Family Science Program
Introduction to Algorithms

Tracy Kimbrel
IBM TJ Watson Research Center

This booklet describes the activities used in the IBM TJ Watson Research Center Saturday Family Science Program session on algorithms. In some cases there may be more detail than is covered during class.

Introduction to Algorithms

At first, perhaps you'll be surprised that in the Family Science session on algorithms (which is one of the Computer Science sessions), we do not use any computers. Some of the most interesting and important problems in computer science are not solved by using computers. Instead, they are solved by using one's head. That is what this session is about.

Many people think mathematics is hard, boring, or both. An important lesson of this session is that math doesn't have to be either hard or boring. Math can be very fun! Mathematics is much more than addition, subtraction, multiplication, and division. Sometimes it doesn't even involve numbers!

The heart of computer science is mathematics. This session aims to give students a taste of the math behind the design and use of computers.

Computers are great tools for getting work done, finding information, and playing games. However, computers are very dumb! Computers can do only exactly what they are told to do. Computers are given instructions by programming them. Programming simply means making up a list of instructions for a computer to follow. The list of instructions is called a program. A program is a way to express a mathematical idea called an algorithm. An algorithm is like a recipe for baking a cake or chocolate chip cookies. In this session, you'll get a chance to work with several algorithms. Perhaps you'll even discover one on your own!

Algorithms are encoded for computers in special languages called programming languages. We won't worry about the details of this. Instead, we'll describe algorithms in English.

Activity 1: searching and sorting

Searching

One of the simplest, yet most important, problems that computers need to solve is the problem of searching a list of items to find a particular item. Computers perform this simple task over and over. For instance, when a bank customer puts his card into a bank machine to withdraw money, the bank machine-which has a computer inside it-must locate the customer's account records among those of all the bank's customers. Each time a grocery store clerk scans a grocery item, the clerk's cash register-which has a computer inside it-must locate the name of the product to determine the correct price. Think of all the different items on a grocery store's shelves. The computer has to search a very long list to find the product! Each time a World Wide Web user visits a Web site, some computer on the Internet has to search the list of Web sites to find the Web page the user requested.

These are just a few of the many different situations in which the problem of searching comes up. You are probably familiar with this problem already. Have you ever looked for a phone number in a telephone book or looked up a word in a dictionary? Imagine how much longer it would

take to perform these tasks if the names in the phone book or the words in the dictionary were not in alphabetical order!

The simplest algorithm for searching a list is called linear search. The item that is being searched for is compared to each item in the list, one at a time, to see if they are the same. The search ends when the item being sought is found, or the end of the list is reached without finding it. This algorithm can be described by the following list of instructions. (Remember that we are using English, rather than a computer programming language, to describe algorithms. Thus we are leaving out a translation step that would be necessary to make this algorithm work on a real computer.)

1. Begin with the first item on the list.
2. Compare the item on the list to the item you are looking for.
3. If they are the same, go to step 5.
4. If there are no more items in the list, stop and report that the item was not found; otherwise, move on to the next item and return to step 2.
5. Report that the item has been found, and give any associated information (such as the definition of a word, if you are searching for a word in a dictionary).

Notice that the linear search algorithm may end up performing three steps for each item in the list. This would take a very long time, if the list is very long. A better, faster algorithm used by computers for searching sorted lists is called binary search. Binary search is similar to the method you use when you look for a word in a dictionary. It works by cutting the list roughly in half repeatedly, and eliminating about half of the items in the list at each step. This requires only about fifty steps (examining only ten of the items) for a list of a thousand items, about a hundred steps (examining only twenty of the items) for a list of a million items, and only about a hundred and fifty steps (examining only thirty of the items) for a list of a billion items!

1. Begin with the middle item on the list.
2. Compare the item on the list to the item you are looking for.
3. If they are the same, go to step 7.
4. If there are no more items in the list, stop and report that the item was not found; otherwise, continue with step 5.
5. If the item you are looking for comes before the item on the list, discard the last half of the list and return to step 1.
6. If the item you are looking for comes after the item on the list, discard the first half of the list and return to step 1.
7. Report that the item has been found, and give any associated information.

Notice that we have left out several details in our algorithm. For instance, how do you find the item in the middle of the list? Which item is in the "middle" if there are an even number of items? You can not cut the list exactly in half if there are an odd number of items. We have also not said what it means for one item to "come before" another. Our algorithm can be used to search list of numbers (ordered by value), words or names (ordered alphabetically), or any other type of item which has a natural ordering. We would have to worry about these things if we were

to write a computer program for this algorithm. However, we don't need to worry about these details to understand what's really important about the algorithm. What's really important is that the algorithm is much faster than linear search (despite the fact that it has more instructions).

We will describe algorithms even more informally in the rest of this booklet. Keep in mind, however, that the details would have to be specified very carefully if we were to make these algorithms work on a real computer.

Sorting

Notice that for binary search to work properly, the list of items must be in order. How does the list get put in order in the first place? This is the problem of sorting. As with the search problem, some algorithms for sorting are faster than others. We will learn about three different algorithms for sorting. They are called insertion sort, quick sort, and merge sort.

Insertion sort works by inserting the items, one at a time, into a list that is always in sorted order. At the beginning, the "list" of sorted items is empty. The first item is easy to add to the list, since there are no other items to worry about. The second item is added to the list either ahead of, or behind, the first item, depending on whether it comes before or after the first item in the proper ordering. Each new item is placed in the list according to the proper ordering. The new item is compared to the first item, then the second, and so on, until an item is found that the new item should come before in the list. The new item takes this place in the list, and all the items later in the list move back one position.

Insertion sort is a simple and natural algorithm, but for long lists it may take a long time. Quick sort and merge sort are algorithms that take less time for long lists. We won't go into the reasons for this, but the algorithms are simple enough to describe and to experiment with.

Quick sort works by first picking an item from the set of items to be sorted. This item will be called the "pivot." Each other item is compared to the pivot to see whether it should come before or after the pivot. The items are divided into two groups, the group that comes before the pivot and the group that comes after. Notice that now we know the correct position of the pivot in the sorted list. The two groups still have to be sorted, but we know that all the items in the first group come before the pivot, and the pivot comes before all the items in the second group. Because of this, we never need to worry about comparing items from the first group to items from the second. How do we sort the two groups? The answer is that we use quick sort again, this time on smaller groups! Quick sort is what is known as a "recursive" algorithm. This means that the algorithm uses itself to solve pieces of the original problem and combines the solutions of the pieces to form the final solution. It is very important that the pieces of the problem get smaller each time the algorithm starts again. Otherwise the algorithm would never finish!

Our final sorting algorithm is merge sort. Merge sort works by merging short sorted lists into longer sorted lists. Suppose that we have two lists that are sorted, and we want to have a single sorted list. We can easily determine which item comes first in the single list, just by looking at the first items in each of the two lists. Whichever one comes earlier must be the first item in the

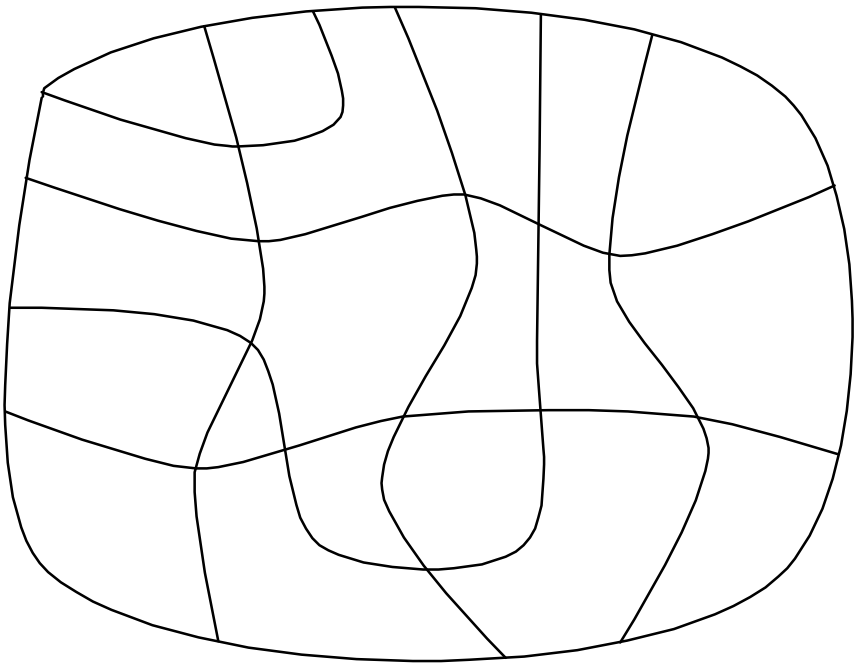
final list. We don't have to worry about any items other than the first ones in each list, because they are sure to come later in the final list. So we remove whichever one came earlier from its list, and put it at the head of the final list. Then we repeat the process. Each time we remove whichever item should come earlier from the head of its list, we add it to the tail of merged list.

Activity 2: map coloring

You may have noticed that maps (of the United States, or of the world) usually are colored in a particular way. States or countries that share a border are given different colors. This way, it is easy to tell the states or countries apart. (If two countries meet at a single point, but do not share any border, it is okay to make them the same color.)

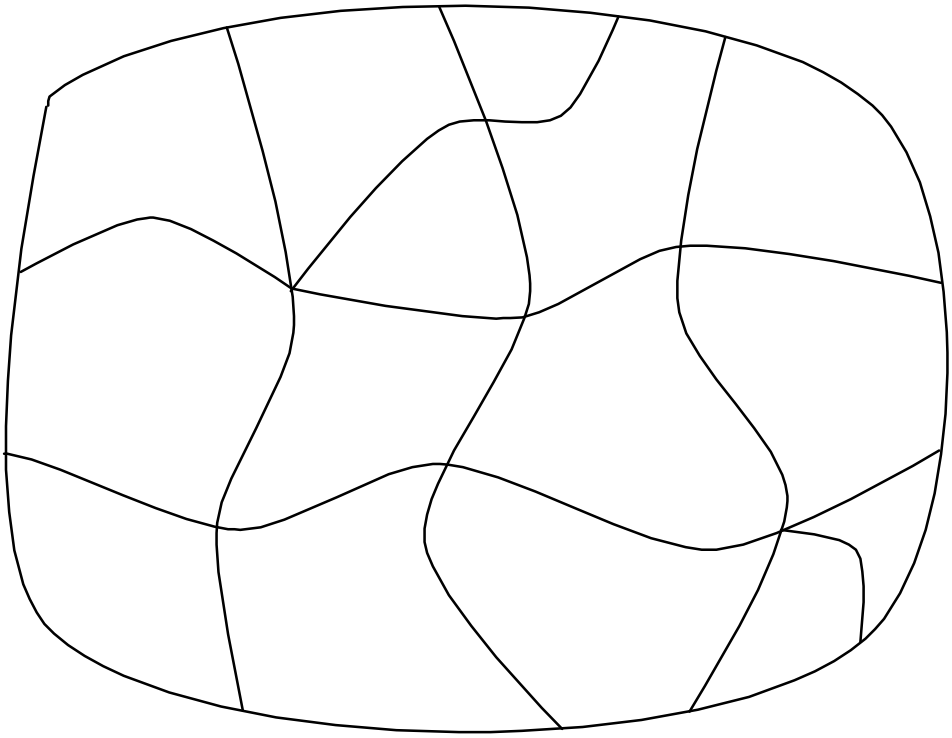
Imagine a very poor map maker who can not afford many crayons. Can you help the map maker color his maps using as few crayons as possible?

Here is a (fictional) map of a continent that requires only two different colors:

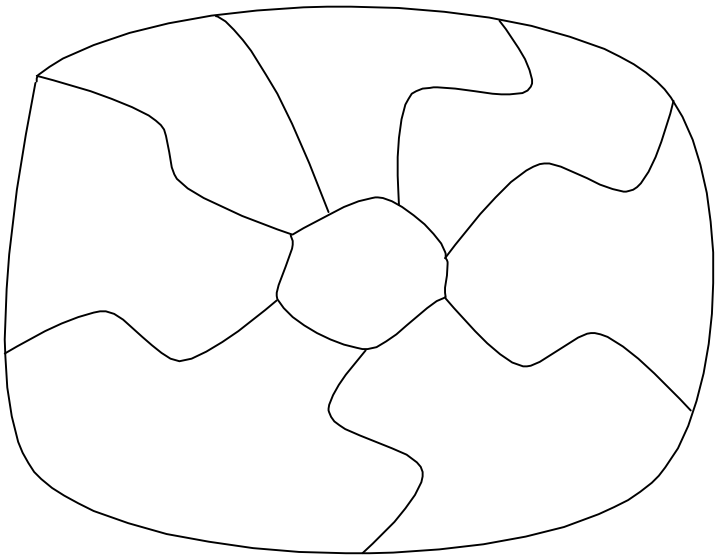


Introduction to Algorithms

This map requires three colors:



This map requires four colors:



Any map can be colored with four colors, but sometimes it is hard to tell whether three colors are enough or four are needed. Mathematicians puzzled over this problem for centuries before proving that four colors are enough for any map, but that's another story.

It is not very hard to tell whether two colors are enough. An algorithm that is sometimes called the Have-to algorithm can be used to decide whether two colors are enough, and to color the map if so. Start with any country, and color it, say, red. Because that country is red, its neighbors have to be another color, say blue. Those countries' neighbors have to be red, and so on. If you ever get stuck, you will have to use more than two colors.

You might want to try making your own maps, for you and your friends or brothers or sisters to color.

Activity 3: minimum spanning trees

You have probably know what the World Wide Web is, and you have probably heard of the Internet. The Internet is just a lot of computers, which contain portions of the World Wide Web (among other things), all connected together by wires of some sort. These wires are called a network. Telephones are also connected together by system of wires, also called a network.

One problem faced by telephone companies and by companies that provide Internet services is that of finding the least expensive places to run the wires that make up the network. Placing wires between a pair of cities is expensive. The cost varies depending on the pair of cities to be connected. This can be for many reasons. If two cities are far apart, a lot of wire is needed. If

there is a mountain or a river between two cities, it costs more to run the wires over, around, or through the obstacle.

In this activity, you will learn about a problem known as the Minimum Spanning Tree problem. You are given a list of cities, and for each of several pairs of these cities, the cost of connecting them by a wire. Your task is to find the least expensive way to connect all the cities. There does not have to be a wire between every pair of cities, but there must be some path along one or more wires between every pair of cities.

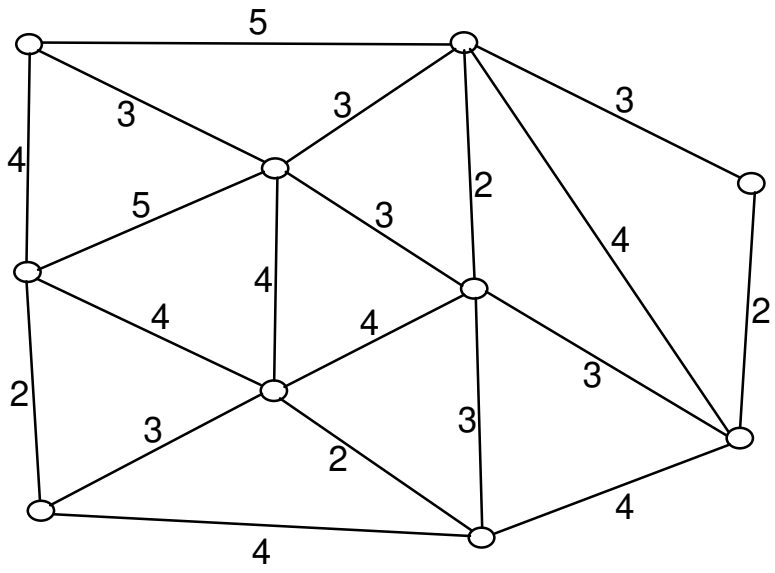
A cycle is a path from a city back to itself, that does not retrace any steps. A set of wires that does not contain any cycles is called a tree. A spanning tree is a tree that connects all the cities. A minimum spanning tree is one that costs the least possible. That is where the Minimum Spanning Tree problem gets its name. It is not hard to see that any set of wires that contains a cycle cannot be the least expensive way to connect all the cities. This is because you could eliminate one of the wires that make up the cycle, and there would still be a path between any two cities.

There are two well-known algorithms for finding a minimum spanning tree. They are known as Kruskal's algorithm and Prim's algorithm.

Kruskal's algorithm works by considering the wires one at a time, in order of increasing cost; that is, from the cheapest to the most expensive. At the start, none of the cities are connected. As each wire is considered, it is included in the network if it connects two cities that are not already connected by a path of wires. That is, if the wire does not form any cycles with the wires that are already part of the network, it is made part of the network; otherwise it is not needed.

Prim's algorithm works by "growing" a tree, starting with any city. At each step, the cheapest wire that connects the starting city to a new city-one that is not already connected by some path to the starting city-is added to the network.

Below is an example of the Minimum Spanning Tree problem. A best possible solution (there are more than one) appears at the end of this booklet.



Activity 4: finite state machines

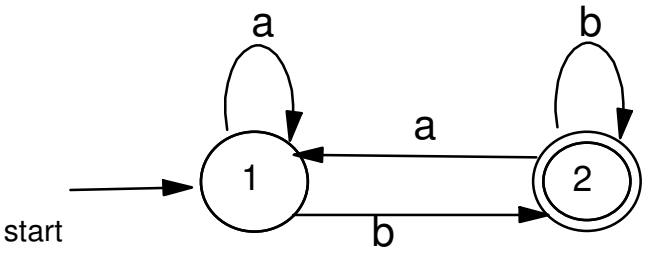
In this activity, you'll learn about a simple model for a computer known as the Finite State Machine. Finite state machines are too simple to model the full power of a general purpose computer. However, many of the activities carried out by a computer can be modeled by finite state machines. Also, many types of computer systems, including aircraft flight safety systems and nuclear reactor controllers, are designed using similar models.

A finite state machine can be thought of as a machine to recognize words from a language. The words don't look much like English words, however! They aren't like Russian or Italian words either, or like words from any other human language. Instead they might look like aaabb or abababccc or xoxoxox. In our examples, the words will be composed of only the letters a and b.

We can think of a finite state machine "liking" some words and not "liking" other words. Different finite state machines like different words. We can take a word, such as abab, and see whether a finite state machine likes the word. Let's see how.

Here is a picture of a simple finite state machine:

Introduction to Algorithms



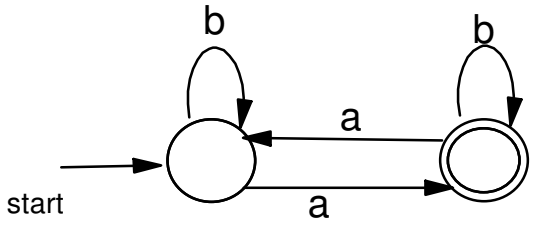
This machine has just two "states," represented by the circles. State 1, which is labeled "start," is a special state, the state of the machine when it starts. State 2 is also special. It is a "final" state, indicated by the double circle. (A finite state machine can have more than one final state. Also, the states don't have to be numbered; they are numbered in this example only so it will be easy to describe the action of the machine.) We can trace the action of the machine when it is given a word as input, as follows.

The machine starts in its start state, of course. Look at the first letter in the word. Let's use the word "abab" as our example, so the first letter is "a." Find the arrow labeled "a" leading from the start state, and see which state it points to. That will be the state the machine enters after reading the letter. In our example, the arrow labeled "a" leads right back to state 1, so the machine stays in state 1. Now look at the next letter in the word- "b" in our example. Follow the arrow; this time the machine moves to state 2. Next we follow the arrow leading out of state 2 labeled by the next letter in the word, the second "a." This takes us back to state 1. Finally, we reach the last letter in our word, the second "b." The arrow out of state 1 labeled "b" leads to state 2. We have read all the letters in our input word, so our machine is done. Now we notice that the machine is in a final state. This means our machine is "happy" because it was fed an input word that it likes! The machine is said to "accept" the word.

Introduction to Algorithms

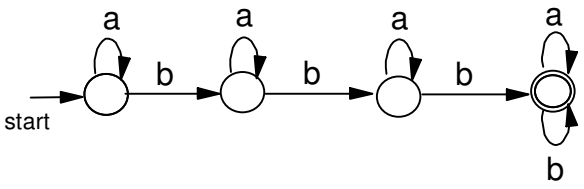
Trace the action of our machine on the word "aaba." This time, the machine goes through the following sequence of states: 1, 1, 2, 1. The machine does not end up in the final state, so it is not happy with this word; it doesn't like it, or "rejects" it. Notice that even though the machine passed through the final state, it still doesn't like the word because it did not end up in the final state. It is happy with a word only when it ends up in a final state. With a little thought, you can see that this machine likes all words that end with "b," and doesn't like any words that end with "a." This is because every time it reads a “b,” it enters or stays in state 2, and every time it reads an “a,” it enters or stays in state 1.

Below is another finite state machine. Trace the action of this machine on each of the following words: abba, aaabb, and bab. Can you describe the kinds of words this machine likes? Hint: notice that each time it reads an "a," the state changes, but each time it reads a "b," the state stays the same. (The answers to the questions in this section are given at the end of this booklet.)

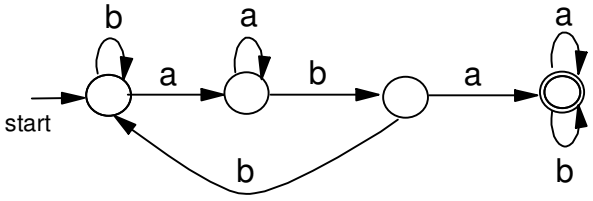


What kinds of words are accepted by the next machine? This machine is fairly easy to understand.

Introduction to Algorithms

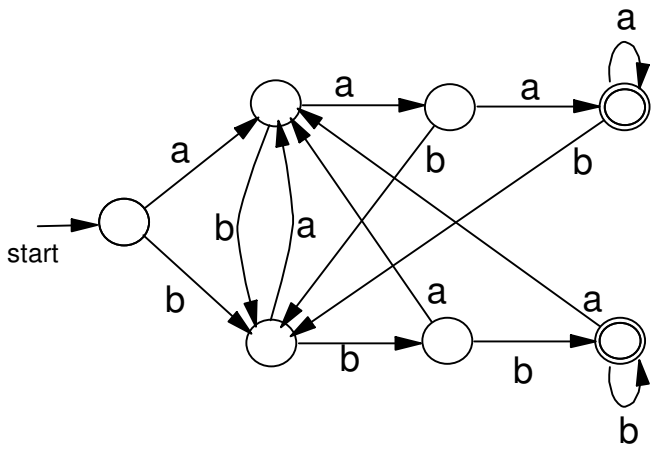


Here is a more complicated machine. Try these words: bbbab, aaaba, and aababb. Can you describe the kinds of words this machine likes?

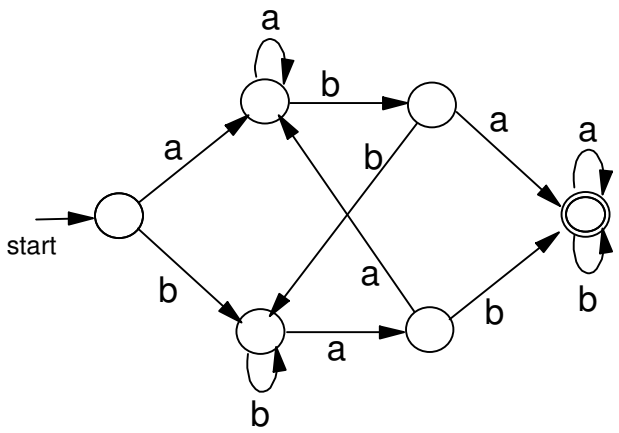


Introduction to Algorithms

Here is another machine. Can you figure out which words it likes? Try these words: abaa, abaaa, aabb, aabbb.



Here is one more machine. These words may help you determine what kinds of words it likes: aabba, aaba, bbba, bbbabaa.



Try designing your own finite state machines! Can you design one that likes all words that end with two a's? All words that contain two a's in a row? Words that contain the sequence aab? Make up your own pattern and design a machine that accepts words that match your pattern!

Activity 5: cryptography

Cryptography means "secret writing." And that's just what it is about: secret codes for writing messages, so that only someone who knows the code can read the messages. There are two sides to the game of cryptography. One side is devising a secret code so that only the sender and receiver can decode the message. The other side is the problem of figuring out or "cracking" someone else's code, to decode their messages.

Cryptography is actually a science that is thousands of years old, dating back much farther than electronic computers. But in the computer age, the need for good cryptographic codes is greater than ever! Remember that computers are dumb machines, capable only of executing the instructions that are given to them. But they can execute those instructions very quickly! Computers can "guess" codes much more quickly than a human can, and so can guess many codes in a short amount of time. Codes that were unbreakable only a few decades ago are easily cracked by modern computers. Luckily, computers can also be used to encrypt messages using much more complicated, and harder to crack, codes than in the past.

There are three main ideas used in cryptographic systems. These are substitution, transposition, and the use of secret keys. Substitution means that each letter of the original message is replaced by a different letter in the encoded version. Transposition means that the letters of the message are rearranged or scrambled in the encoded version. A secret key can be used to indicate the way

in which the substitutions or transpositions are done. All three of these techniques can be combined. In this activity, we will use secret codes that illustrate these three ideas.

Our first code is a simple substitution code. Below are two copies of the alphabet, one above the other. However, the letters are shifted so that the alphabet begins with n in the second copy, and z is followed by a.

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d	e	f	g	h	i	j	k	l	m

We can use this to encode a message. Simply replace each letter in the message with the letter that appears directly below it in the second, shifted copy of the alphabet. For instance, the letter s has the letter f underneath it, e has r underneath it, c has p underneath it, r has e underneath it, e has r underneath it, and t has g underneath it, so the word "secret" is encoded as "frperg." Can you decode the message "zngu vf sha?" (The answer is given at the end of this booklet.) We could call this code "Shift 13," since the second copy of the alphabet has been shifted by 13 letters.

Here is another code, in which the second copy of the alphabet has been shifted by only two letters. We can call this "Shift 2."

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b

Make up your own message, encode it using Shift 2, and give it to a friend. See if your friend can decode the message. Then explain how it was encoded, so your friend can decode your message.

Next we will learn a transposition code, called a grid code. A grid code works by writing out the letters of the message in a grid. The letters are then read off the grid in a different order from the order in which they were written. Let's encode the message "Thanks for coming to Family Science." We will leave out punctuation and use only lower case letters, since these would give clues to the code. (Remember that we don't want anyone who doesn't know the code to be able to read our messages!) We write out the letters across the rows of a rectangle, called a grid. We'll use a grid with five columns. Here is the grid:

t	h	a	n	k
s	f	o	r	c
o	m	i	n	g
t	o	f	a	m
i	l	y	s	c
i	e	n	c	e

Now, read the letters off by columns! Read the first column, then the second, and so on. Just squeeze all the letters together into one long string. This gives the message "tsotiihfmoleaoifynnrnasckcgmce." This looks pretty hard to understand! Decoding is just the reverse process. Write out the letters down in columns, and you get the grid above. Now you (or someone you send the message to) can read the letters off the rows. The result is "thanksforcomingtofamilyscience." It is not too hard to figure out where to add spaces to get the original message.

You can encode any message you like using this method. There are a few things to be careful about, though! First, you have to make sure the person receiving the message knows how many rows and columns are in the grid. Also, what happens if the number of letters in the message doesn't happen to fill up the grid exactly? This wasn't a problem in our example. If this happens, just add some extra letters (called "nulls") to fill out the grid. Use whichever letters you like, but don't use the same letter many times! Your code could be cracked by someone who detects the pattern created by using the same letter repeatedly. Here is an example of the use of nulls to fill out a grid. The message "Secret codes are fun" is encoded as "stsuecancorardebeefc." Decoding gives us “secretcodesarefunabc.” It is easy to put back the spaces and delete the nulls to arrive at the original message.

s	e	c	r	e
t	c	o	d	e
s	a	r	e	f
u	n	a	b	c

Introduction to Algorithms

Now let's see an example of using a key to help encode a message. We will use a grid code again, but this time we'll read the columns off in a different order. The key will tell us the order in which to read off the columns. We can use the same message and grid as our first example. We'll use the key word "tracy." (Any word will work as long as it has as many letters as there are columns in the grid, and no letters are repeated.) Replace each letter in the key by its position in alphabetical order among the letters of the key. For our key, this gives us 43125. Write the numbers above the grid, like this:

4	3	1	2	5
t	h	a	n	k
s	f	o	r	c
o	m	i	n	g
t	o	f	a	m
i	l	y	s	c
i	e	n	c	e

Now read the columns off in the order indicated. The middle column has the number 1 above it, so we read off aoifyn. The column with a 2 above it is nrnasc, and so on. We end up with the encoded message "aoifynnrnaschfmoletsotiikcgmce." Decoding is the reverse process. We would write the first six letters in the middle column, the next six letters in the fourth column, and so on. Notice that even if a person trying to crack the code knows that we used the grid code, he would still have a hard time decoding our message unless he had the key.

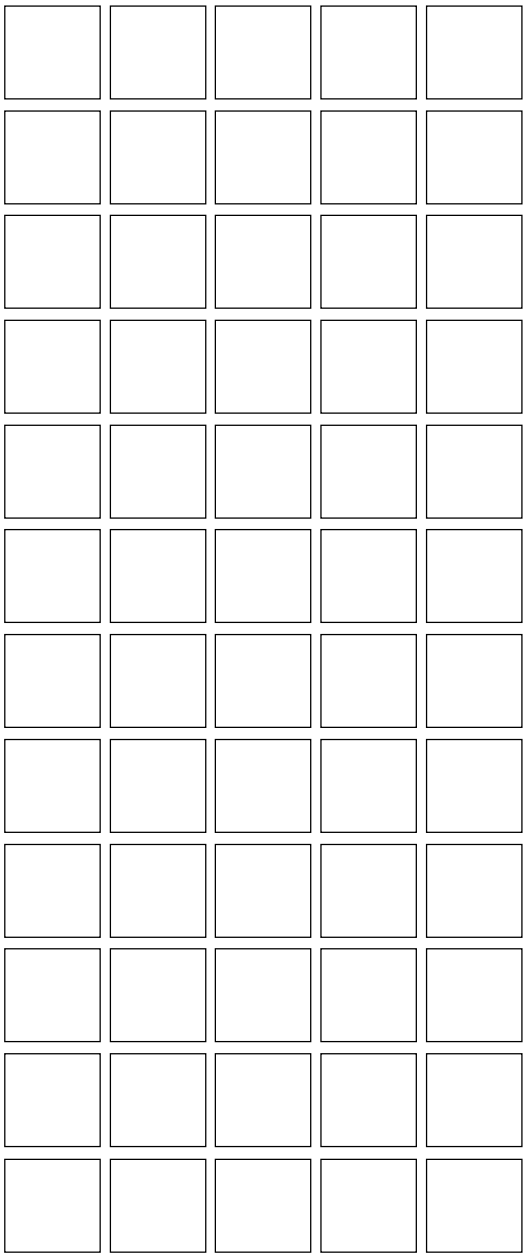
The methods of encrypting messages we have used can be combined to make a code that is even harder to crack! You could first use a substitution code, and then write the message in a grid, and even use a key to change the order of the columns. Make up your own variations on the ideas we've used! We have barely scratched the surface of the many ways in which messages can be encoded. Your local library or bookstore has books containing many more encryption schemes.

Activity 6: probability theory

Probability theory is a branch of mathematics concerned with the likelihood of events that are not known with certainty. You are probably (no pun intended) familiar with probabilities in weather forecasts. The forecast might say there is a thirty percent chance of snow. If the people making the forecasts are doing their job correctly, this means that out of every hundred days they say there is a thirty percent chance of snow, it will snow on about thirty of those days.

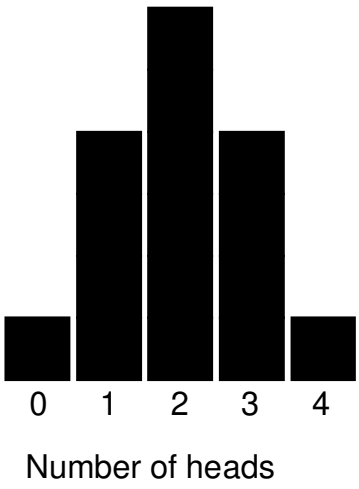
We won't have time to explore the many connections between probabilities and computers, or the many ways in which computer scientists use probability theory. However, in this activity, you will learn a very interesting thing about probability theory. Some of the most important problems in probability theory are really no more difficult than just the first thing you ever learned about mathematics and numbers: counting!

You will need four coins and a pen or pencil. It is helpful to use four different kinds of coins, such as a penny, a nickel, a dime, and a quarter. You will toss all four coins at once, and count the number that come up heads. The next page contains a blank chart that can be used to record the numbers of times zero, one, two, three, or four heads are seen. Each time a particular number of heads comes up, fill in one of the blank squares in the corresponding column. Start at the bottom and work your way up. Toss the coins 16 times or so, and mark the number of heads seen each time.


0
1
2
3
4
Number of heads

Introduction to Algorithms

Below is a chart (these charts are called histograms, by the way) showing the most likely outcome. Your histogram will probably be slightly different, but will have the same general shape. You will probably see two coins come up heads most often, followed by one and three heads. Four heads or zero heads (which is the same as all coins coming up tails) are least likely.

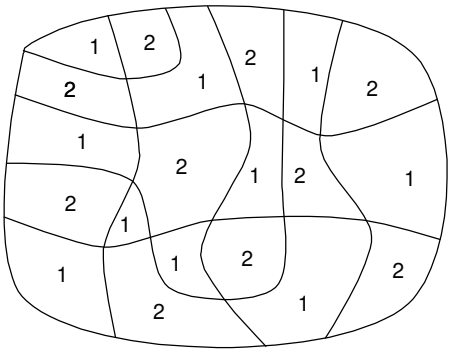


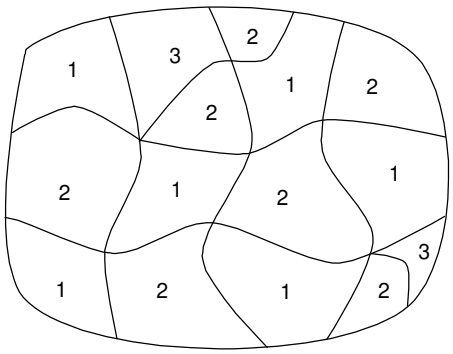
Why are one, two, or three heads much more likely than zero or four? The answer is easy to understand, simply by counting. How many combinations of the possible coin tosses result in zero heads? The answer is that there is only one such outcome: all four coins must come up tails. How many combinations result in one coin coming up heads? There are four such outcomes. The penny could come up heads, and the other three coins could come up tails. Or the nickel could come up heads, and the penny, the dime and the quarter could come up tails. You should be able to name two other combinations that result in one coin out of the four coming up heads. Finding the number of combinations that result in two of the four coins coming up heads is a little bit more complicated, but not too hard. You should be able to count six different combinations. You should also be able to determine the number of combinations that result in three heads and the number that result in four heads.

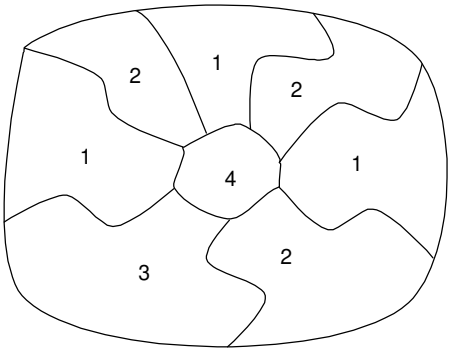
Introduction to Algorithms

Answers

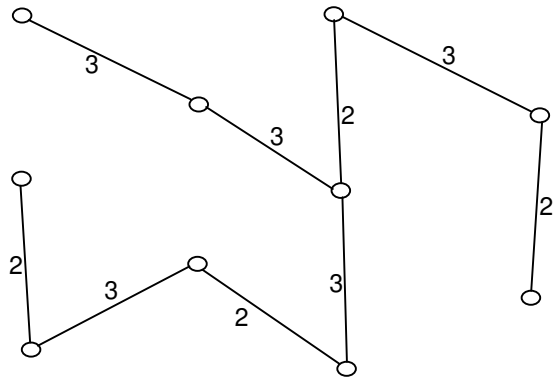
Here are “colorings” of the maps; substitute your favorite colors for the numbers 1, 2, 3, and 4.







Here is a solution to our
minimum spanning tree problem with cost 23:



The second finite state machine likes words that have an odd number of a's, and doesn't like words with an even number of a's. The third machine likes words with at least three b's. The fourth machine likes words that contain the sequence aba. The fifth machine likes words that end with three a's in a row or three b's in a row. The last machine likes words that contain the sequence aba or the sequence bab.

Using the substitution code Shift 13, the message "zngu vf sha" decodes to "math is fun."

Introduction to Algorithms

Acknowledgments

The author wishes to thank Professor Mike Fellows of the University of Victoria for relating his experiences teaching computer science and mathematics topics to children and for providing materials. The idea for the map coloring activity and the minimum spanning tree activity, including the example problem, are taken from an unpublished manuscript by Fellows and Professor Neal Koblitz of the University of Washington. The idea for the finite state machine activity is taken from the MegaMathematics workbook. The cryptographic codes used here (and many others) are described in the book by Gardner.

Bibliography

Casey, Nancy, and Fellows, Mike.
This is MEGA-Mathematics -- Stories and activities for mathematical thinking, problem solving, and communication -- The Los Alamos Workbook.
World Wide Web version at http://www.c3.lanl.gov/mega-math.

Fellows, Michael R., and Koblitz, Neal.
Combinatorially based cryptography for children (and adults).
Unpublished manuscript, 1999.

Gardner, Martin.
Codes, Ciphers, and Secret Writing.
Dover Publications, New York, 1984 (originally published in 1972).

Introduction to Algorithms

Notes for Teachers

This section describes how these activities were presented during the IBM T.J. Watson Family Science Program in the Winter of 2000. The author would be delighted to think that anyone was using this booklet in any way, but offers his experience in the Family Science Program to those interested. Both the children and the parents seemed to enjoy these activities very much.

To illustrate the advantage of binary search over linear search, I used several copies of a popular fiction book and several copies of a dictionary. I handed out the books and asked people to find a simple word. I used the word iceberg; I don't think it actually occurred in the fiction book. Of course, those who received dictionaries found the word quickly, and those who received the fiction book gave up. I used the scanner at a grocery store (and the computer inside it) as an example of a system in which records sorted by keys of some kind make binary search much more efficient. The large number of different items on the shelves of a supermarket made a good example that was familiar to everyone. I then asked the children to guess how many records need to be examined by binary search in lists of a thousand, a million, and a billion records (ten, twenty, and thirty, respectively).

Search was a natural motivation for sorting, of course. To illustrate sorting algorithms, I asked the children and their parents to stand up in an area large enough for everyone to move about and with a wall against which all the people could line up. I started with insertion sort, and sorted by first names. I made sure people understood that to act like a computer, we had to perform one simple step at a time, i.e., one comparison at a time. Thus to insert a person in the list required several comparisons, once the list started getting long. I usually asked if people were getting bored after about ten insertions, gave up on insertion sort, and mentioned that the algorithm was correct, but slow. I then sorted the people by birthdate (month and day only, not year) using Quicksort. It was not significantly faster on lists of thirty or so people,of course, but I explained that on longer lists it was much faster. I carried out Quicksort to completion. Finally I divided the people into three or four groups, and asked the groups to sort themselves by height. Then I merged the lines into one line, emphasizing the fact that I had only to look at the first person in each line to decide which would come next in the final ordering.

For the map coloring activity, I handed out letter-size copies of the maps included in this booklet and colored tabs that are available in office supply stores. The tabs have two advantages over crayons: they are faster and you can change your mind about which color to assign to a region. Nearly everyone could find a two-coloring of the two-colorable map, and often the children could explain the algorithm to find it. Usually the children could explain why the odd cycles in the three-colorable map prevented a two-coloring, and why the odd cycle along with the region adjacent to all members of the cycle prevented a three-coloring of the four-colorable map. After they were done coloring the maps, I mentioned that this was a problem that mathematicians worked on for centuries before proving that four colors are always enough, and explained that for large and complex maps, deciding between three- and four-colorability is thought to be a problem too hard (I didn't use the term NP-hard) even for computers. I also tried to motivate the problem more practically by mentioning related problems such as scheduling airplanes and crews.

Introduction to Algorithms

I introduced MSTs by asking the children what the Internet and WWW are physically made of, looking for "a bunch of computers connected by wires" as the answer. I also mentioned telephone networks, and asked whether if there are wires from New York to Chicago and from Chicago to Los Angeles, there needs to be a wire from New York to Los Angeles to make a call. I handed out the MST instance in this booklet, along with a copy with all the edges removed (and pencils), and asked people to find the cheapest network they could that connected all the points. After several minutes, I would usually point out that any network with cycle could be made cheaper; often people had cycles in their solutions under construction. Once most of the people had found an optimal solution, I would illustrate Kruskal's and Prim's algorithms on a very simple example.

I made a large finite state machine on the floor (the last of the examples in this booklet) with masking tape. States and labels were pre-printed using a computer, and taped down, and then arrows were made of masking tape. I gave each child four inputs to test, and they walked through the FSM to see if each word was accepted. "Winning" words were rewarded with a piece of licorice. Make sure each child gets the same numbers of winners and losers! (I used two of each for each child.) Then I handed out copies of the FSMs in this booklet, and asked for each one whether the children could give examples of accepted and rejected words, and then asked if anyone could identify the pattern and explain why it was correct. We usually got through the first three or four, and I left the rest as "homework."

For the cryptography activity, I motivated the concept with the example of using a credit card number to make purchases on the World Wide Web. I would hand out copies of the substitution codes and write down the first message "math is fun"- encoded, of course- and ask whether anyone could read it. Then I revealed the code and let them decipher it. Once they knew the message, I would ask whether anyone though math was hard or boring. I would admit that long division is perhaps not the most fun activity in the world, but did mention that it was important to know how to do it nonetheless. I then asked what we had been doing for the last hour and a half or so, and often got the response "mathematics." Everyone agreed that what we had been doing was fun. I then handed out small blank sheets of paper, and asked them to use the second substitution code to send a message to a neighbor. After a few minutes, I wrote the encoded version of the grid code message "secret codes are fun,"and then showed them how to decode it using a blank grid that I handed out. I also asked them to send messages to one another using a grid code. There wasn't time to use the version of the grid code with a key. I would briefly describe the use of keys in cryptography, and then tell them that they had learned about the three basic concepts used in cryptography- substitution, transposition, and the use of secret keys- although, of course, that the codes we had used were very easy to crack by computer, and that more sophisticated versions of these methods were used in practice.

For the last activity on coin tossing, I handed out small paper cups each containing a penny, a nickel, a dime, and a quarter. I made sure to illustrate covering the cup with one hand and shaking very well to ensure a random trial. After everyone had done a large number of trials (many insisted on continuing until one column "won the race" and was filled in to the top), I would reveal a prediction printed in large type by computer, to make it clear I had done it in

advance. The prediction was “More twos than zeroes or fours.” I asked whether they thought I was psychic, or whether there was some simple explanation for the fact that I could make such a prediction. I would then illustrate the probabilities of each outcome by counting the ways it can come about. Zero, one, three and four are easy to describe in words only. I drew a chart to illustrate the six ways in which the outcome two can be reached.