “pair up the numbers in the set”, look at each pair of elements and discard the smaller, keep the larger. Continue this process with each half left, until you have only two numbers left. Compare these two numbers: the largest is the largest of the entire set. At any stage, if you have an extra number (so the numbers cannot be evenly divided into pairs), just pass the extra number along to the next stage with the largest numbers of each pair.

As a class you can implement this process by having all students stand up at the outset; after the first iteration, about half the class sits down, then another half, etc. The last student remaining has the largest number.

How many comparisons are made with this process, called an algorithm?
Computational thinking alert (algorithm)

An **algorithm** is a finite sequence of steps for solving a problem. An important aspect of computational thinking is defining a solution process that is repeatable, so that when you are faced with the same problem again, or a variation of it, you will be able to solve it efficiently by leveraging past experience. Framing your solution process as an algorithm makes it systematic and repeatable. By defining an algorithm you create a tool that you can invoke the next time you encounter the same type of problem.

How many comparisons are made if there are 5 numbers? How about if there are 10 numbers? Can you generalize and say how many comparisons are made if there are n numbers?

If there are n elements in the set, there are n/2 initial comparisons and you are left with a set of n/2 elements (if n is even) or (n-1)/2+1 elements (if n is odd). For the n/2 (or (n-1)/2 + 1) elements in the new set, make pairwise comparisons to get the largest element in each pair. Continue the process until you have the largest overall element.

The table below should record the number of comparisons at each stage starting from n=2 to n=10. Note that if n=1 then that element is the largest of the set so no comparison is needed.

Let C(n) denote the number of comparisons need to find the largest element of a set of n elements. Thus C(1) = 0 and C(2) = 1. What is a formula for computing C(3), the number of comparisons when n=3? Express the formula in terms of smaller C(-)-s. Complete the table showing of the number of comparisons performed for each value of n.

There are several ways the general formula could be expressed. Here’s one way:

- C(3) - we have one initial comparison plus 1 comparing the largest of the first two with the third. So C(3) = 2.
- \( C(4) \) - since it requires two initial comparisons and then a comparison of the two largest from the initial comparisons. \( C(4) = 2 + C(2) = 2 + 1 = 3. \)
- \( C(5) \) - since there are two initial comparisons, leaving a 3 element set which needs to be compared, \( C(5) = 2 + C(3) = 2 + 2 = 4. \)

Note we could write \( C(3) = 2 + C(0) \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( C(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 1:

It looks like a pattern is forming, which would yield \( C(n) = 1 + C(n-1) \) with \( C(1) = 0 \) or alternatively \( C(n) = 2 + C(n-2) \) with \( C(2) = 1 \).

See if each works for \( C(6) \) and \( C(7) \).

If it does (and there is a proof that it does) we have defined \( C(n) \) not in terms of the preceding \( n \), but in terms of two back. It is still a recursive process for solving the problem of finding the largest number in a set, even if it goes back one further.

**Homework for section 1:**

1. If you deposit $5,000.00 in a savings account with an interest rate of 3% how much will you have in the account in 5 years? Write the recurrence relationship that represents the solution to this problem.
$5,796.37 after 5 years; \ S(n) = 1.03S(n-1) \ with \ S(1) \ or \ S(0) = 5000, \ or \ for \ students \ who \ know \ exponents \ S(n) = 5000(1.03)^n$

2. What happens if the interest rate drops to 2% in problem 1?
$5,520.40 \ and \ S(n) = 1.02S(n-1) \ with \ S(0) = 5000$

3. A classic SAT question gives you a sequence of numbers: 1, 6, 26, 106, .... And asks you to give the 7th term. Write the recurrence relation for this sequence and compute the 7th term.
$R(n) = 4R(n-1) + 2 \ with \ R(1) = 1. \ Therefore \ R(7) = 6,826$

4. A classic SAT question gives you a sequence of numbers and asks you to write a formula which gives the nth number of the sequence. For example, write a recurrence relation for this sequence:
$1000, \ 1120, \ 1254, \ 1254.4, \ 1404.928, \ ...$
$C(n) = 1.12C(n-1) \ with \ C(1) = 1000$

5. Make up two sequences of your own to be used on SAT exams. Give the recurrence relations that describe them.
$2,4,8,16,... \ A(n) = 2A(n-1)$
$20, \ 146, \ 1028, \ 7202,... \ A(n) = 7A(n-1) + 6$

6. Find the largest number in the set of numbers:
$43,2,5,21,17,24,32,8,3,52,36,28,10,19$ using the algorithm talked about in class.
How many comparisons did you have to make?
The largest number is 52. The number of comparisons, C(n), is given by the formula $C(n) = 2 + C(n-2) \ or \ C(n)=1+C(n-1)$, students will remember this as one less than the previous so $C(14) = C(12) + 2 = C(10) + 2 + 2 = C(10) + 4 = 9+4 = 13$

Section 2 (Day 2)
Review the solutions to the first set of homework problems. Talk about the procedure to solve problem 6.
A suggestion is to do this as a whole class problem to start the class. As they come in give each student a number. Run them through the algorithm. Record the number of comparisons. Include a sheet of numbers that teachers can copy, cut apart and hand to students.

Another solution to the previous problem, that of finding the largest number in a set (or the largest number held by the set of students), is to split the set in half and find the largest number in each half by dividing each half in half again, and half again, etc., until you have sets with two elements in it, then build it back up again. Is this the same process as described before?

Try this method on problem 6 from the homework.
Have students work in small groups.
The recurrence relation this time looks different, but is it?
Have students give reasons pro and con.

Have the students in small groups compute the largest number in a set of 10 numbers and for a set of 15 numbers using this method.
If L(n) = number of comparisons needed to find the largest number, then in this case L(1) = 0 and L(2) = 1, L(3) = 2 and L(4) =3 etc
L(n) = 2L(n/2) + 1 for n even and L(n) = 2L((n + 1)/2) for n odd
Is one of the two methods more efficient than the other in terms of number of comparisons, or are they the same? Complete the table that shows the first 10 counts:

<table>
<thead>
<tr>
<th>n</th>
<th>C(n)</th>
<th>L(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
In each case the process is the same, as described by this computation tree:

Each box represents a value, and each pair of lines represents a comparison being performed, with the larger of the two values going into the box that the two lines connect to.

Activity 1.5: Hopefully, you are beginning to understand recursive thinking. Try this game and see how you do.

It is best to do this in groups of 4-5. Create a spinner (or use a pair of dice) with as many numbers as you wish on the spinner. Each student spins the spinner (rolls the dice) to get a number for his/her payoff for each correct guess.

Create a table with the results of a recursive relationship, but with some numbers missing. Use different missing numbers for each group.
For example consider the table below. This table, once completed, indicates the recursive relationship \( A(n) = 3(A(n-1)) - 1 \).

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>5</th>
<th>14</th>
<th>41</th>
<th>122</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>365</td>
<td>1094</td>
<td>3,281</td>
<td>9,842</td>
<td>29,525</td>
</tr>
</tbody>
</table>

Player 1 guesses a whole number between 1 and 100000. If it appears in the table, then the player gets the number of points indicated on the spinner. One player in each group will need to be the scorekeeper. Each group member will have to have a copy of the sequence. Play continues until the table is completed. If a player can guess the recurrence relation, then they get an extra 5 points. The player with the most points wins. You can construct a different table for each group. Depending on your class, make the relationship as easy or hard as you wish.

**Activity 1.4:** Find a recurrence relation for the sequence 1, 2, 6, 24, 120, …. Can you identify this sequence using something you learned in mathematics classes?

This sequence is the sequence of factorials starting with 1.

\( n! \) is defined recursively as \( n! = n(n-1)! \), and \( 0! = 1 \)

This homework set is more challenging than the day 1 homework set. You may want to start some of the problems in class. Encourage students to draw diagrams when thinking about the problems.

**Homework 2:**

1. Find a recurrence relation to count the number of regions when \( n \) lines intersect, no three in the same point and no two parallel. Except in the case \( n=1 \), all lines intersect. For example, when two lines intersect we get 4 regions of the plane. \( T(n) = T(n-1) + n \) with \( T(1) = 2 \).
2. Find a recurrence relation to count the number of regions when n circles intersect, no two concentric. For example when 2 circles intersect there are 4 regions – think of (and draw) a Venn diagram to help you visualize this case. 

\[ C(n) = C(n-1) + 2(n-1) \text{ with } C(1) = 2 \]
3. The cost of a year in college is said to increase by 15% a year. Write a recurrence relation to express the cost n years from now, if currently the average cost is $30,000 a year for all expenses.

\[ C(n) = 1.15C(n-1) \text{ with } C(1) = 30,000 \]

<table>
<thead>
<tr>
<th>N</th>
<th>C(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>44</td>
</tr>
<tr>
<td>8</td>
<td>58</td>
</tr>
<tr>
<td>9</td>
<td>74</td>
</tr>
<tr>
<td>10</td>
<td>92</td>
</tr>
</tbody>
</table>
4. In a party, every partygoer wants to shake hands with everyone at the party. Write a recurrence relation that gives the total number of handshakes in a room of \( n \) people.

This problem is good small group problem. Hint for students: think about the number of handshakes that occurred before the \( n \)th person arrived. How many handshakes will the new arrival add to the total?

\[
R(n) = R(n-1) + (n-1) \quad \text{with } R(2) = 1
\]
5. In a round robin tournament, \( n \) players play every other player in the tournament. Write a recurrence relation to count the total number of games played.

*This is the same recurrence as in question 4. You can start either recurrence at \( R(1)=0 \) if desired.*

### Section 3: Recurrences based on more than one preceding term.

**Driving Question:** How do we define recurrences when the value of \( n \)th occurrence depends on more than one term that precedes it?

**Example 1:** You are always in a hurry, so often you climb stairs two at a time as well as one at a time. How many different ways are there to climb a set of \( n \) stairs when you are allowed to go up one step or two?

<table>
<thead>
<tr>
<th>( n )</th>
<th>( R(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
</tr>
<tr>
<td>10</td>
<td>45</td>
</tr>
</tbody>
</table>
Go to the stairwell and try it! Have students create a table of values.
For example, if $n=1$ there is only one way to go up the stair (one at a time), but if $n = 2$ then there are two ways: go up one step at a time, or go up two steps at the same time.
If $T(n)$ counts the number of possible ways, then $T(1) = 1$ and $T(2) = 2$.
Compute $T(3)$, $T(4)$, and $T(5)$. Guess at a formula – a recurrence relation.
$T(3) = 3$ since you can go up each one separately, or the first two then the last, or vice versa. $T(4) = 5$ since you can go up each one separately, or 2,2 or 1,1,2, or 2,1,1 or 1,2,1. By now they should see that $T(4) = T(3) + T(2)$ and $T(3) = T(2) + T(1)$. Have them compute $T(5)$ to confirm that $T(5) = T(4) = T(3)$. Have them describe why $T(n) = T(n-1) + T(n-2)$ logically.

Are there any requirements on $n$? $n > 2$, since there are two base cases ($n=1$ and $n=2$), and $T(n)$ will depend on both $T(n-1)$ and $T(n-2)$. Can you think of ways of counting the ways of climbing the stairs without using actual stairs?
Some possible answers are (1) to draw a picture, as above, or (2) to count in how many ways change can be given for N using coins of value 1 and 2.

**Example 2:** Given 1x1 tiles and 1x2 tiles, how many different ways can you form a 1xn strip?
Have students cut out their own 1x1 and 1x2 tiles from paper, or give them paper tiles. Students can then use these tiles, either individually or as part of a team, to explore physically what the possibilities are.
Have the students complete a table, as in previous exercises, so that they can record their findings. These results will assist them in finding the recurrence. They will fairly quickly discover that $T(1) = 1$ and $T(2) = 2$. From there they may find that $T(3) = 3$ and $T(4) = 5$. They should guess at this point that $T(n) = T(n-1) + T(n-2)$ and that the base step is $T(1) = 1$ and $T(2) = 2$.
Some students should begin to recognize that this is the same as the stair-climbing problem.
How many 2xn strips can you make? Have students complete a table.
Example 3: Fibonacci Numbers

The question that Leonardo de Pisa (better known as Fibonacci) posed is as follows:
If we start with one pair of newborn rabbits of opposite sex, then how many pairs of rabbits are there at the end of one month, two months, three months, four months, …if we assume that rabbits reproduce as follows:

(1) It takes a newborn rabbit two months to mature into an adult rabbit and be able to reproduce.
(2) A pair of adult rabbits of opposite sex produces one pair of newborn rabbits of opposite sex, and
(3) Rabbits never die.

A denotes an adult rabbit pair.
N denotes a newborn rabbit pair.
We have 1 pair of rabbits at the end of month 1, and 1 pair of rabbits remains at the end of month 2. By the end of month 3 this pair remains and produces a new born pair yielding 2 pairs. By the end of month 4, the pair remains and produces another newborn pair and the newborn pair from month 3 remains yielding 3 pairs of rabbits. So far we have 1, 1, 2, 3, … On your paper continue the pattern one more step.

The resulting sequence 1, 1, 2, 3, … is the first part of what is called the Fibonacci Sequence.

Questions:
1. How many pairs of rabbits are there at the end of month 6? How many of these are adults? How many are newborns?
   8 pairs, 5 adult pairs and 3 newborn pairs at end of month 6.
2. How many rabbits are there at the end of month 7? How many of these are adults? How many are newborns?
   13 pairs, 8 adult pairs and 4 newborn pairs at end of month 7.
3. Can you find a formula to compute the number of pairs of rabbits at the end of month \( n \)? Test it with your answers to problems (1) and (2).
   If \( F(n) \) denotes the number of pairs of rabbits at the end of month \( n \) then
   \[ F(n) = F(n-1) + F(n-2) \]
4. How is this formula related to the stair climbing and the tiling formulas?
   It is the same recurrence relation except for the starting point or base steps.
   Here we start at \( F(1) = 1 \) and \( F(2) = 1 \). Before we started at \( T(1) = 1 \) and \( T(2) = 2 \). An extra 1 starts the sequence.
5. Suppose rabbits can produce 2 pairs of newborns at a time? How does that change the recurrence relation?
   The recurrence will be \( T(1)=1, T(2)=2, T(n) = 2T(n-2) + T(n-1) \). The first few terms of the sequence are: 1, 1, 3, 5, 11, ...

The **Fibonacci Sequence** 1, 1, 2, 3, 5, 8, … is defined by \( F(1) = 1 \) and \( F(2) = 1 \) and
\[ F(n) = F(n-1) + F(n-2). \]

Bring branches of various trees, a pineapple, a pine cone, and some flowers in to class. Have the students count the leaves along a branch as they keep track of the number of times they go around the branch. Similarly, have them count the whorls around the pineapple as they turn the pineapple around, and count the petals around the center of a flower.

Chose a species of animal and find out how they reproduce. Determine how many (or how many pairs) will exist at the end of each month (or year).
**Phyllotaxis** is the field of botany that studies the arrangement of leaves around the stem, scales on a pine cone, florets on a flower, whorls on a pineapple, etc. In some trees, like the elm or basswood, the leaves along a branch seem to occur alternately on opposite sides and we say the elm has $\frac{1}{2}$ phyllotaxis. An apple tree and a rose bush have a leaf every 144 degrees around the branch or five leaves for every two turns around the branch (144x5 = 360), thus has $\frac{2}{5}$ phyllotaxis. The pear and the apple tree have $\frac{3}{8}$ phyllotaxis. The **phyllotaxis** of a branch = $p/q$ where $p$ represents the number of leaves for $q$ turns of the branch.

Do you notice anything interesting about the ratios? They are ratios of Fibonacci sequence numbers.

**Teaser Discussion Question** (frames the next day’s lesson)

Create 24 signs with different numbers, not necessarily in numerical order, on them (enough so each student in the class gets one). Randomly hand one to each student. How many different ways can you find to sort the numbers (people) from lowest number to highest number? As a method is suggested, try it out with the students standing in a line. Observe how long it took to get the signs (people) in numerical order. What measure did you use to determine how long it took?

**Homework 3:**

1. Triangular numbers are numbers that can be created by sets of dots that form an equilateral triangle.

   ![Triangular Numbers Diagram]

   $T(1) = 1 \quad T(2) = 3 \quad T(3) = 6$

Write the recurrence relation for triangular numbers, \( T(n) \).

\[ T(1) = 1, \quad T(n) = T(n-1) + n \]

2. Bee reproduction: A male bee is produced by one female bee, but a female bee has two parents – a male and a female. So a male bee has 1 parent, two grandparents, 3 great grandparents, 5 great great grandparents etc. Draw a family tree for a male bee, and another for a female bee. For the female bee how many generations. Write recurrence relations for each of their sets of ancestors. Include the family trees as part of the teacher section.

For males:
\[ M(n) = M(n-1) + M(n-2) + 1 \]  for males and \( F(n) = F(n-1) + F(n-2) + 2 \) for females

with \( M(0) = 1 \), \( M(1) = 1 \) and \( F(0) = 1 \) and \( F(1) = 2 \)

3. How many ways can you tile a strip of \( n \) unit squares if you have size 1x1 and 1x3 tiles?

\[ T(n) = T(n-1) + T(n-3) \text{ where } T(1)=1, \ T(2)=1, \ T(3)=2 \]

4. Using a piece of graph paper tile a single rectangle with squares of side length 1, 1, 2, 3, 5, and 8, how big a rectangle do you get?

Have them use different sizes of rectangles to try it. Have your students cut out the squares from paper, or provide them to your students. You can color the various size squares. The size of the rectangle is 8x13.
5. What would be the next biggest rectangle you can tile with squares corresponding to the Fibonacci numbers 1, 1, 2, 3, 5, 8, and 13? Show it.

21x 13

How is this related to the previous question?

6. Assume you have an apple worth $1, a package of crackers worth $2, and a milkshake worth $3. How many ways can you hand out $n$ worth of food? For example, you can hand out $0$ worth in only one way – pass out nothing. You can hand out $1$ worth in one way – give them an apple. You can hand out $2$ worth of food in 2 ways – pass out two apples or one package of crackers. Let $A(n)$ denote the number of ways of handing out $n$ worth of food. Assume order does not make a difference. What is $A(1)$? What is $A(2)$ and $A(3)$? What is $A(4)$? Write a recurrence relation for $A(n)$.

\[
A(n) = A(n-1) + A(n-2) - A(n-4) - A(n-5) + A(n-6) \quad \text{with}
\]

\[A(1) = 1, \quad A(2) = 2, \quad A(3) = 3, \quad A(4) = 4, \quad A(5) = 5 \quad \text{and} \quad A(6) = 7\]

or

\[
A(n) = 1 + A(n-2) + A(n-3) - A(n-5) \quad \text{and same initial conditions up to 5.}
\]

7. How many ways are there to arrange $n$ total distinguishable students in a row of $n$ seats? Let $A(n) =$ number of ways of arranging $n$ students. Find $A(1)$, $A(2)$, $A(3)$, and the recurrence relation for $A$. What is this the same as?

\[A(1) = 1 \quad A(2) = 2, \quad A(3) = 6 \quad A(n) = nA(n-1)\]

8. If $B(n) = B(n-1) + B(n-2) + B(n-3)$ is a recurrence relation. How many initial conditions do you need? Give an example of when you might use this recurrence relation.

You need three initial conditions; $B(n-1)$, $B(n-2)$, and $B(n-3)$ indicate this.
Section 4: Computational Efficiency, Part I - Decrease and Conquer Algorithms

Many problems can be solved by breaking them apart into smaller pieces, solving the smaller pieces, and combining the answers to those smaller problems to produce an answer to the problem as a whole. That is precisely what many algorithms do. These kinds of algorithms are classified as either decrease and conquer or divide and conquer algorithms. A decrease and conquer algorithm breaks a large problem into just one smaller problem whereas a divide and conquer algorithm breaks a large problem into two or more smaller problems. The power of dividing a large problem into two or more smaller problems (divide and conquer algorithms) will be explored in more detail in section 5.

Much of the discussion in sections 4 and 5 has to do with sorting. There are two senses of sorting: (1) putting things in order, and (2) grouping things into categories. We use “sorting” in the first sense.

The problem of putting things in some particular order, as in from smallest to largest, or from cheapest to most expensive, is called sorting. Sorting a set of data is a good example of a problem where dividing the data set into smaller pieces and working with the smaller pieces leads to a solution of the whole problem.

Example: presenting items in an on-line store from least expensive to most expensive. This involves sorting (ordering) simply on the price of each item. The ordering is that determined by ‘<’ (less than). If two items have the very same price some additional criterion must be used to determine their relative order.

Example 4.1: Putting books in alphabetical order by author’s name.
‘Alphabetical order’ relies on a lexicographic order: words are ordered first by their first letter; words that match on their first letter are ordered second by their second letter, and so on.

If students know about cross products of sets you can explain that a lexicographic ordering of AXB (the cross product of A and B, consisting of pairs of elements (a,b), where $a \in A$ and $b \in B$) first orders the pairs by of elements of A, then by an ordering of
elements of B. Since a word consists of a sequence of letters; if the set of letters is denoted by L, then normal alphabetical order is a lexicographical ordering of LXLXLX,..XL,

**Example 4.2:** Putting playing cards in order.

Many orderings are possible. Here’s one lexicographical ordering of playing cards:

Order of suits defined by:

- Spade < Heart < Club < Diamond

Order of values within a suit defined by:

- Ace < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K

This is a lexicographic order of cards: Suit X Value.

There are many very well understood methods to sort data. Some of these algorithms, while intuitive, do not scale up well, in other words handle large amounts of data well. Others are significantly more efficient: they are able to sort data sets containing several million items in the time it takes an inefficient algorithm to sort a few thousand.

Sorting is everywhere – we want to sort many lists of things from phone numbers, to rank in class, to grades on a test, to diving scores, and more. Suppose you are interested in sports, like hockey. You can visit the NHL website to view statistics on teams: [http://www.nhl.com/ice/teamstats.htm](http://www.nhl.com/ice/teamstats.htm). Once on this page you can sort teams by various characteristics, such as the name of the team, the number of games played, the number of wins, total points, and others.

If you are exploring career choices, check out the Bureau of Labor Statistics websites and see examples of lists that have been sorted:

- [http://www.bls.gov/ooh/most-new-jobs.htm](http://www.bls.gov/ooh/most-new-jobs.htm)
- [http://www.bls.gov/ooh/highest-paying.htm](http://www.bls.gov/ooh/highest-paying.htm)
- [http://www.bls.gov/ooh/fastest-growing.htm](http://www.bls.gov/ooh/fastest-growing.htm)
When you shop on-line you probably sort products by price or rating. For example, at http://www.amazon.com you can sort by “relevance”, “most popular”, “price (high to low)”, “price (low to high)” or “average customer review”. Choosing a different criterion changes the ordering.

Example 4.3: Lining children up by height.
Have the students offer suggestions on how they would line up the students in the class according to their height.

Question: Give examples of other things they might want to sort, such as sort students by grades in the class, sort them by birthday, etc.

SORTING
As you probably discovered from the Teaser in section 3, there are many ways to sort data, where by sort we mean to arrange a list in order from smallest to largest, alphabetically first to last, etc. Sorting is quite a common thing to do when computing, because many data processing tasks can be done more efficiently when the data is sorted compared to when it is not. We compare two approaches to sorting, one of which is more efficient than the other. Instead of just talking about recursive processes, we will talk about algorithms that are recursive.

The first sorting algorithm we explore is called selection sort. The algorithm begins with a list of items that need to be put in order. Selection sort is described by a process with two cases:

- Case 1: If there is just one item to sort, stop – you are done!
- Case 2: If there is more than one item to sort, look through the unsorted items and remove the smallest one (first alphabetically) and placing it between those items already sorted and the rest of the unsorted items. Finally, (recursively) sort the rest of the items.

When we say to recursively sort the rest of the data we mean apply exactly the same process to sort the remainder of the items.
Example:
Let's start with an informal presentation of the algorithm.
Suppose we have a list of numbers (maybe the scores that students received on a test): 95, 86, 97, 72, 83. If we apply Selection sort to put these scores in order from smallest to largest:
Find the smallest value in the list. It is 72. Put this as the first score in the answer, leaving behind 95, 86, 97, 83.
Find the smallest value amongst the remaining values. It is 83. Place it as the second value in the answer: 72, 83, leaving behind 95, 86, 97.
Find the smallest value amongst the remaining values. It is 86. Place it as the third value in the answer: 72, 83, 86, leaving behind 95, 97.
Find the smallest value amongst the remaining values. It is 95. Place it as the fourth value in the answer: 72, 83, 86, 95, leaving behind 97.
Find the smallest value amongst the remaining values. It is 97. Place it as the fifth value in the answer: 72, 83, 86, 95, 97. Since there are no values remaining the process is finished, and the scores are in order from smallest to largest.

You can find some nice visualizations of how selection sort operates on-line, for example in:


Every recursive process can be described with at least two cases, as above. One case, called the “base case”, handles simple problems for which an answer can be determined immediately. The other case, called the “recursive case”, solves a large problem by breaking it into a smaller problem of the same kind, solves the smaller problem, and then figures out an answer to the original problem using the answer to the smaller problem.

**Activity 4.1:** Break students into groups of at least 6. Each group must have the same number of chairs as students, arranged in a row. Start out with the students sitting
down at random in the chairs. Have the students apply the selection sort algorithm to arrange themselves in alphabetical order by first name.

In the example, you should do first, we can arrange the names in alphabetical order using the selection sort algorithm. The selection sort algorithm divides the data into two groups: those that have been sorted, and those that have not been sorted. To begin with, all data items are in the unsorted group. (Even if the data items are initially in the correct order, the algorithm doesn’t KNOW that – the algorithm must still do its work to verify that the data is indeed in order from smallest to largest). In the following the sorted items are underlined.

**Example:**

For our second example we will go through the steps of the algorithm in a little more detail. Suppose our initial data set is a list of names:

| Sue | Zoe | Meg | Cam | Joe | Amy | Lea | Bob |

As long as the unsorted group has at least two members we can find the “smallest” word (the one that comes first alphabetically) using a method from the teaser or Activity 1.1. Of all the unsorted students, Amy comes first alphabetically, so Amy and Sue switch places. (How many comparisons did that take? Remember, a comparison determines which of two values comes first in the order. A comparison between x and y is x < y.)

Let’s go over in more detail how to determine the smallest value in a list.

To find the smallest value assume that the first unsorted element is the smallest, and then compare it to the next unsorted item. If the next unsorted item is smaller, then our hypothesis as to the smallest value is updated to this new value (otherwise it stays the same). Continuing in this fashion we can be assured that our assumption is the smallest value only once the hypothesis has been compared to each unsorted value. In this example the hypothesis starts out as Sue. We proceed as follows:

1) Sue is compared to Zoe. Sue comes before Zoe so nothing changes.
<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

2) **Sue** is compared to the next value, **Meg**. Meg is smaller, so Meg now becomes the one compared to others.

<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

3) **Meg** is compared to the next value, **Cam**. Cam is smaller, so Cam now becomes the one compared to others.

<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

4) **Cam** is compared to the next value, **Joe**. Cam comes before Joe so the nothing changes.

<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

5) **Cam** is compared to the next value, **Amy**. Amy is smaller, so Amy now becomes the one compared to others.

<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

6) **Amy** is compared to the next value, **Lea**. Amy comes before Lea so the nothing changes.

<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

7) **Amy** is compared to the next value, **Bob**. Amy comes before Bob so the nothing changes.

<table>
<thead>
<tr>
<th>Sue</th>
<th>Zoe</th>
<th>Meg</th>
<th>Cam</th>
<th>Joe</th>
<th>Amy</th>
<th>Lea</th>
<th>Bob</th>
</tr>
</thead>
</table>

Now we are done – and **Amy** is the smallest (first one alphabetically).
Question: How many comparisons did we make?

Note: We underline the ones in the sorted group. Remember that we swapped Amy with Sue to put Amy in the right place.

Now recursively sort the (now smaller) unsorted group: Zoe, Meg, Cam, Joe, Sue, Lea and Bob. (How many comparisons does this require?)

This is how the recursive sort algorithm rearranges students’ names:

We first compared them all to find the first alphabetical one, in this case we made 7 comparisons. Once that one was moved, we compared the second one to each of the rest by making 6 comparisons. Once the second one was in the right place we need to make 5 comparisons, then 4 comparisons, then 3, 2 and finally 1 comparison and everything is where it should be. The total number of comparisons can be written as
7 + 6 + 5 + 4 + 3 + 2 + 1 = 28. We see at level k there are k-1 comparisons, called order k (written as O(k)). See the computational thinking alert after an example.

Now observe that there are n total levels, so the sum can be written as (n-1)((n-1)+1)/2, or (n-1)n/2. Check that this formula works for summing 1 to 7 to get 28.

If students have had counting in algebra 2, they may know the following formula expressed with a summation sign, or just as n(n+1)/2:

\[
\sum_{k=1}^{n} k = \frac{n(n+1)}{2}
\]

Overall we say that this is an \(n^2\) algorithm since it has \(\frac{1}{2} (n^2 + n)\).

**Question**: Suppose there are 20 students in line. How many comparisons are needed to put them in the correct alphabetical order?

Counting directly, we have 19 + 18 + 17 + … + 2 + 1 = 190

If your students know the sum of a set of consecutive integers formula then they can write it as \(n + (n-1)+ \ldots + 2 + 1 = n(n+1)/2 = n^2/2 + n/2\). So the answer to the question is 19(20)/2 = 190

In general if there are n elements to sort this way, we need to make

\((n-1) + (n-2) + \ldots + 2 + 1\) comparisons = \((n-1)n/2\) using the formula for the sum of consecutive integers 1, 2, 3, …, (n-1).

We say algorithms of this type are of order \(n^2\), since the formula is a quadratic. Therefore the selection sort algorithm is said to be order \(n^2\).

**Computational Thinking Alert**

**Big O (order of algorithm)**

Big-O measures how well an operation will “scale” (handle a large amount of data) when you increase the amount of “things” it operates on. Big-O can be used to describe how fast an algorithm will run by giving an estimate of how much work it
needs to do to solve a given problem in terms of the input size. In the case of the selection sort algorithm a useful measure of the amount of work done is given by the number of comparisons that are needed to put data elements in order.

Some basic $O(n)$ descriptions are:

$O(1)$ means that no matter how large the input is, the time taken doesn't change. For example determining if a number is even or odd is $O(1)$. We say that an algorithm that is $O(1)$ runs in constant time.

$O(n)$ means that for every element, you are doing a constant number of operations, such as comparing each element to a known value. Since there are $n$ items to process the total time grows as $n$ gets bigger. For example determining the smallest in a set of $n$ values is $O(n)$ because we have to look at each item once. We say that an algorithm that is $O(n)$ runs in linear time.

$O(n^2)$ means that for every element, you do something that takes linear time. We say that an algorithm that is $O(n^2)$ runs in quadratic time.

Big-O is one of a family of so-called asymptotic notations. Big-O gives an upper bound. Other notations include little-o, Omega $\Omega$ and Theta $\Theta$.

Using the big-O notation we say that selection sort is $O(n^2)$.

**Activity 4.2:**

Materials: For each pair of students, A and B, provide a deck of cards numbered 1 to 16. Have A shuffle their cards ten times, then hand the cards over to B. B must sort the cards, using selection sort. While B is sorting, A must keep track of how many $x < y$ comparisons are done to find the smaller of two cards. Students can then reverse roles.
[NOTE: an x<y comparison is different from an x for y swap. If x < y is true, then x and y are in the correct relative order, and no swap needs to be done. However, an x < y comparison was still done, and must be counted.]

[IDEA: this can be done for 4 card, 8 cards, 16 cards, 32 cards, 64 cards, etc. The class as a whole can gather data on the number of comparisons performed. To save time different pairs of students can sort different numbers of cards. More students can do the smaller decks, or students doing smaller decks can repeat multiple times to get some average number of comparisons. Students should find that all attempts to sort the same number of cards uses the same number of comparisons. This is an interesting property of Selection Sort.]

Fill in Table 4.1.

<table>
<thead>
<tr>
<th>Number of cards or Problem size</th>
<th># comparisons</th>
<th>What happens to the average # of comparisons when problem size doubles?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trial 1</td>
<td>Trial 2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>…and so on…</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1

Homework for Section 4:

1. Give an example of an algorithm that can be described recursively and is not one of the ones discussed in this section.
   Example: algorithm for folding laundry:
   base case – pile is empty
recursive case – take one thing from the pile, fold it, then fold rest of pile

2. The following is a set of zip codes that must be sorted from **lowest to highest**: 01234, 04321, 08976, 14231, 35678, 29454, 87654, 15234. Show how to sort them using selection sort and give the number of comparisons that you had to perform.

   - **Starting with** 01234, 04321, 08976, 14231, 35678, 29454, 87654, 15234 it will take 7 comparisons to find the smallest value, 01234. Partial answer is 01234.
   - **Starting with** 04321, 08976, 14231, 35678, 29454, 87654, 15234 it will take 6 comparisons to find the smallest value, 04321. Partial answer is 01234, 04321.
   - **Starting with** 08976, 14231, 35678, 29454, 87654, 15234 it will take 5 comparisons to find the smallest value, 08976. Partial answer is 01234, 04321, 08976.
   - **Starting with** 14231, 35678, 29454, 87654, 15234 it will take 4 comparisons to find the smallest value, 14231. Partial answer is 01234, 04321, 08976, 14231.
   - **Starting with** 35678, 29454, 87654, 15234 it will take 3 comparisons to find the smallest value, 15234. Partial answer is 01234, 04321, 08976, 14231, 15234.
   - **Starting with** 35678, 29454, 87654, 15234 it will take 2 comparisons to find the smallest value, 29454. Partial answer is 01234, 04321, 08976, 14231, 15234, 29454.
   - **Starting with** 35678, 87654, it will take 1 comparison to find the smallest value, 35678. Partial answer is 01234, 04321, 08976, 14231, 15234, 29454, 35678.
   - **Starting with** 87654, it will take 0 comparisons to find the smallest value, 87654. Final answer is 01234, 04321, 08976, 14231, 15234, 29454, 35678, 87654.

   The total number of comparisons needed is 7+6+5+4+3+2+1+0, or 28.

3. The following passwords need to be sorted from **highest to lowest**.
2345, 1234, 4321, 5678, 3251, 9123, 4123, 5317. Show how you would modify selection sort to do this sort and do it.

4. How many comparisons are needed to do a selection sort for
   a. 40 items \((39)(40)/2\)
   b. 100 items \((99)(100)/2\)
   c. 35 items \((34)(35)/2\)

5. If a sorting algorithm requires \(n^3+5\) comparisons for an \(n\) element set it is \(O(\?)\)
   If a sorting algorithm requires \(n^2/2+8\) comparisons for an \(n\) element set it is \(O(\?)\)

6. If you have 32 hats to match with their 32 owners, but you do not know which one belongs to which owner how would you use a sorting algorithm to make the matches? Hint: think of what properties of hats/owners correspond and might also be distinct from one hat/person to the next.
   There is no one correct answer. Here are some ideas. (1) We could sort hats by the inside circumference of their openings, and people by the outside circumference of their heads. (2) We could measure the length of the front-to-back opening of each hat, as well as the width of the left-to-right opening of each hat, and order the hats lexicographically first by front-to-back distance and then by left-to-right distance. People could be matched to hats by ordering them similarly.

7. You have just moved into a new apartment. You are entering the neighborhood and you realize you don’t know your building number. You looked at the building and did not find a number there. You knocked on a random apartment, and asked, “what is the number of this building?” The guy who opened the door smiled and said, “I don’t know the number of this building, but if you look at the number of the building next to us and add 1, you will know the building number.”
   Now let’s think for a moment. The problem you are trying to solve is knowing the number of your house. It was suggested that you look at the number of the building next to you and add 1. Your problem is reduced to a problem exactly like the first problem – at first your problem was to find the number of your house and now it is to find the number of the house (next to it).
   Does this define a problem that can be solved recursively? If so, how?
8. Recursion pops up in mathematical definitions quite often. For example, the factorial function: 0! = 1, n! = n * (n-1)! for n > 0. Do you know of any other recursively defined functions? Can you invent some of your own? Can you define addition or multiplication recursively?

We can easily define recursive functions. Consider a function f(n) that computes the sum of the first n positive even numbers (where n>0): f(1) = 2, f(n) = 2n + f(n-1), for n > 1.

Addition: 0 + y = y, x + y = 1 + (x-1) + y, for x>0.

Multiplication: 1 * y = y, x * y = y + (x-1)*y, for x>1.

9. Recursion pops up in language too. Sentences can be embedded within sentences like ‘John said that X’. For example, we can make arbitrarily long sentences like this: Sue said that Bill said that Mary said that John said that Cloë likes to read.

(a) Can you think of verbs other than 'say' which permit a whole sentence to follow?

Some are: believe, doubt, remember, think. There are many others.

(b) Can you write a rule that expresses the recursion?

Unless students have had some formal grammar training they probably will not express the rule in a formal way, which is fine. They might notice that some verbs cannot take sentences as complements (e.g. ‘eat’, ‘purchase’, ‘dance’, ‘give’) while some can. Call the former SimpleVerbs, and the latter SententialComplementVerbs. A base case sentence is one that uses a SimpleVerb, whereas a recursive case sentence is one that uses SententialComplementVerb.

Section 5: Computational efficiency, part II – Expression Trees and Divide and Conquer Algorithms

This section explores the idea of divide-and-conquer. This is a very power problem-solving technique, widely used to build computationally efficient algorithms. The first example, expression trees, is used to give some grounding
in multi-way recursive structures: a binary operators, such as addition (‘+’) takes two arguments, each of which is an expression. Because there are two subexpressions that are a part of the addition expression there is two-way recursion involved.

Expressions
You are probably familiar with arithmetic expressions. Some examples are:

\[
\begin{align*}
3 + 4 \\
7 - 2 \\
4 \times (5 - 3) \\
17
\end{align*}
\]

Something all these expressions have in common is that they have a value. The value of an arithmetic expression is what you get when you carry out all the operations. The values of the expressions listed above are:

\[
\begin{align*}
7 \\
5 \\
8 \\
17
\end{align*}
\]

Figuring out the value of the third expression, \(4 \times (5 - 3)\), requires us to first figure out the value of the expression \(5 - 3\). Since this expression has value 2 we then multiply 4 by 2 to get the final answer of 8. The last expression, 17, is very simple since 17 is just a number. The value of a number is itself!

We can think of expressions as having a recursive structure. The base case of the structure is a simple expression: the number. The recursive case of an expression is an expression that has an arithmetic operation: +, -, \times, or ÷. Each arithmetic operation, such as +, is a binary operation, meaning it needs two operands.

We can visualize the structure of this example expression by drawing it as a tree, called an expression tree. An expression tree for a number is just the number written in a circle. The expression tree for 17 is therefore:
The expression tree for an expression with an operator is written with the operator in a circle, along with the expression trees for its two operands written just below and on either side of the operator. The operator is connected to its two operands with lines. For example, the expression tree for $3 + 4$ is:

```
+  
|  
3  4
```

Question: what is the expression tree for $7 - 2$?

Answer:

```
-  
|  
7  2
```

Question: what is the expression tree for $4 \times (5 - 3)$?

Answer:

```
\times  
|  
4  
\-  
\|  
5  3
```

Now let’s consider a larger example, such as $4 \times 3 + 10 \div 2$. 
From this diagram we can see more clearly that the expression is the sum of the values of two sub-expressions: $4 \times 3$ and $10 \div 2$. Each of these is also a binary expression, meaning it combines two things. The one on the left is the product of the values of two sub-expressions, 4 and 3, whereas one on the right is the quotient of the two sub-expressions 10 and 2.

In general we can describe the structure of an arithmetic expression as consisting of either

- a number, or
- an operator and two operands, each of which is an expression.

We can express this using a slightly more formal notation for the structure of an expression as follows:

- `<expression> → <number>`
- `<expression> → <expression> <operator> <expression>`

This is a definition that is given recursively: there is a base case for the definition (an expression can consist of just a number) and a recursive case (an expression can consist of an expression, an operator and another expression).

The value of an expression is determined as follows:

The value of a `<number>` is just the number itself

The value of `<expression₁> + <expression₂>` is the sum of the values of `<expression₁>` and `<expression₂>`
The value of $\text{expression}_1 - \text{expression}_2$ is the difference of the values of $\text{expression}_1$ and $\text{expression}_2$

The value of $\text{expression}_1 \times \text{expression}_2$ is the product of the values of $\text{expression}_1$ and $\text{expression}_2$

The value of $\text{expression}_1 \div \text{expression}_2$ is the quotient of the values of $\text{expression}_1$ and $\text{expression}_2$

This means that the value of this expression $(4 \times 3 + 10 \div 2)$:

![Diagram of expression](image)

is the sum of the values of these two expressions $(4 \times 3)$ and $(10 \div 2)$:

![Diagram of expressions](image)

The value of the expression on the left is the product of the values of these two expressions $(4$ and $3)$:

![Product diagram](image)
Each of these expressions consists of just a number, so there is no further evaluation that needs to take place. The value of the expression 4 is 4, and the value of the expression 3 is 3.

The value of the product expression is therefore the product of 4 and 3, or 12.

Similarly, the value of the quotient expression is the quotient of the values of the two expressions 10 and 2, or 5.

Finally, the value of the overall expression is the sum of the values of these expressions,

\[
\begin{align*}
\times & \quad \div \\
4 & \quad 3 & 10 & \quad 2
\end{align*}
\]

which we now know is the same as the sum of these values,

\[
\begin{align*}
12 & \quad 5
\end{align*}
\]

which is 17.

**Activity 5.1:**
Use an expression tree to illustrate the structure of the following and find the value of this expression:

\[
( 2 + 3 ) \times ( ( 15 \div ( 4 - 1 ) ) \times 6 )
\]

**MergeSort**
The second sorting algorithm we explore is called *mergesort*. It sort (orders its input from smallest to largest) using a divide-and-conquer approach. It is described by the following process:
• If there is just one item to sort, stop – you are done!
• If there is more than one item to sort, first split the data into two disjoint parts, a left and a right part.¹ Recursively sort each part, then merge the two (now sorted) parts into a single sorted list.

There are some nice visualizations of how merge sort operates on-line, for example at: http://en.wikipedia.org/wiki/Merge_sort.

Activity 5.2:

Materials: Provide a deck of cards numbered 1 to 16 for each pair of students, A and B, A shuffles the cards ten times, then hands the cards over to B. B must sort the cards, using merge sort. While B is sorting A must keep track of how many $x < y$ comparisons are done to find the smallest card. Students can then reverse roles.

[IDEA: this can be done for 8 cards, 16 cards, 32 cards, 64 cards, etc. The class as a whole can gather data on the number of comparisons performed. To save time different pairs of students can sort different numbers of cards. More students can do the smaller decks, or students doing smaller decks can repeat multiple times to get some average number of comparisons.

Fill in a table in the following format:

<table>
<thead>
<tr>
<th>Problem size</th>
<th># comparisons</th>
<th>What happens to average # comparisons when problems size doubles?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Trial 1</td>
<td>Trial 2</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>... and so on...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

¹Note: If there are an even number of elements then the split can be into equally-sized parts. If there are an odd number of elements one of the two partitions will have one extra item. Let’s assume that the extra element consistently goes into the right partition.
**Question:** Using mergesort, how many comparisons are needed when you sort 2 cards? How many comparisons are needed to sort 4 cards? 8 cards, etc.?

To sort two cards, divide them into 2 parts each with one card. These parts are already sorted. Merge the two cards with lowest number on the bottom. It took only 1 comparison – of the two cards. Let’s keep track of the number of comparisons with C denoting the number of comparisons. With four cards, divide them into two parts each with two cards and sort these parts. Once sorted, merge the sorted pairs together.

**Optional material**

If your students have had or are currently taking Algebra 2 and understand logs try asking how one would sort N cards with k students where k is an integer power of 2: $k = \text{ceiling}(\log_2(N))$. For example, to sort 32 items, use 6 students. The first student $S_1$ splits the pile of 32 cards into two equal partitions, L and R, each of 16 cards. $S_1$ hands L (the left pile) to student $S_2$, who sorts it using mergesort and returns the sorted pile to $S_1$. $S_1$ then hands R (the right pile) to student $S_2$, who sorts it using mergesort and returns the sorted pile to $S_1$. $S_1$ merges the two piles as follows: $S_1$ picks the smallest of the two top cards from L and R, putting it at the back of the answer pile. $S_1$ continues choosing the smallest of the top cards from L and R, putting the result at the back of the answer pile, until one of L and R is empty. At that point all remaining cards are put at the back of the answer pile.

Here’s a visual depiction of how the sorting proceeds in the general case:
First, I will split my pile of $n$ cards into two piles of $n/2$ cards. (Technically I will split into one pile of floor($n/2$) cards and one of ceiling($n/2$) cards. This works when $n$ is even or odd, and keeps me from having to tear any cards in half! 😊)

Now I give one pile of unsorted cards to a friend to sort (also using merge sort).

I have $n/2$ cards to sort. I will sort using the merge sort algorithm!

I'm done – I sorted my pile of cards using merge sort. I will give this pile of sorted cards back to the friend who gave them to me in the first place.
Here you go, friend!

Thanks! I am well on my way to sorting all of my cards. Now I give my other pile of unsorted cards to a friend to sort (also using mergesort).
Hey friend, here's another pile of unsorted cards for you to put in order.

Gee, thanks. I guess this is what friends are for!

I have n/2 cards to sort. I will sort using the mergesort algorithm!
I'm done – I sorted my pile of cards using mergesort. I will give this pile of sorted cards back to the friend who gave them to me in the first place.

Here you go, friend!

Thanks! Now I have two piles of cards, each pile sorted. All I need to do now is merge these two sorted piles into one sorted pile.
Another way of thinking about how this algorithm proceeds is to take a look at a concrete example. The problem is to put eight names into alphabetical order using merge sort: (Sue, Zoe, Meg, Cam, Joe, Amy, Lea, Bob). **Note this is the same list we started with in section 4.**

To sort we first split the list of names into two disjoint parts. Since we (conveniently) have 8 names, we split into two lists each with 4 names: (Sue, Zoe, Meg, Cam) and (Joe, Amy, Lea, Bob).

Next, we recursively mergesort both of these smaller lists. **For the time being let's not worry about how that happens but simply assume that it does happen.** Once each of the two smaller lists has been sorted we have: (Cam, Meg, Sue, Zoe) and (Amy, Bob, Joe, Lea).

Now we illustrate the last step of the algorithm, the *merge* step. To merge the two lists into a single list we **compare** the first name from the front of each list to determine which belongs first alphabetically: Cam vs. Amy. Since Amy comes before Cam we remove Amy from the right list and put it into a new list which we'll call our merged list. Our left list is still (Cam, Meg, Sue, Zoe) but our right list is just (Bob, Joe, Lea). Our merged list is (Amy).

We continue with the merge process until both the left and right lists are empty and all eight items are in the merged list. To illustrate the next step, we again **compare** the first name from the front of the left and right lists to determine which belongs first
alphabetically: Cam vs. Bob. Since Bob comes before Cam we remove Bob from the right list and put it at the back of our merged list. Our left list is still (Cam, Meg, Sue, Zoe) but our right list is just (Joe, Lea). Our merged list is (Amy, Bob).

Continuing along we again compare the first name from the front of the left and right lists to determine which belongs first alphabetically: Cam vs. Joe. Since Cam comes before Joe we remove Cam from the left list and put it at the back of our merged list. Our left list is now (Meg, Sue, Zoe) and our right list is unchanged (Joe, Lea). Our merged list is (Amy, Bob, Cam).

Summarizing the work thus far in merging the left and right lists:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Merged list</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Cam, Meg, Sue, Zoe)</td>
<td>(Amy, Bob, Joe, Lea)</td>
<td>()</td>
</tr>
<tr>
<td>(Cam, Meg, Sue, Zoe)</td>
<td>(Bob, Joe, Lea)</td>
<td>(Amy)</td>
</tr>
<tr>
<td>(Cam, Meg, Sue, Zoe)</td>
<td>(Joe, Lea)</td>
<td>(Amy, Bob)</td>
</tr>
<tr>
<td>(Meg, Sue, Zoe)</td>
<td>(Joe, Lea)</td>
<td>(Amy, Bob, Cam)</td>
</tr>
<tr>
<td>(Meg, Sue, Zoe)</td>
<td>(Lea)</td>
<td>(Amy, Bob, Cam, Joe)</td>
</tr>
</tbody>
</table>

The next steps in the merge process look like this:

<table>
<thead>
<tr>
<th>Left list</th>
<th>Right list</th>
<th>Merged list</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Meg, Sue, Zoe)</td>
<td>(Lea)</td>
<td>(Amy, Bob, Cam, Joe)</td>
</tr>
<tr>
<td>(Meg, Sue, Zoe)</td>
<td>()</td>
<td>(Amy, Bob, Cam, Joe, Lea)</td>
</tr>
<tr>
<td>(Sue, Zoe)</td>
<td>()</td>
<td>(Amy, Bob, Cam, Joe, Lea, Meg)</td>
</tr>
<tr>
<td>(Zoe)</td>
<td>()</td>
<td>(Amy, Bob, Cam, Joe, Lea, Meg, Sue)</td>
</tr>
<tr>
<td>()</td>
<td>()</td>
<td>(Amy, Bob, Cam, Joe, Lea, Meg, Sue, Zoe)</td>
</tr>
</tbody>
</table>

Notice that comparisons are only needed during the merge step.

So how do we sort the lists (Sue, Zoe, Meg, Cam) and (Joe, Amy, Lea, Bob)? Above we just said “don’t worry about it” and to assume that it just happens. This is the neat thing about recursive processes – once we have described the overall algorithm by...
giving a base case and a recursive case, the algorithm is fully described. The recursive case expresses a solution to the original (large) problem in terms of solutions to smaller instances of the same problem. Thus, we sort the lists (Sue, Zoe, Meg, Cam) and (Joe, Amy, Lea, Bob) recursively, applying mergesort to each one in turn and then merge the two lists (combine them).

Step-by-step the process unfolds as follows:

To mergesort this list: (Sue, Zoe, Meg, Cam, Joe, Amy, Lea, Bob)
1) Mergesort left half of (Sue, Zoe, Meg, Cam, Joe, Amy, Lea, Bob):
   To mergesort this list: (Sue, Zoe, Meg, Cam)
   1.1) Mergesort left half of (Sue, Zoe, Meg, Cam)
       To mergesort this list: (Sue, Zoe)
       1.1.1) Mergesort left half of (Sue, Zoe)
               (Sue)
       1.1.2) Mergesort right half of (Sue, Zoe)
               (Zoe)
       1.1.3) Merge the two sorted halves:
               (Sue, Zoe)
   1.2) Mergesort left half of (Sue, Zoe, Meg, Cam)
       To mergesort this list: (Meg, Cam)
       1.2.1) Mergesort left half of (Meg, Cam)
               (Meg)
       1.2.2) Mergesort right half of (Meg, Cam)
               (Cam)
       1.2.3) Merge the two sorted halves:
               (Cam, Meg)
   1.3) Merge the two sorted halves:
       (Cam, Meg, Sue, Zoe)
2) Mergesort right half of (Sue, Zoe, Meg, Cam, Joe, Amy, Lea, Bob):
   To mergesort this list: (Joe, Amy, Lea, Bob)
2.1) Mergesort left half of (Joe, Amy, Lea, Bob)
   To mergesort this list: (Joe, Amy)
   2.1.1) Mergesort left half of (Joe, Amy)
         (Joe)
   2.1.2) Mergesort right half of (Joe, Amy)
         (Amy)
   2.1.3) Merge the two sorted halves:
         (Amy, Joe)

2.2) Mergesort right half of (Joe, Amy, Lea, Bob)
   To mergesort this list: (Lea, Bob)
   2.2.1) Mergesort left half of (Lea, Bob)
         (Lea)
   2.2.2) Mergesort right half of (Lea, Bob)
         (Bob)
   2.2.3) Merge the two sorted halves:
         (Bob, Lea)

2.3) Merge the two sorted halves:
     (Amy, Bob, Joe, Lea)

3) Merge the two sorted halves:
   (Amy, Bob, Cam, Joe, Lea, Meg, Sue, Zoe)

A good way to visualize the entire process of sorting is by way of a computation tree:
Question: How many merges did we do?

With 8 names, there are three merges to do the left half of the tree and 3 merges to do the right side of the tree and then 1 additional merge, or 7 total to get a final alphabetically ordered list.

Try to derive a formula for how many comparisons are needed, in terms of the problems’ size (call it $n$).

Answer: If you did the optional section then it is $n \times \log_2(n)$. If you didn’t, see if they realize it is order bigger than $O(n)$ and less than $O(n^2)$ by comparing it to the number of comparisons made in the selection sort of the same list.

Informal explanation: Looking at the computation tree, the top half corresponds to the splitting phase of the algorithm (blue arrows) while the bottom half corresponds to the merging phase (green arrows).
Comparisons only take place during the merge step. To merge two partitions, each of size n/2, requires at most n-1 comparisons (one for each position in the result, except the last one). We can overestimate a little bit and assume that for each position in a merged partition we need to do a comparison. With this assumption, each merge level in the computation tree then requires n comparisons. The big question is how many levels there are. It turns out that there are log₂(n) levels: this is how many times a partition of size n must be halved to get partitions of size 1.

Recall that for the computation tree for selection sort there is O(n) work done per level in finding the smallest value, and there are n levels in the tree, giving O(n x n). In mergesort there is still O(n) work to be done per level, but there are only log(n) levels in the tree, giving an overall runtime of (n x log(n))

We can compare this to the computation tree of selection sort – (n-1) comparisons on top level, (n-2) comparisons on next level, etc. There are n levels in the tree.

**Question:** Which is faster (in terms of the number of comparisons) - trees that are shallower or trees that are taller in a mergesort?

Savings in number of comparisons comes from making the computation tree shallower in mergesort.

Mergesort is a nice example of what is called a divide-and-conquer technique for designing algorithms. The idea is that solving large problems is hard but solving small problems, like sorting a single item, or merging two sets, each already sorted, into one sorted set, is easy. To recursively define how to find a solution to a problem you have to identify:

1) base cases – instances of the problem for which solutions are easily known
2) a recursive decomposition of a problem into smaller instances (pieces) of the same type of problem
3) a way to combine solutions of smaller instances of the same type of problem into a solution for the larger problem

**Activity 5.3:**

**Materials:** Pencil and paper, maybe a calculator  Although a typical calculator will not compute logarithms in base 2, students can use the formula \( \log_b(n) = \frac{\log_{10}(n)}{\log_{10}(2)} \).

Predict how large a problem can be solved using selection sort and merge sort, in a given amount of time. Assume that the only relevant determinant of the time it takes each of the algorithms to run is the number of comparisons they perform, \( O(n^2) \) for selection sort and \( O(n \log n) \) for merge sort.

We did some experiments to see how long it took selection sort and mergesort to sort random data sets of various sizes. To sort 8,000 random numbers selection sort needed 279 milliseconds (a millisecond is 1/1,000 of a second), while merge sort needed just 2 milliseconds. How much time do you think each algorithm would need to sort 64,000 random numbers? How about 1,024,000? Fill in the table that follows.

Note: For each row the problem size is being doubled (going from \( n \) to \( 2n \)). The selection sort time is expected to quadruple, since it is \( O(n^2) \) and \( (2n)^2 \) is \( 4n^2 \). The mergesort time is a little trickier to guess, but it should go up by a little more than double in each row. This is not true for small problems (less than about 8,000 items). Because the algorithm finishes so quickly the clock records the time as just 1 millisecond for both the 2,000 and 4,000 problem sizes. You can have students compute the expected ratios:

\[
\frac{2n \log_2 2n}{n \log_2 n}
\]

for different values of \( n \) if they understand logarithms.

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Selection Sort time</th>
<th>Merge Sort time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>4,000</td>
<td>74</td>
<td>1</td>
</tr>
<tr>
<td>8,000</td>
<td>279</td>
<td>2</td>
</tr>
<tr>
<td>16,000</td>
<td>1127</td>
<td>6</td>
</tr>
</tbody>
</table>
These are the observed times in our experiment. The selection sort numbers do not go beyond a problem size of 256,000 as the predicted time is excessive (516,388 milliseconds is approximately 8 minutes and 36 seconds.) For a problem of size 512,000 the predicted time to complete is 2,065,552 milliseconds, or approximately 34 minutes and 25 seconds. It is instructive for students to realize that to sort one million items with selection sort would take approximately 2 hours, whereas with merge sort it takes about 1 second!

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Selection Sort time</th>
<th>Prediction (formula)</th>
<th>Merge Sort time</th>
<th>Prediction (formula)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,000</td>
<td>18</td>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4,000</td>
<td>74</td>
<td>74</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8,000</td>
<td>279</td>
<td>279</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>16,000</td>
<td>1,127</td>
<td>1127</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>32,000</td>
<td>4,653</td>
<td>(1,127 \times 4 = 4,508)</td>
<td>12</td>
<td>(6 \times (2)^{(\log_{2} 32000/\log_{2} 16)} = 13)</td>
</tr>
<tr>
<td>64,000</td>
<td>18,126</td>
<td>(4,508 \times 4 = 18,032)</td>
<td>28</td>
<td>(13 \times (2)^{(\log_{2} 64000/\log_{2} 32000)} = 28)</td>
</tr>
<tr>
<td>128,000</td>
<td>74,264</td>
<td>...</td>
<td>57</td>
<td>...</td>
</tr>
<tr>
<td>256,000</td>
<td>516,388</td>
<td></td>
<td>142</td>
<td></td>
</tr>
<tr>
<td>512,000</td>
<td></td>
<td></td>
<td>342</td>
<td></td>
</tr>
<tr>
<td>1,024,000</td>
<td></td>
<td></td>
<td>1,018</td>
<td></td>
</tr>
<tr>
<td>2,048,000</td>
<td></td>
<td></td>
<td>2,083</td>
<td></td>
</tr>
<tr>
<td>4,096,000</td>
<td></td>
<td></td>
<td>4,879</td>
<td></td>
</tr>
<tr>
<td>8,192,000</td>
<td></td>
<td></td>
<td>11,997</td>
<td></td>
</tr>
<tr>
<td>16,384,000</td>
<td></td>
<td></td>
<td>27,836</td>
<td></td>
</tr>
<tr>
<td>32,768,000</td>
<td></td>
<td></td>
<td>66,095</td>
<td></td>
</tr>
</tbody>
</table>
There are some good graphs showing the effects of algorithmic efficiency by the size of problems that are realistically solvable:
http://science.slc.edu/~jmarshall/courses/2002/spring/cs510/BigO/

**Homework for Section 5:**

1. Draw an expression tree for and evaluate the expression \((8 \times 6) / (5 + 3)\).
2. Draw an expression tree for and evaluate the expression \((36 / 6) \times (54 / 9)\).
3. The following is a set of zip codes that must be sorted from **lowest** to **highest**:
   01234, 04321, 08976, 14231, 35678, 29454, 87654, 15234. Show how to sort them using mergesort and give the number of comparisons that you had to perform. Compare your answer to the one you got for question 2 in section 4.
4. The following passwords need to be sorted from **highest** to **lowest**.
   2345, 1234, 4321, 5678, 3251, 9123, 4123, 5317. Show how you would modify mergesort to do this and then do it.
5. How many comparisons are needed to do a merge sort for
   a. 32 items
   b. 128 items
   c. 70 items
6. How many comparisons are needed to do a merge sort for
   d. 40 items
   e. 100 items
   f. 35 items
7. Optional: In propositional logic formulas are constructed from simple propositions (like p and q), logical operators that act on formulas, and parentheses. Typical operators are AND, OR and NOT. There are rules that govern what is considered a **well-formed formula** (sometimes called a **wff**) and what is just a nonsense sequence of symbols. For example, the following are all wffs:
   a. p
b. \( p \text{ AND } q \)

c. \( \text{NOT } p \text{ AND } q \)

d. \( \text{NOT (} p \text{ AND } q) \)

Non-wffs include:

  e. \( p \text{ AND } q \)

  f. \( p \text{ AND AND } q \)

  g. \( \text{OR } p \text{ AND } q \)

  h. \( p \text{ NOT AND } q \)

Write a set of recursive rules which characterize the well-formed propositional logic formulas.

- A proposition \((p, q, r, \text{ etc})\) is a \textit{wff}.
- If \( x \) is a \textit{wff}, then so is \( \text{NOT } x \).
- If \( x \) is a \textit{wff}, then so is \((x)\).
- If \( x \) and \( y \) are \textit{wffs}, then so are \( x \text{ AND } y \) and \( x \text{ OR } y \).

6.1 Final Assessment:

You have three pegs and 3 rings like the child’s stacking toy. At the outset all the rings are on one peg largest, then middle size, then smallest. How many moves does it take to move all three pegs to another peg with largest on the bottom, and smallest on top? At no time are you allowed to have the wrong order on a peg. Suppose you have just 2 rings? Guess a recurrence relation for the number of moves with \( n \) rings in terms of smaller numbers of rings? Test it with 4 rings. This problem is called the Temple of Benares problem. Legend has it that God started with 64 rings and pegs made of solid gold and He said that the Temple would crash and that the world would end when all of the rings were correctly transferred to another peg.

\[ M(3) = 5, \quad M(1) = 1, \quad M(2) = 2, \quad M(4) = 11, \]
\[ M(N) = 2M(N-1) + 1 \]

6.2 Project:
**Time analysis of everyday tasks**

Think of working as a librarian who has to shelve returned fiction books in the library.

**a.** Describe an algorithm that tells how you would do this task. As a group, name 10 novels (title and author) and hand-simulate the algorithm on these books.

**b.** We want to calculate the cost (in effort) of this task. Assume that you start at the A authors. You take 1 book off your cart, move to the corresponding shelf, then pick up another book and repeat. Assume there is 1 bookcase of books for each letter of the alphabet containing all authors whose last name starts with that letter. Assume it costs 5 steps per letter of the alphabet to walk past each bookcase on the way to the first letter of the author of the book you are to shelve. For example, it costs 10 steps to get to bookcase C starting at bookcase A, in order to shelve a book by James F. Cooper. With these assumptions, calculate the costs of shelving your selected 10 novels,

- a. if they are not in any order on the book cart?
- b. if they are put into sorted order by author's last name on the book cart?

Describe the pattern of author last names for the books to be shelved associated with this **worst case** cost.

Given your estimates for 1 and 2 above, then can you estimate the **worst case** cost of shelving n books

- if they are not in any order on the book cart?
- if they are put into sorted order by author's last name on the book cart?

Describe the pattern of author last names for the books to be shelved associated with this **worst case** cost.