UNIVERSITY OF WATERLOO
Faculty of Mathematics
Waterloo, Ontario N2L 3G1

Centre for Education in
Mathematics and Computing

Grade 7/8 Math Circles
Computer Science & Algorithms - Solutions
October 20/21, 2015

The Technological Revolution

Our lives have been shaped by the changing technology around us. The internet has grown from just an idea, to a an integral part of our daily lives. Just a few years ago, cell phones were just that - phones. They could talk, and they could text, but they weren't the gateway to the world they have become today. None of this would be possible without computers, and more specifically, without the computer scientists and engineers that build them.

Today, researchers and engineers are continuing to work towards building computers that can learn and think like human beings. The way that they do this is through the logic and programming that is built into the machine. Today, we will be looking at one of the basic building blocks of this logic and a key piece of mathematics - algorithms.

What Are Algorithms?

In the simplest definition, an algorithm is simply a set of steps or group of instructions to be followed. Just today you've likely followed an algorithm without even knowing about it. But if an algorithm is just a set of steps, what makes them so useful and so fundamentally important to both mathematics and computer science?

While there may be no right answer to that question, one major reason that algorithms are so valuable is they help us solve problems. While there are many problems we can solve on our own, our world has become so complex that there are a great deal of problems that require so much work that a single person couldn't solve it in a reasonable amount of time - even when the work isn't very hard. This is where computers come in. They can complete calculations in a fraction of the time humans can, and with more accuracy. The challenge then, is telling the computer what to do. This is where algorithms come in. Algorithms can tell the computer what calculations to do at each step, and an algorithm can tell the computer whether to follow option A or option B depending on the result. Some basic computer programming is essentially translating the algorithm or process into information the computer processes.

Sorting Algorithms

One application where algorithms are used extensively is in sorting. There are countless sorting algorithms used today.

Exercise 1: Name as many different sorting algorithms as you can.

For our sorting algorithms we will just sort numbers, but we can use the same algorithms to sort any type of item, as long as we know how to compare them.

Exercise 2: Sort the following numbers from lowest to highest

$$14, 6, 11, 5, 20, 17, 1$$

Think about how you sorted that list. Did you look for a specific number in the list? Or did you break the list up into smaller parts?

The first type of algorithm we will look at involve looking for a specific item or place in the list. Specifically, we are going to look at *selection sort* and *insertion sort*. Both of these algorithms work by maintaining two lists - the original unsorted list, and a new sorted list. At each step, the algorithm moves one item from the unsorted list to the sorted list.

Selection Sort
This algorithm is similar to what many people do when they are sorting a list. It looks for the smallest (or largest) item in the unsorted list and adds it to the end of the sorted list. To better see how this algorithm works, look at how it sorts the list from *Exercise 2*.

Sorted	Unsorted
	$14, 6, 11, 5, 20, 17, 1$
1	$14, 6, 11, 5, 20, 17$
$1, 5$	$14, 6, 11, 20, 17$
$1, 5, 6$	$14, 11, 20, 17$
$1, 5, 6, 11$	$14, 20, 17$
$1, 5, 6, 11, 14$	$20, 17$
$1, 5, 6, 11, 14, 17$	20
$1, 5, 6, 11, 14, 17, 20$	

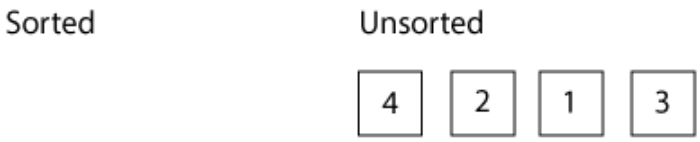
At the beginning of the process, our sorted list is empty. At each iteration, we add one new number to the sorted list. This means that if our list has length n we need to do this process n times.

Exercise 3 - Think about the Selection Sort algorithm. What guarantees that the end result will be a properly sorted list? At each step we have one less number to sort. Eventually we will run out of numbers

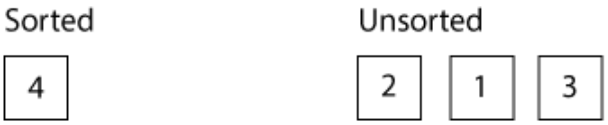
Insertion Sort The second sorting algorithm we will look at is similar to selection sort. It starts with an unsorted list and an initially empty sorted list. At each step it moves on element from the unsorted list to the sorted list. The difference between insertion sort and selection sort is in which element is moved.

Insertion sort simply takes the element from the first spot in the unsorted list and *inserts* it into the correct place in the sorted list. To illustrate this, suppose we were sorting the list $4, 2, 1, 3$ in increasing.

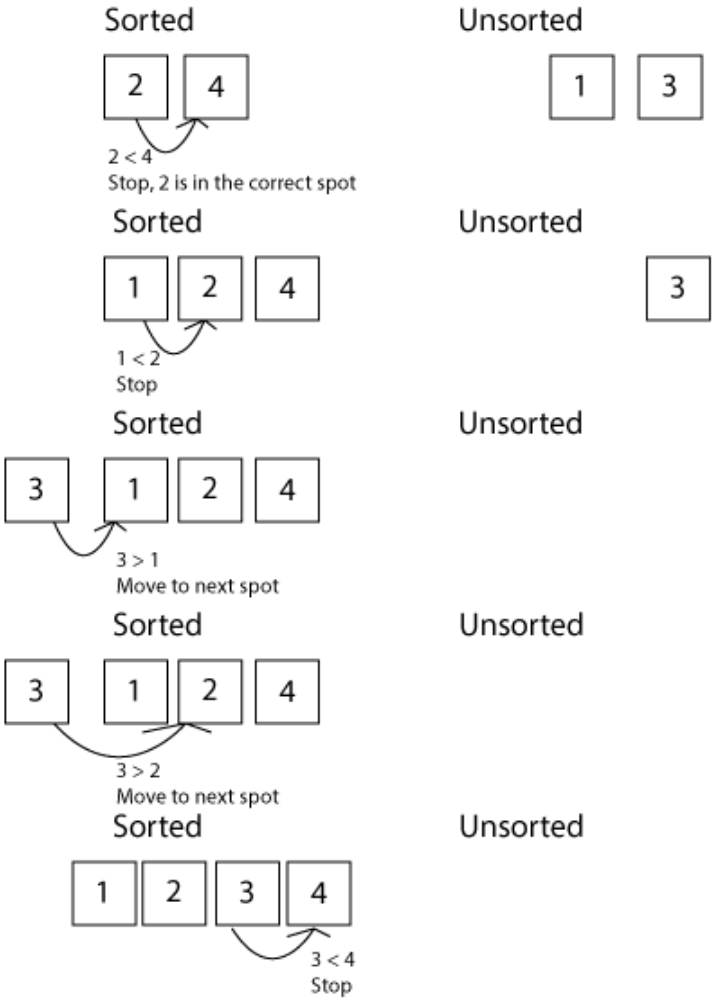
Our initial lists would look like this:



At each step, we take the first element from the unsorted list and put it in the sorted list. In this case, we move 4 to the sorted list.



At the next step we need to move 2 to the sorted list. But where should we put 2? This is where the algorithm has a decision to make. We know that the sorted list is in increasing order, so we can just check each spot for the new element, and check if the next element is larger or smaller. If it is smaller, then we move to the next spot. If it is larger, then we found the right spot and put the new element there.



Exercise 4 - Sort the list from *Exercise 2* using the insertion sort algorithm. The list to sort is $14, 6, 11, 5, 20, 17, 1$

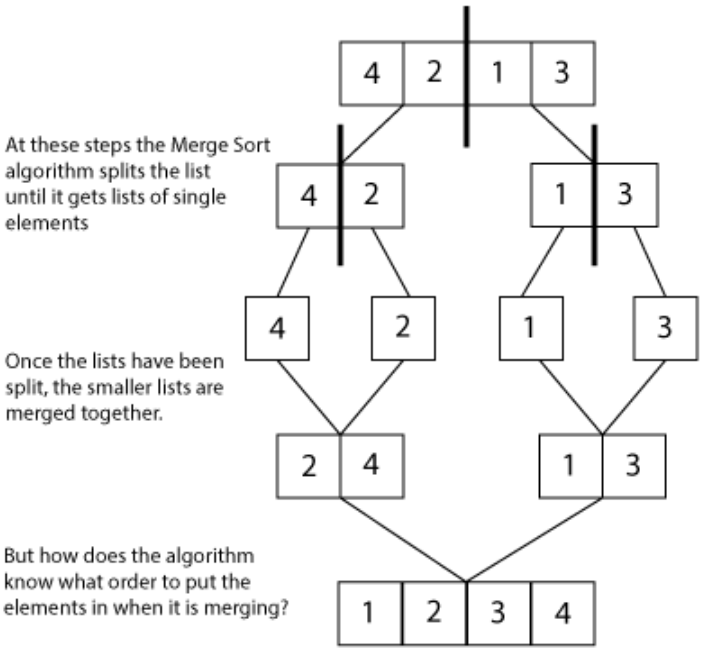
Sorted	Unsorted
	$14, 6, 11, 5, 20, 17, 1$
14	$6, 11, 5, 20, 17, 1$
$6, 14$	$11, 5, 20, 17, 1$
$6, 11, 14$	$5, 20, 17, 1$
$5, 6, 11, 14$	$20, 17, 1$
$5, 6, 11, 14, 20$	$17, 1$
$5, 6, 11, 14, 17, 20$	1
$1, 5, 6, 11, 14, 17, 20$	

While the *Selection Sort* and *Insertion Sort* algorithms both work by taking elements one by one from an unsorted list, that isn't the only approach to sorting. Another common approach to sorting (and problem solving in general) is to break the problem down into smaller parts. Often these smaller parts are easier to solve, and then we can build the solution to the whole problem by combining the smaller solutions.

Merge Sort

This algorithm is one of the best examples of breaking down a problem into smaller pieces and then putting those pieces back together - because this is exactly what *Merge Sort* does.

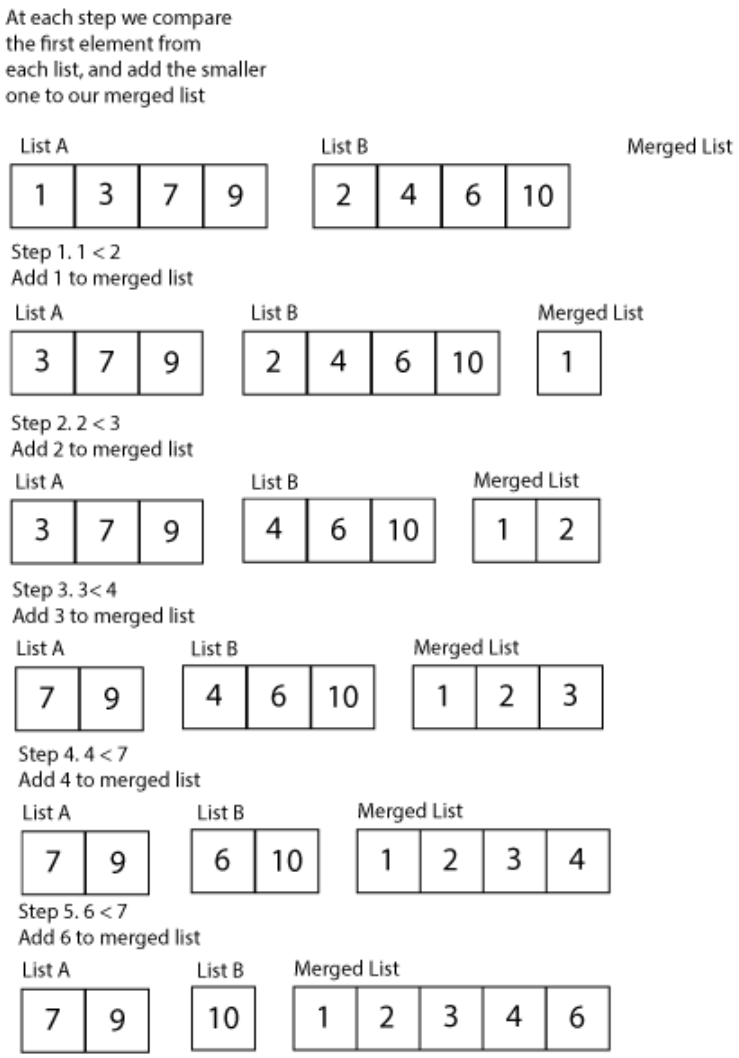
The algorithm takes an unsorted list and breaks that list in half. It keeps breaking the resulting lists in half until there is only one element in each list. These lists are very easy to sort! The single element lists are then paired up, with each 2 element list getting sorted. This merging process continues until we have one list again.



While the idea of breaking the list into smaller and smaller pieces seems to make sense, how does the merging process actually work?

At each merging step the algorithm exploits the fact that each list it is merging together is already sorted. This means it only has to compare the first element in each list - if the smallest in list A is smaller than the smallest in list B, then it must be the smallest in both.

To see exactly how merging works, consider merging the lists $A = 1, 3, 7, 9$ and $B = 2, 4, 6, 10$



Exercise 5 - Complete the merging process from the above image.

List A	List B	Merged List
9	10	$1, 2, 3, 4, 6, 7$
	10	$1, 2, 3, 4, 6, 7, 9$
		$1, 2, 3, 4, 6, 7, 10$

Huffman Coding

While we can understand and express algorithms in English, computers can't immediately act on instructions written in English or any other languages. Instead, that language is interpreted and converted into binary - what is essentially the language of computers. Binary consists of just 0's and 1's, and these digits can represent numbers, letters, and even pictures.

If we look at text, each character (including spaces and punctuation) in the English alphabet can be converted into a an 8-digit (or 8-bit, we can also call 8 bits one byte) code. Each bit (or digit) is either a 0 or a 1. For example, the letter *a* can be coded as 01100001.

If we know all of these 8-bit codes, we could convert an entire message into binary. For example, we could write "Math!" as 010011010110000101110100011010000010000 1. This is a lot longer, and if we were to write a larger document (like a book) the binary code would be extremely long. To save on space and make message transmissions easier, computer scientists have developed a number of ways to compress data to make it take up less space.

Some of these algorithms are *lossy* - they lose parts of the message. Let's say that you decided to compress your text messages by removing all of the vowels. You want to send your friend a message that says "Math is Awesome". Right now, we need $8 \times 15 = 120$ bits. But if we remove the vowels our message becomes "Mth s wsm" and then we only need $8 \times 9 = 72$ bits. That's 40% less space! But can anyone actually understand the message?

This is why we have developed methods for compressing text without losing any information. The standard method for text compression is Huffman Coding.

Huffman coding works by taking a message and assigning short (e.g. 1 or 2 digit) codes to the letters that appear most frequently. Less frequent letters are given longer codes.

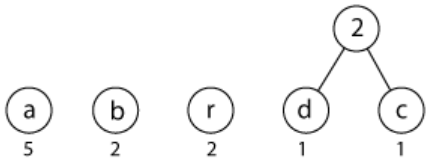
To see exactly how Huffman Coding works, let's look at how we can compress the word "abracadabra". Right now, this would take $8 \times 11 = 88$ bits to encode. Let's see how much smaller we can make this message with Huffman Coding.

First, we count the frequencies of each letter:

Letter	Frequency
a	5
b	2
c	1
d	1
r	2

Now that we know the frequency, we can start to build our tree. To start, we create vertices for each letter, and write the corresponding frequency beside it. We arrange these vertices with the highest number on the left.

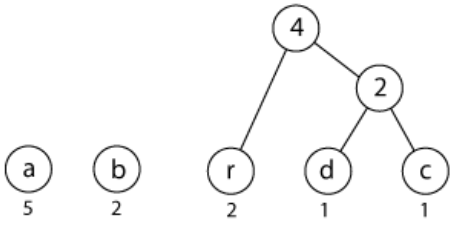
We then take the two letters with the smallest numbers (in this case c and d) and make a new vertex that joins these letters. This number in this vertex is just the sum of the frequencies.



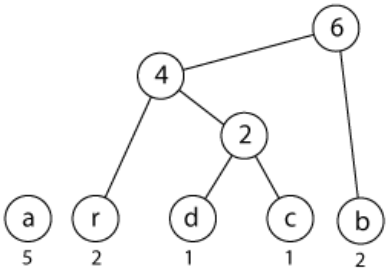
At each step, we take the two smallest numbers, and join them at a vertex. That vertex is labeled with the sum of the two numbers. We only care about the number at the top of a chain. In our example, since we have already joined c and d, we don't care about the number 1's beside them. We only care about the number 2 in the vertex that joins them.

When there are multiple options for the two smallest numbers, then you can choose either. Each choice will result in a different coding, but the size of the compressed message will be the same. What you want to keep consistent is having the smallest number on the right.

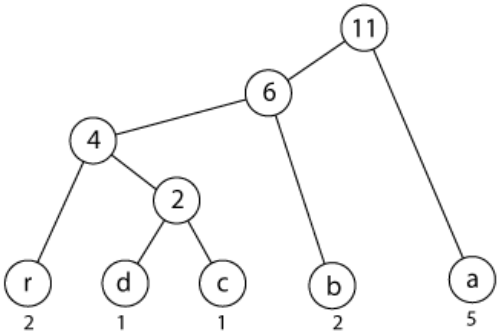
Using this method, we can complete our tree as follows:



The smallest numbers are 2 so we pick a pair of vertices and join them



4 and 2 are the smallest numbers. For consistency we keep the smallest number on the right



Once we join the last two parts, the tree is complete.

Now that we have our tree, we need to know how this try represents binary codes for each letter. The easiest way is to assign 0 to the left direction, and 1 to the right. To determine what the code is for each letter, we follow the tree down. Every time we go to the left, we add a 0 to the code and every time we go to the right, we add a 1.

This gives the following codes for each letter:

Letter	Code	Code Length	Frequency	Total Length
a	1	1	5	5
b	01	2	2	4
c	0011	4	1	4
d	0010	4	1	4
r	000	3	2	6

As we can see from the table, total length of the entire word is just 23 bits. That's almost 75% less space! The final binary message is simply 10100010011100101010001. These space savings would be magnified in a larger file with even more repeated letters, and it illustrates the power of some simple algorithms for solving problems and making our lives that much easier.

Problem Set

1. In your own words, describe what an algorithm is, and one reason why we use them today. An algorithm is a process or set of steps we use to solve a problem. Algorithms can allow us to solve a certain type of problem (e.g. sorting) even in complex cases (e.g. lists of millions of elements). Algorithms also help us describe a process so we can translate it into a work computer program. Without an understanding of the process the computer needs to go through to solve the problem, we can't easily tell the computer what to do.

2. Describe an algorithm for making a paper airplane. Be as detailed as possible. Trade algorithms with a partner, and follow their algorithm to build a plane. See whose plane flies the farthest. Answers will vary.

3. Series are often defined recursively, meaning that the next number in a pattern is determined by the previous terms.

   (a) The Fibonacci Sequence is $1, 1, 2, \ldots$ Each term is sum of the previous two terms. Formally, we write $f_1 = 1$, $f_2 = 1$ and $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$. For example, we know that $f_3 = 2$, but we could also recursively find this as $f_3 = f_2 + f_1$ and since we know $f_1 = f_2 = 1$, then $f_3 = 2$.

      i. Using the recursive definition $f_n = f_{n-1} + f_{n-2}$ find the 7^{th} Fibonacci number
      Using the recursive definition we have:

$$f_7 = f_6 + f_5$$
$$f_6 = f_5 + f_4$$
$$f_5 = f_4 + f_3$$
$$f_4 = f_3 + f_2$$

      We also know that $f_3 = 2$ and $f_2 = 1$. Therefore,

$$f_4 = f_3 + f_2 = 2 + 1 = 3$$
$$f_5 = f_4 + f_3 = 3 + 2 = 5$$
$$f_6 = f_5 + f_4 = 5 + 3 = 8$$
$$f_7 = f_6 + f_5 = 8 + 5 = 13$$

      ii. Find the 11^{th} Fibonacci number

Similar to the previous part, we can use the definition to find that:

$$f_8 = 13 + 8 = 21$$
$$f_9 = 21 + 13 = 34$$
$$f_10 = 34 + 21 = 55$$
$$f_11 = 55 + 34 = 89$$

iii. What is the value of $f_9 - f_6$
$f_9 - f_6 = 34 - 8 = 26$

(b) Given the following information $a_1 = 1$, $a_2 = 4$, $a_3 = 13$ and $a_n = 2a_{n-1} - a_{n-2} + 3a_{n-3}$ for $n \geq 4$. Find:

i. a_5
Using the recursive definition we have:

$$a_5 = 2(a_4) - a_3 + 3(a_2)$$
$$a_4 = 2(a_3) - a_2 + 3(a_1)$$

We already know the values for a_1, a_2 and a_3, and this means:

$$a_4 = 2(a_3) - a_2 + 3(a_1) = 2(13) - 4 + 3(1) = 25$$
$$a_5 = 2(a_4) - a_3 + 3(a_2) = 2(25) - 13 + 3(4) = 49$$

ii. a_7
Similarily, we know:

$$a_6 = 2(a_5) - a_4 + 3(a_3) = 2(49) - 25 + 3(13) = 112$$
$$a_7 = 2(a_6) - a_5 + 3(a_4) = 2(112) - 49 + 3(25) = 250$$

iii. $a_7 - a_6$
$a_7 - a_6 = 250 - 112 = 138$

4. Sort the list $53, 8, 32, 78, 29, 82, 5, 83, 67$ from highest to lowest using:

   (a) Selection Sort

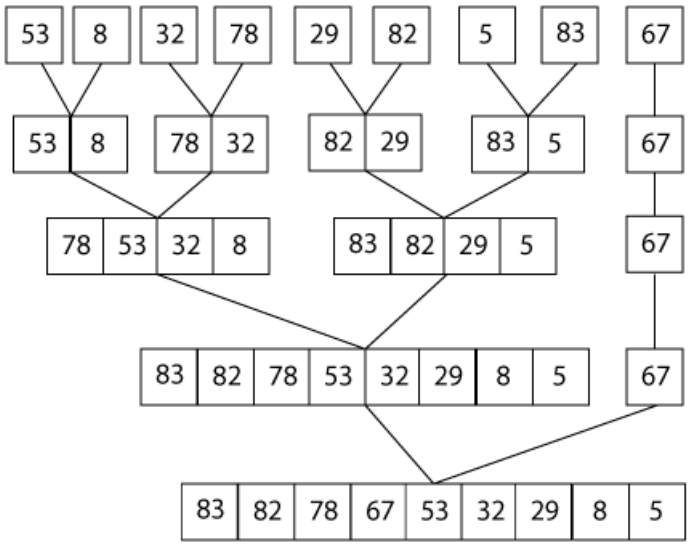
Sorted	Unsorted
	$53, 8, 32, 78, 29, 82, 5, 83, 67$
83	$53, 8, 32, 78, 29, 82, 5, 67$
$83, 82$	$53, 8, 32, 78, 29, 5, 67$
$83, 82, 78$	$53, 8, 32, 29, 5, 83, 67$
$83, 82, 78, 67$	$53, 8, 32, 29, 5$
$83, 82, 78, 67, 53$	$8, 32, 29, 5$
$83, 82, 78, 67, 53, 32$	$8, 29, 5$
$83, 82, 78, 67, 53, 32, 29$	$8, 5$
$83, 82, 78, 67, 53, 32, 29, 8$	5
$83, 82, 78, 67, 53, 32, 29, 8, 5$	

   (b) Insertion Sort

Sorted	Unsorted
	$53, 8, 32, 78, 29, 82, 5, 83, 67$
53	$8, 32, 78, 29, 82, 5, 83, 67$
$53, 8$	$32, 78, 29, 82, 5, 83, 67$
$53, 32, 8$	$78, 29, 82, 5, 83, 67$
$78, 53, 32, 8$	$29, 82, 5, 83, 67$
$78, 53, 32, 29, 8$	$82, 5, 83, 67$
$82, 78, 53, 32, 29, 8$	$5, 83, 67$
$82, 78, 53, 32, 29, 8, 5$	$83, 67$
$83, 82, 78, 53, 32, 29, 8, 5$	67
$83, 82, 78, 67, 53, 32, 29, 8, 5$	

   (c) Merge Sort

   Only the merging process is shown as the splitting process simply breaks the list into 9 single element lists.



5. Suppose you had a list that was already sorted such as $1, 2, 3, 4, 5$.

   (a) How would selection sort work? Where would it always find the smallest element?
   Selection sort will always take the first element in the list.

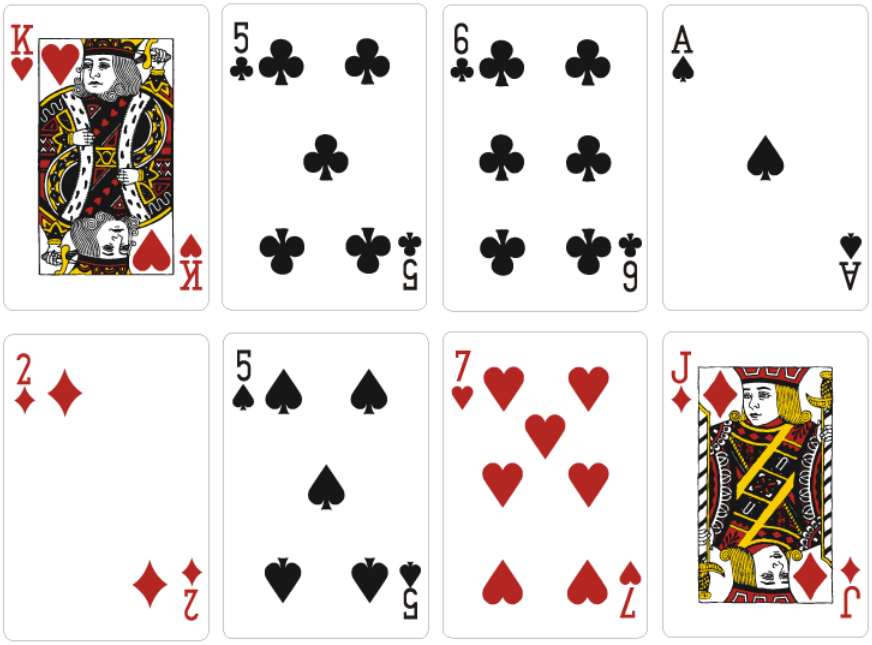
   (b) How would insertion sort work? Where would it always insert the next element?
   Insertions sort will always insert the element at the end of the list

   (c) Given a sorted list of n elements, which of selection and insertion sort do you think would run faster?
   Using the methods discussed in class, they will run in almost identical times. However, we could modify insertion sort to run faster in the case of a sorted list. We could do this by changing where we start the insertion process. As we did in this Math Circle, we started checking where to insert elements starting at the beginning of the sorted list. We could also have started at the end of the list. If we had done this, then insertion sort would always start its checks in the correct spot.
   In contrast, Selection Sort will always have to scan the entire unsorted list. This is because it needs to guarantee that the element it moves into the sorted list is the smallest. This can only be done by checking each element (since the computer doesn't know if the list is sorted to begin with or not).

6. Suppose we have a set of cards that we want to sort. We want to sort the cards first by suit - clubs then diamonds then spades and finally hearts - and then by number in increasing order - 2 to Ace.

   (a) Describe your own method for sorting the cards Answers will vary

   (b) Sort the following cards using both your algorithm and insertion sort.
   Only insertion sort is shown here. Each card is represented as a 2-digit code. The first digit (2-A) is the card number, the second (C, D, S, H) represents the suit.

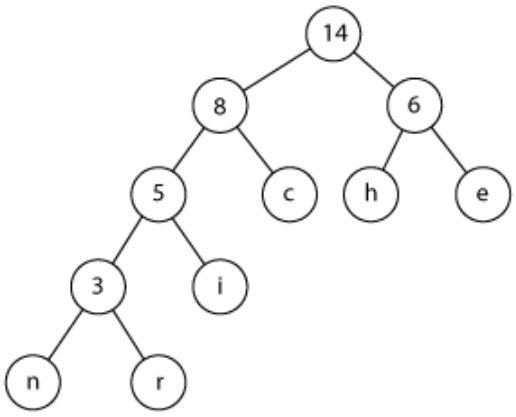


Sorted	Unsorted
	KH, 5C, 6C, AS, 2D, 5S, 7H, JD
KH	5C, 6C, AS, 2D, 5S, 7H, JD
5C, KH	6C, AS, 2D, 5S, 7H, JD
5C, 6C, KH	AS, 2D, 5S, 7H, JD
5C, 6C, AS, KH	2D, 5S, 7H, JD
5C, 6C, 2D, AS, KH	5S, 7H, JD
5C, 6C, 2D, 5S, AS, KH	7H, JD
5C, 6C, 2D, 5S, AS, 7H, KH	JD
5C, 6C, 2D, JD, 5S, AS, 7H, KH	

(c) Which algorithm was easier for you to follow? Why?
Answers will vary

(d) Which algorithm do you think would be easier for a computer to follow? Why?
Answers will vary

7. Using the following Huffman Coding tree, decode the message:
01100010000011011000100100000 1101111



We assign a 1 to to moves to the right, and 0 to moves to the left. This gives the following codes for each letter: We then use these codes to split the binary string.

Letter	Code
c	01
e	11
h	10
i	001
n	0000
r	0001

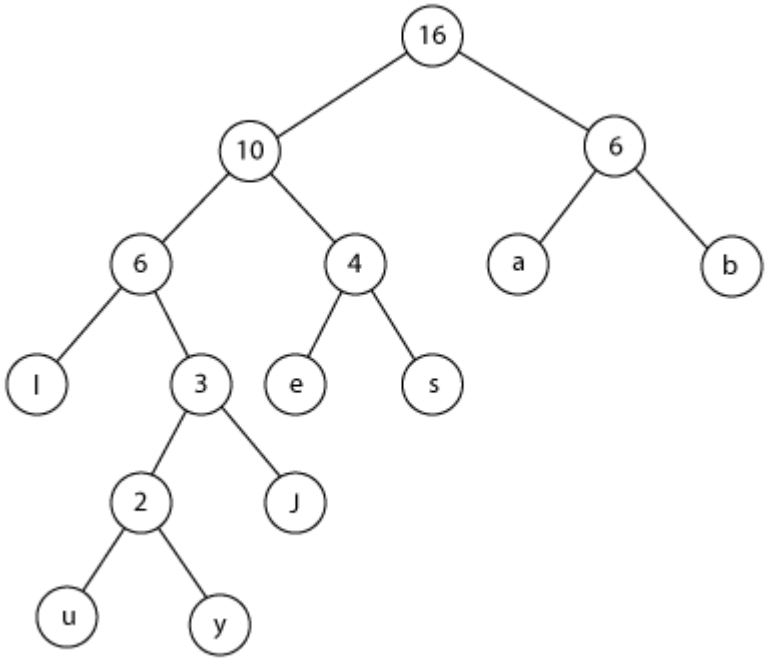
Each separation represents a new letter.
01 10 001 0000 01 10 11 0001 001 0000 01 10 11 11 Converting these codes to letters give the word *chincherinchee*

8. (a) Encode the message "BlueJaysBaseball" into binary using the Huffman Coding algorithm

   (b) Calculate the total space saved (in bits) by compressing the message. What percentage of space saved is this?
   We first calculate the frequencies of each letter. This is summarized in the table below: We then create the Huffman Coding tree. Note that there are multiple

Letter	Frequency
a	3
b	3
l	3
e	2
s	2
j	1
u	1
y	1

trees possible, as there are many steps where nodes have the same number. This is one possible tree.



Assigning 0 and 1 to the left and right paths respectively yields the following codes and code lengths. We can find the total length by multiplying the frequency by the code length. This

Letter	Frequency	Code	Code Length	Total Length
a	3	10	2	6
b	3	11	2	6
l	3	000	3	9
e	2	010	3	6
s	2	011	3	6
j	1	0011	4	4
u	1	00100	5	5
y	1	00101	5	5

means the total length is $6+6+9+6+6+4+5+5=47$. Originally, if we had used 8 bits for each letter, we would have needed $16 \times 8 = 128$ bits. This means the total space save is 81 bits, or 63%.

Challenge Problems

1. Think about why Huffman Coding works. Why is it that we can know exactly what letters each code represents when they are all different lengths? What if one code was 00 and another was 001? Can that even happen?
   We can use different code lengths for different letters because Huffman Coding is what is called a *prefix code*. This means that no code is the prefix (or start) of another code. So if we have the letter a as code 10 then Huffman Coding ensures that there are no codes like 1001 or 101. This is due to the structure of the coding tree. The letters are always at the end of the tree - there is nothing further down a path through a letter. Each path in the tree represents a different code. The only way for one code to be the start of a different one would be if there was a path that continued through the letter - except this doesn't happen because we only add things above the letters - never below.

2. Given a list of n elements in any order, which of selection sort, insertion sort, or merge sort would work the fastest (on average)? Why?
   On average merge sort will run the fastest. This is because it makes less overall comparisons than the other algorithms. An algorithm like selection sort or insertion sort needs to go through the majority of the list each time (either to find the minimum element or to find the correct place in the sorted list. These checks need to occur at each step. In contrast, merge sort makes these comparisons at each level. However, the number of levels is much smaller than the length of the list because at each level we divide the list in half rather than just decreasing the length of the unsorted list by one. This means the comparisons are done less often and so merge sort will run faster on average. For a more detailed and rigourous discussion of this, interested students can research *algorithmic complexity*.

3. Think about other sorting algorithms you may know (e.g. bubble sort, bozo sort, quicksort). Which of these do you think would work the fastest (on average) on a given list? Why? Is there any scenario where one algorithm may be the fastest, but in a different situation it wouldn't be?
   Answers will vary. Interested students are once again encouraged to look into algorithmic complexity