Summer Homework for Mr. Carr AP Classes

There are seven questions scattered throughout the assignment.

Artificial Intelligence

1. Write a minimum one-page paper (typed, double-spaced) on one way in which artificial intelligence might be incorporated by a church and argue whether it would it be a good thing or bad thing. These are examples. Will AI write the minister's message for Sunday? Will it find people who are likely to accept Christ as their savior? Use one of these questions or make up your own and argue pros and cons.

Encryption

Go to [The Black Chamber - Caesar Cipher (simonsingh.net)]

2. Use an offset of 3 and the Caesar encryption algorithm and encrypt the

   “THIS IS AN EASY ENCRYPTION ALGORITHM”

   The result will be:

   _____

3. Use the Digraph encryption algorithm to encrypt the phrase”
   “THIS IS A HARDER ALGORITHM”

   The result will be:

   _____

4. Explain a Digraph encryption algorithm.

Decidability and Efficiency

So Many Problems!

Why is computer science riddled with so many problems? Because computer scientists are in the business of solving problems. Let's define *problem* and some special types of problems in the context of computer science.

- A **problem** is a general description of a task that can (or cannot) be solved algorithmically. So a problem is not that something is wrong or a bad thing is happening, a problem is just a task to be solved.
- An **instance** of a problem is a specific task that needs to be solved with specific input. For example, sorting is a problem; sorting a specific list such as {2,3,1,7} is an instance of the problem.
- A **decision** problem is a problem with a yes/no answer (e.g., is there a path from A to B?).
- An **optimization** problem is a problem with the goal of finding the "best" solution among many (e.g., what is the shortest path from A to B?).
- A **decidable problem** is a decision problem for which an algorithm can be written to produce a correct output for all inputs (e.g., "Is the number even?").
- An **undecidable problem** is one in which no algorithm can be constructed that always leads to a correct yes-or-no answer.

Can Computers Solve Every Problem?

It probably doesn't surprise you to hear the answer is *no*. However, it may surprise you to learn that this is *mathematically proven* to be impossible. In fact, Alan Turing, one of the most influential figures in computer science, theorized what would later become the digital computer as part of his proof.

Turing proved that at least one task is computationally *undecidable*-**The Halting Problem**. He proved that you cannot write a program that will tell you which computer programs will halt (i.e., exit) and which ones will continue forever (i.e., infinitely loop). An undecidable problem may have some instances that have an algorithmic solution, but there is no algorithmic solution that could solve all instances of the problem.

Many Ways to Solve a Problem

However, most computer scientists spend time constructing programs for problems that computers *can* solve. A programmer's knowledge and skill affects how a program is developed and how it is used to solve a problem. Even so, there are often many different approaches to solving the same problem. Multiple solutions may be equally valid with no single one being exclusively better than the other. In fact, for most of the problems that you'll encounter in your life, there is **almost never just one, single way that the problem can be solved**. Instead, there are almost always multiple ways of solving a given problem-some of which might be better choices than others.

Your challenge as a computational thinker is to 1) design a solution that works and 2) recognize how your algorithm can be made more efficient.

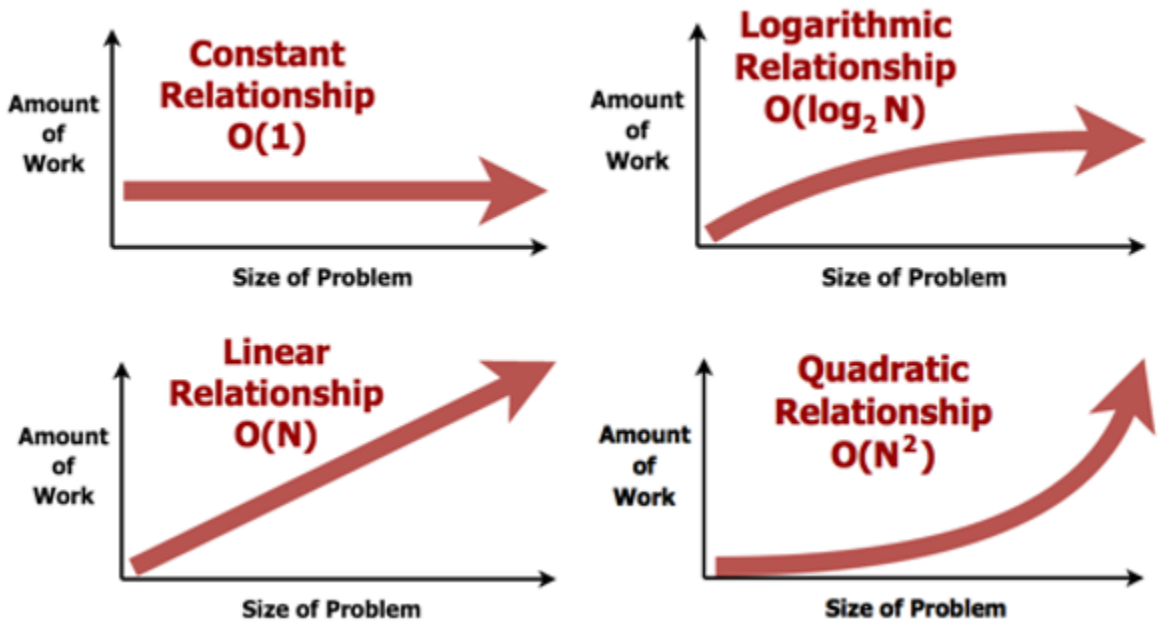
Efficiency and Scalability

So, how do we know that one algorithmic solution is any better than another? In order to answer that question, we first need a way of measuring the relative performance or efficiency that an algorithm offers. An algorithm's efficiency is determined through formal or mathematical reasoning. An algorithm's efficiency can be informally measured by determining the number of times a statement or group of statements executes.

Another factor to consider is scalability. That is, how well does the algorithm perform at larger and larger scales? If the algorithm is efficient for a small data set, is it also efficient when applied to a much larger data set? For example, if we *double* the number of items in a list, how much harder does that make the task of finding a particular item in that list? What if we *triple* the size of the list? *Quadruple* it? What if we increase the size of the overall problem by a factor of *N*? How much does that increase the amount of work required by the algorithm?

Scalability is the capacity for a system to change in size and scale to meet new demands.

Computer scientists use something called "Big-O Notation" (a mathematical concept that is outside the scope of this course) to describe how well different algorithms scale. This allows them to classify different solutions into different categories of relative performance.



Efficiency is an estimation of the amount of computational resources used by an algorithm. Efficiency is typically expressed as a function of the size of the input.

- **Constant:** No matter how much a problem grows, the amount of work stays more or less the same.

- **Logarithmic:** Every doubling of the size of a problem only requires one extra unit of work.
- **Linear:** As the size of a problem grows, the amount of work required grows at approximately the same rate.
- **Quadratic:** As the size of a problem grows, the amount of extra work required increases much more quickly.

Algorithms with a polynomial efficiency or slower (constant, linear, square, cube, etc.) are said to run in a *reasonable* amount of time. Algorithms with exponential or factorial efficiencies are examples of algorithms that run in an *unreasonable* amount of time.

Performance Comparisons

Fun facts about algorithms:

- Algorithms can be written in different ways and still accomplish the same tasks.
- Algorithms that appear similar can yield different side effects or results.
- Different algorithms can be developed or used to solve the same problem.
- Different correct algorithms for the same problem can have different efficiencies.

Questions:

5. Explain the existence of undecidable problems in computer science.

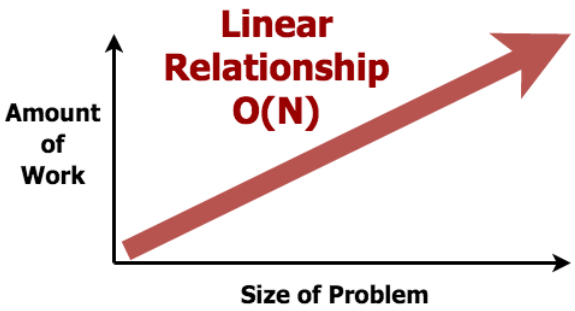
Search Algorithms

Imagine you needed to search through a list of 1,000,000 items.

Sequential Search

Linear search or sequential search algorithms check each element of a list, in order, until the desired value is found or all elements in the list have been checked.

If the item were toward the front of the list, Sequential Search would find the item with very few examinations. However, if the item were toward the end of the list, Sequential Search might need to look through as many as one million items! On average, depending on where the item might be located in the list, Sequential Search will require examining about 500,000 items. This is referred to as having linear performance because there is a linear relationship between the size of the problem (i.e., the total number of items you must search through) and the amount of work required to solve the problem (i.e., how many items you must actually look at before finding what you are looking for).



Binary Search

Compare that to a binary search, which does not see the same one-to-one relationship between the size of the problem and the amount of work.

The **binary search algorithm** starts at the middle of a sorted data set of numbers and eliminates half of the data; this process repeats until the desired value is found or all elements have been eliminated. In order to implement a binary search on a data set, the data must be in sorted order first.

Because of its divide-and-conquer approach, the amount of work required to find an item grows much more slowly with Binary Search than with Sequential Search. In fact, with this logarithmic behavior, you can double the total number of items in the list and it only costs one more unit of additional work. This means that if the size of the problem grows incredibly large, the amount of extra work that the algorithm will require will stay relatively small. Binary search is often more efficient than sequential/linear search when applied to sorted data.

Problems with Scalability

Small and simple problems are often easy to solve because even an inefficient solution likely doesn't require much work in the first place. But as problems grow larger, the efficiency of an algorithm becomes more and more important. So important that it might mean the difference between the problem even being solvable or not. Improvements in algorithms, hardware, and software increase the kinds of problems and the size of problems solvable by programming.

Case Study: Brute Forcing a Password

On the other hand, suppose law enforcement wants to develop a piece of forensics software that is intended to break the encryption on whatever data they need to investigate. The easiest solution is to take a "brute force" approach that simply tries all possible password combinations. It is essentially a Sequential Search through every possible password, which makes it a linear solution, which certainly seems like it should be a viable solution to the problem. But how well does it really scale?

In this case, it is not so much a problem with how efficiently the algorithm scales, but rather a problem with how quickly size and complexity of the problem can scale.

Imagine you have a four-digit (0–9) passcode, like you might have on your phone or home security system. How many possible passcodes exist? Well, if the code consists of four digits, each of which is one of 10 possible digits (0–9), then there are 10^4, or 10,000, unique combinations (0000 through 9999). A brute force approach would only need to make no more than 10,000 attempts to ensure success. Ten thousand is a lot, but trivial with the help of computational technology.

How can we make it harder to crack?

1. Make the passcode longer.
    - Every single digit added to the length of the passcode increases the number of possibilities by a factor of 10.
2. Increase the number of characters to use.
    - Instead of limiting the passcode to numerical digits, 0 through 9, it could use uppercase letters, lowercase letters, punctuation, symbols, emoji, etc.

Digits	Characters	Combinations
4	0-9 10	10^4 = 10,000
5	0-9 10	10^5 = 100,000

Digits	Characters	Combinations
4	0-9, a-z 36	36^4 = 1,679,616
4	0-9, a-z, A-Z 62	62^4 = 14,776,336
5	0-9, a-z, A-Z 62	62^5 = 916,132,832
8	0-9, a-z, A-Z 62	62^8 = 218,340,105,584,896
12	0-9, a-z, A-Z 62	62^{12} = 3.2E21

Notice how quick and easy it is to increase the size of the problem. Even just a meager, eight-digit passcode made up of alphanumeric characters (0–9, a–z, A–Z) has more than 200 trillion possible combinations, meaning the brute force approach could similarly require more than 200 trillion attempts before cracking the encryption. Even at only 12 digits long, such a passcode would allow so many possible passcodes that a brute force attempt taking only 1ms per attempt would still require ten times the age of the universe. Clearly, a brute force solution does not scale well.

6. Passwords can be variable in length. Many sites require a minimum length (e.g, eight characters)-why? How is this an issue of scalability?

7. How does forcing special characters and capital letters affect scalability? Is this is a good thing or a bad thing?

Note: This material is drawn from Codio - 1.8 Decidability and Performance-2