Spring 2020

Randomized Algorithms and How Society Uses Them Everyday

Rosaley Milano
Otterbein University, rosaley.milano@otterbein.edu

Randomized Algorithms and How Society Uses Them Everyday

Otterbein University

Department of Mathematics

Westerville, Ohio 43081

Rosaley Milano

8 April 2020

Submitted in partial fulfilment of the requirements

for graduation with Honors

David Stucki, MS

Project Advisor — Advisor's Signature

Pei Pei, Ph.D

Second Reader — Second Reader's Signature

Erica Van Drop, MS, ACE-CPT, ACSM-GEI, TSAC-F

Honors Representative — Honor Rep's Signature

Acknowledgments

First, I would like to thank my advisor Professor Stucki. From the beginning he has been a constant source of support. I ended up in his office after struggling for weeks to find a topic. But that day he assured me that we would find a topic together and whether he would be a good fit as an advisor or not, he would be there to help. It was this unwavering support that got me through some hard times during this project. No matter what happened I always felt like I had someone in my corner and without his guidance and help I am not sure that I would have completed this project. I am also eternally indebted to him for introducing me to computer science. Thank you for sharing your passion with me and in turn helping me realize mine.

Next I would like to thank Dr. Pei, Dr. Berndt and Dr. Van Drop for their time and input. I would also like to thank my parents for their consistent support and encouragement during this project and throughout my time at Otterbein. They know more than anyone how difficult these four years have been and throughout it all I have been so grateful to have them as my parents.

Abstract

Randomness is an interesting and very beneficial phenomenon. In computer science randomness facilitates great advances in efficiency but topics like randomized algorithms aren't taught until someone enters graduate school. This paper provides undergraduates as well as people unacquainted with computer science an opportunity to explore the topic of randomness by guiding them from essential topics all the way through the graduate level topic of randomized algorithms. Topics like what an algorithm is, how they are represented and the history that brought them into existence bring the reader up to speed before diving deeper into randomized algorithms. A discussion of complexity theory is used to motivate the reader as to why randomized algorithms are so important and beneficial. Then the remainder of the paper is used to describe the categorization of randomized algorithm by goal as well as some of the applications of randomized algorithms including hashing, random sampling and generating random numbers. The paper concludes with a discussion of a real-world application of randomized algorithms, the Kidney Exchange Problem, where incompatible patient-donor pairs are matched with others to facilitate successful kidney transplants.

Table of Contents

Introduction

Chaos and order, each only exist in the absence of the other. Following Newton's discoveries in calculus and classical mechanics, scientists started leaning towards the belief that everything in the universe could be determined, there was no such thing as chaos. Pierre-Simon de Laplace, a French physicist, was one scientist who believed this. He claimed that if someone (now known as Laplace's demon) knows the position and forces acting on every atom in the universe, all past and future movements could be calculated. But as later discovered by Heisenberg, measuring the position and velocity of a particle at the same time is impossible, thus knocking Laplace's theory off its feet. As we will explore in this paper, chaos and randomness are not only possible but also provide surprising benefits for computers in the form of great advances in efficiency.

Algorithms

Almost every activity in life can be expressed as an algorithm. To motivate this let's first consider a simple example, a recipe. If you have ever cooked using a recipe you know that the steps are clearly ordered and include all the information needed to carry each one out, provided you have some knowledge about cooking. In addition, the steps in a recipe can be completed so a result is produced, and it only requires a finite amount of time to complete all the steps (Schneider 13). The average person reading a recipe doesn't need to acknowledge that it meets all these requirements but for them to complete the recipe, each must be true. Let's consider another example, something less structured, making a bed. Consider the steps below to make a bed.

1. Check to make sure the fitted sheet is around the mattress. If it isn't, pull each corner around the mattress. Otherwise continue to the next step.
2. Pull the bottom most layer up towards the top of the bed. Continue this step until there is only one layer left and that layer is the comforter.
3. Place the pillows at the top of the bed near the head board and pull the comforter up over them.

Just like a recipe, all the steps are clearly ordered and contain all the information needed to complete them. All the steps can be completed by any capable human to produce a bed that was made in a reasonable amount of time.

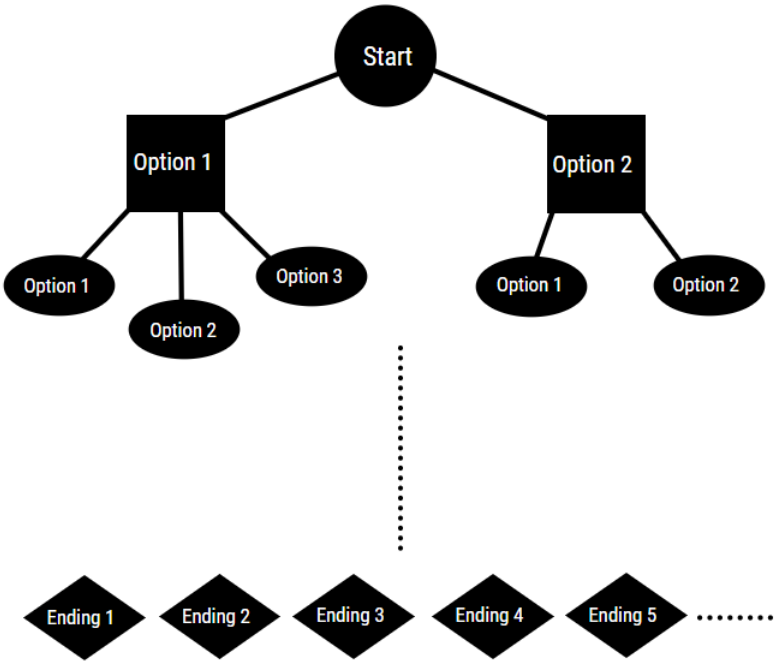
As stated above, most everyday activities can be expressed as algorithms. After analyzing recipes and how to make a bed you may be catching on to the requirements of an algorithm. According to Schneider and Gersting, "An **algorithm** is a well-ordered collection of unambiguous and effectively computable operations that, when executed, produce a result and halt in a finite amount of time" (Schneider 12). But what does all that mean?

- Well ordered: The order of each operation or step is clearly defined.
- Unambiguous: Each operation is defined such that there is only one way to interpret and carry it out.
- Effectively Computable: Each operation or step can be mechanized for a machine to carry out.
- Produces a result: At the end of the algorithm something is produced. It doesn't have to be the result we were attempting to produce but there is some indication that the algorithm executed.

- Halts in a finite amount of time: A result can be produced and will not take forever (Schneider 12-16).

So far, we have only provided examples of one type of algorithm, a deterministic algorithm. The defining factor of a **deterministic algorithm** is that for some fixed input it will always produce the same output. Think back to our recipe example and suppose you are making cookies. If you follow the recipe, you will always end up with cookies. Each batch will mostly likely vary slightly but using the same recipe will always produce cookies and not something else like a cake. The fact that a recipe will always produce the same thing is what makes it a deterministic algorithm.

A **randomized algorithm** is an algorithm that uses randomness to decide what to do next. Thus, randomized algorithms fall into the category of **nondeterministic algorithms** because for some fixed input it will always produce a different output. A simple example of a randomized algorithm is a modified version of a choose your own adventure story. Instead of choosing what path to take, at each crossroad a random number would be generated and that would



Choose your own adventure

determine what path to take. This modified version of a choose your own adventure story is a randomized algorithm because randomness is dictating what happens and even with the same starting point there will be a different ending. Now that we have covered the basics of what algorithms are, we can begin to address what they have to do with computers.

Representing Algorithms

Computers are inherently stupid. Without explicit instructions that are given in a language they can understand, they are unable to do anything. Humans write **computer programs**, a set of instructions that the computer can understand, to be able to communicate with computers. It is important to note that an algorithm and a computer program are two separate entities. An algorithm can be written in natural languages, such as English, like the example above or it can be written into a computer program. And a computer program can have an algorithm in it, or it can just be a meaningless set of instructions. But for all intents and purposes, it is valid to view programming as the act of translating algorithms so computers can carry them out (Hromkovic 47).

There are many different programming languages each with their own rules and syntax. You may be asking yourself how there can be so many different programming languages, aren't computers supposed to be stupid? Computer hardware that your program runs on never actually sees the code that is displayed on your screen. A compiler takes the code you have written in whatever language and converts it to machine language which is what is run on the computer's hardware. So, although there are many different programming languages, they are only used to make the programmers' job easier by

allowing them to not have to deal directly with the machine language. Every computer at its core runs on machine language.

Often when first learning about algorithms and how to represent them, pseudocode is used**. Pseudocode** is a structured subset of English but without all the details of a specific programming language (Schneider 14). So, pseudocode mimics all the structures of a program without having to learn all the details of a specific programming language. There are three main categories of operations that we can represent; sequential, conditional and iterative operations. These types of operations are seen in most programming languages but, as mentioned before, have different syntax.

Sequential Operations

A sequential algorithm is an algorithm that uses sequential operations in order to execute each operation one after another from top to bottom. So **sequential operations** are operations that execute one after another without affecting the sequence of the algorithm. The three types of sequential operations are computations, input and output. Input and output operations are used to communicate with the user; they either get or give information. Computational operations are operations that do arithmetic and any kind of data manipulation (Schneider 48-50).

- Computations

  Set the value of "variable" to "arithmetic operation"

  Ex. Set the value of x to 4+5

- Input

    Get a value for “variable1”, “variable2”, …

    Ex. Get a value for x, y and z

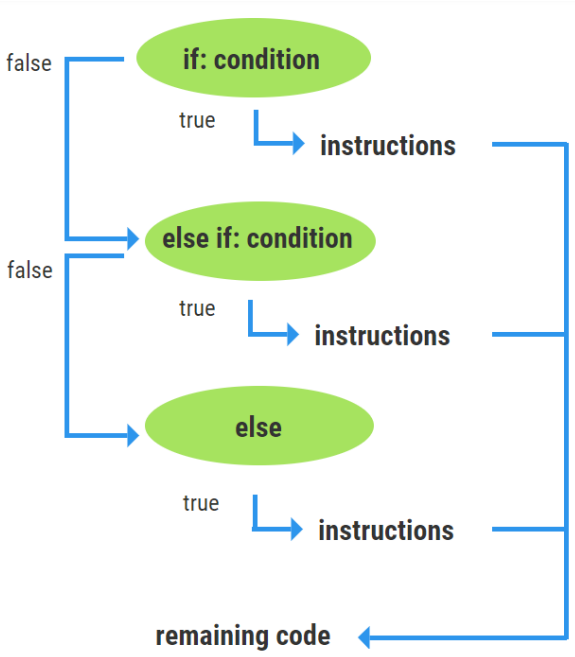
- Output

    Print the value of “variable1”, “variable2”, …

    Ex. Print the value of x, y and z

Conditional Operations

Conditional operations, unlike sequential operations, affect the flow of the algorithm. As the name suggests when a **conditional operation** is used there is a condition that is checked. And depending on if that condition is true or false, different sets of instructions are executed. Once the correct set of instructions has been executed the program moves on to the instructions outside of the conditional operation. In pseudocode conditional operations are represented by an if statement (Schneider 51-53).



- If statements

    if “a true or false condition” is true then

        instructions

    else (the above condition was false)

        other instructions

Ex.

if (x = 5) then

Set the value of x to x + 2

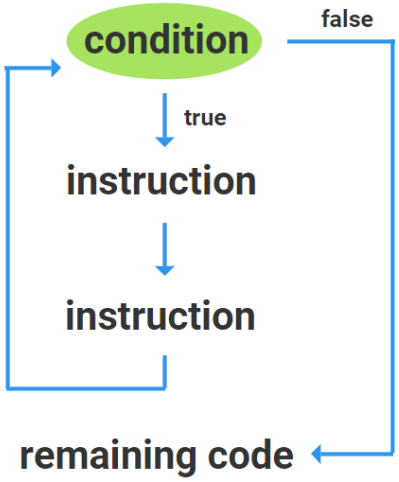
else

Set the value of x to x + 3

Value of x before the if statement	X = 2	X = 3	X = 5
Value of x after the if statement	X = 5 (2 + 3)	X = 6 (3 + 3)	X = 7 (5 + 2)

Iterative Operations

Iterations operations are also operations that affect the flow of the algorithm. But unlike conditional operations, **iterative operations** will continue to execute until the condition is no longer true which is why they are typically called loops. There are two types of iterative operations, while do and do while loops (Schneider 54-58).



- While do

1. Evaluate the condition
    a. If the condition is true execute the following instruction
        i. Once you get to the bottom of the instructions go back up to the condition and reevaluate it.
    b. If the condition is false, skip the block of instructions

Ex.

While (x < 5) do

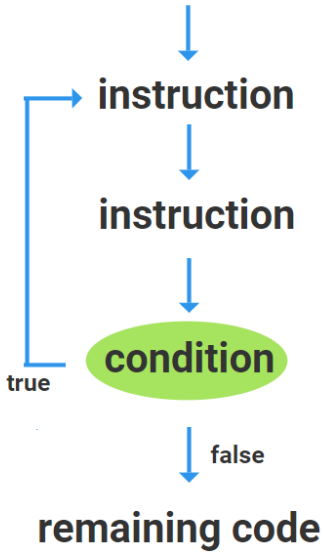
Print the value of x

Set the value of x to x + 1

Get a value for y

Value of X before the while do loop	X = 2	X = 6
Value of X after the while do loop	X = 5	X = 6
Output	2 3 4	No output, the while do never executes

- Do while
  1. Do the set of instructions
  2. Evaluate the condition
     a. If the condition is true return to the top of the block of instructions and execute them again
     b. If the condition is false, move on to the remaining code in the program



Ex.

Do

Print the value of x

Set the value of x to x + 1

While (x < 5)

Get a value for y

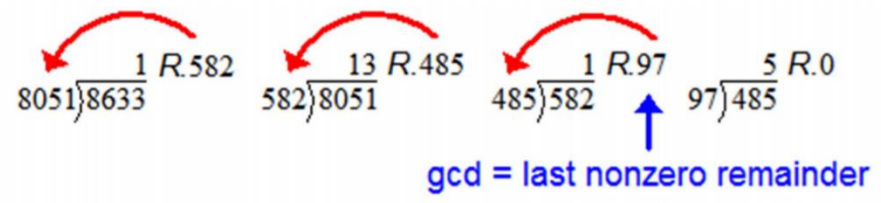
Value of X before the do while loop	X = 2	X = 6
Value of X after the do while loop	X = 5	X = 7
Output	2 3 4	6

It is important to note that a do while loop will always execute at least once as with the case where X = 6 in our last example. The same is not true for while do loops.

Alan Turing, an influential figure in modern computing, established that sequential, conditional and iterative operations are the fundamental operations; any problem that can be solved by an algorithm will only need these operations.

History

The history of algorithms starts with the Euclidean Algorithm, a deterministic algorithm from 300 BC. The Euclidean algorithm, which is still used today, finds the largest integer that divides A and B, which are also both integers. The greatest common divisor of A and B, GCD(A, B), is found by first dividing B into A and labeling the remainder as r1. The remainder r1 is then divided into B and the new remainder is labeled r2. This is repeated



until the remainder is zero. The GCD is the last nonzero remainder. The example above shows how to use the Euclidean algorithm to find GCD(8633,8051). This very basic deterministic algorithm can be used to reduce fractions or find the solutions to a linear equation. Refer to the appendix for a more in depth discussion of Euclid's algorithm and consequences of it (Burger).

After the development of the Euclidean algorithm, deterministic algorithms to solve a wide range of problems continued to be discovered and integrated into the world. And like the Euclidean algorithm, they were all being carried out manually by human computers (prior to WWII the word computer was a reference to a profession rather than a mechanical device).

During the industrial revolution in the 19th century that all changed. Instead of humans performing algorithms, machines were being produced to perform algorithms that

required repetitive mechanical tasks. Unlike humans, they could be given step by step instructions and perform a task all day without getting bored or needing to take breaks.

Then in the 1940's there was another shift from machines performing mechanical tasks to machines performing mental tasks. The machine that allowed this shift to occur was the digital electronic computer which in the early days employed deterministic algorithms to solve problems. As computers started to become more popular, it became obvious that it was much more beneficial to society if computers could do the lower level work, leaving humans more time for higher level thinking. This has persisted throughout the years as, although computers are used for a multitude of tasks, at their core they are still used to help solve problems by executing a series of deterministic algorithms.

Consider one of the simplest tasks that computers can perform, addition. Surprisingly the deterministic algorithm that computers use to perform addition is very similar to the way that everyone learns how to add in elementary school.

1. Add all the values in the ones position. If the sum is larger than 9, carry the tens digit, always 1, to the next column.
2. Add all the values in the tens position. If the sum is larger than 9, carry the tens digit, always 1, to the next column.

This process continues until every position had been added together. Ultimately addition is an extremely easy thing to do, we learn it very early in our education. But even though it is an easy task it is more beneficial to have a computer add than do it ourselves. By having a computer deal with the easy things like addition, it allows us humans to focus on the harder more important task at hand. Take for example a calculus student trying to

solve a problem. By passing easy arithmetic like addition off to the computer, the student can focus on the more important task at hand, using calculus to solve the problem.

Since the development of computers spurred the development of deterministic algorithms, what spurred the development of randomized algorithms? Randomized algorithms really started gaining traction in the 1970's when computer scientists realized they were able to,

1. Outperform, in terms of runtime, the best-known deterministic algorithms for the same problem
2. They are much easier to implement than deterministic algorithms of comparable performance (Motwani 6).

A huge influence in randomized algorithms taking off was the development of two efficient algorithms to determine if extremely large numbers, of approximately a hundred digits, are prime. One algorithm was developed by Solovay and Strassen while another was developed by Rabin which was based on work from Miller.

One may be asking themselves why it is so important to determine if a number that large is prime or not, but large primes are the back bone of cryptology. **Cryptology** is a branch of computer science which deals with the writing and solving of ciphers. A **cipher** is an algorithm to encode or decode a message. A simple example of a cipher is a Caesar cipher. Using this encryption technique involves writing the message out as normal. Then each letter in the message is replaced with the letter 3 places to the right in the alphabet. For example, anywhere in the message there is an "A" it gets replaces with "D", "B" gets

replaced by “E” and so on until you get to “X” which gets replaced with “A”, “Y” which is replaced with “B” and “Z” which is replaced with “C” (Schneider 409-410).

Encoding														
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	…
⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	
A	B	C	D	E	F	G	H	I	J	K	L	M	N	…

Ex. Original Message: Computer science is awesome.

Encrypted Message: Frpsxwhu vflhqfh lv dzhvrph.

The message can then be sent and if anyone intercepts it, it will look like gibberish. Once the message reaches the person it was intended for, they do the exact opposite to get the message back to English, the alphabet it shifted 3 letters to the left. So anywhere there is a “D” it gets replaced with “A”, “E” gets replaced by “B” and so on until you get to “A” which gets replaced with “X”, “B” which is replaced with “Y” and “C” which is replaced with “Z”.

Decoding														
A	B	C	D	E	F	G	H	I	J	K	L	M	N	…
⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	⬇	
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	…

Ex. Encrypted Message: Exw pdwk lv ehwwhu.

Original Message: But math is better.

But in all reality, this is not a very secure way to encode a message. Modern encryption techniques are much more complicated, thus the need to find extremely large prime numbers. **RSA** is an encryption technique that makes use of large prime numbers. In this scheme there is a key to encoding the message which is public and is the product of two very large prime numbers, found with a primality test. But the key to decode the message is kept private and in order to find it you must find the prime factors of the public key. Thus, RSA's success is based on the difficultly in finding the prime factors of a very large number, the public key. So, if someone needs to send some important information over the internet it is important to use an encryption technique like RSA to keep anyone who may intercept the information from figuring out what it says (Schneider 417-418).

Thus, an algorithm that can efficiently find prime numbers to make a public encryption key is very important. If a deterministic algorithm was used to try to figure out if a number, n, with one hundred digits was prime, it would need to check if any number between 2 and $\sqrt{n}$ would evenly divide the value. Checking values up to $\sqrt{n}$ confirms the primality of n because n is made of two factors, a and b. We know that a or b will be less than $\sqrt{n}$ because otherwise $a * b$ would be larger than n. For prime numbers $a = 1$ and $b = n$, or vice versa. So, if we can't find values for a and b between 2 and $\sqrt{n}$, n is prime (Dietzfelbinger 1). Using this mathematical property makes things much easier but not easy enough. Consider a value with three digits, say 100. A deterministic algorithm would have to check if any value from 2 to 10 would divide 100. But if you jump up to seven digits, say 1,000,000, now it must check all values from 2 to 1000 to see if any of them divide 1,000,000. These two examples are not a big deal for a computer but when you are trying

to find a value that divides a number with hundred-digit, it becomes unrealistic for a computer to do using a deterministic algorithm. We are left to use randomized algorithms.

Solovay and Strassen as well as Rabin were both able to make randomized algorithms with polynomial bound run-times. Both have two major similarities

1. If the input is prime, the output of the algorithm is a 0.
2. If the input is composite, the output is a 0 or 1. The probability that the outcome is wrong is 1/2.

An error bound of 1/2 is not that great but by repeating the algorithm multiple times the bound on the error can be dropped significantly. For example, the probability of being wrong twice in a row would only be 1/4, three times in a row would be 1/8, etc. So, using either of their algorithms allows someone to in polynomial time find out if a very large number is prime or not (Dietzfelbinger 7-8).

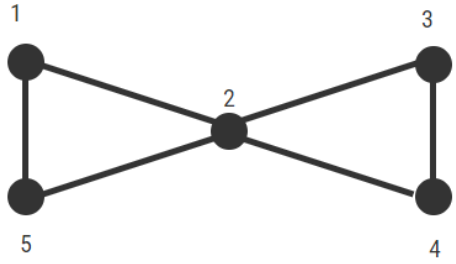
Before the 20th century people didn't generally accept the idea of randomness. Religion was a huge factor in this, but the development of natural sciences and mechanical engineering also reassured people that everything could be explained by cause and effect. Randomness was also connected with chaos, uncertainty and unpredictability, all things that people related with fear. So, rejecting the possibility of randomness helped to ease people's fear about the chaos and uncertainty surrounding it. In the 19th and 20th centuries with the discovery of both statistical and quantum mechanics, randomness became an accepted phenomenon. As you can imagine it is extremely difficult to prove that randomness does exist, and it is very unlikely that we will have an answer anytime soon. But for computer scientists randomness is accepted not only because of physics but also

because randomness allows for great improvements in efficiency. It seems unlikely that something that provides for such gains would be false (Hromkovic 206-209).

Complexity and Efficiency

Complexity theory is important in determining if an algorithm is efficient and solvable. The goal of **complexity theory** is to determine the amount of resources needed to carry out an algorithm which then allows us to classify problems as solvable and unsolvable. The two main uses of a computer's resources by an algorithm are time and space. The **time complexity** of an algorithm is defined as the amount of work, or fundamental instructions such as conditional and iterative operations, that the computer must perform during the execution of the algorithm. It is important to note that when fundamental operations are nested within each other that the time complexity of the algorithm drastically increases. The **space complexity** of an algorithm is defined as the amount of memory space that the algorithm uses which includes all the data that the algorithm will operate on. An algorithm is efficient when as little resources as possible are used to produce the result. So, when the time and space complexities are small the algorithm is more efficient. When we have a complexity for two different algorithms it allows us to estimate each of the algorithm's runtime as well as compare the two algorithms to determine which is the better option. Big O is an analysis tool to describe the time and space complexity of an algorithm, it represents the upper bound of an algorithm's run-time. The lower the magnitude of Big O the more efficient the algorithm is (Schneider 95-96, 100-101).

While determining the time and space complexity of an algorithm is important, complexity theory can also be used to determine if a problem is solvable or not. When an algorithm to solve a problem has a time complexity less than or equal to some polynomial function of input, we say the problem is solvable and lives in the **complexity class P**. P is a set of problems that all satisfy this condition.



A Eulerian Cycle is 2 1 5 2 3 4 2 but there is not a Hamiltonian Cycle in this graph.

Problems that can't be solved by algorithms with time complexity less than or equal to a polynomial function of input can be classified as **hard**. It is important to note that even a slight change to a hard problem's constraints can cause it to become a member of P and no longer be hard. This reduction from being hard to being a member of P can be seen in graph theory with Hamiltonian cycles and Eulerian cycles. A Hamiltonian cycle is a path around a graph that touches every vertex only once, but a Eulerian cycle is a path around a graph that touches every edge only once. Finding a Hamiltonian cycle is classified as a hard problem but by twisting the problem statement and looking for a Eulerian cycle the problem now lives within the complexity class P.

Algorithms for problems in P are easy to implement but the same is not true of hard problems. When dealing with a hard problem, our goal is to find an algorithm, one without unreasonable performance costs, to solve the problem and that is where randomized algorithms become useful (Hromkovic 177, 179-180).

As mentioned earlier from Motwani, randomized algorithms have smaller time complexities than the best-known deterministic algorithms, but why is this? In short it is because some randomized algorithms work to produce a solution that isn't perfect but just good enough. But in addition to randomized algorithms being very efficient, they can also be more reliable than deterministic algorithms. This is because the probability of a hardware error increases the longer an algorithm is running. So, a fast-randomized algorithm can be more reliable than a slow deterministic algorithm because you are reducing the probability of a hardware error. Thus, although you aren't getting the perfect answer you are getting an answer faster and more reliably (Hromkovic 202).

Monte Carlos vs. Las Vegas

Randomized algorithms can be classified into two groups based on how goals are prioritized. The first type is a **Monte Carlo algorithm** which is a randomized algorithm with a goal to produce a result in a small amount of time but doesn't always produce the right solution (although the probability of an incorrect solution can be bounded). One important thing to note is that if a Monte Carlo algorithm is run many times, the probability of an incorrect solution will become arbitrarily small. It is also important to note that to reduce the probability of an incorrect solution, the run time of the algorithm gets longer.

One example that demonstrates the success of a Monte Carlo algorithm is estimating the value of π. π is an irrational number meaning it is nonterminating and nonrepeating. So, if we had attempted to get an exact value of π, the algorithm would literally go on forever. Thus, it is important to get a value that is good enough instead of waiting forever.

In this method a circle with a radius of 0.5 is enclosed by a 1x1 box. Recall,

$$A_c = Area\ of\ Cirle = \pi r^2 \qquad A_c = \pi(.5)^2 = \frac{\pi}{4} \qquad A_s = Area\ of\ Square = S^2$$

$$r = radius\ of\ the\ circle \qquad A_s = (1)^2 = 1 \qquad S = length\ of\ a\ side$$

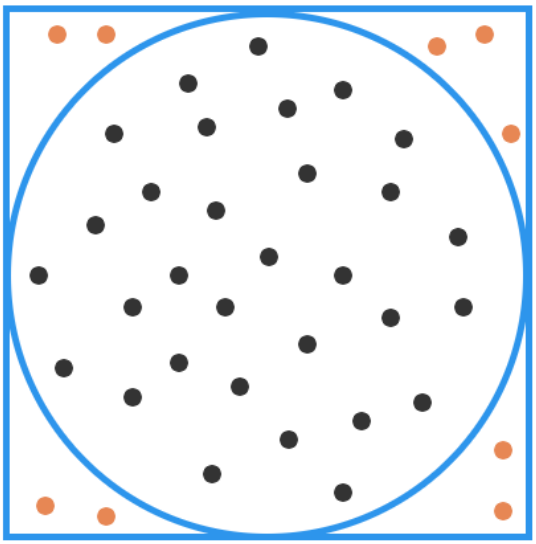
Then using a computer, a large amount of points are randomly distributed in the 1x1 box. The number of points in the circle is compared to the total number of points. This ratio should be very close to the ratio of the area of the circle and the square.

$$P_{in} = The\ number\ of\ points\ in\ the\ circle$$
$$P_{total} = The\ total\ number\ of\ points\ within\ the\ box \qquad \frac{P_{in}}{P_{total}} \approx \frac{A_c}{A_s}$$

So,

$$\frac{\pi}{4} \approx \frac{P_{in}}{P_{total}} \qquad \pi \approx (4)\frac{P_{in}}{P_{total}}$$

As explained above, the computer randomly places points in a designated region and can quickly and easily find a value of π which isn't perfect but is "good enough;" the goal of a Monte Carlo randomization algorithm (Estimating Pi Using the Monte Carlo Method). In practice the value of π is not calculated in this manner but this provides us with a good illustration of a Monte Carlo algorithm.



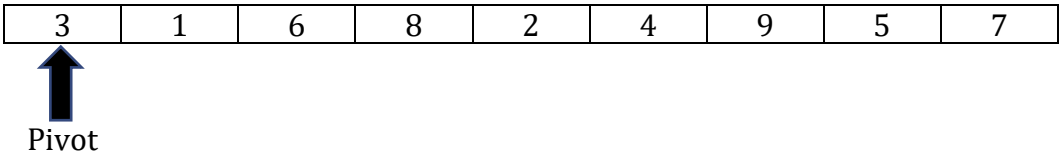
There are 31 total dots in the circle and 40 total dots.

4 * (31/40) = 3.1 ≈ π

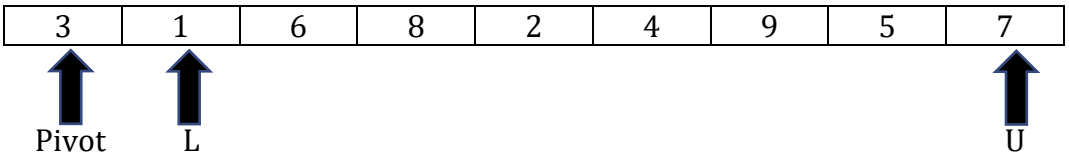
A **Las Vegas algorithm** is another type of randomized algorithm, but the goal is to always produce the correct solution. It is important to note that although a Las Vegas

algorithm will always produce the correct solution, the amount of time it takes to get the correct solution will vary with each run. A classic example of a Las Vegas algorithm is a sorting algorithm called quickSort. **QuickSort** is an algorithm used to sort data in ascending order. We will first cover how a quickSort algorithm operates and then discuss how a randomized quickSort algorithm differs from that.

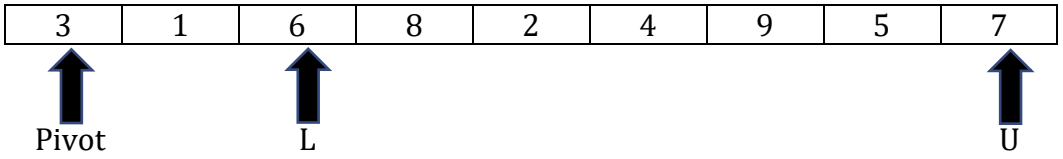
1. The first step in a quickSort algorithm is to select a data point to be the pivot. The easiest way to do this is to select the first value as the pivot.



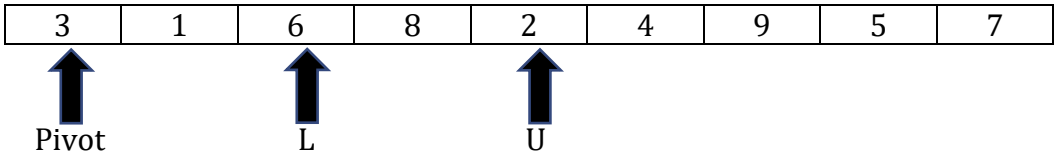
2. Once the pivot has been selected, identifiers, called L for lower and U for Upper, are added in at the first element after the pivot and the last element in the list.



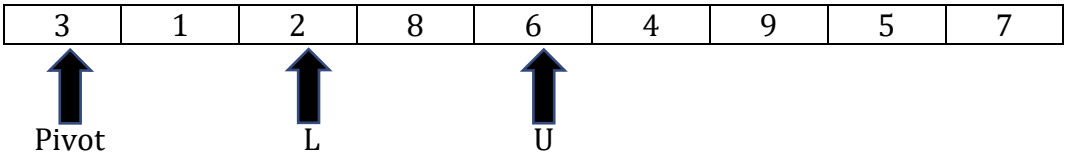
3. The identifier L is moved to the right until it runs into a value that is larger then the pivot.



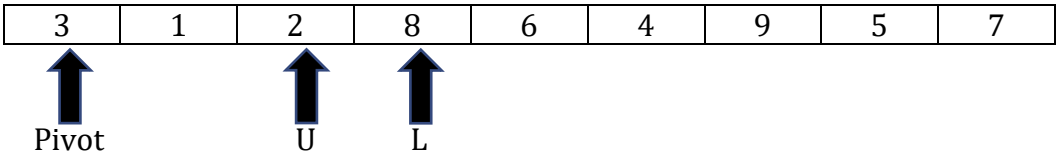
4. The identifier U is moved to the left until it runs into a value that is smaller than the pivot



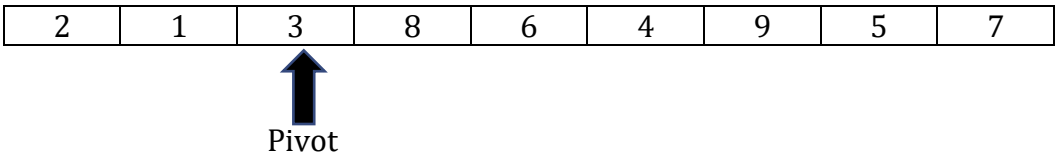
5. If L is the left of U, i.e. L and U have not crossed each other, the values at L and U are swapped. In this example the value 6 is moved from the 3rd spot to the 5th spot and the value 2 is moved from the 5th spot to the 3rd spot.



6. Like what we did above continue moving L to the right until it encounters an element larger than pivot. Then move U to the left until it encounters an element smaller than the pivot.



7. At this point U is on the left of L, i.e. U and L have crosses, so the elements at U and pivot are swapped.



8. Notice the pivot is now separating two lists of values. The values on the left of the pivot are all values that are smaller than the pivot and the values on the right of the pivot are all values larger than the pivot.

9. The steps above are repeated on each subsection, values smaller than pivot and values larger than pivot, until a sorted list is produced.

So how does a quickSort algorithm become a randomized quickSort algorithm? Recall that in the previous explanation the pivot was set to be the first element in the list. In a randomized quickSort algorithm, the pivot is selected by randomly generating a number and then using the value at that position in the list as the pivot.

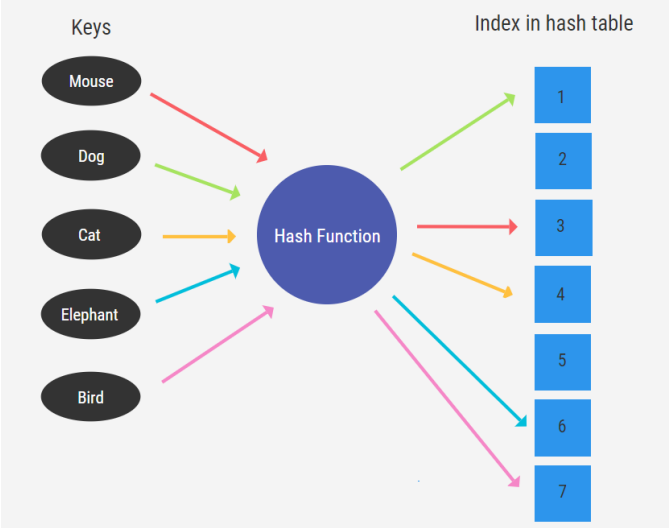
But what is the benefit of introducing randomization to the quickSort algorithm? When a pivot is randomly selected it is very unlikely that one of the extremes will be

selected, most likely a value around the mean will be selected. And when this occurs, every execution of the quickSort algorithm recursively splits the list into smaller near-equivalent portions making the remaining work balanced rather than uneven. When the pivot is set to the first element in the list, there is chance that the first element is an extreme. In the case that the list is already sorted or sorted backwards, the pivot will always be an extreme and the list won't be split evenly. This causes the worst-case runtime of a quickSort algorithm which is n^2. Since randomly selecting the pivot drastically reduces the chances that an extreme will be selected as the pivot every time the run time of a randomized quickSort algorithm is likely be closer to the best-case of $n\ log(n)$ instead of the worst case n^2. Refer to the appendix for a deeper analysis of quickSort (Landman).

A Las Vegas algorithm is said to be efficient if its *expected* run time is bounded by a polynomial function of the input size. And a Monte Carlo algorithm is said to be efficient if its *worst-case* run time is bounded by a polynomial function of the input size (Motwani 10).

One may be asking themselves which type of algorithm is better, but it depends on the problem that the algorithm is trying to solve. Imagine you are on the Apollo 11 mission to the moon and you need to recalculate your trajectory. Is it incredibly important to get the correct answer and as such using anything but a Las Vegas algorithm could lead to catastrophe. But suppose an answer that is good enough will suffice, then a Monte Carlo algorithm should be used.

Academic Applications



Hash function that doesn't have any collisions

Hashing

Computer scientists often deal with large databases. Suppose there is a database of all the phone numbers with a 614, Columbus, Ohio, area code. If someone needs to search through the database or add/ delete a phone number, we want an efficient way to do this. Hashing, a type of data structure, provides an efficient way to do all these operations by employing randomness when organizing the data.

Data is organized into a **hash table** using a hash function. A **hash functions** takes a key and operates on it by pointing to the location of that element in the hash table. For a hash function to have good performance we want a few things.

1. We want the keys to be spread out without making the hash table too much bigger than the number of elements
2. We want the hash function to be simple

But when keeping the size of the hash table approximately the same size as the number of elements, there is a higher probability that two keys will be mapped to the same location in the hash table. Collisions are not ideal and one way to avoid them is to implement randomization. A set of hash functions is said to be **2-universal** if, when selecting a hash function randomly, the probability that two nonequal elements will collide is less than or

equal to 1/M where M is the size of the hash table (Wagner). So, if a hash function is randomly selected, independent of the keys that are going to be stored, from the 2-Universal hash function family a similar phenomenon to that of the randomized quickSort is observed. By selecting our hash function randomly, it guarantees that a certain input won't always produce the worst-case behavior. The user is guaranteed good average-case performance for any input. The appendix proves a detailed proof of a 2-Universal set (Cormen 232).

Random sampling

If you have ever taken a course in statistics, you are surely familiar with random sampling. But if you have never heard of **random sampling**, it is the process of taking a group of subjects and randomly selecting a certain number of them. Random sampling is a kind of randomized algorithm because the algorithm to choose subjects uses randomness. Suppose that we are a group of researchers and we want to find out how many people in the City of Los Angeles have a job where they earn over $100,000 a year. It would be very time consuming and expensive to send a survey to every resident in the city. So instead we randomly select a group of individuals to survey. By randomly selecting individuals we get a smaller group of people that represent the city as a whole. So whatever conclusion we can draw from the subjects that were surveyed, can be applied to the whole City of Los Angeles. The algorithm to choose individuals uses randomness and as such it is a randomized algorithm (Landman).

Pseudo Random Number Generator

Throughout this text when referring to random number it was implied that these values were truly random: the next random value to be produced couldn't be predicted and a series of numbers would only repeat if it were a complete accident. But it quite inefficient to generate truly random numbers because it involves taking some physical phenomenon and extract randomness from it. The simplest, but unrealistic, example would be to attach a dice to a computer. Then whenever the user requests a random number the computer would roll its dice and display the number that it rolled. But in reality, to generate truly random values a computer has to examine some kind of physical phenomena such as variations in mouse motion, radioactive decay, background noise or any other physical phenomenon in the environment where the computer is at. Whatever physical phenomenon is used the random number is generated by detecting small, unpredictable changes in the data collected from the environment.

It is important to make sure that the physical phenomenon that is selected to produce random numbers is itself random. For example, suppose that we are going to generate random numbers by detecting the background noise in the office. The background noise in an office is generally random but what if the fan on your computer running. A fan is a consistent, rotating device and as such will introduce a non-random sound into your environment. So, if we use this physical phenomenon, we can't expect to get random values.

To bypass the inefficiency that is the result of generating truly random numbers, a computer will generate numbers that appear to random. They do this by using a pseudorandom number generator which uses a precalculated table or an algorithm to

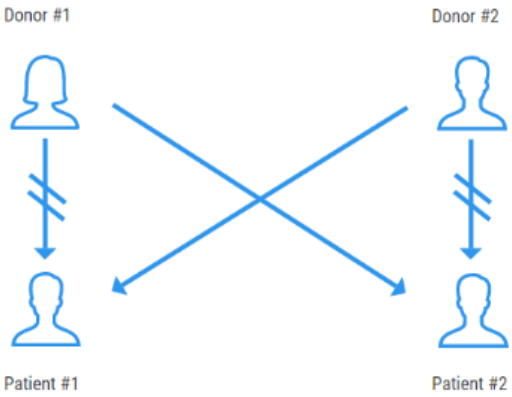
produce values. To produce a precalculated table of random numbers, one might roll a dice a set amount of times and record the values that are produced in the computer. Then when the user asks for a random value the next value in line is outputted. To the user the value seems random but, in all reality, it was a predetermined value. The most common method to compute pseudorandom number is to use a linear congruential generator. This method uses an algorithm to produce the next random number from the seed (the starting value) or the last calculated random number.

$$r_{n+1} = (a * r_n + c) \, mod \, m$$

$$r_0 = seed \qquad r_1, r_2, r_3 \ldots = the \; random \; numbers \qquad a, c \; and \; m \; are \; constants$$

Notice that for both methods although they are much more efficient than true random number generators, they can reproduce a sequence of numbers if you know the starting value and they will eventually repeat themselves. It is sometime helpful for the user to be able to reproduce a sequence of values, but it is not beneficial for the sequence to loop. Often though, the amount of values that it will take for the pseudorandom number generator to repeat itself is so long that it can be ignored for most practical purposes. Pseudorandom number generators are often used with simulation and modeling applications but are not suitable for applications in which you need values that can't be predicted like in cryptology. Pseudorandom number generators are often used with randomized algorithms to preserve their efficiency (Haahr).

Applications



Thus far there have been numerous examples of how randomized algorithms can be used. Take the example where the value of π was estimated using a Monte Carlo algorithm. More realistic applications include randomized quickSort to sort a large amount of data or random sampling to draw conclusions about a population. Randomized algorithms can be used in many ways, even to help to match people to facilitate kidney transplants. Often when someone finds out that they will need a kidney transplant, friends and family will offer to donate. But many times, it turns out that these people don't end up being matches for the patient. In 1986 Rapaport proposed matching incompatible patient-donor pairs with other incompatible patient-donor pairs so that both patients received a compatible kidney. In this scenario all pairs are happy in the end, the patient receives the kidney they needed and the donor, although they didn't directly donate to their incompatible pair, enabled the donation by giving their kidney to another pair. The Kidney Exchange Problem works to match incompatible patient-donor pairs with other pairs to facilitate the most exchanges possible. There are some logistical issues that arise when implementing this in real life. Ideally the amount of exchanges would be large to include many incompatible patient-donor pairs. But to execute multiple exchanges would require that all the operations be happening simultaneously. For one it is important to get the organ out and into the next body as quick as possible. But in addition to this, if a donor's

patient pair receives their kidney before they donate, they may be tempted to back out of their donation. The donor has already received what they wanted, their patient pair to receive a kidney, so why, other than moral integrity, would they feel inclined to go through with their donation? But this presents the need for a huge surgical staff as well as space to operate on and keep people for recovery after surgery. In general, the number of exchanges in the Kidney Exchange Problem is bounded between 2 and 5. When the number of exchanges is greater than or equal to 3, the Kidney Exchange Problem is considered NP-Hard, the problem doesn't have an algorithm with a polynomial bounded run time. Using a randomized algorithm with the Kidney Exchange Problem where 3 or more exchanges are involved has a better bound on runtime than any previous algorithm. While logistically it would be difficult to perform a lot of exchanges, having a randomized algorithm that can identify matches for many people increases the odds that a patient will receive the organ they need (Lin).

Conclusion

Randomness, something that used to be viewed as a negative because of its association with chaos, now provides for great improvements in algorithms' time and space complexity. While the process to get truly random values can be difficult, algorithms that use numbers that appear to be random still experience great leaps in efficiency. With better, more efficient algorithms society spends less money on tasks such as conducting a survey on the people of a city. Employers don't have to pay an employee to search through a database or match patients to donors; that money can be used in other more important areas. Early scientists believed that the whole world was deterministic, free of chaos, but

with new discoveries, chaos and randomness are not only accepted phenomena, but used to make algorithms more efficient.

Appendix

QuickSort Analysis

The expected performance of quickSort is

$$T_a(n) = n + \frac{1}{n}\sum_{p=1}^{n} T_a(p-1) + T_a(n-p)$$

$$n = the\ number\ of\ elements\ in\ the\ list\ to\ sort$$

$$p = the\ position\ of\ the\ pivot\ in\ the\ list$$

$$Base\ Cases: Clearly\ if\ n = 0\ or\ n = 1 \quad T_a(n) = 0$$

Explanation of the equation

- First term: The complexity of the partition is O(n). This means that it takes linear time to place the pivot such that all that all elements smaller than it are to the left and all elements larger than it are to the right.
- You multiple the recursive summation by $\frac{1}{n}$ because you are finding the average performance and there are n possibilities positions for the pivot.
- Inside the summation you are making recursive calls and applying this same equation to the lower list, (p – 1), and the upper list (n – p).
    - Note that for each recursive call you will look at all possible pivot positions

Solving the equation

$$T_a(n) = n + \frac{1}{n}\sum_{p=1}^{n} T_a(p-1) + T_a(n-p) \tag{1}$$

Notice that

$$\sum_{p=1}^{n} T_a(p-1) = \sum_{p=1}^{n} T_a(n-p) \tag{2}$$

Each summation goes from the first position to the length of the list. The first summation finds the number of comparisons for a list of length 0, 1, 2, … up to n – 1. The second summation finds the number of comparisons for a list of length n – 1, n – 2, … 1, 0. Thus they are equal, and we can rewrite our equation

$$T_a(n) = n + \frac{2}{n}\sum_{p=1}^{n} T_a(p-1)$$

(3)

Now multiplying by n

$$n * T_a(n) = n^2 + 2\sum_{p=1}^{n} T_a(p-1)$$

(4)

and substituting n – 1 for n we have

$$(n-1) * T_a(n-1) = (n-1)^2 + 2\sum_{p=1}^{n-1} T_a(p-1)$$

(5)

If we subtract equation 5 from equation 4, we obtain

$$nT_a(n) - (n-1)T_a(n-1) = n^2 - (n-1)^2 + 2T_a(n-1)$$

(6)

Since the summation in equation 4 runs to n, we have one more term than the summation in equation 5 and thus that is why the last term is $2T_a(n-1)$. Simplifying the equation, we have

$$nT_a(n) = (n+1)\,T_a(n-1) + 2n - 1$$

(7)

Next divide the equation by n (n + 1) and we obtain

$$\frac{nT_a(n)}{n\,(n+1)} = \frac{(n+1)\,T_a(n-1)}{n\,(n+1)} + \frac{2n}{n\,(n+1)} + \frac{-1}{n\,(n+1)}$$

(8)

$$\frac{T_a(n)}{(n+1)} = \frac{T_a(n-1)}{n} + \frac{2}{(n+1)} + \frac{-1}{n\,(n+1)}$$

(9)

Dropping the last term creates the inequality

$$\frac{T_a(n)}{(n+1)} \leq \frac{T_a(n-1)}{n} + \frac{2}{(n+1)} \tag{10}$$

Defining the equation

$$F(n) = \frac{T_a(n)}{n+1} \tag{11}$$

and substituting it in equation 10 we obtain

$$F(n) \leq F(n-1) + \frac{2}{n} \tag{12}$$

Expanding this equation, we get

$$F(n) \leq F(n-2) + \frac{2}{n-1} + \frac{2}{n} \tag{13}$$

This is accomplished by taking $F(n)$ and substituting n - 1 for n.

$$F(n-1) \leq F(n-2) + \frac{2}{n-1}$$

Then plugging this equation of $F(n-1)$ back into $F(n)$ we obtain equation 13. Continuing in this fashion we obtain

$$F(n) \leq 2\left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} + \frac{1}{n}\right) \tag{14}$$

This is a partial harmonic series which can be written as

$$\sum_{k=1}^{n} \frac{1}{k} \tag{15}$$

Which evaluates to

$$\int_1^n \frac{1}{k} = \log n$$

(16)

In calculus, you probably learned that the above integral evaluates to $\ln n$. But since the base of $\ln n$ and $\log n$ only differs by a constant and constants have no weight in Big O, it is legal to say that the integral evaluates to $\log n$. Thus

$$F(n)\ \epsilon\ \Theta\ (\log n)$$

(17)

Because of our definition of F(n) we have

$$F(n) = \frac{T_a(n)}{n+1}$$

$$T_a(n) = F(n) * (n+1)$$

$$T_a(n)\ \epsilon\ \Theta\ \big(nlog(n)\big)$$

(18)

Recall that constants don't matter in Big O, which is why we drop the constant 1 (Stucki).

Euclid's Extended Algorithm

I will quickly review Euclid's Algorithm to find the greatest common divisor of two numbers.

Given two integers a and b you start by dividing a by b and label the remainder as r1. Now you will divide the previous value of b by r1 and label this quotients remainder as r2. Continue in the fashion until the remainder is zero. The GCD is the last non-zero remainder.

Ex. GCD(204, 15) = 3

a = 204
b = 15

```
   13
15 ) 204
   - 195
  r1 =  9
```

$$204 = 15 * 13 + 9$$

a = 15
b = 9

```
    1
9 )  15
   - 9
r2 =  6
```

$$15 = 9 * 1 + 6$$

a = 9
b = 6

```
    1
6 )  9
   - 6
 r3 = 3
```

$$9 = 6 * 1 + 3$$

a = 6
b = 3

```
    2
3 )  6
   - 6
r4 =  0
```

$$6 = 3 * 2 + 0$$

Here is a recursive implement of Euclid's Algorithm in Java

```
import java.util.Scanner;

public class GCD {
	public static void main(String[] args) {
		System.out.println("GCD Calculator\t\tGCD(x, y)");
		Scanner in = new Scanner(System.in);
		//Allows the user to enter
		System.out.print("Enter the a value: ");
		int a = in.nextInt();
		System.out.print("Enter the b value: ");
		int b = in.nextInt();
		int GCD = greatestCommonDivisor(a, b);

		System.out.print("The GCD of " + a + " and " + b + " is " + GCD + ".");
	}

	//Method to recursively find the GCD
	public static int greatestCommonDivisor(int a, int b) {
		//If the remainder is zero, return the last non-zero remainder, b.
		if (a % b == 0) {
			return b;
		}
		/*
		 * Recursively calls itself passing in the divisor (b) and the remainder
		 * of the division (a/b)
		 */
		else {
			return greatestCommonDivisor (b, a % b);
		}
	}
}
```

So now that we have reviewed what the Euclidean Algorithm is, we can look at the Extended Euclidean Algorithm. This algorithm computes x and y values such that

$$GCD(a, b) = ax + by$$

We will prove that integers, x and y, exist later but first let's see how to find the value of x and y with the example above.

Step 1: Find the GCD (a, b).

GCD(204, 15) = 3

a = 204
b = 15

```
   13
15 ) 204
   - 195
     9
```

$$204 = 13 * 15 + 9 \quad (1)$$

a = 15
b = 9

```
    1
9 )  15
   - 9
     6
```

$$15 = 1 * 9 + 6 \quad (2)$$

a = 9
b = 6

```
    1
6 )  9
   - 6
     3
```

$$9 = 1 * 6 + 3 \quad (3)$$

a = 6
b = 3

```
    2
3 )  6
   - 6
     0
```

$$6 = 2 * 3 + 0 \quad (4)$$

Step 2: Take the last equation with a non-zero remainder (ie. the equation where the GCD is the remainder) and rearrange the terms so that the GCD is on the left by itself. In our case this is equation 3.

$$9 = 1 * 6 + 3 \qquad Original\ Equation\ 3$$

$$3 = 9 - 1 * 6 \qquad Equation\ 3\ Rearranged$$

Step 3: Notice the value of b in equation 3, 6, is the remainder in the previous equation, equation 2. So, rewrite equation 2 in terms of its remainder (ie. the value of b in equation 3).

$$15 = 1 * 9 + 6 \qquad Original\ Equation\ 2$$

$$6 = 15 - 1 * 9 \qquad Equation\ 2\ Rearranged$$

Step 4: Substitute rearranged equation 2 into the rearranged equation 3.

$$3 = 9 - 1 * (15 - 1 * 9)$$

Step 5: Distribute without multiplying values out and collect like terms, the 9's.

$$3 = 2 * 9 - 1 * 15$$

Step 6: Notice the value of b in equation 2, 9 in our case, is the remainder in the previous equation, equation 1 in our case. So, rewrite equation 1 in terms of its remainder (ie. the value of b in equation 3).

$$204 = 13 * 15 + 9 \qquad Original\ Equation\ 1$$

$$9 = 204 - 13 * 15 \qquad Equation\ 1\ Rearranged$$

Step 7: Substitute rearranged equation 1 into our equation from step 5.

$$3 = 2 * (204 - 13 * 15) - 1 * 15$$

Step 8: Distribute without multiplying values out and collect like terms, the 15's.

$$3 = 2 * 204 - 27 * 15$$

Step 9: Notice that this equation contains the GCD on the left-hand side and the right-hand side is a linear combination of our original values of a and b. We are done, we have found the value of x and y! (Extended Euclidean Algorithm).

$$GCD(a, b) = ax + by$$

$$x = 2\ and\ y = -27$$

You may have noticed a pattern. Starting with the equation that has the GCD as the remainder, you continue up the chain of equations, rearranging them in terms of their remainder and substituting them in until you have constants multiplied by the original values of a and b.

Now that we have covered how to find the values of x and y, we will prove that these values exist. This proof is formally named Bezout's Identity.

Theorem: For nonzero integers a and b, let d be the greatest common divisor, d = gcd(a, b). Then, there exist integers x and y such that ax + by = d.

Proof: First let's consider the set $S = \{ s > 0 \mid s = ax + by\ for\ some\ x, y \in Z \}$*, the set of positive linear combinations. Clearly |a| + |b| > 0 as*

$$a > 0\ and\ b > 0 \quad |a| + |b| = 1 * a + 1 * b$$

$$a > 0\ and\ b < 0 \quad |a| + |b| = 1 * a + (-1) * b$$

$$a < 0\ and\ b > 0 \quad |a| + |b| = (-1) * a + 1 * b$$

$$a < 0\ and\ b < 0 \quad |a| + |b| = (-1) * a + (-1) * b$$

So we know that it is a member of S and so S is non-empty. By the well ordering principle which states that every non-empty set of positive integers contains a smallest element, S contains a smallest element. Suppose that the smallest element is $s_0 = ax_0 + by_0$*.*

By the division algorithm we know there exists q, r ϵ Z such that $a = s_0 q + r$ *with* $0 \leq r < s_0$*. Note that since* $r < s_0, r \notin S$ *since* s_0 *is the smallest element in S. Notice that rearranging our equation results in* $r = a - s_0 q$*. Then substituting in for* s_0*, distributing and rearranging terms we obtain*

$$r = a - (ax_0 + by_0)\, q = a\,(1 - x_0 q) + b\,(-y_0 q).$$

From this, it appears that r is in S since it is a positive linear combination of a and b. But we have already established that it is not in S so for this to be the case, r must be zero. Thus $a = s_0 q$ *and we can clearly see that* $s_0 | a$*. Similarly,* $s_0 | b$*.*

By definition of the greatest common divisor, we know that $d \mid a$ *and* $d \mid b$*, and so we have* $d \mid ax_0 + by_0 = s_0$*,* $ie. d | s_0$*. Note that* $d \leq s_0$*, by the definition of divisibility, but since* $s_0 | a$ *and* $s_0 | b$ *it is a common divisor. So, since d is the greatest common divisor and* s_0 *is a common divisor that is greater than or equal to d, d must equal* s_0*. So substitute d in for* s_0 *and we obtain* $d = ax_0 + by_0$*. So we have an x and y such that* $d = ax + by$ *(Chen; AITKEN).*

So, where has all this been going. There is something called the modular multiplicative inverse which is used in cryptography. The modular multiplicative inverse of an integer a is an integer x such that

$$ax \equiv 1 \,(mod\ b)$$

and the Extended Euclidean Algorithm provides a very fast algorithm to find this integer x. It should be noted that for a to have a modular multiplicative inverse the GCD(a, b) must be 1. We will work through a simple example to demonstrate how to find a values modular multiplicative inverse.

Step 1: Confirm that the GCD(a,b) = 1 and find integers such that GCD(a, b) = ax + by (the Extended Euclidean Algorithm)

GCD(13, 5) = 1

a = 13
b = 5

```
    2
5 ) 13
   -10
    3
```

$13 = 2 * 5 + 3$ *(1)*

a = 5
b = 3

```
    1
3 ) 5
  - 3
    2
```

$5 = 1 * 3 + 2$ *(2)*

a = 3
b = 2

```
    1
2 ) 3
  - 2
    1
```

$3 = 1 * 2 + 1$ *(3)*

a = 2
b = 1

```
    2
1 ) 2
  - 2
    0
```

$2 = 2 * 1 + 0$ *(4)*

So, the GCD is 1. Now let perform the Extended Euclidean Algorithm. Refer to earlier in this section for clarification on what is going on in each step.

$1 = 3 - 1 * 2$ *Equation 3 rearranged*

$2 = 5 - 1 * 3$ *Equation 2 rearranged*

$1 = 3 - 1 * (5 - 1 * 3)$ *Substitute equation 2 into equation 3*

$1 = 2 * 3 - 1 * 5$ *Simplify*

$3 = 13 - 2 * 5$ *Rearrange equation 1*

$1 = 2 * (13 - 2 * 5) - 1 * 5$ *Substitute it into the working equation*

$1 = 2 * 13 - 5 * 5$ *Simplify*

This is a linear combination of a and b so we are done!

$$x= 2 \text{ and } y = -5$$

Step 2: Now that we have found the solution to the extended Euclidean algorithm, we can move on to finding the modular multiplicative inverse. First apply mod b on both sides of the equation we found with the extended Euclidean algorithm, note that I have just rearranged terms to match the format

$$ax + by = GCD(a, b)$$

$$13 * 2 + 5 * -5 = 1$$

$$(13 * 2 + 5 * -5) \ (mod\ 5) \equiv 1 \ (mod\ 5)$$

Note that the term 5 * -5 (mod 5) will always be zero, any multiple of 5 will always be reduced to zero, so we can drop this term. And we are left with

$$13 * 2 \equiv 1 \ (mod\ 5)$$

$$The\ modular\ multiplicative\ inverse\ is\ 2.$$

Thus when the GCD(a, b) = 1, calculating the extended Euclidean algorithm and then applying mod b to both sides reveals the modular multiplicative inverse (Ankur).

Constructing a 2-Universal Set of Hash Functions

First recall the definition of 2-Universal.

***2-Universal**: A collection H of hash functions $h: \{0, 1, \dots, M\} \to \{0, 1, \dots, m-1\}$ is said to be 2 – Universal if for every $x, y \in M$ such that $x \neq y$ has*

$$Pr[h(x) = h(y)] \leq \frac{1}{m}.$$

Now we will construct a set of hash functions and prove that it is 2-Universal.

Theorem: Let $h_{ab}(x) = (ax + b \bmod p) \bmod m$, where $a \in Z_p \setminus \{0\}$, $b \in Z_p$ and p is a prime such that $p \geq m$. Then $\{h_{ab}\}$ is 2 – Universal.

Proof: To prove that $\{h_{ab}\}$ is 2-Universal we first have to prove that for each distinct $x, y \in \{0, \dots, m-1\}$, the number of pairs of a and b such that $h_{a,b}(x) = h_{a,b}(y)$ is at most $\frac{p(p-1)}{m}$

First let's define $g_{a,b}(x) = ax + b \pmod p$.

When a collision occurs or $h_{a,b}(x) = h_{a,b}(y)$ this means that there are values $s, t \in \{0, \dots, p-1\}$ such that $s = t \pmod m$ where $g_{a,b}(x) = s$ and $g_{a,b}(y) = t$. Let's begin by counting the number of these pairs.

It is important to note before we begin counting that $g_{a,b}(x) \neq g_{a,b}(y)$. If $g_{a,b}(x) = g_{a,b}(y)$ then

$$ax + b = ay + b \pmod p$$

$$a(x - y) = 0 \pmod p$$

But $a \neq 0 \pmod p$, from our definition of a being a member of the set {1, ..., p-1}. So, this would mean that $(x - y) = 0 \pmod p$ or $x = y \pmod p$. But this contradicts our assumption that x and y are distinct. So it must be true that $g_{a,b}(x) \neq g_{a,b}(y)$. So we should never count a pair (s, t) where s = t.

We will say that a pair (s, t) with $s, t \in \{0, \dots, p-1\}$ is a valid pair if $s = t \pmod n$ and $s \neq t$. Note that there are p choices for s. So once we fix s, there are $\lceil p/m \rceil$ values in {0,..., p-1} with $s = t \pmod n$ for t. Remember that we need to exclude the pair where s = t so we actually have $\lceil p/m \rceil - 1$ choices for t. Finally, the total number of valid pairs is $p(\lceil p/m \rceil - 1)$.

Notice that $\lceil p/m \rceil \leq (p + m - 1)/m$. So, we can rewrite our total number of valid pairs as

$$p\left(\left\lceil \frac{p}{m} \right\rceil - 1\right) \leq p\left(\frac{p + m - 1}{m} - \frac{m}{m}\right) \leq p\left(\frac{p-1}{m}\right)$$

So, at most we have $p\left(\frac{p-1}{m}\right)$ valid pairs (s, t).

Now that we have valid pairs (s, t), we need to count the number of pairs (a, b) such that $g_{a,b}(x) = s \pmod p$ *and* $g_{a,b}(y) = t \pmod p$*. Essentially, we are finding the number of solutions to the system of equations for each valid pair (s, t). Remember that* $a, b \in \{0, \dots, p-1\}$

$$\begin{Bmatrix} ax + b = s \pmod p \\ ay + b = t \pmod p \end{Bmatrix}$$

Notice that we have two equations and two unknowns. Since p is prime, we know that the solution is unique. So, for each pair (s, t) we have a pair (a, b) with $g_{a,b}(x) = s$ *and* $g_{a,b}(y) = t$*. Thus, we have at most* $p\left(\frac{p-1}{m}\right)$ *pairs for (a, b) such that* $h_{a,b}(x) = h_{a,b}(y)$*. Now recall that we have p(p-1) functions in the family since we have p-1 choices for a and p choice for b. Thus, the probability that*

$$h_{a,b}(x) = h_{a,b}(y) \; is \; \leq \frac{1}{m}$$

So, from the definition of 2-Universal, we can see that our family of functions is indeed 2-Universal (Wagner).

Bibliography

AITKEN. “BEZOUT’S IDENTITY, EUCLIDEAN ALGORITHM.” 2009.

Ankur. “Modular Multiplicative Inverse.” *GeeksforGeeks*, 23 Sept. 2019, www.geeksforgeeks.org/multiplicative-inverse-under-modulo-m/.

Aspnes, James. *Notes on Randomized Algorithms CPSC 469/569: Fall 2016*. 19 Dec. 2016, www.cs.yale.edu/homes/aspnes/classes/469/notes-2016.pdf.

Bradley, Larry. “Laplace's Demon.” *Chaos & Fractals*, http://www.stsci.edu/~lbradley/seminar/laplace.html.

Burger. *The Euclidean Algorithm and Diophantine Equations*. California State University, Fresno, zimmer.csufresno.edu/~lburger/Math149_diophantine%20I.pdf.

Chen, Jason. “Bezout's Identity on Linear Combinations.” *Expii*, www.expii.com/t/bezouts-identity-on-linear-combinations-3397.

Cormen, Thomas H.., et al. *Introduction to Algorithms*. 2nd ed., The MIT Press, 2001.

Dietzfelbinger, Martin. *Primality Testing in Polynomial Time: From Randomized Algorithms to 'Primes Is in P'*. Springer, 2004.

“Estimating Pi Using the Monte Carlo Method.” *Academo*, 2016.

“Extended Euclidean Algorithm.” *Brilliant Math & Science Wiki*, brilliant.org/wiki/extended-euclidean-algorithm/.

Haahr, Mads. “Introduction to Randomness and Random Numbers.” *RANDOM.ORG*, www.random.org/randomness/.

Hromkovic, Juraj. *Algorithmic Adventures: From Knowledge to Magic*. Springer, 2009.

Landman, Nathan, and Christopher Williams. “Randomized Algorithms.” *Brilliant Math & Science Wiki*, brilliant.org/wiki/randomized-algorithms-overview/.

“Laplace's Demon.” *Wikipedia*, Wikimedia Foundation, 13 Oct. 2019, https://en.wikipedia.org/wiki/Laplace's_demon.

Lin, Mugang, et al. *Randomized Parameterized Algorithms for the Kidney Exchange Problem*. 2019, pp. 1–13, *Randomized Parameterized Algorithms for the Kidney Exchange*

Problem, eds.b.ebscohost.com/eds/pdfviewer/pdfviewer?vid=6&sid=12cbd1f6-c6b1-489f-a063-ab0278022704%40pdc-v-sessmgr04.

Motwani, Rajeev, and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

Schneider, G. Michael, and Judith L. Gersting. *Invitation to Computer Science*. 8th ed., Cengage Learning, 2019.

Stucki, David. "QuickSort Analysis." Lecture Notes.

Wagner, David. "UC Berkeley CS170: Efficient Algorithms and Intractable Problems." 25 Feb. 2003. https://people.eecs.berkeley.edu/~daw/teaching/cs170-s03/Notes/lecture9.pdf.