



Grade 7/8 Math Circles Computer Science & Algorithms

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 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?

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:

Sorted

Unsorted



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.

Sorted

Unsorted



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.

Sorted

Unsorted



$2 < 4$
Stop, 2 is in the correct spot

Sorted

Unsorted



$1 < 2$
Stop

Sorted

Unsorted



$3 > 1$
Move to next spot

Sorted

Unsorted



$3 > 2$
Move to next spot

Sorted

Unsorted



$3 < 4$
Stop

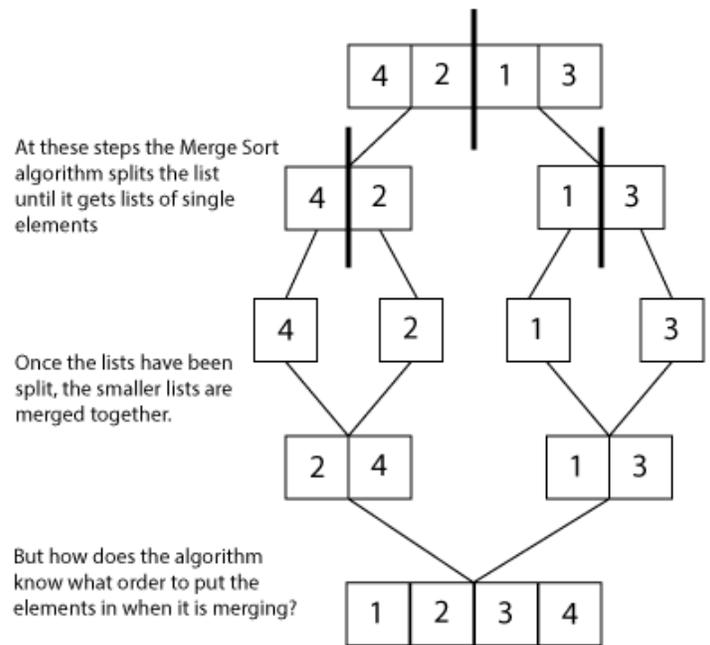
Exercise 4 - Sort the list from *Exercise 2* using the insertion sort algorithm. The list to sort is 14, 6, 11, 5, 20, 17, 1

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?

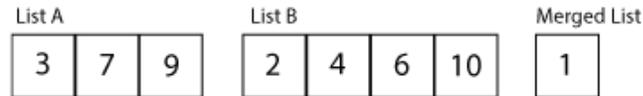
At each merging step the algorithm exploits the fact that each list it is merging together is already sorted. This means it only has to compare the first element in each list - if the smallest in list A is smaller than the smallest in list B , then it must be the smallest in both.

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

At each step we compare the first element from each list, and add the smaller one to our merged list



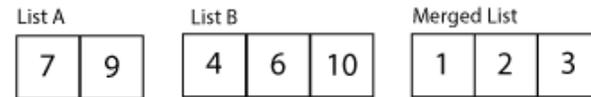
Step 1. $1 < 2$
Add 1 to merged list



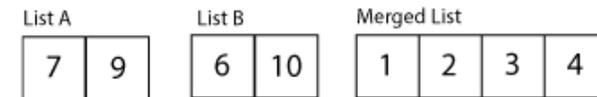
Step 2. $2 < 3$
Add 2 to merged list



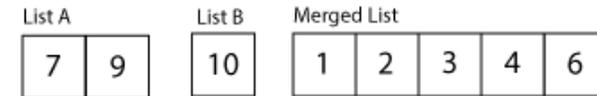
Step 3. $3 < 4$
Add 3 to merged list



Step 4. $4 < 7$
Add 4 to merged list



Step 5. $6 < 7$
Add 6 to merged list



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

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 0100110101100001011101000110100000100001. 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.

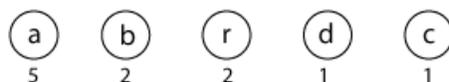
Huffman coding works by taking a message and assigning short (e.g. 1 or 2 digit) codes to the letters that appear most frequently. Less frequent letters are given longer codes.

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.

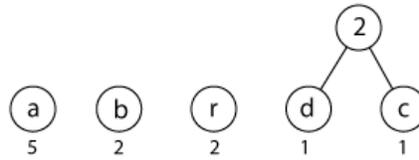
First, we count the frequencies of each letter:

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.



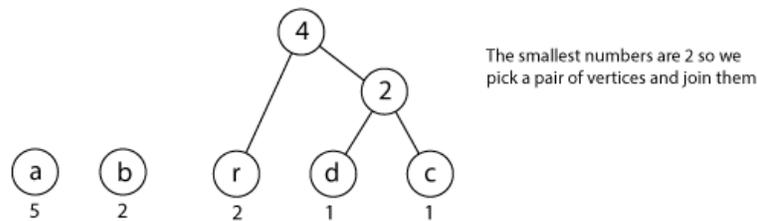
We then take the two letters with the smallest numbers (in this case c and d) and make a new vertex that joins these letters. This number in this vertex is just the sum of the frequencies.



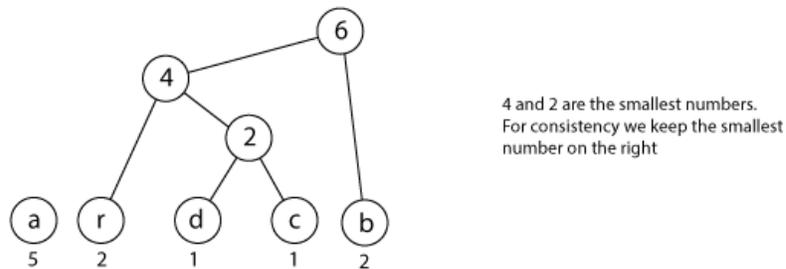
At each step, we take the two smallest numbers, and join them at a vertex. That vertex is labeled with the sum of the two numbers. We only care about the number at the top of a chain. In our example, since we have already joined c and d, we don't care about the number 1's beside them. We only care about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

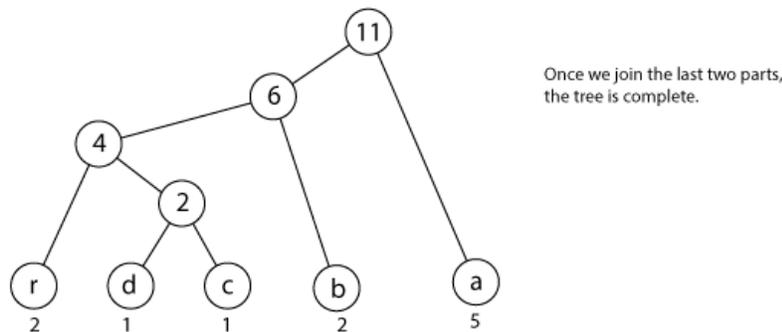
Using this method, we can complete our tree as follows:



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 tree 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.

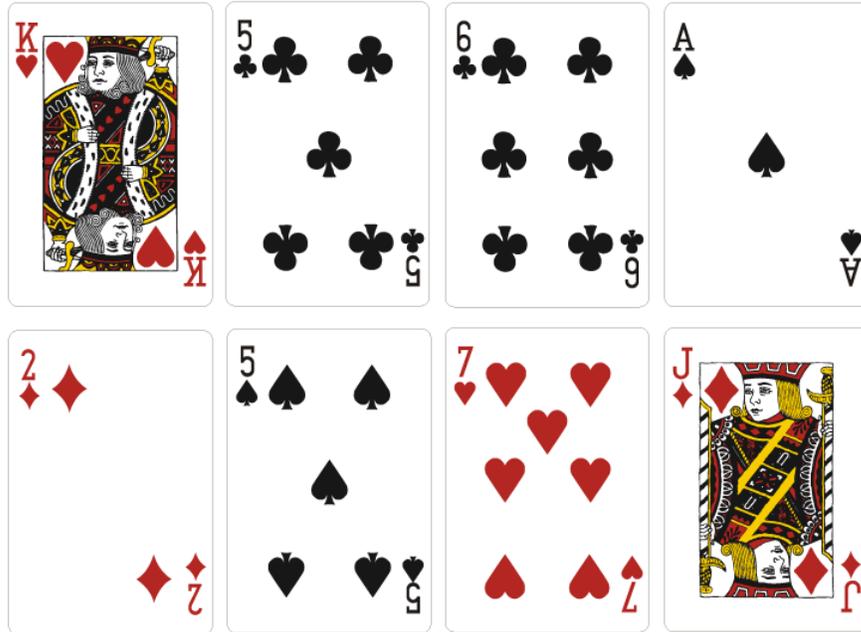
This gives the following codes for each letter:

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

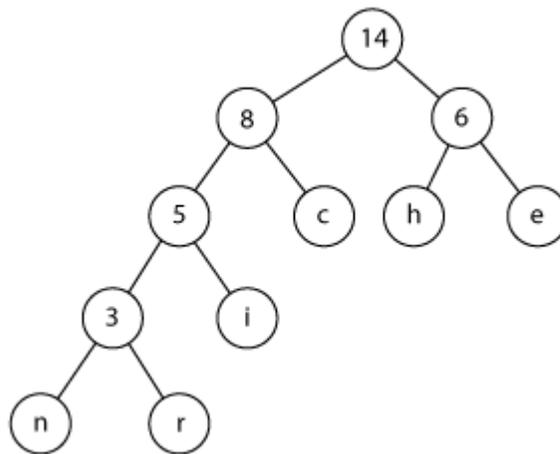
As we can see from the table, total length of the entire word is just 23 bits. That's almost 75% less space! The final binary message is simply 10100010011100101010001. These space savings would be magnified in a larger file with even more repeated letters, and it illustrates the power of some simple 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.
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.
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 $1, 1, 2, \dots$. 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 7th Fibonacci number
 - ii. Find the 11th Fibonacci number
 - iii. What is the value of $f_9 - f_6$
 - (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
 - ii. a_7
 - iii. $a_7 - a_6$
4. Sort the list 53, 8, 32, 78, 29, 82, 5, 83, 67 from highest to lowest using:
 - (a) Selection Sort
 - (b) Insertion Sort
 - (c) Merge Sort
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?
 - (b) How would insertion sort work? Where would it always insert the next element?
 - (c) Given a sorted list of n elements, which of selection and insertion sort do you think would run faster?
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
 - (b) Sort the following cards using both your algorithm and insertion sort.



- (c) Which algorithm was easier for you to follow? Why?
 - (d) Which algorithm do you think would be easier for a computer to follow? Why?
7. Using the following Huffman Coding tree, decode the message:
011000100000110110001001000001101111



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?

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?
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?
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?