

Introduction to Algorithms & Data Structures 1

A solid foundation for the real world of machine learning and
data analytics



Bolakale Aremu

Ojula Technology Innovations

This is an electronic version of the print textbook. Due to electronic rights restrictions, some third-party content may be suppressed. Editorial review has deemed that any suppressed content does not materially affect the overall learning experience. The publisher reserves the right to remove content from this title at any time if subsequent rights restrictions require it. For valuable information on pricing, previous editions, changes to current editions, and alternate formats, please contact OjulaTech@gmail.com and inquire ISBN number, author, or title for materials in your areas of interest.

Introduction to Algorithms & Data Structures

First Edition



© 2023 Ojula Technology Innovations®

ISBN: 9791222093178

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented.

Ojula Technology Innovations is a leading provider of customized learning solutions with employees residing in nearly 45 different countries and sales in more than 130 countries around the world. For more information, please contact OjulaTech@gmail.com.

Printed in the United States of America

Print Number: 01

Print Year: April 2023

I am indebted to my mother for her love, understanding and support throughout the time of writing this textbook.

Bolakale Aremu

Table of Contents

0. What You Will Learn & How to Get Help

0.1. Benefits of learning about algorithms and data structures

0.2. Course Structure

1. Introduction to Algorithms

1.1. Playing a Counting Game

1.1.1. What is an Algorithm?

1.1.2. Guess the Number Game

1.1.3. Algorithm Guidelines

1.1.4. Practice Exercise 1

1.1.5. Answers to Practice Exercise 1

1.1.6. Evaluating Linear Search

1.1.7. Evaluating Binary Search

1.1.8. Practice Exercise 2

1.1.9. Answers to Practice Exercise 2

1.2. Time Complexity

1.2.1. Efficiency of an Algorithm

1.2.2. The Big O

1.2.3. Constant and Logarithmic Time

1.2.4. Linear & Quadratic Time

1.2.5. Cubic Runtime

1.2.6. Quasilinear Runtime

1.2.7. Polynomial Runtimes

1.2.8. Exponential Runtimes

1.2.9. How to Determine the Complexity of an Algorithm

1.2.10. Practice Exercise 3

1.2.11. Answers to Practice Exercise 3

1.3. Algorithms in Code

1.3.1. Linear Search in Code

1.3.2. Binary Search in Code

1.3.3. Recursive Binary Search in Code

1.3.4. Practice Exercise 4

1.3.5. Answers to Practice Exercise 4

1.4. Recursion and Space Complexity

1.4.1. Recursive Functions

1.4.2. Space Complexity

1.4.3. A Recap of What You Learned

1.4.4. Practice Exercise 5

1.4.5. Answers to Practice Exercise 5

1.5. Download Training Resources & Get Further Help

About The Author

My name is Bolakale Aremu. My educational background is in software development. I have a few colleagues who are software developers and system engineers. I spent over 17 years as a software developer, and I've done a bunch of other things too. I've been involved in SDLC/process, data science, operating system security and architecture, and many more. My most recent project is serverless computing where I simplify the building and running of distributed systems. I always use a practical approach in my projects and courses.

Bolakale Aremu
CEO, Ojula Technology Innovations
Web developer and Software Engineer
Ojulaweb.com

0. What You Will Learn & How to Get Help

The design of an efficient algorithm for the solution of the problem calls for the inclusion of appropriate data structures. In the field of computer science, data structures are used to store and organize data in a way that is easy to understand and use. They are used to organize and represent data in a way that will make it easier for computers to retrieve and analyze it. These are the fundamental building blocks that any programmer must know how to use correctly in order to build their own programs.

0.1. Benefits of learning about algorithms and data structures

First, they will help you become a better programmer. Another benefit is that they will make you think more logically. Furthermore, they can help you design better systems for storing and processing data. They also serve as a tool for optimization and problem-solving.

As a result, the concepts of algorithms and data structures are very valuable in any field. For example, you can use them when building a web app or writing software for other devices. You can apply them to machine learning and data analytics, which are two hot areas right now. If you are a hacker, algorithms and data structures in Python are also important for you everywhere.

Now, whatever your preferred learning style, I've got you covered. If you're a visual learner, you'll love my clear diagrams and illustrations throughout this book. If you're a practical learner, you'll love my hands-on lessons so that you can get practical with algorithms and data structures and learn in a hands-on way.

0.2. Course Structure

There are three volumes in this course. This is volume one. In this volume, you'll take a deep dive into the world of algorithms. With increasing frequency, algorithms are starting to shape our lives in many ways - from the products recommended to us, to the friends we interact with on social media, to even important social issues like policing, privacy and healthcare. So, the first part of this course covers what **algorithms** are, how they work, and where they can be found (real life applications).

In the second volume, you'll work through an introduction to data structures. You're going to learn about two introductory data structures - **arrays** and **linked lists**. You'll look at common operations and how the runtimes of these operations affect our everyday code.

In the third volume, you're going to bring your knowledge of algorithms and data structures together to solve the problem of sorting data using the Merge Sort algorithm. We will look at algorithms in two categories: **sorting** and **searching**. You'll implement well-known sorting algorithms like Selection Sort, Quicksort, and Merge Sort. You'll also learn basic search algorithms like Sequential Search and Binary Search.

At the end of many sections of this course, short practice exercises are provided to test your understanding of the topic discussed. Answers are also provided so you can check how well you have performed in each section. At the end of the course, you will find a **link to download more helpful resources, such as codes and screenshots used in this book, and more practice exercises**. You can use them for quick references and revision as well. My **support link is also provided** so you to contact me any time if you have questions or need further help.

By the end of this course, you will understand what algorithms and data structures are, how they are measured and evaluated, and how they are used to solve real-life problems. So, everything you need is right here in this book. I really hope you'll enjoy it. Are you ready? Let's dive in!

1. Introduction to Algorithms

Whether you are a high school or college student, a developer in the industry or someone who is learning to code, you have undoubtedly run into the term algorithm. For many people, this word is kind of scary. It represents this body of knowledge that seems just out of reach. Only people with computer science degrees know about algorithms.

Now to others, this brings up feelings of imposter syndrome. You might already know how to code, but you're not a real developer because you don't know anything about algorithms. It makes some developers frame certain jobs as above their skill level because the interview contained algorithm questions. Well, whatever your reasons are, in this course, our goal is to dispel all those feelings and get you comfortable with the basics of algorithms. Like any other subject, I like to start my courses with what the course is and what is not.

In this part of the course, we're going to cover the very basic set of knowledge that you need as a foundation for learning about algorithms. This part is less about specific algorithms and more about the tools you will need to evaluate algorithms, understand how they perform, compare them to each other, and make a statement about the utility of an algorithm in a given context.

Now don't worry. None of this will be theoretical, and we will learn these concepts by using well-known algorithms. We'll also be writing code, so I do expect you to have some programming experience if you intend to continue with this topic. You can definitely stick around even if you don't know how to code, but you might want to learn the basics of programming.

In the meantime, we will be using the Python programming language. Python reads a lot like regular English and is the language you will most likely encounter when learning about algorithms these days. If you don't know how to code, or if you know how to code in a different language, check out the following book that completely simplifies Python programming language for beginners:

<https://www.scribd.com/book/513773394/Python-Programming-from-Beginner-to-Paid-Professional-Part-1-Learn-Python-for-Automation-IT-with-Video-Tutorials> (or use this BIT.LY short link: <http://bit.ly/3ZKREhB>).

Even if you know nothing about Python, as long as you understand the fundamentals of programming, you should be able to follow along pretty well. If you're a JavaScript developer or a student who's learning JavaScript for example, chances are good that you'll still be able to understand the code we write later. I'll be sure to provide links along the way if you need anything to follow up on. Let's start with something simple.

1.1. Playing a Counting Game

Let's get a gentle introduction to algorithms by playing a simple game with two of my colleagues! Over the next few lessons, we'll talk about algorithmic thinking, what an algorithm is and take our first look at two popular search algorithms.

1.1.1. What is an Algorithm?

An algorithm is a set of steps or instructions for completing a task. This might sound like an oversimplification, but really that's precisely what an algorithm is. A **recipe** is an algorithm. Your **morning routine** when you wake up is an algorithm and the **driving directions** you follow to get to a destination is also an algorithm. In computer science, the term algorithm more specifically means the set of steps a program takes to finish a task.

If you've written code before (such as `print("Hello World")` or any code really), generally speaking, you have written an algorithm. Given that much of the code we write can be considered an algorithm, what do people mean when they say you should know about algorithms? Now consider this.

Let's say I'm a teacher in a classroom and I tell everyone I have an assignment for them. On their desks, they have a picture of a maze and their task is to come up with the way to find the quickest way out of the maze. Everyone does their thing and comes up with a solution.



Figure 1.1.1: Students solved a maze problem in a classroom

Every single one of these solutions is a viable solution and is a valid example of an algorithm: the steps one needs to take to get out of the maze.



Figure 1.1.2: Three solutions to the maze problem were drawn on the board by the teacher

But from being in classrooms or any group of any sort, you know that some people will have better ideas than others. We all have a diverse array of skillsets. Over time, our class picks the best of these solutions, and anytime we want to solve a maze, we go with one of these solutions. This is what the field of algorithms is about. There are many problems in computer science, but some of them are pretty common. Regardless of what project you're working on, different people have come up with different solutions to these common problems, and over time, the field of computer science has identified several that do the job well for a given task.

When we talk of algorithms, we're referring to two points. We're primarily saying there's an established body of knowledge on how to solve particular problems well, and it's important to know what these solutions are. Now, why is it important? If you're unaware that a solution exists, you might try to come up with one yourself and there's a likelihood that your solution won't be as good or efficient, whatever that means, compared to those that have been thoroughly reviewed. But there's a second component to it as well.

Part of understanding algorithms is not just knowing that an algorithm exists, but understanding when to apply it. Understanding when to apply an algorithm requires properly understanding the problem at hand, and this arguably is the most important part of learning about algorithms and data structures. As you progress through this chapter, you should be able to look at a problem and break it down into distinct steps. When you have a set of steps, you should then be able to identify which algorithm or data structure is best for the task at hand.

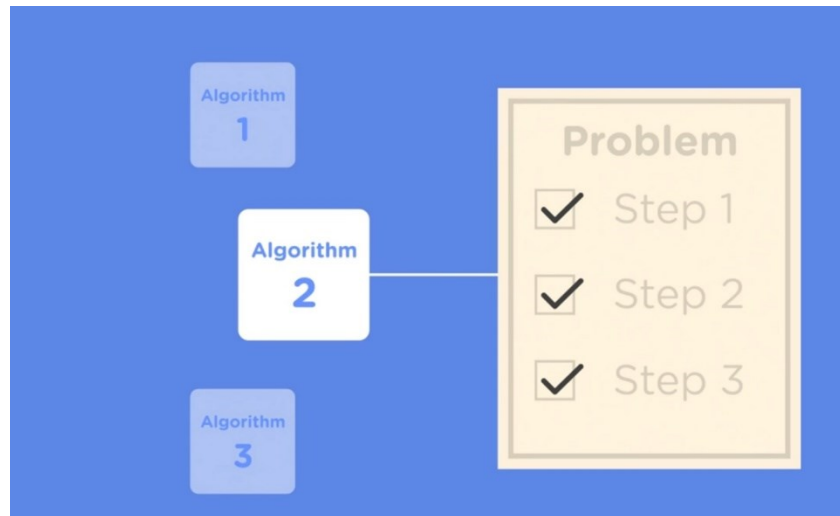


Figure 1.1.3: Algorithmic thinking: Algorithm 2 was found to be the best for solving the problem

This concept is called **algorithmic thinking**, and it's something we're going to try and cultivate together as we work through this chapter. Lastly, learning about algorithms gives you a deeper understanding about complexity and efficiency in programming. Having a better sense of how your code will perform in different situations is something that you'll always want to develop and hone.

Algorithmic thinking is why algorithms also come up in big tech interviews. Interviewers don't care as much that you are able to write a specific algorithm in code, but more about the fact that you can break a seemingly insurmountable problem into distinct components, and identify the right tools to solve each distinct component. That is what we plan on doing. In this course though, we're going to focus on some of the tools and concepts you'll need to be aware of before we can dive into the topic of algorithms. If you're ready, let's get started!

1.1.2. Guess the Number Game

In this section, I'm going to do something unusual. I have two friends, Britney and John, who are going to play a game with me. This game is really simple, and you may have played it before. It goes something like this. I'm going to think of a number between 1 and 10, and they have to guess what the number is. Easy, right? When they guess a number, I'll tell them if their guess is too high or too low. The winner is the one with the *fewest* tries. Between you and me, the answer is 3. All right, let's start with John.

Me: I'm thinking of a number between 1 and 10. What is it?

John: Uh, quick question. Does the range include 1 and 10?

That is a really good question. What John did right there was to establish the bounds of our problem. No solution works on every problem, and an important part of algorithmic thinking is to clearly define what the problem set is, and clarify what values count as inputs.

Me: Yeah, 1 and 10 are both included.

John: Is it 1?

Me: Too low?

John: Is it 2?

Me: Too low?

John: Is it 3?

Me: Correct!

Okay, that was an easy one. It took John **three tries** to get the answer. Let's switch over to Britney and play another round using the same number as the answer.

Me: Britney, I'm thinking of a number between 1 and 10 inclusive, so both 1 and 10 are in the range. What number am I thinking of?

Britney: Is it 5?

Me: Too high.

Britney: Two?

Me: Too low?

Britney: Is it 3?

Me: Correct!

What we had there was two very different ways of playing the same game. Somehow with even such a simple game, we saw different approaches to figuring out a solution. To go back to algorithmic thinking for a second, this means that with any given problem, there's no one best solution. Instead, what we should try and figure out is what solution works better for the current problem. In this first pass at the game, they both took the same amount of turns to find the answer, so it's not obvious who has the better approach, and that's mostly because the game was easy. Let's try this one more time. This time the answer is 10. Now, John first.

John: Is it 1?

Me: Too low.

John: Is it 2?

Me: Still too low.

John: Is it 3?

Me: Too low.

John: Is it 4?

Me: Too low.

John: Is it 5?

Me: Still too low.

John: Is it 6?

Me: Too low.

John: Is it 7?

Me: Too low.

John: Is it 8?
Me: Too low.
John: Is it 9?
Me: Too low.
John: Is it 10?
Me: Correct. You got it!

Now let's do the same thing, but with Britney this time,

Britney: Is it 5?
Me: Too low.
Britney: 8?
Me: Too low.
Britney: Is it 9?
Me: Still too low.
Britney: It's 10.

Here, we start to see a difference between their strategies. When the answer was 3, they both took the same number of turns. This is important. When the number was larger, but not that much larger, 10 in this case, we start to see that Britney's strategy did better. She took **four** tries while John took **ten**. We've played two rounds so far, and we've seen a difference set of results based on the number they were looking for.

If you look at John's way of doing things, then the answer being 10 (the round we just played), is his worst-case scenario. He will take the maximum number of turns (ten) to guess it. When we picked a random number like 3, it was hard to differentiate which strategy was better because they both performed exactly the same. But in John's worst case scenario, a clear winner in terms of strategy emerges. In terms of algorithmic thinking, we're starting to get a sense that the specific value they're searching for may not matter as much as where that value lies in the range that they have been given.

Identifying this helps us understand our problem better. Let's do this again for a range of numbers from 1 to 100. We'll start by picking 5 as an answer to trick them. This time, we're going to run through the exercise again, but this time from 1 to 100 and both 1 and 100 are included.

At this point, without even having to run through the exercise, we can guess how many tries John is going to take. Since he starts at 1 and keeps going, he's going to take **five tries**. For Britney's turn, let's run through the exercise.

Britney: Is it 50?
Me: Too high.
Britney: Is it 25?

Me: Still too high.

Britney: Is it 13?

Me: Too high.

Britney: Is it 7?

Me: Too high.

Britney: Is it 4?

Me: Too low.

Britney: Is it 6?

Me: Too high.

Britney: Is it 5?

Me: Correct!

Let's evaluate. John took five tries. Britney on the other hand took seven tries, so John wins this round. But again, in determining whose strategy is preferred, there's no clear winner right now. What this tells us is that it's not particularly useful to look at the easy answers where we arrive at the number fairly quickly because it's at the start of the range. Instead, let's try one where we know John is going to do poorly. Let's look at his worst-case scenario where the answer is 100 and see how Britney performs in such a scenario.

If I do this exercise one more time with John, 1 through 100 again, you can guess what happens. John takes 100 tries.

Me: Now, Britney, you're up.

Britney: Is it 50?

Me: Too low.

Britney: Is it 75?

Me: Too low.

Britney: 88?

Me: Too low,

Britney: 94?

Me: Too low.

Britney: Is it 97?

Me: Too low.

Britney: 99?

Me: Too Low.

Britney: 100.

Me: correct.

That took Britney seven turns again and this time she is the clear winner. If you compare their individual performances for the same number set, you'll see that Britney's approach leaves John's in the dust. When the answer was 5, so right around the start of the range, John took five turns, but when the answer was 100 right at the end of the range, he took 100 tries. It took him 20 times

the amount of tries to get that answer compared to Britney.

On the other hand, if you compare Britney's efforts when the number was five, she took seven tries, but when the number was 100, she took the same amount of tries. This is pretty impressive. If we pretend that the number of tries is the number of seconds it takes Britney and John to run through their attempts, this is a good estimate for how fast their solutions are.

Okay, we've done this a couple times and Britney and John are getting tired. Let's take a break. In the next section, we'll talk about the point of this exercise.

1.1.3. Algorithm Guidelines

In the last section we ran through an exercise where I had some of my coworkers guess what number I was thinking. What was the point of that exercise? You might be thinking, hey, I thought I was here to learn about algorithms. The exercise we just did was an example of a real-life situation you'll run into when building websites, apps and writing code. Both approaches taken by John and Britney to find the number I was thinking of are examples of searching for a value. It might be weird to think that there's more than one way to search, but as you saw in the game, the speed at which the result was obtained differed between John and Britney.

Think about this problem from the perspective of a company like Facebook. At the time of writing this book, Facebook has over 2.96 billion active users. Let's say you're traveling in a different country and meet someone you want to add on Facebook. You go into the search bar and type out this person's name. If we simplify how the Facebook app works, it has to search across these 2.96 billion records and find the person you are looking for.

The speed at which you find this person really matters. Imagine what kind of experience it would be if when you search for a friend, Facebook put up a spinning activity indicator and said, come back in a couple hours. I don't think we'd use Facebook as much if that was the case. From the company's perspective, working on making search as fast as possible using different strategies really matters.

Now, I said that the two strategies Britney and John used were examples of search. More specifically, these are **search algorithms**. The strategy John took where he started at the beginning of the range and just counted one number after the other is a type of search called **linear search**. It is also called **sequential search**, which is a better description of how it works, or even, **simple search**, since it really is quite simple.

But what makes his approach an algorithm as opposed to just looking for something? Remember we said that **an algorithm is a set of steps or instructions to complete a task**? Linear search is a search algorithm and we can visualize it like this. We start at the beginning of the list or the range of values. Then we compare the current value to the target. If the current value is the target value that we're looking for, we're done (step 1). If it's not, we'll move on sequentially to the next

value in the list (step 2) and then repeat step 2. If we reach the end of the list, then the target value is not in the list.

This definition has nothing to do with programming and in fact you can use it in the real world. For example, I could tell you to walk into a bookstore and find me a particular book, and one of the ways you could do it is using the linear search algorithm. You could start at the front of the bookstore and read the cover or the spine of every book to check that it matches the book that you're looking for. If it doesn't, you go to the next book and repeat until you find it or run out of books.

What makes this an algorithm is the **specificity** of how it is defined. In contrast to just jumping into a problem and solving it as we go along, an algorithm follows a certain set of guidelines and we use the same steps to solve the problem each time we face it. An important first step to defining the algorithm is not the algorithm itself, but the problem we're trying to solve.

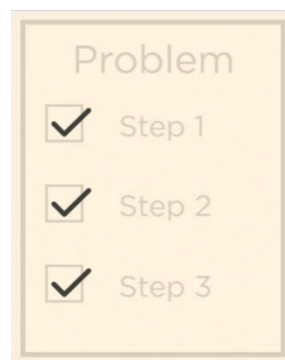


Figure 1.1.4: The number of steps of an Algorithm for solving a problem

Our first guideline is that an algorithm must have a **clear problem statement**. It's pretty hard to define an instruction set when you don't have a clear idea of what problem you're trying to solve. In defining the problem, we need to specify how the input is defined and what the output looks like when the algorithm has done its job. For linear search, the input can be generally described as a series of values and the output is a value matching the one we're looking for.

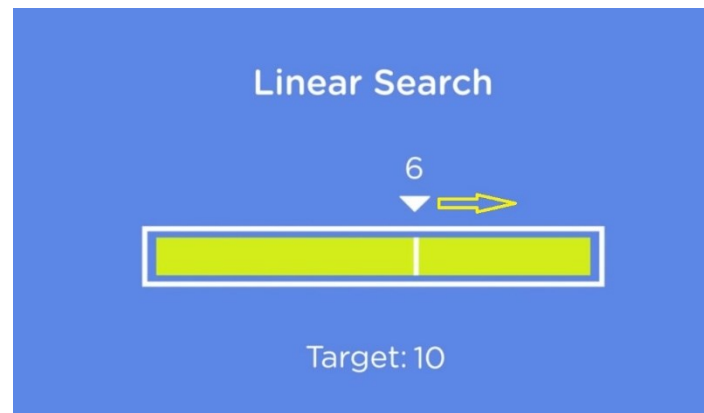


Figure 1.1.5: An example of a linear search Algorithm

Right now, we're trying to stay away from anything code related, so this problem statement definition is pretty generic. But once we get to code, we can actually tighten this up. Once we have a problem. An algorithm is a set of steps that solves this problem.

Given that the next guideline is that **an algorithm definition must contain a specific set of instructions in a particular order**, we really need to be clear about the order in which these instructions are executed. So specific order is really important.

Algorithm

- 1 Start at beginning
- 2 Compare current value to target
- 3 Move sequentially
- 4 Reach end of list

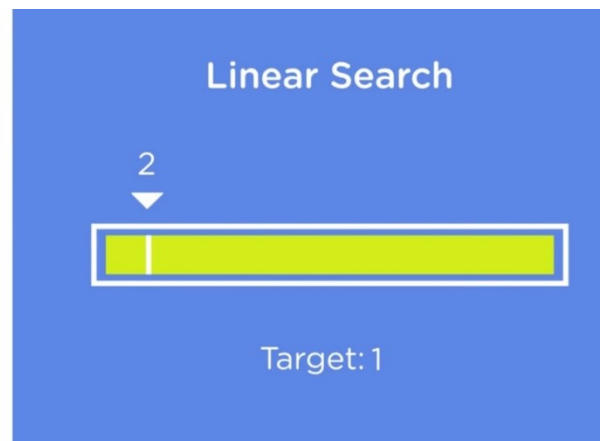


Figure 1.1.6: An ordered list of steps required to complete a linear search Algorithm

Taking our simple definition of linear search, if I switched up the order in Figure 1.1.6 and said, move sequentially to the next value before specifying that first comparison step, if the first value were the target one, our algorithm wouldn't find it because we moved to the second value before comparing. Now you might think, okay, that's just an avoidable mistake and kind of common sense. The thing is computers don't know any of that and just do exactly as we tell them, so specific order is really important.

The third guideline is that **each step in our algorithm definition must not be a complex one** and needs to be explicitly clear. What I mean by that is that you shouldn't be able to break down any of the steps into further or additional subtasks. Each step needs to be a distinct one. We can't

define linear search as search until you find this value because that can be interpreted in many ways and further broken down into many more steps. It's not clear.

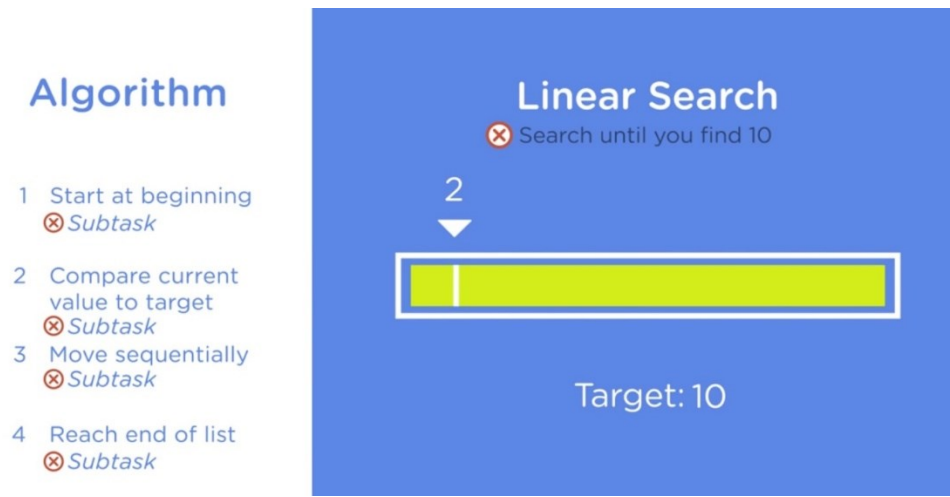


Figure 1.1.7: Each step of an Algorithm needs to be a distinct one; subtasks are not allowed

Next, **algorithms should produce a result**. This one might seem obvious. If it didn't, how would we know whether the algorithm works or not. To be able to verify that our algorithm works correctly, we need a result. Now, when using a search algorithm, the end result can actually be nothing, which indicates that the value wasn't found. But that's perfectly fine. There are several ways to represent “nothing” in code, and as long as the algorithm can produce some result, we can understand its behavior.

The last guideline is that the **algorithm should actually complete and cannot take an infinite amount of time**. If we let John loose in the world's largest library and ask him to find a novel, we have no way of knowing whether he succeeded or not, unless he came back to us with a result.

Now, here's a quick recap. What makes an algorithm an algorithm and not just something you do?

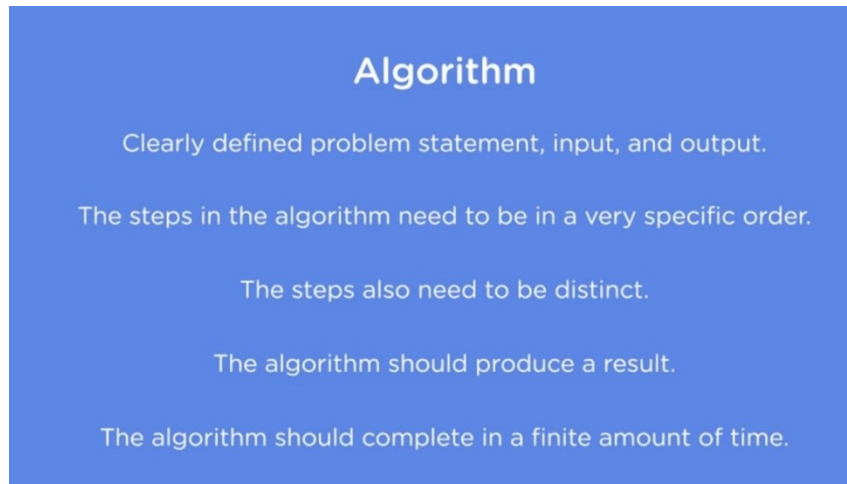


Figure 1.1.8: A recap of 5 Algorithm guidelines

1. It needs to have a clearly defined problem statement, input, and output. When using linear search, the input needs to be just a series of values, but to actually use Britney's strategy, there's one additional precondition so to speak. If you think about her strategy, it required that the numbers be sorted in ascending order. This means that where the input for John is just a series of values to solve the problem, the input to Britney's algorithm needs to be a sorted series of values. So, clearly defined problem statement, clearly defined input and clearly defined output.
2. The steps in the algorithm need to be in a very specific order.
3. The steps also need to be distinct. You should not be able to break it down into further sub-tasks.
4. Next, the algorithm should produce a result, and finally,
5. The algorithm should complete in a finite amount of time.

These guidelines not only help us define what an algorithm is, but also helps us verify that the algorithm is correct. Executing the steps in an algorithm for a given input must result in **the same output every time**. If in the game I played above the answer was 50 every time, then every single time John must take 50 turns to find out that the answer is 50. If somehow he takes 50 turns in one round, then 30 the next, we technically don't have a correct algorithm. Consistent results for the same set of values is how we know that the algorithm is correct.

I should stress that we're not going to be designing any algorithms on our own and we'll start off and spend most of our time learning the tried-and-true algorithms that are known to efficiently solve problems. The reason for talking about what makes for a good algorithm though is that the same set of guidelines makes for good algorithmic thinking, which is one of the most important skills we want to cultivate. When we encounter a problem, before rushing in and thinking about solutions, what we want to do is work through the guidelines. First, we break down the problem

into any possible number of smaller problems where each problem can be clearly defined in terms of an input and an output.

1.1.4. Practice Exercise 1

Q1. If we cannot determine the output of a program, can we call it an algorithm even if it solves the problem at hand?

- A. No.
- B. Yes.

Q2. For which of the following reasons do we learn about algorithms and data structures?
Choose the correct answer below:

- A. To be aware of existing solutions to common problems
- B. To understand the efficiency of solutions to common problems
- C. To be able to evaluate and break down problems into distinct parts
- D. All of the above

1.1.5. Answers to Practice Exercise 1

- 1. A
- 2. D

1.1.6. Evaluating Linear Search

Now that we know how to generally define an algorithm, let's talk about what it means to have a good algorithm. An important thing to keep in mind is that there's no one single way to measure whether an algorithm is the right solution because it is all about context. Earlier we touched on two concepts, **correctness** and **efficiency**. Let's define correctness more clearly because before we can evaluate an algorithm on efficiency, we need to ensure it's correctness. Before we define our algorithms, we start by defining our problem.

In the definition of that problem, we have a clearly defined input, satisfying any preconditions and a clearly defined output. An algorithm is deemed correct if on every run of the algorithm against all possible values in the input data, we always get the output we expect. Part of correctness means that for any possible input, the algorithm should always terminate or end. If these two are not true, then our algorithm isn't correct.

If you were to pick up an Algorithms textbook and look up *correctness*, you will run into a bunch of mathematical theory. This is because traditionally algorithm correctness is proved by mathematical induction, which is a form of reasoning used in mathematics to verify that a statement is correct. This approach involves writing what is called a specification and a correctness proof. We won't be going into that in this course. Proof through induction is an important part of designing algorithms, but we're confident that you can understand algorithms

both in terms of how and when to use them without getting into the math, so if you pick up a textbook and feel daunted, don't worry. I do too. But we can still figure things out without it.

So, once we have a correct algorithm, we can start to talk about **how efficient** an algorithm is. Remember that this efficiency ultimately matters because they help us solve problems faster and deliver a better end user experience in a variety of fields. For example, algorithms are used in the sequencing of DNA and more efficient sequencing algorithms allow us to research and understand diseases better and faster, but let's not get ahead of ourselves. We'll start simple by evaluating John's linear search algorithm in terms of its efficiency.

First, what do we mean by efficiency? There are **two measures of efficiency** when it comes to algorithms, time and space. Sounds really cool and very sci-fi, huh? Efficiency measured by time, something you'll hear called **time complexity**, is a measure of how long it takes the algorithm to run. Time complexity can be understood generally outside the context of code and computers because how long it takes to complete a job is a universal measure of efficiency. The less time you take, the more efficient you are.

The second measure of efficiency is called **space complexity**, and this is pretty computer specific. It deals with the amount of memory taken up on the computer. Good algorithms need to balance between these two measures to be useful. For example, you can have a blazingly fast algorithm, but it might not matter if the algorithm consumes more memory than you have available.

Both of these concepts, time and space complexity, are measured using the same metric, but it is a very technical sounding metric, so let's build up to it slowly and start simple. In section 1.1.2, I played a game with John and Britney where they tried to guess the number I was thinking of. Effectively, they were searching for a value. So, how do we figure out how efficient each algorithm is and which algorithm was more suited to our purposes?

If we consider the number of tries they took to guess or search for the value as an indicator of the time they take to run through the exercise, this is a good indicator of how long the algorithm runs for a given set of values. This measurement is called the **running time** of an algorithm and we'll use it to define time complexity in the game. We played it 4 rounds. Let's recap those in Figure 1.1.9, focusing on John's performance.

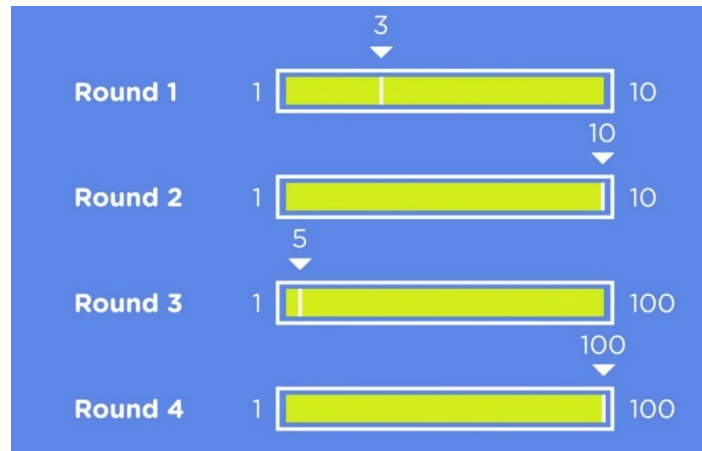


Figure 1.1.9: Chart showing John's performance in 4 rounds of a game played in section 1.1.2

In round 1, we had 10 values, the target was 3 and John took 3 turns. In round 2, we had 10 values. The target was 10 and John took 10 turns. In round 3, we had 100 values, the target was 5, John took 5 tries. Finally in round 4, when the target was 100, given 100 values, John took 100 tries. On paper, it's hard to gauge anything about this performance. When it comes to anything with numbers though, I like to put it up on a graph and compare visually on the vertical or Y-axis.

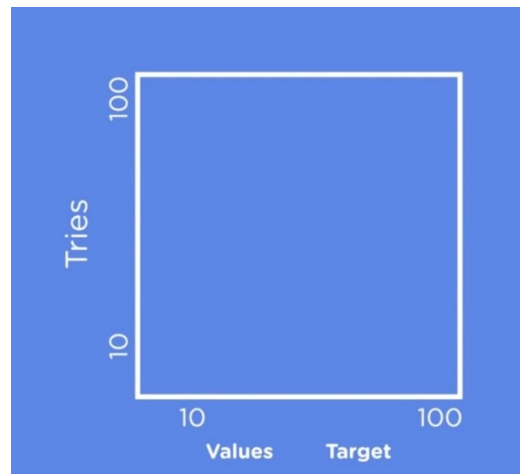


Figure 1.1.10: A graph to compare John's number of tries (running time) against the number of values/target value

Let's measure the number of tries it took John to guess the answer, or the running time of the algorithm. On the horizontal or x-axis, what do we put for each turn? We have a number of **Values** as well as a **Target** value. We could plot the target value on the horizontal axis, but that leaves some context and meaning behind. It's far more impressive that John took 5 tries when the range went up to a 100 than when he took 3 tries for a maximum of 10 values. We could plot the maximum range of values, but then we're leaving out the other half of the picture. There are data points, however that satisfy both requirements.

If we only plot the values where the target, the number John was looking for, was the same as the maximum range of values, we have a data point that includes both the size of the data set as well as his effort. There's an additional benefit to this approach as well. There are three ways we can measure how well John does or in general, how well any algorithm does.

First, we can check how well John does in the best case or good scenarios from the perspective of his strategy in the range of 100 values. The answer being a lone number like 3 at the start of the range is a good scenario. He can guess it fairly quickly. 1 is his best-case scenario.

Alternatively, we could check how well he does on average. We could run this game a bunch of times and average out the running time. This would give us a much better picture of John's performance over time, but our estimates would be too high if the value he was searching for was at the start of the range or far too low if it was at the end of the range.

Let's imagine a scenario where Facebook naively implements linear search when finding friends. They looked at the latest US census, saw that 50% of names start with the letters A through J, which is the first 40% of the alphabet, and thought, okay, on average linear search serves us well, but what about the rest of those whose names start with the letter *after* J in the alphabet? Searching for a name like Peter would take longer than the average and much longer for someone whose name starts with a letter Z.

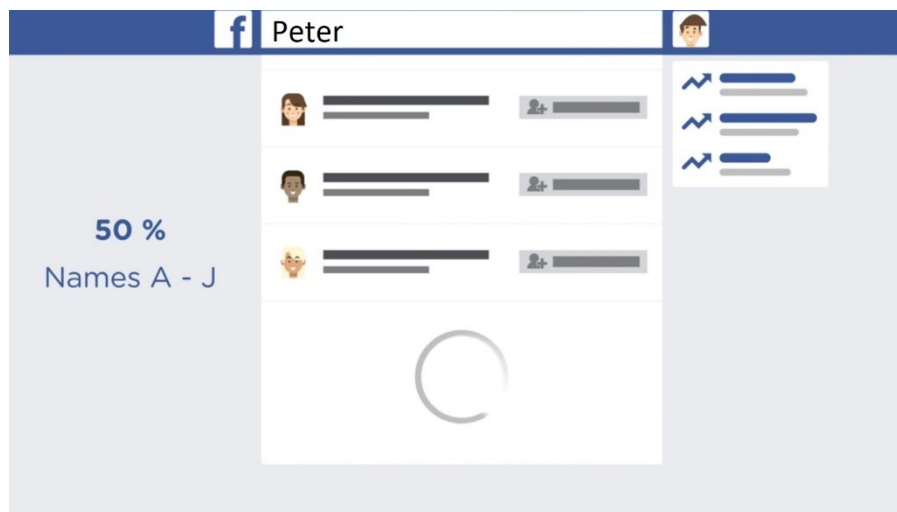


Figure 1.1.11: An example of implementation of linear search on Facebook

So, while measuring the runtime of an algorithm, on average, might seem like a good strategy, it won't necessarily provide an accurate picture. By picking the maximum in the range, we're measuring how our algorithm does in the worst-case scenario. Analyzing the worst-case scenario is quite useful because it indicates that the algorithm will never perform worse than we expect. There's no room for surprises.

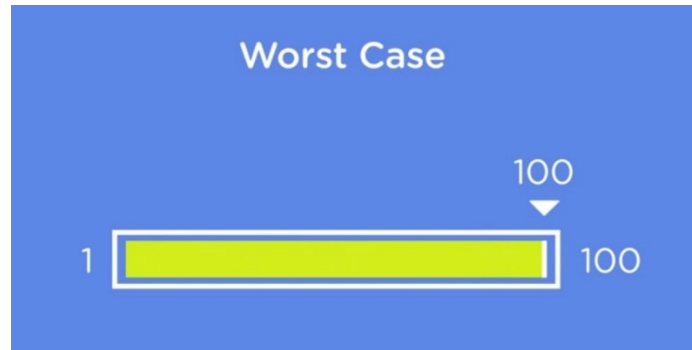


Figure 1.1.12: The worst-case scenario of our algorithm

Back to our graph in Figure 1.1.10, we're going to plot the number of tries, a proxy for running time of the algorithm, against the number of values in the range, which will shorten to n . " n " here also represents John's worst case scenario. When n is 10, he takes 10 turns. When n is 100, he takes 100 turns. But these two values alone are insufficient to really get any sort of visual understanding. Moreover, it's not realistic. John may take a long time to work through 100 numbers, but a computer can do that in no time. To evaluate the performance of linear search in the context of a computer, we should probably throw some harder and larger ranges of values added.

The nice thing is by evaluating a worst-case scenario, we don't actually have to do that work. We know what the result will be. For a given value of n , using linear search, it will take n tries to find the value in the worst-case scenario. So, let's add a few values in there to build out the graph shown in Figure 1.1.13.

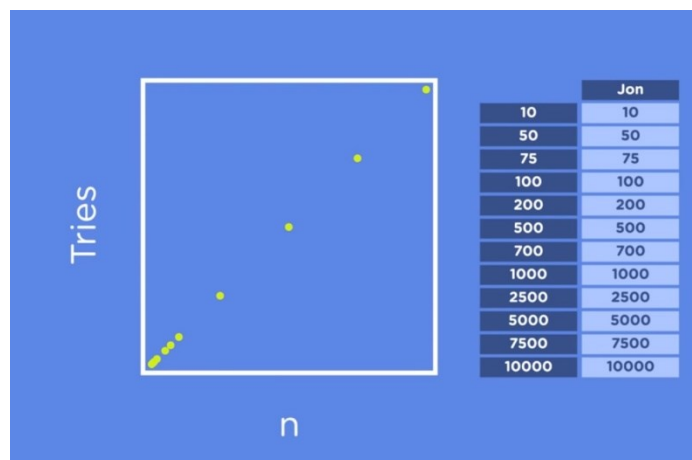


Figure 1.1.13: John's performance graph with larger ranges of values of n added

Okay, so we have a good picture of what this is starting to look like. As the values get really large, the running time of the algorithm gets large as well. Well, we sort of already knew that. Before we dig into this runtime any deeper, let's switch tracks and evaluate Britney's work. By having something to compare against, it should become easier to build a mental model around

time complexity.

1.1.7. Evaluating Binary Search

The algorithm John used, linear search, seemed familiar to us, and you could understand it because it's how most of us search for things in real life anyway,

Britney's approach on the other hand got results quickly, but it was a bit harder to understand, so let's break it down. Just like John's approach, Britney started with a series of values, or a list of numbers, as her input. Where John just started at the beginning of the list and searched sequentially, Britney's strategy is to always start in the middle of the range. From there, she asks a comparison question.

Is the number in the middle of the range equal to the answer she's looking for, and if it's not, is it greater than or less than the answer? If it's greater than, she can eliminate all the values *less than* the one she's currently evaluating, as illustrated in Figure 1.1.14. If it's lesser than the answer, she can eliminate all the values *greater than* the one she's currently evaluating with the range of values that she's left over with. She repeats this process until she arrives at the answer.

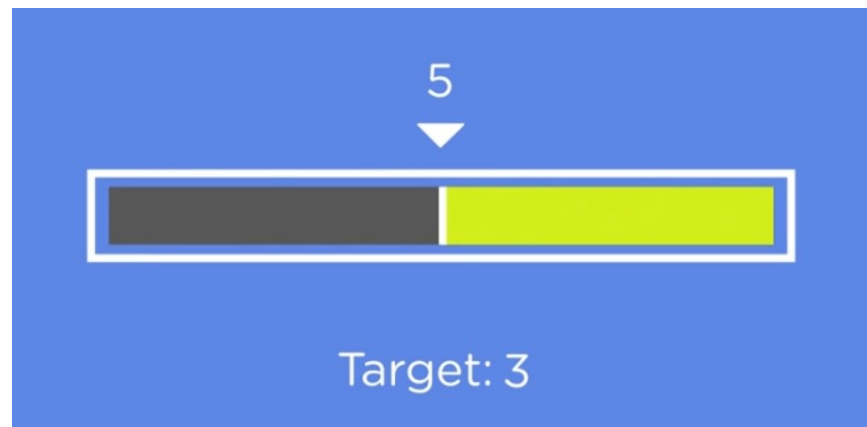


Figure 1.1.14: Chart illustrating how Britney eliminates all the values less than the one she's currently evaluating

Let's visualize how she did this by looking at round 3. In round 3, the number of values in the range was 100. The answer was 5. The bar in Figure 1.1.15 represents the range of values, 1 at the left, 100 at the right, and the pointer represents the value Britney chooses to evaluate. So, she starts in the middle at 50. She asks, is it equal to the answer? I say it's too high. This tells her that the value she's evaluating is greater than our target value, which means there's no point in searching any of the values to the right of 50. that is, values greater than 50 in this range, so she can discard those values altogether.

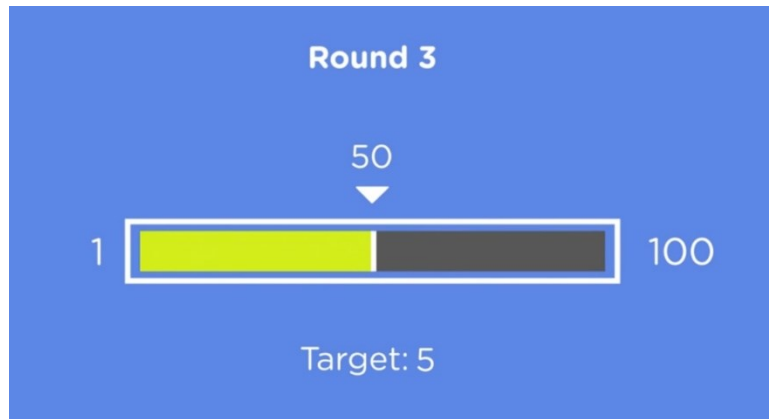


Figure 1.1.15: Chart illustrating how Britney eliminates all the values greater than 50

She only has to consider values from 1 to 50. Now, the beauty of this strategy and the reason why Britney was able to find the answer in such few turns is that with every value she evaluates, she can discard half of the current range. On her second turn, she picks the value in the middle of the current range, which is 25. She asks the same question. I say that the value is too high again, and this tells her that she can discard everything greater than 25 and the range of values drops from 1 to 25.

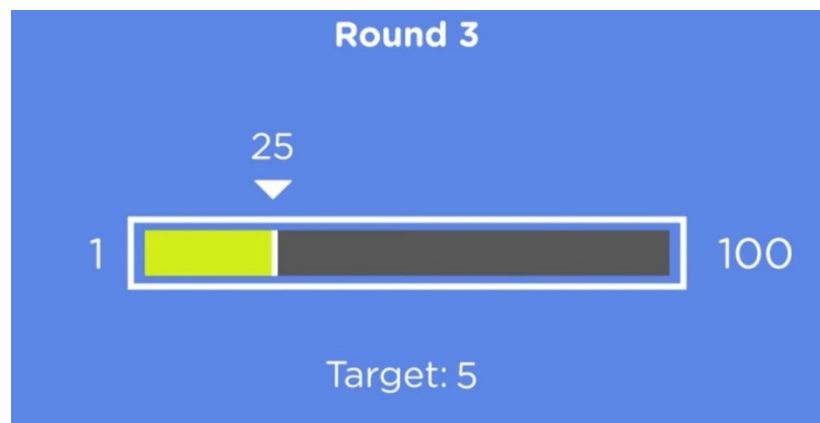


Figure 1.1.16: Chart illustrating how Britney eliminates all the values greater than 25

Again, she evaluates the number in the middle roughly, so that would be 13 in Figure 1.1.16. I tell her this is still too high. She discards the values greater, moves to value at 7, which is still too high. Then she moves to 4, which is now **too low**. She can now discard everything less than 4, which leaves the numbers 4 through 7. Then she picks 6, which was too high, This only leaves one value: **5**.

This seems like a lot of work, but being able to get rid of half the values with each turn is what makes this algorithm much more efficient. Now, there's one subtlety to using binary search, and you might have caught onto this. For this search method to work, as we've mentioned, the values need to be sorted.

With linear search, it doesn't matter if the values are sorted since a linear search algorithm just progresses sequentially, checking every element in the list. If the target value exists in the list, it will be found. But let's say this range of values 1 to 100 was unsorted, Britney would start at the middle with something like 14, and ask if this value was too low or too high. I say it's too high, so she discards everything less than 14.

Now, this example starts to fall apart here because while Britney knows what numbers are less than 14 and greater than 1, she doesn't need an actual range of values to solve this. A computer however does need that. Remember, search algorithms are run against lists containing all sorts of data. It's not always just a range of values containing numbers. In a real use case of binary search, which we're going to implement in a bit, the algorithm would not return the target value (5) because we already know that. It's a search algorithm, so we're providing something to search for. Instead, what it returns is the position in the list that the target occupies.

Without the list being sorted, a binary search algorithm would discard all the values to the left of 14, which could include the position where our target value is. Eventually, we'd get a result back saying the target value doesn't exist in the list, which is inaccurate. Earlier when defining linear simple search, I said that the input was a list of values and the output was the target value, or more specifically the position of the target value in the list.

Therefore, with binary search, there's also that precondition: the input list must be sorted. So, let's formally define binary search.

The **input** is a sorted list of values.

The **output** is the position in the list of the target value we're searching for, or some sort of values indicate that the target does not exist in the list.

Remember our guidelines for defining an algorithm? Read them up again really quick in Figure 1.1.8. Let's use those to define this algorithm.

- **Step 1:** We determine the middle position of the sorted list.
- **Step 2:** We compare the element in the middle position to the target element.
- **Step 3:** If the elements match, we return the middle position and end. If they don't match then step 4.
- **Step 4:** We check whether the element in the middle position is smaller than the target element. If it is, then we go back to step 2 with a new list that goes from the middle position of the current list to the end of the current list.
- **Step 5:** If the element in the middle position is greater than the target element, then again we go back to step 2 with a new list that goes from the start of the current list to the middle position of the current list.

We repeat this process until the target element is found, or until a sublist contains only one

element. If that single element sublist does not match the target element, then we end the algorithm, indicating that the element does not exist in the list.

That is the magic behind how Britney managed to solve the round much faster. In the next section, we'll talk about the efficiency of binary search.

1.1.8. Practice Exercise 2

Q1. Which of the following best defines the input for binary search? Choose the correct answer below:

- A. A sorted, sequential series of arbitrary data
- B. A sequential series of arbitrary data

Q2. An algorithm is considered correct if, on every run of the algorithm, against all possible values in the input data we always get the output we expect. Choose the correct answer below.

- A. False
- B. True

Q3. Choose the correct answer below. Time complexity can be described as

- A. The efficiency of an algorithm measured by the amount of time it takes in relation to the size of the data set.
- B. The efficiency of an algorithm measured by the amount of memory it consumes in relation to the size of the data set.

1.1.9. Answers to Practice Exercise 2

- 1. A
- 2. B
- 3. A

1.2. Time Complexity

Over the last few lessons, we saw two of my colleagues play a game with different strategies and different end results. These strategies were examples of real-world algorithms but we still don't have a good understanding of which one was better. In this section, we're going to take a look at time complexity and measuring the efficiency of algorithms.

1.2.1. Efficiency of an Algorithm

We have a vague understanding that Britney's approach is better in most cases, but just like with linear search, it helps to visualize this. Much like we did with linear search, when determining the efficiency of an algorithm (remember, we're still only looking at efficiency in terms of time - time complexity as it's called), we always want to evaluate how the algorithm performs in the worst-case scenario.

Now, you might be thinking, well, that doesn't seem fair because given a series of data, if the target value we're searching for is somewhere near the front of the list, then linear search may perform just as well, if not slightly better than binary search. That is totally true. Remember, a crucial part of learning algorithms is understanding what works better in a given context. When measuring efficiency though, we always use the worst-case scenarios as a benchmark because remember, it can never perform worse than the worst case. Let's plot these values on the graph we started earlier.

With the number of tries, or the runtime of the algorithm, on the Y-axis, and the maximum number of values in the series or n on the horizontal axis, to represent the worst-case scenario, we have two data points. When n equals 10, Britney took 4 tries using binary search, and when n equals 100, it took 7 tries.

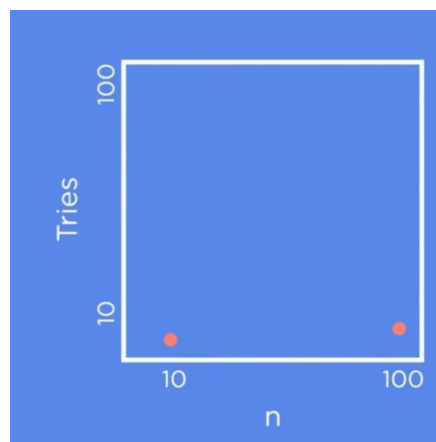


Figure 1.2.1: Graph representing the number of tries and 10 to 100 values of n for a binary search

But even side by side, these data points are sort of meaningless. Remember that while there is quite a difference between the runtime of linear search and binary search at an n value of a 100,

for a computer that shouldn't matter. What we should check out is how the algorithm performs at levels of n that might actually slow a computer down. As n grows larger and larger, how do these algorithms compare to one another? Let's add that to the graph.

Now a picture starts to emerge. As n gets really large, the performances of these two algorithms differ significantly. The difference is kind of staggering actually. Even with the simple game, we saw that binary search was better, but now we have a much more complete idea of how much better.

For example, when n is 1000, the runtime of linear search measured by the number of operations or turns is also 1000.

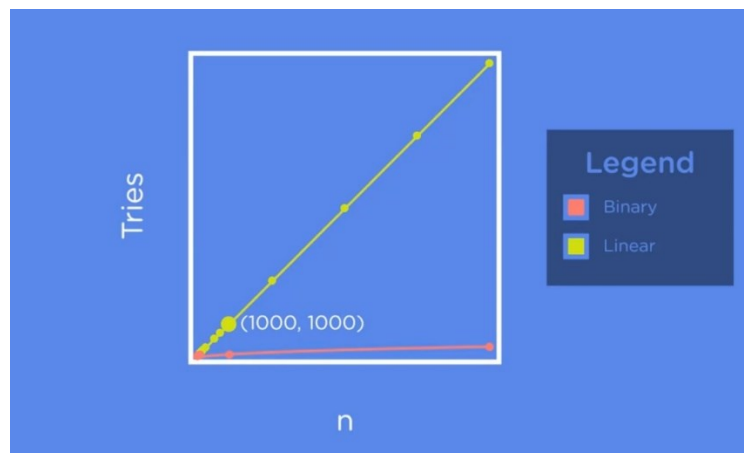


Figure 1.2.2: Graph highlighting the performance of linear search for 1000 tries and 10000 values of n

For binary search, it takes just 10 operations. Now, let's look at what happens when we increase n by factor of 10. See Figure 1.2.3.

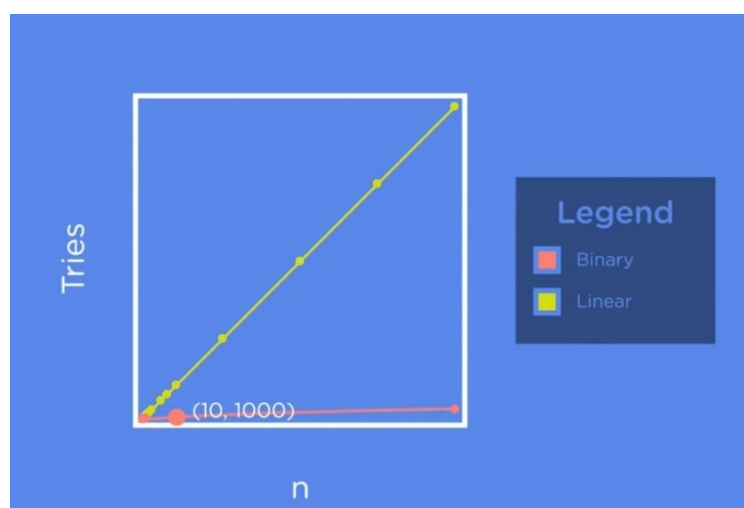


Figure 1.2.3: Graph highlighting the performance of binary search for 1000 tries and 10000 values of n

At 10,000, linear search takes 10,000 operations while binary search takes 14 operations.

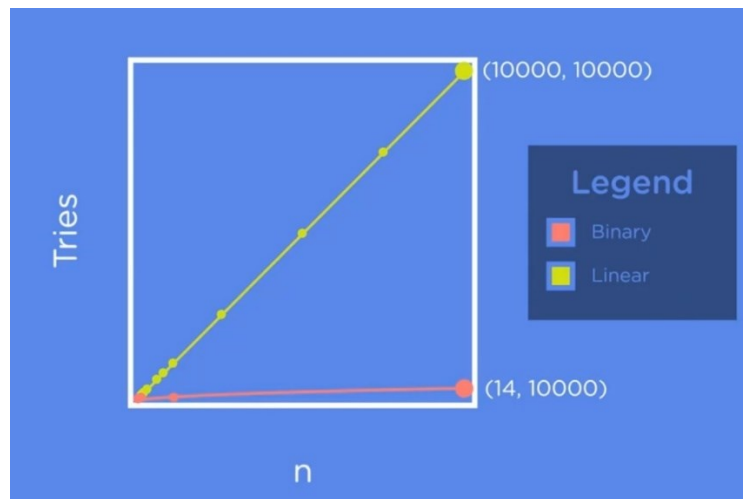


Figure 1.2.4: Graph comparing linear and binary searches for 10,000 tries and 10,000 value of n

As you can see in Figure 1.2.4, an increase by a factor of 10, and binary search only needs 4 more operations to find a value. If we increase it again by factor of 10, once more to an end value of 100,000, binary search takes only 17 operations, it is blazing fast! What we've done here is plotted on a graph how the algorithm performs as the input set it is working on increases. In other words, we've plotted the **growth rate** of the algorithm, also known as the **order of growth**.

Different algorithms grow at different rates, and by evaluating their growth rates, we get a much better picture of their performance because we know how the algorithm will hold up as n grows larger. This is so important. In fact, it is the standard way of evaluating an algorithm and brings us to a concept called **Big O**.

1.2.2. The Big O

You might have heard this word thrown about, and if you found it confusing, don't worry. We've already built up a definition in the past few lessons. We just need to bring it all together. Let's start with a common statement you'll see in studies on algorithms.

Big O is a theoretical definition of the complexity of an algorithm as a function of the size.

Wow, what a mouthful! This sounds really intimidating, but it's really not. Let's break it down. Big O is a notation used to describe complexity. What I mean by notation is that it simplifies everything we've talked about down into a single variable. An example of complexity written in terms of Big O looks like this.

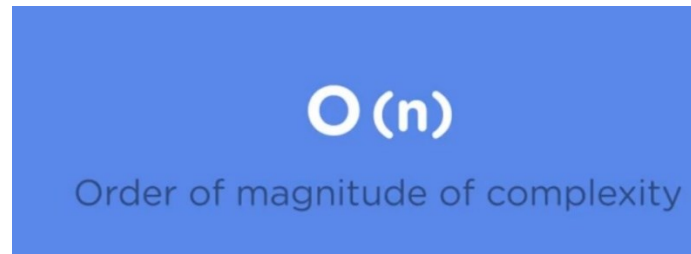


Figure 1.2.5: The Big O: Order of magnitude of complexity of an algorithm

As you can see, it starts with an uppercase letter O. That's why we call it Big O. It's literally a big O. The O comes from order of magnitude of complexity, so that's where we get the big O from. Now, complexity here refers to the exercise we've been carrying out in measuring efficiency. If it takes Britney 4 tries when n is 10, how long does the algorithm take when n is 10 million? When we use big O for this, the variable used, which we'll get to, distills that information down so that by reading the variable you get a big picture view without having to run through data points and graphs just like we did.

It's important to remember that complexity is relative. When we evaluate the complexity of the binary search algorithm, we're doing it relative to other search algorithms, not all algorithms. Big O is a useful notation for understanding both time and space complexity, but only when comparing amongst algorithms that solve the same problem. The last bit (n) in that definition of Big O is a **function of the size**, and all this means is that big O measures complexity as the input size grows because it's not important to understand how an algorithm performs in a single data set, but in all possible data sets.

You'll also see big O referred to as the **Upper Bounds** of the algorithm. What that means is that big O measures how the algorithm performs in the worst-case scenario. That's all Big O is. Nothing special! It's just a notation that condenses the data points and graphs that we've built up down to one variable.

What do these variables look like? For John's strategy, linear search, we say that it has a time complexity of **O(n)**, that is, Big O and then an "n" inside parenthesis. For Britney's strategy binary search, we say that it has a time complexity of **O(log n)**, that is, Big O log n. That's Big O with something called a "log" and an "n" inside parentheses. Now, don't worry if you don't understand that. We'll go into that in more detail later on in the course.

Each of these has a special meaning, but it helps to work through all of them to get a big picture view. So, over the next few lessons, we'll examine what are called common complexities or common values of Big O that you'll run into and should internalize.

1.2.3. Constant and Logarithmic Time

In our discussions of complexity, we made one assumption that the algorithm as a whole had a single measure of complexity. That isn't true, and we'll get at how we arrive at these measures for

the entire algorithm at the end of this exercise. But each step in the algorithm has its own space and time complexity. In linear search, for example, there are multiple steps and the algorithm goes like this:

- Start at the beginning of the list or range of values.
- Compare the current value to the target.
- If the current value is the target value that we're looking for, we're done.
- If it's not, we'll move on sequentially to the next value in the list and repeat step two.
- If we reach the end of the list, then the target value is not in the list.

Let's go back to step two for a second. Comparing the current value to the target, does the size of the dataset matter for this step? When we're at step two, we're already at that position in the list and all we're doing is reading the value to make a comparison. Reading the value is a single operation, and if we were to plot it on a graph of runtime per operations against n , it looks like Figure 1.2.6. A straight line that takes constant time regardless of the size of n .

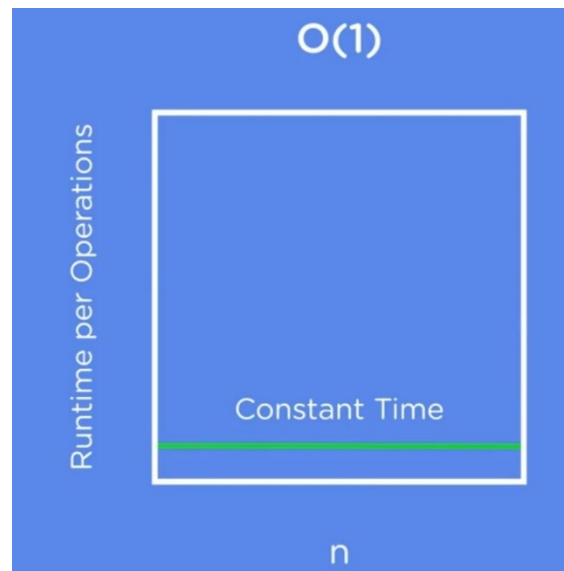


Figure 1.2.6: The graph of runtime per operations against n

Since this takes the same amount of time in any given case, we say that the runtime is constant time. It doesn't change in Big O notation, so we represent this as big O with a one inside parenthesis. Now, when I first started learning all this, I was really confused as to how to read this. Even if it was in my own head, should I say Big O of 1? When you see this written, you are going to read this as constant time, so reading a value in a list is a **constant time operation**. This is the most ideal case when it comes to runtimes because input size does not matter. Also, we know that regardless of the size of n , the algorithm runtime will remain the same.

The next step up in complexity, so to speak, is the situation we encountered with the binary search algorithm. Traditionally, explaining the time complexity of binary search involves math,

and I'm going to try to do it both with and without. When we played the game using **binary search**, we noticed that with every turn we were able to discard half of the data. But there's another pattern that emerges that we didn't explore. Let's say n equals 10. How long does it take to find an item at the 10th position of the list? We can write this out as shown in Figure 1.2.7. So, we go from 10 to 5 to 8 to 9 and then down to 10. Here it takes us **4** tries to cut down the list to just one element and find the value we're looking for.



Figure 1.2.7: Finding how long does it take to find an item at the 10th position of the list in binary search

Let's double the value of n to 20 and see how long it takes for us to find an item at the 20th position.

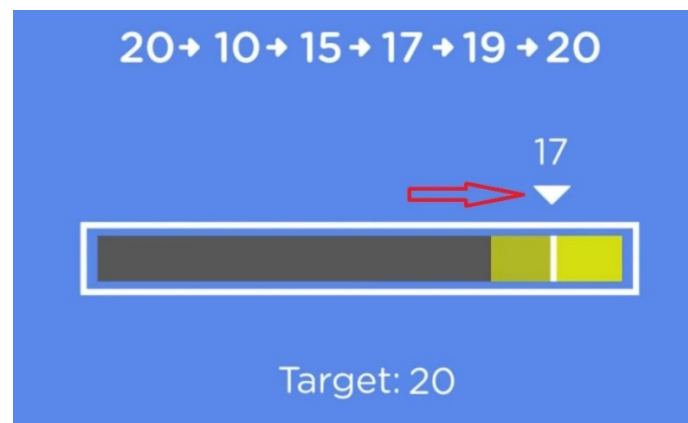


Figure 1.2.8: Finding how long does it take to find an item at the 20th position of the list in binary search

So, we started 20 and then we picked 10. From there we go to 15, 17, 19, and finally 20. So here, it takes us **5** tries. Now, let's double it again so that n is 40 and we try to find the item in the 40th position. Try it. When we start at 40, the first midpoint we're going to pick is 20. From there we go to 30, then 35, 37, 39, and then 40. It takes **6** tries.

Notice that every time we double the value of n , the number of operations it takes to reduce the list down to a single element only increases by **1**. There's a mathematical relationship to this

pattern and it's called a **logarithm of n**. You don't really have to know what algorithms truly are, but if you like underlying explainers, I'll give you a quick one. If you've taken algebra classes, you may have learned about exponent. Here's a quick refresher.

$2 \times 1 = 2$. Now, this can be written as $2^1 = 2$ (two raised to the first power) because it is our base case.

Now, $2 \times 2 = 4$. This can be written as $2^2 = 4$ (two raised to the second power) because we're multiplying 2 twice. First, we multiplied 2 times 1. Then we multiply the result of that by 2.

Now, $2 \times 2 \times 2 = 8$, and we can write this as $2^3 = 8$ (two raised to the third power) because we're multiplying 2 three times. In 2^2 and 2^3 , the 2 and 3 there are called exponents, and they define how the number grows.

With 2^3 , we start with the base value (2) and multiply itself three times. The **inverse** of an exponent is called a logarithm. So, if I say $\log_2 8 = 3$, I'm basically saying the opposite of an exponent. Instead of saying how many times do I have to multiply this value, I'm asking how many times do I have to divide 8 by 2 to get the value 1. This is 3 operations.

What about the result of log to the base 2 of 16? That evaluates to four. That is, $\log_2 16 = 4$.

Why does any of this matter? Notice that this is sort of how binary search works. Log to the base 2 of 16 is 4. If n was 16, how many tries does it take to get to that last element? Well, that is **5** tries, or $(\log_2 16 + 1)$.

In general, for a given value of n, the number of tries it takes to find the worst-case scenario is $\log_2 n + 1$. Because this pattern is overall a logarithmic pattern, we say that the runtime of such algorithms is logarithmic. If we plot these data points on our graph, a logarithmic runtime looks like Figure 1.2.9.

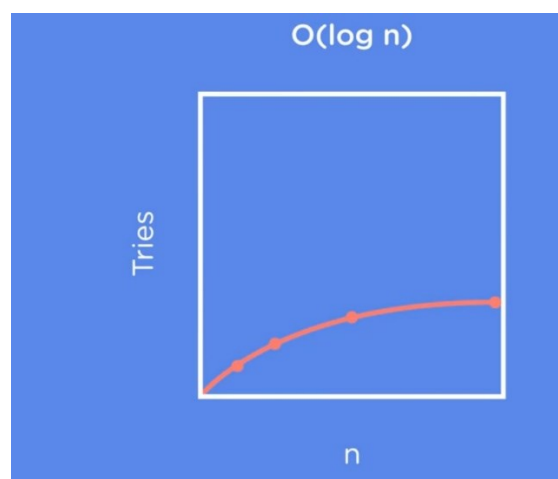


Figure 1.2.9: Graph of a logarithmic runtime

In big O notation, we represent a logarithmic runtime as **$O(\log n)$** , which is written as big O with log n inside parenthesis, or even sometimes as **$O(\ln n)$** . When you see this, read it as **logarithmic time**. As you can see on the graph, as n grows really large, the number of operations grows very slowly and eventually flattens out. Since this line is below the line for a linear runtime, which you can see in Figure 1.2.10, you might often hear algorithms with logarithmic runtimes being called **sublinear**.

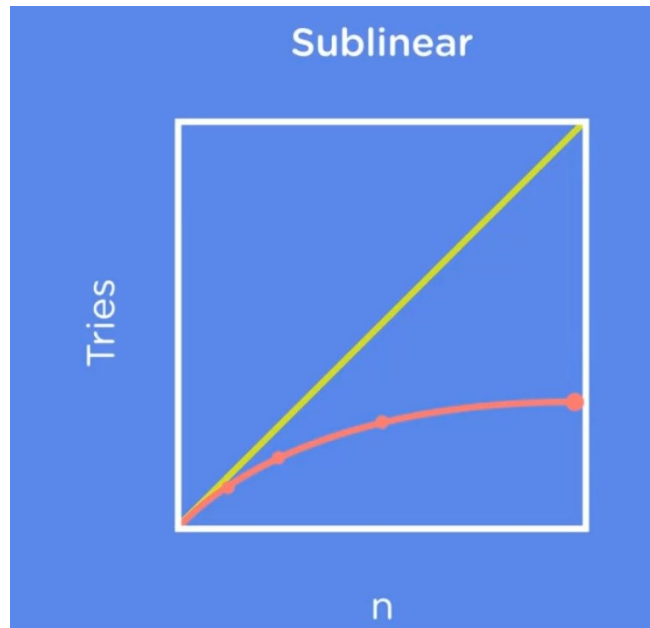


Figure 1.2.10: Linear (yellow) and Sublinear/Logarithmic (orange) runtimes

Logarithmic or subline runtimes are preferred to linear because they're more efficient. But in practice, linear search has its own set of advantages, which we'll take a look at in the next section.

Let's look at the situation we encountered with the linear search algorithm. We saw that in the worst-case scenario, whatever the value of n was, John took exactly that many tries to find the answer. As in linear search, when the number of operations to determine the result in the worst-case scenario is at most the same as n, we say that the algorithm runs in linear time. We represent this as $O(n)$. You can read that as Big O of n, or you can say linear time, which is more common. When we put that up on a graph against constant time and logarithmic time, we get a line that looks like Figure 1.2.11.

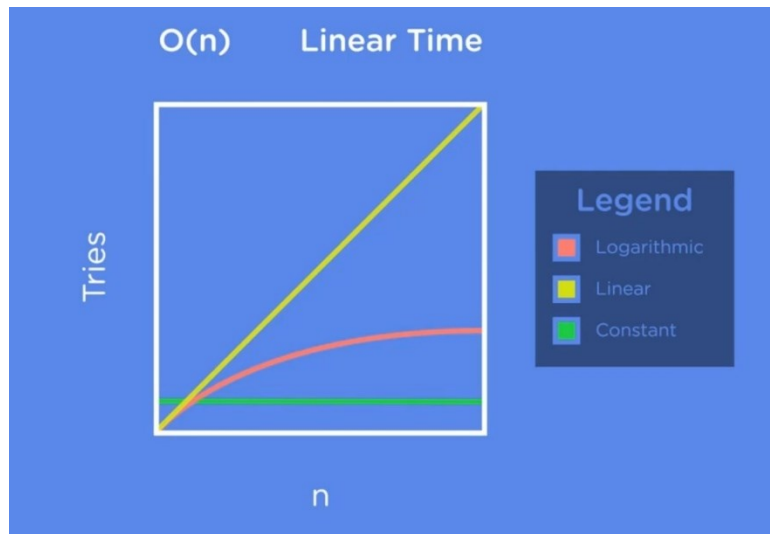


Figure 1.2.11: Graph of Linear (yellow), Sublinear/Logarithmic (orange) and Constant runtimes

Any algorithm that sequentially reads the input will have linear time. So, remember, anytime you know a problem involves reading every item in a list, that means a linear runtime. As you saw from the game we played, Britney's strategy using binary search was clearly better, and we can see that on the graph. So, if we had the option, why would we use linear search which runs in linear time?

Remember that binary search had a precondition, **the input set had to be sorted**. While we won't be looking at sorting algorithms in this course, as you learn more about algorithms, you'll find that sorting algorithms have varying complexities themselves, just like search does. So, we have to do additional work prior to using binary search. For this reason, in practice, linear search ends up being more performant up to a certain value of n because the combination of sorting first and then searching using binary search adds up.

1.2.4. Linear & Quadratic Time

The next common complexity you'll hear about is when an algorithm runs in quadratic time. If the word quadratic sounds familiar to you, it's because you might have heard about it in math class. Quadratic is a word that means an operation raised to the second power (such as x^2), or when something is squared.

Let's say you and your friends are playing a tower defense game and to start it off, you're going to draw a map of the terrain. This map is going to be a grid and you pick a random number to determine how large this grid is. Let's set n , the size of the grid to 4.

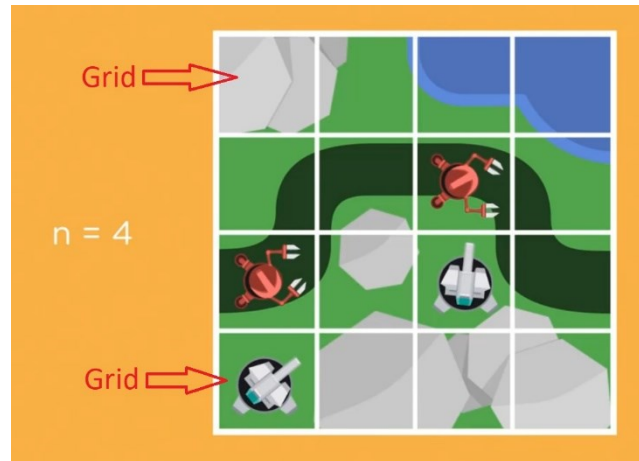


Figure 1.2.12: Map of a terrain with 4 grids in a tower defense game

Next, you need to come up with a list of coordinates so you can place towers, enemies and other stuff on this map. So how would we do this? If we start at the bottom and move horizontally, we'd have coordinate points that go (1, 1), (1, 2), (1, 3) and (1, 4). Then you go up one level vertically and we have points (2, 1), (2, 2), (2, 3) and (2, 4). Go up one more and you have the points (3, 1), (3, 2), (3, 3) and (3, 4). On the last (top) row you have the points (4, 1), (4, 2), (4, 3) and (4, 4). Notice that we have a pattern as you can see in Figure 1.2.13.

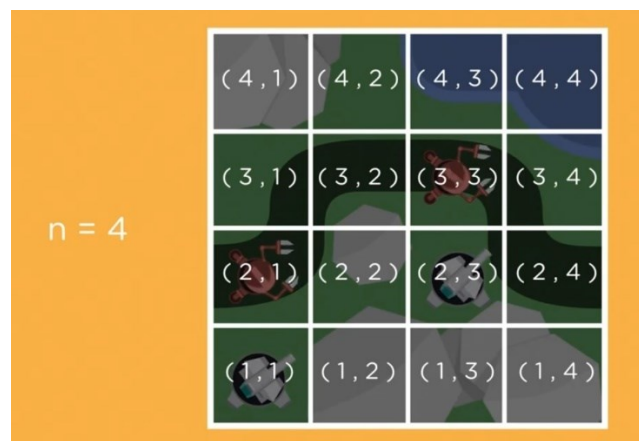


Figure 1.2.13: Map of the terrain with 4 grids and coordinates

For each row we take the value and then create a point by adding to that every column value. The range of values go from one to the value of n . We can generally think of it this way: for the range of values from one to n , for each value in that range, we create a point by combining that value with the range of values from one to n .

Again, doing it this way for each value in the range of one to n , we create an n number of values and we end up with 16 points, which is also n times n or n^2 . This is an algorithm with a **quadratic runtime** because for any given value of n , we carry out n^2 number of operations.

Now I pick a relatively easy, so to speak, example here because in English at least we often denote map sizes by height times width (h x w), so we would call this a 4 by 4 grid, which is just another way of saying 4^2 or n^2 . See Figure 1.2.14.

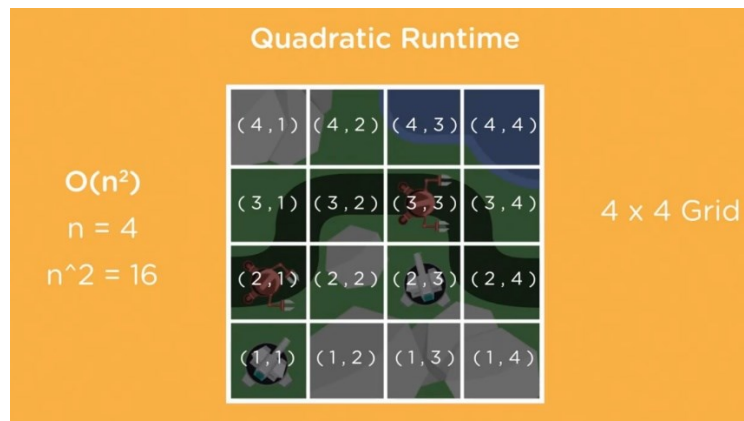


Figure 1.2.14: This map illustrates an algorithm with a quadratic time

In Big O notation, we would write this as $O(n^2)$ or say that this is an **algorithm with a quadratic runtime**. Many search algorithms have a worse case quadratic runtime, which you'll learn about soon.

1.2.5. Cubic Runtime

Now, in addition to quadratic runtimes, you may also run into cubic runtimes as you encounter different algorithms. In such an algorithm, for given value of n , the algorithm executes n^3 number of operations. Cubic runtimes are not as common as quadratic algorithms though, so we won't look at any examples. But I think it's worth mentioning. Thrown up on our graph, quadratic and cubic runtimes look like Figure 1.2.15.

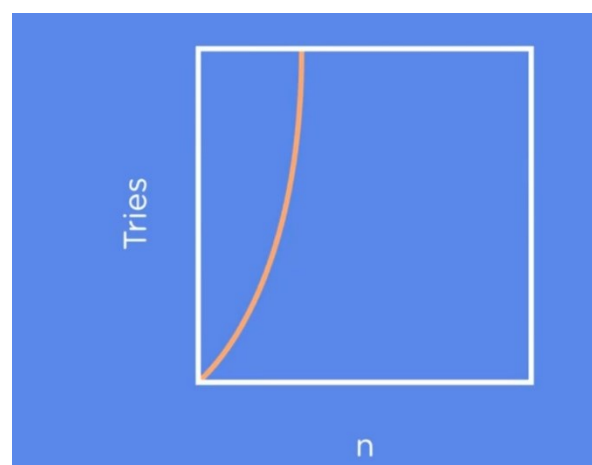


Figure 1.2.15: Graph of quadratic and cubic runtimes

This is starting to look pretty expensive computationally, as they say. We can see here that for

small changes in 2, there's a pretty significant change in the number of operations that we need to carry out.

1.2.6. Quasilinear Runtime

The next worst-case runtime we're going to look at is one that is called quasi linear. It's sort of easier to understand. I'll start with the Big O notation. Quasilinear runtimes are written out as **$O(n \log n)$** . We learned what $\log n$ was right? A logarithmic runtime, where as n grows, the number of operations only increased by a small factor. With a quasilinear runtime, what we're saying is that for every value of n , we're going to execute a $\log n$ number of operations, hence the runtime of n times $\log n$.

Furthermore, you saw earlier with the quadratic runtime that for each value of n , we conducted n operations. It's sort of the same in that as we go through the range of values in n , we're executing $\log n$ operations. In comparison to other runtimes, a quasilinear algorithm has a runtime that lies somewhere between a linear runtime and a quadratic runtime. See Figure 1.2.16.

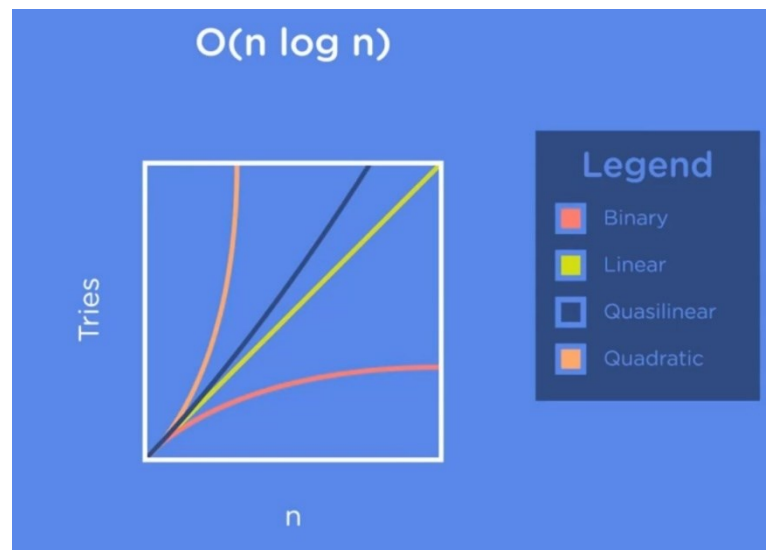


Figure 1.2.16: Graphical comparison of binary, linear, quasilinear and quadratic runtimes

Where would we expect to see quasilinear runtime in practical use? Well, sorting algorithms is one place you'll definitely see it. Merge Sort for example is a sorting algorithm that has a worst-case runtime of $O(n \log n)$. Let's take a look at a quick example. Let's say we start off with a list of numbers that looks like Figure 1.2.17 and we need to sort it.

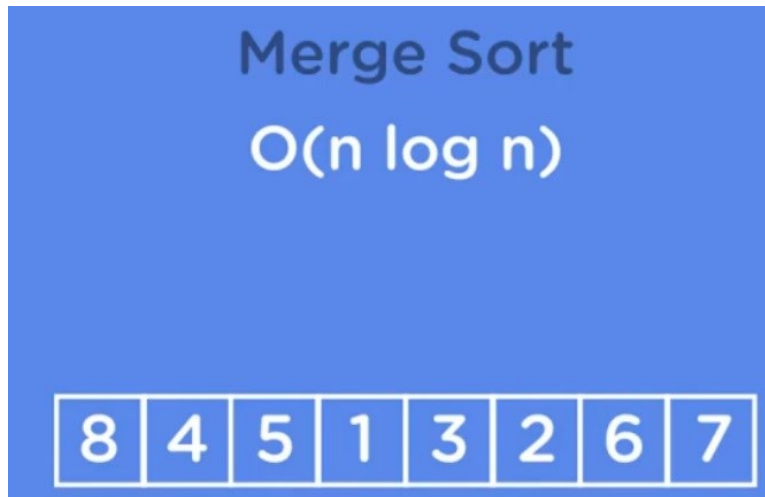


Figure 1.2.17: A list of eight numbers to be sorted using Merge Sort algorithm

Merge Sort starts by splitting this list into two lists down the middle. Figure 1.2.18.

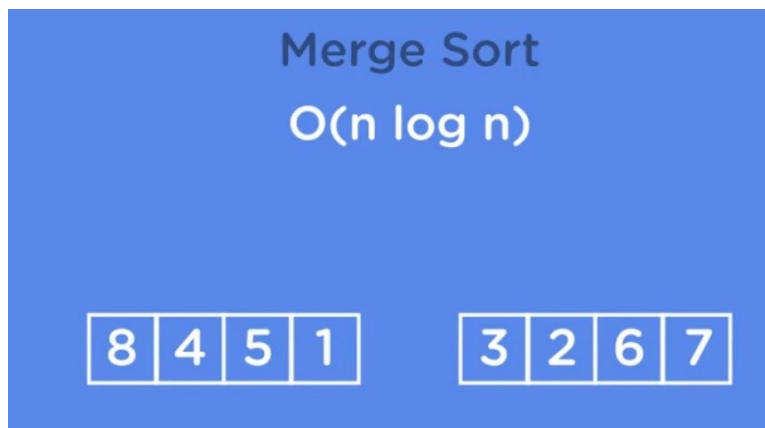


Figure 1.2.18: Merge Sort splits the list down the middle into two lists

It then takes each sublist and splits that in half down the middle.



Figure 1.2.19: Merge Sort splits each sublist in half down the middle again

Again, it keeps doing this until we end up with a list of just single numbers.

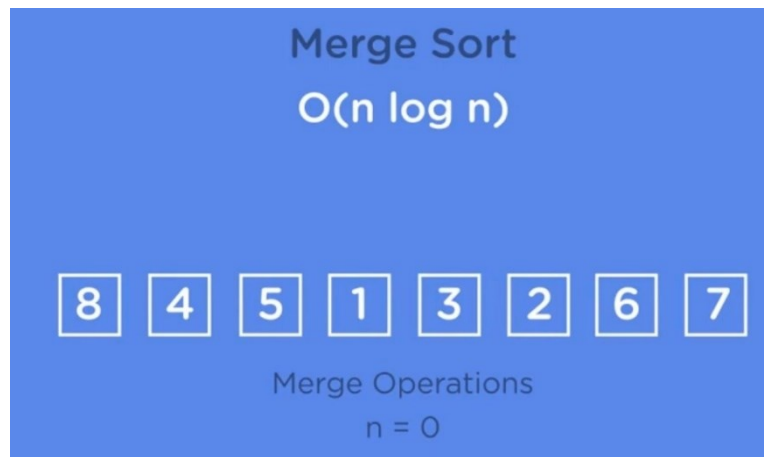


Figure 1.2.20: Merge Sort keeps splitting each sublist until we end up with a list of just single numbers

When we're down to single numbers, we can do one sort operation and merge the sublists back in the opposite direction.



Figure 1.2.21: Merge Sort carries out one sort operation by merging the sublists back in the opposite direction ($n=1$)

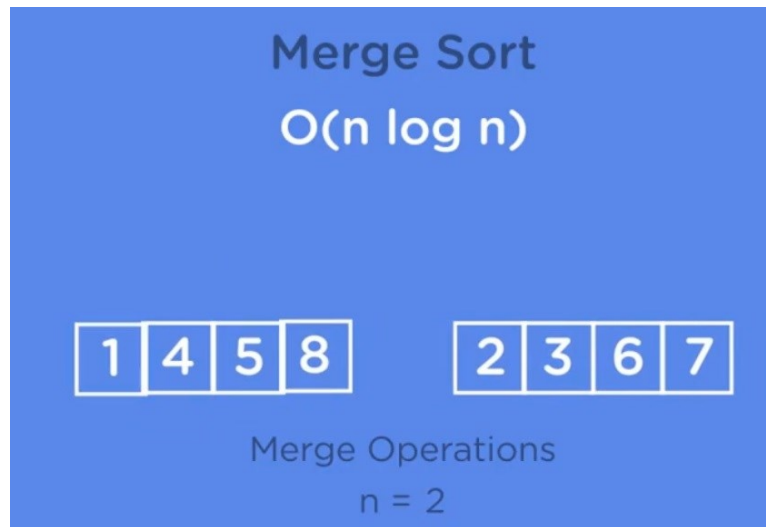


Figure 1.2.22: Merge Sort carries out one more sort operation ($n=2$)



Figure 1.2.23: Merge Sort carries out yet one more sort operation ($n=3$) and sorting is complete

The first part of Merge Sort cuts those lists into sublists with half the numbers. This is similar to binary search where each comparison operation cuts down the range to half the values. You know the worst-case runtime in binary search is $\log n$. So, the splitting operations have the same runtime, $O(\log n)$ or logarithmic. But splitting into half is not the only thing we need to do with Merge Sort. We also need to carry out **comparison operations** so we can sort those values. If you look at each step of this algorithm, we carry out an n number of comparison operations, and that brings the worst-case runtime of this algorithm to $O(n \log n)$, also known as quasilinear.

1.2.7. Polynomial Runtimes

Don't worry if you didn't understand how Merge Sort works. That wasn't the point of the demonstration. We will be covering Merge Sort later. The runtimes we've looked at so far are all called **polynomial runtimes**.

An algorithm is considered to have a polynomial runtime if for a given value of n , its worst-case runtime is in the form of $O(n^k)$, where k just means some value. So, it could be n^2 where $k = 2$ for a quadratic runtime, and n^3 where $k = 3$ for a cubic runtime, and so on. All of those are in the form of n raised to some power. Anything that is bounded by this, that is, anything that falls under the graph in Figure 1.2.24 is considered to have a polynomial runtime. Algorithms with an upper bound or a runtime with a big O value that is polynomial are considered efficient algorithms, and are likely to be used in practice.

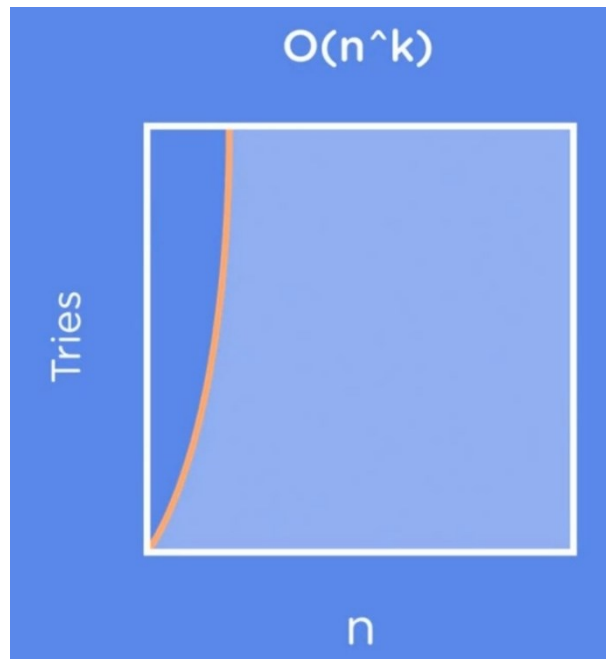


Figure 1.2.24: Graph of a polynomial runtimes

1.2.8. Exponential Runtimes

Now, the next class of runtimes that we're going to look at are runtimes that we don't consider efficient. These are called **exponential runtimes**. With these runtimes, as n increases slightly, the number of operations increases exponentially. As we'll see in a second, these algorithms are far too expensive to be used. An exponential runtime is an algorithm with a big O value of some number x raised to the n th power, $O(x^n)$.

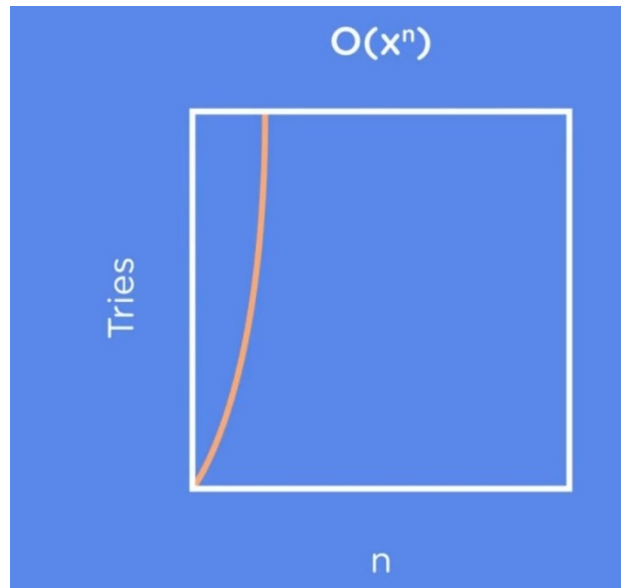


Figure 1.2.25: Graph of a exponential runtimes

Imagine that you wanted to break into a locker that had a padlock on it. Let's assume you forgot your code. This lock takes a two-digit code and the digit for the code ranges from **0** to **9**. You start by setting the dials to **00**. With the first dial remaining on 0, you change the second dial to 1 and try and open it. If it doesn't work, you set it to 2, then try again. You would keep doing this and if you still haven't succeeded with the second dial set to 9, then you go back to that first dial, set it to 1 and start the second dial over. The range of values you'd have to go through is **00** to **99**, which is 100 values.

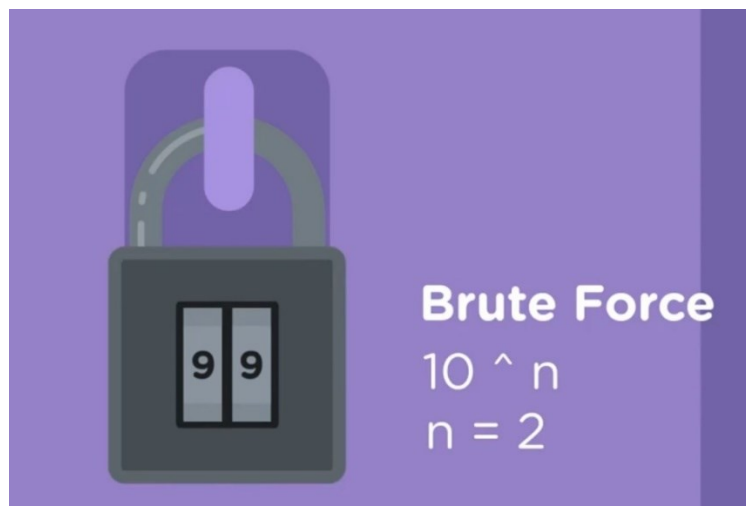


Figure 1.2.26: A digital locker with two digits

This can be generalized as 10^2 , since there are 10 values on each dial raised to two dials. Searching through each individual value until you stumble on the right one is a strategy called **brute force**, and **brute force algorithms have exponential runtimes**. Here, there are two dials,

so $n = 2$, and each dial has 10 values. So again, we can generalize this algorithm as 10^n where n represents the number of dials. The reason that this algorithm is so inefficient is because with just one more dial on the lock, the number of operations increases significantly. With three dials, the number of combinations in the worst-case scenario where the correct code is the last digit in the range, is 10^3 or 1000 values. With an additional wheel it becomes 10^4 or 10,000 values. As n increases, the number of operations increases exponentially to a point where it's unsolvable in a realistic amount of time.

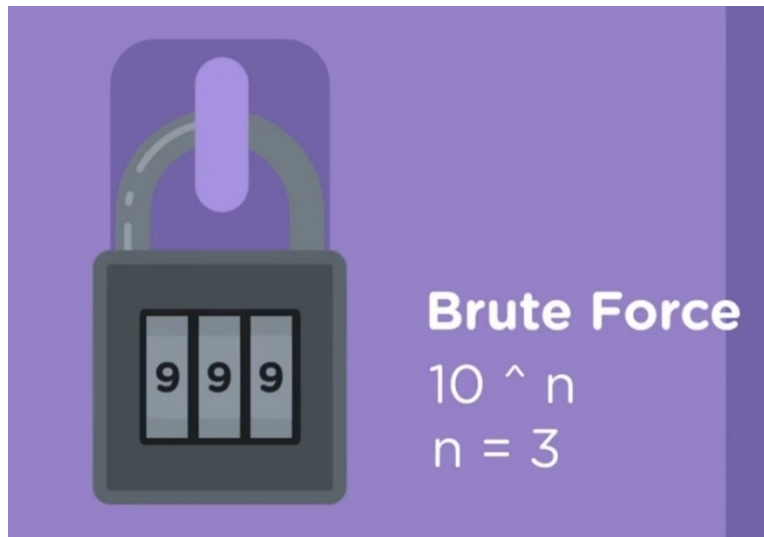


Figure 1.2.27: A digital locker with three digits

Now you might think, well any computer can crack a four-digit numerical lock and that's true because n here is sufficiently small. But this is the **same principle that we use for passwords**. In a typical password field, implemented well, users are allowed to use letters of the English alphabet up to 26, numbers from 0 to 9, and a set of special characters of which there can be around 33. So, $26 + 10 + 33 = 69$ characters. An example is shown in Figure 1.2.28.

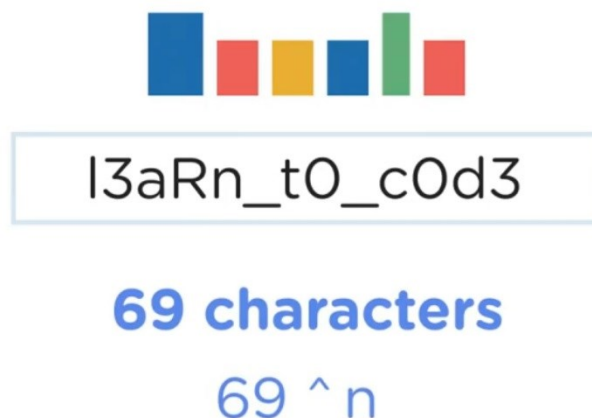


Figure 1.2.28: A website password field that can take up to 69^n

So, typically that means each character in a password can be 1 out of 69 values. This means that for a one-character password, it takes 69 to the nth power (69^n). So, for $n = 1$ we have 69 operations. In the worst-case scenario, to figure out the password. Just increasing n to 2 increases the number of operations needed to guess the password to 69^2 or 4,761 operations.

Now usually on a secure website there isn't really a limit, but in general, passwords are limited to around 20 characters in length. With each character being a possible 69 values, and there being 20 characters, the number of operations needed to guess the password in the worst-case scenario is 69 raised to the 20th power (69^{20}) or approximately 6×10^{36} of operations. An Intel CPU with 5 cores can carry out roughly about 65,000 million instructions per second. That's a funny number. To crack our 20 digit passcode in this very simplistic model, it would take this intel CPU 2×10^{20} years to brute force the password.

So, while this algorithm would eventually produce a result, it is so inefficient that it is pointless. This is one of the reasons why people recommend you have longer passwords, since brute forcing is exponential in the worst case, each character you add increases the number of combinations by an exponent.

The next class of exponential algorithms is best highlighted by a popular problem known as the **traveling salesman**. The problem statement goes like this. Given a list of cities and the distance between each pair of cities, what is the shortest possible route that visits each city and then returns to the origin city? This seems like a simple question, but let's start with a simple case, three cities, A, B, and C. To figure out what the shortest route is, we need to come up with all the possible routes. With three cities, we have six routes.

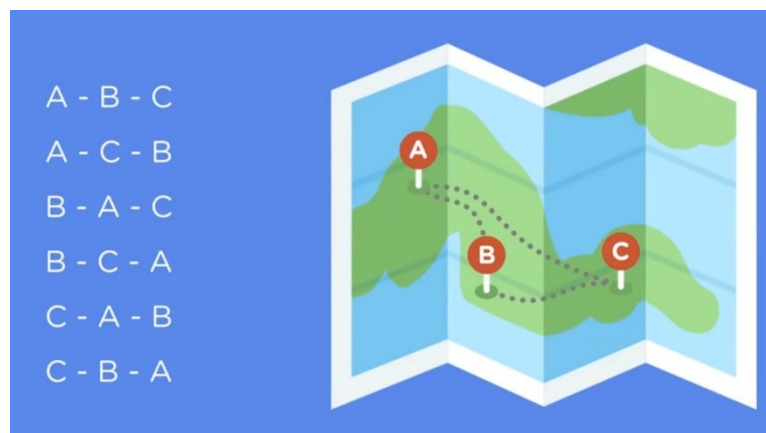


Figure 1.2.29: Calculating the shortest possible route for each of three cities A, B, C

In theory at least some of these routes can be discarded. For example, A - B - C is the same as C - B - A, but in the opposite direction. So, we can discard the last three routes in Figure 1.2.29, including C - A - B, and B - C - A. But as we do know, sometimes going from A to C through B may go through a different route than C to A through B, so we'll stick to the six routes and from there we could determine the shortest. No big deal.

Now if we increase this to 4 cities, we jump to 24 combinations.

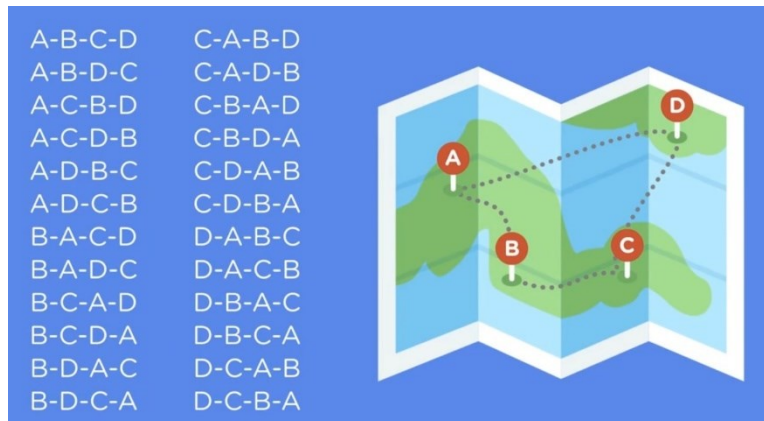


Figure 1.2.30: Calculating the shortest possible route for each of four cities A, B, C, D

The mathematical relationship that defines this is called a **factorial** and is written out as n followed by an exclamation point ($n!$). Factorials are basically obtained as follows:

$$n! = n(n-1)(n-2)(n-3)\dots(2)(1).$$

That is, n times n minus 1 times n minus 2 and so on, until you reach the number 1. For example, the factorial of 3 is

$$3! = 3 \times 2 \times 1 = 6$$

which is the number of combinations we came up with for 3 cities. The factorial of 4 is

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

which is the number of combinations we arrived at with 4 cities. In solving the traveling salesman problem, the most efficient algorithm will have a **factorial runtime** or a **combinatorial runtime** as it's also called. At low values of n , algorithms with a factorial runtime may be used. But with an n value of say 200 (which is $200!$), it would take longer than humans have been alive to solve the problem!

For sake of completeness, let's plot a combinatorial runtime on our graph so that we can compare. An algorithm such as one that solves the traveling salesman problem has a worst case runtime of $O(n!)$.

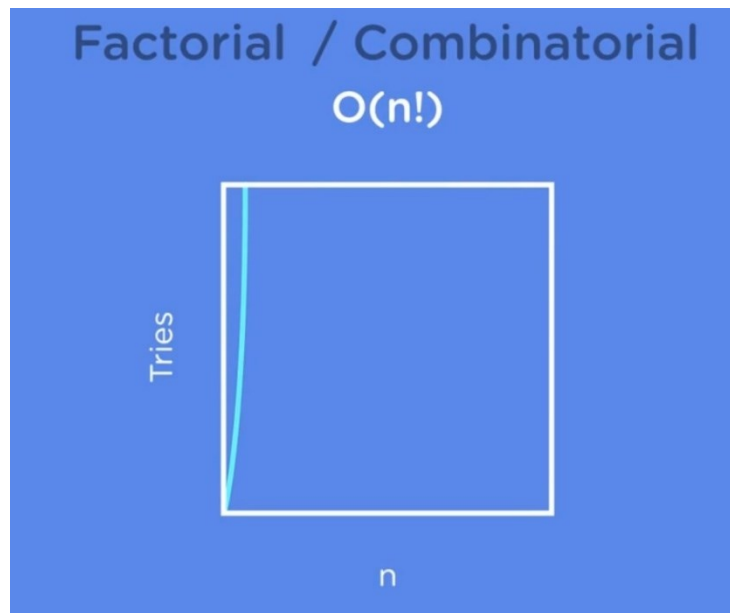


Figure 1.2.31: Graph of a combinatorial runtime

Studying exponential runtimes like this are useful for two reasons. First, in studying how to make such algorithms efficient, we develop strategies that are useful across the board and can potentially be used to make existing algorithms even more efficient. Second, it's important to be aware of problems that take a long time to solve.

Knowing right off the bat that a problem is somewhat unsolvable in a realistic time means you can focus your efforts on other aspects of the problem. As a beginner though, you're going to steer clear of all this and focus your efforts on algorithms with polynomial runtimes since we're much more likely to work with and learn about such algorithms.

Now that we know some of the common complexities, in the next section, we'll talk about how we determined the complexity of an algorithm because there are some nuances.

1.2.9. How to Determine the Complexity of an Algorithm

Over the last few sections, we took a look at common complexities that we would encounter in studying algorithms. But the question remains, how do we determine what the worst-case complexity of an algorithm is? Earlier, I mentioned that even though we say that an algorithm has a particular upper bound or worst-case runtime, each step in a given algorithm can have different runtimes.

Let's bring up the steps for binary search. Again, assuming the list is sorted, the first step is to determine the middle position of the list. In general, this is going to be a **constant time operation**. Many programming languages hold on to information about the size of the list, so we don't actually need to walk through the list to determine the size. Now if we didn't have information about the size of the list, we would need to walk through counting each item one by

one until we reach the end of the list, and this is a **linear time operation**. Realistically this is a $O(1)$ or constant time.

Step two is to compare the element in the middle position to the target element. We can assume that in most modern programming languages this is also a constant time operation because the documentation for the language tells us it is.

Step three is our success case and the algorithm ends. This is our best case and so far, we have only incurred two constant time operations, so we would say that the **best-case runtime of binary search is constant time**, which is actually true. But remember that best case is not a useful metric, except for if we don't match is splitting the list into sublists.

Assuming the worst-case scenario, the algorithm would keep splitting into sublists until a single element list is reached with the value that we're searching for. The runtime for this step is **logarithmic** since we discard half the values each time. So, in our algorithm, we have a couple steps that are constant time and one step that is logarithmic overall.

When evaluating the runtime for an algorithm, we say that the algorithm has as its upper bound, the same runtime as the least efficient step in the algorithm. Think of it this way. Let's say you're participating in a triathlon which is a race that has a swimming, running and a cycling component. You could be a phenomenal swimmer and a really good cyclist, but you're a pretty terrible runner. No matter how fast you are at swimming or cycling, your overall race time is going to be impacted the most by your running race time because that's the part that takes you the longest.

If you take 1 hour 30 minutes to finish the running component, 55 minutes to swim, and 38 minutes to bike, it won't matter if you can fine tune your swimming technique down to finish in 48 minutes and your cycle time to 35 because you're still bounded at the top by your running time, which is close to almost double your bike time.

Similarly with the binary search algorithm, it doesn't matter how fast we make the other steps, they're already as fast as they can be in the worst-case scenario. The splitting of the list down to a single element list is what will impact the overall running time of your algorithm. This is why we say that the time complexity or runtime of the algorithm in the worst case is $O(\log n)$ or logarithmic. As I alluded to though, your algorithm may hit a best-case runtime and in between the two, best and worst case, have an average runtime as well. This is important to understand because algorithms don't always hit their worst case. But this is getting a bit too complex for us.

For now, we can safely ignore average case performances and focus only on the worst case in the future. If you decide to stick around, we'll circle back and talk about this more. Now that you know about algorithms, complexities and Big O, take some practice exercise in the next section. Then you can take a break from all of that and write code in the section after.

1.2.10. Practice Exercise 3

Q1. I started following a workout plan recently and one of the routines goes like this. I have to do a certain number of pull-ups and for each pull-up that I complete, I have to complete an equivalent number of pushups. For example, if I want to do 10 pull-ups, after each pull-up, I do 10 pushups. How can you generalize the running time of such a routine? Choose the correct answer below.

- A. Quadratic time because the total number of exercises completed for any given routine is n times n .
- B. Constant time. I'm superhuman and I can complete any number of exercises in equal amounts of time.
- C. Linear time because the total number of exercises increases proportionally to the size of n .

Q2. My son throws toys all over the floor and I clean them up by picking up one at a time. How would you generalize the runtime of my "cleaning" algorithm? Choose the correct answer below.

- A. $O(n^2)$ or quadratic time
- B. $O(n)$ or linear time
- C. $O(1)$ or constant time

Q3. You have a box of Lego bricks of various colors that you want to organize into one pile ordered by color. You work with a friend and split the bricks into the smallest piles possible, organize each pile and combine piles as they are organized. How would you generally define the runtime of this algorithm as?

- A. Linear or $O(n)$
- B. Quasilinear or $O(n \log n)$
- C. Quadratic or $O(n^2)$

Q4. Every step in an algorithm has a cost or time complexity of its own. Choose the correct answer below.

- A. True
- B. False

Q5. If an algorithm has a runtime of $O(1)$ how much longer will it take when $n = 10000$ compared to when $n = 10$.

- A. Time to completion increases by a factor of 1000.
- B. Time to completion increases by a factor of $\log 1000$.
- C. The same amount of time.

1.2.11. Answers to Practice Exercise 3

1. A.

2. B.

3. B.

4. A.

5. C.

1.3. Algorithms in Code

So far, we've only been looking at algorithms in the abstract but you're here to learn how to implement them as well! Over the next few sections, we're going to write code to implement both linear and binary search.

1.3.1. Linear Search in Code

So far, we've spent a lot of time in theory and while these things are all important things to know, you get a much better understanding of how algorithms work when you start writing some code. As I mentioned in the introductory chapter, we're going to be writing Python code in this and all subsequent algorithm courses. If you do have programming experience but in another language, check out the following book that completely simplifies Python programming language for beginners, including all necessary setup and installation:

<https://www.scribd.com/book/513773394/Python-Programming-from-Beginner-to-Paid-Professional-Part-1-Learn-Python-for-Automation-IT-with-Video-Tutorials> (or use this BIT.LY short link: <http://bit.ly/3ZKREhB>).

If you don't have any experience writing code, I'll try my best to explain as we go along in this section. Try to download Python (www.python.org/downloads) and have it installed on your system first. You're also going to need a coding environment called IDE, such as Pycharm (www.jetbrains.com/pycharm/download), Visual studio, or even an ordinary notepad (Windows) or TextEdit (MacBook). This is where we will open a workspace to write all of our code in. If you're familiar with using Python in your local environment, then feel free to keep doing so. Otherwise, there's a bunch of free videos available on Youtube that you can follow to set up Python and Pycharm on your system.

If you don't want to install anything, you can instead write your code online at <https://www.programiz.com/python-programming/online-compiler/>. Once you're ready, just follow along on writing and evaluating with me.

Pycharm workspaces is quite straightforward to use. On the left of Figure 1.3.1 we have a file navigator pane, which is currently empty since we haven't created a new file.

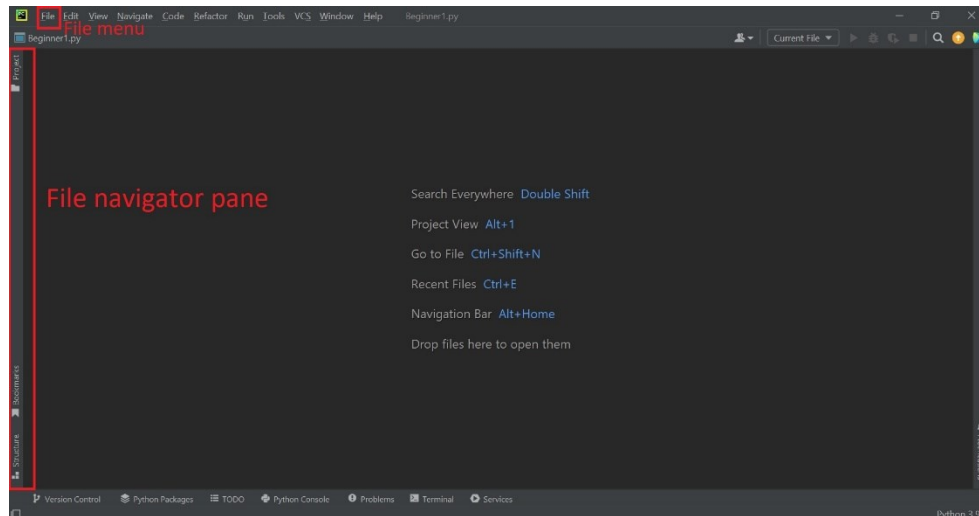


Figure 1.3.1: Pycharm IDE window showing the navigator pane and file menu

Well, let's open a new project. At the top left, follow this path:

File (1) > New Project ... (2). See Figure 1.3.2. A window pops up.

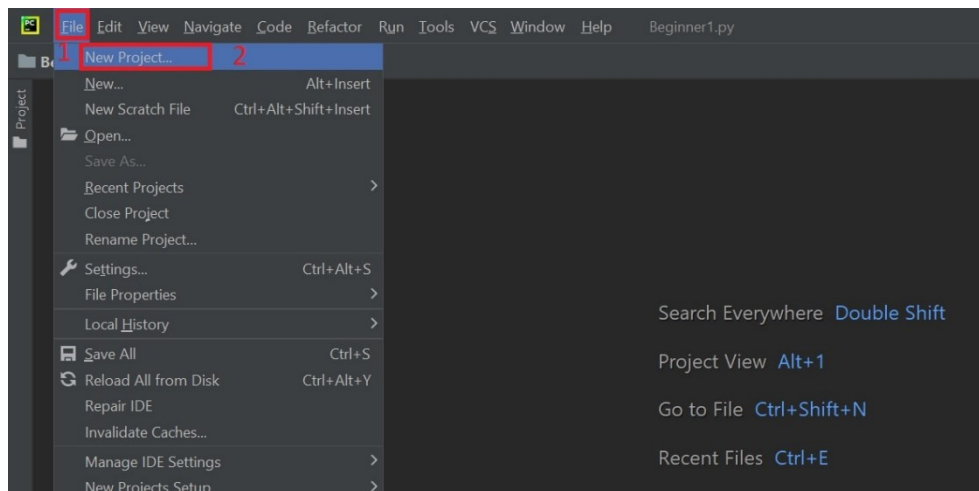


Figure 1.3.2: Pycharm IDE window showing how to create a new project

From this window, create a new folder with a title of Algorithm, or select an existing folder from any location you like on your computer (such as Desktop) where you want to save your new file. Then click **Create** at the bottom right of the window. Now, create a new file in this folder and name it, *linear_search.py*. “py” is the extension for a python file. See Figure 1.3.2.

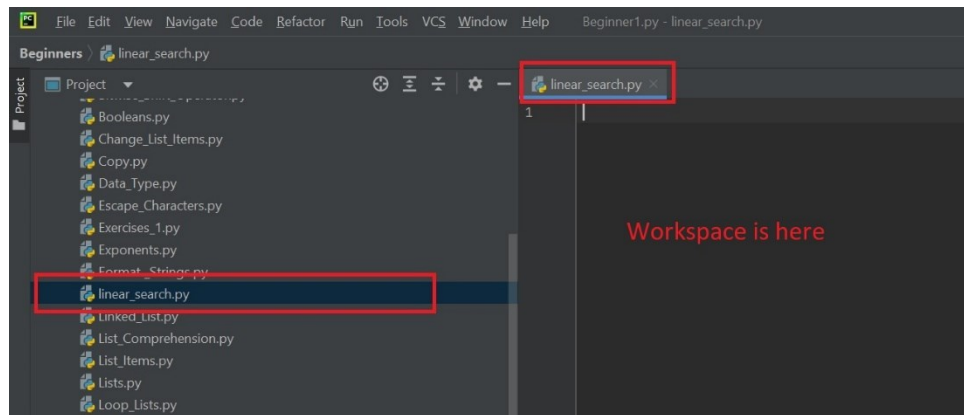


Figure 1.3.3: A new file created in Pycharm showing our workspace is ready for coding

In the workspace, we're going to define our linear search algorithm as a standalone function. We start with the keyword `def`, which defines a function or a block of code, and then we give it the name `linear_search`. This function will accept two arguments. First, the list we're searching through and then the target value we're looking for. Both of these arguments are enclosed in a set of parentheses, and there's no space between the name of the function and the arguments. After that we have a colon. Here's the code so far.

```
def linear_search(list, target):
```

Now, there might be a bit of confusion here since we already have this target value, what are we searching for? Like the game we played at the beginning where John's job was to find the value in a true implementation of linear search, we're looking for the position in the list where the value exists. If the target is in the list, then we return its position and since this is a list, that position is going to be denoted by an index value.

Now if the target is not found, we're going to return `none`. The choice of what to return in the failure case may be different in other implementations of linear search, you can return `-1` since that isn't typically an index value. You can also raise an exception, which is Python's speak for indicating an error occurred.

Now I think for us the most straightforward value we can return here is `none`. Let's add a comment to clarify this. So hit enter to go to the next line and then we're going to add three single quotes. Then below that on the next line, we'll say, *Returns the index position of the target if found, else returns None*. And then on the next line we'll close off those three quotes. This is called a *docstring* and is a Python convention for documenting your code. Here's the updated code:

```
def linear_search(list, target):  
    """  
    Returns the index position of the target if found, else returns None  
    """
```

To save your code, go to the main menu at the top of Pycharm window and follow this path:

File > Save All or use the keyboard shortcut *Ctrl + S*.

The linear search algorithm is a sequential algorithm that compares each item in the list until the target is found. To iterate or loop (or walk) through our list sequentially, we can use a *for* loop. Now, typically when iterating over a list in Python, we would use a loop like this: *for item in list*.

This assigns the value at each index position to that local variable *item*. We don't want this though since we primarily care about the index position. Instead, we're going to use the *range* function in Python to create a range of values that start at zero and end at the number of items in the list. So, we'll say,

```
for i in range(0, len(list)):
```

We start at 0 and go all the way up to the length of the list. Now going back to our talk on complexity and how individual steps in an algorithm can have its own runtimes, this is a line of code that we would have to be careful about. Python keeps track of the length of a list. So, this function called *len(list)*, is a **constant time** operation.

Now if this were a naive implementation, for example let's say we wrote the implementation of the list and we iterate over the list, every time we call this length function, then we've already incurred a linear cost. So, once we have a range of values that represent index positions in this list, we're going to iterate over that using the *for* loop and assign each index value to this local variable *i*. Using this index value, we can obtain the item at that position using subscript notation on the list.

Now, this is also a constant time operation because the language says so. So, we'll write

```
if list[i]== target:
    return i
```

Once we have this value, we'll check if it matches the target. So, if the value at *i* equals target, then we'll return that index value because we want the position. Once we hit this *return* statement, we're going to terminate our function.

If the entire *for* loop is executed and we don't hit this return statement, then the target does not exist in the list. So, at the bottom of our code we'll say *return None*. Here's the updated code.

```
def linear_search(list, target):
    """
    Returns the index position of the target if found, else returns None
    """
    for i in range(0, len(list)):
        if list[i]== target:
```

```
        return i
    return None
```

Even though all the individual operations in our algorithm run in constant time, in the worst-case scenario, the *for* loop above will have to go through the entire range of values and read every single element in the list before giving the algorithm a Big O value of n or running in linear time.

Now, if you've written code before, you've definitely written code like this a number of times and I bet you didn't know that all along you were implementing what is essentially a well-known algorithm. So, I hope this goes to show you that algorithms are a pretty approachable topic.

Like everything else, this does get advanced, but as long as you take things slow, there's no reason for it to be impossible. Remember that not any block of code counts as an algorithm. To be a proper implementation of linear search, this block of code must return a value, must complete execution in a finite amount of time, and must output the same result every time for a given input set.

Now, let's verify this code with a small test. Let's write a function called “*verify*” that accepts an index value. If the value is not *None*, it prints the index position. If it is *None*, it informs us that the target was not found in the list. So, we add the following to our existing code:

```
def verify(index):
    if index is not None:
        print("Target found at index: ", index)
    else:
        print("Target not found in list")
```

There we go. Now, using this function, let's define a range of numbers which will be a list of numbers. We'll just go from 1 to 10.

```
numbers = [1,2,3,4,5,6,7,8,9,10]
```

Now if you've written Python code before, you know that I can use a **list comprehension** to make this list of numbers easier, but we'll keep things simple. We can now use our linear search function to search for the position of a target value in this list. So, we can say

```
result = linear_search(numbers,12)
verify(result)
```

Notice that we passed in the *numbers* list. That's the one we're searching through, and we want to look for the position where the value 12 exists. Lastly, we'll verify this result. If our algorithm works correctly, the *verify* function should inform us that the target did not exist. Finally, make sure you save the file. Here's the complete code:

```
def linear_search(list, target):
    """
    Returns the index position of the target if found, else returns None
```

```

"""
    for i in range(0, len(list)):
        if list[i]== target:
            return i
    return None

def verify(index):
    if index is not None:
        print("Target found at index: ", index)
    else:
        print("Target not found in list")

numbers = [1,2,3,4,5,6,7,8,9,10]

result = linear_search(numbers,12)
verify(result)

```

Figure 1.3.4 shows how the code looks inside Pycharm. If you've been using Pycharm to write your code, select **Current File** (1) from the drop-down menu at the top right. Then click the green arrow (2) to run your code.

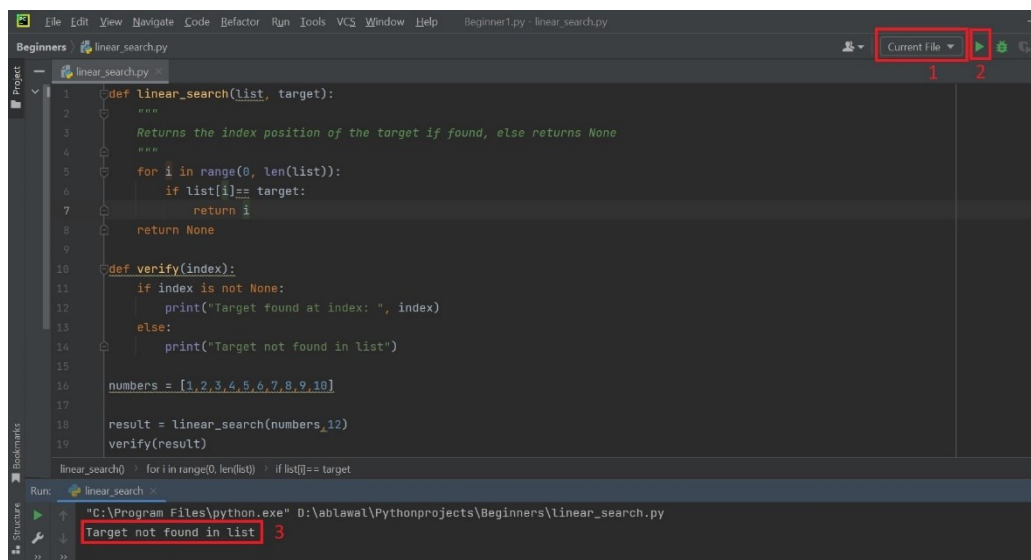


Figure 1.3.4: The linear_search code displayed inside Pycharm workspace. The result is also displayed below (3)

If you've been using Pycharm to write your code, select **Current File** (1) from the drop-down menu at the top right. Then click the green arrow (2) to run your code. As you can see in Figure 1.3.4, the output of the code is correct because the target **12** was not found in the list (3). So, the output of our script is what we expect.

For a second test, let's search for the value **6** in the list. To do this, just go back to your code and change 12 to 6.

```

result = linear_search(numbers,6)
verify(result)

```

Save your code and then hit the green arrow (2) again to run your updated code. Here's the result:

```
Target found at index: 5
```

Again, you'll see that it works correctly. Remember that the first element (1) in the list has index 0. This is why returned **5** as the correct index for the element 6. Run the program on your end and make sure everything works as expected. Our algorithm returns a result in each case it is executed in a finite time, and the results are the ones we expect. In the next section, we'll tackle binary search.

1.3.2. Binary Search in Code

In the last section we left off with an implementation of linear search. Let's do the same for binary search so that we get an understanding of how this is represented in code. We'll do this in a new file. So, create a new file and name this one *binary_search.py*.

Like before, we're going to start with a function named `binary search`. So, we'll say

```
def binary_search(list, target):
```

This takes a list and a target. If you remember, binary search works by breaking the array or list down into smaller sets until we find the value we're looking for. We need a way to keep track of the position of the list that we're working with. So, let's create two variables *first* and *last* to point to the beginning and end of the array.

```
    first = 0
    last = len(list) - 1
```

Now if you're new to programming, list positions are represented by index values that start at 0 instead of 1. So here, we're setting *first* to 0 to point to the first element in the list. *Last* is going to point to the last element in the list but we'll say $len(list) - 1$, not $len(list)$. Now this may be confusing to you, so here's a quick sidebar to explain what's going on.

Let's say we have a list containing 5 elements. If we called *Len* on that list, we should get 5 back because there are 5 elements. But remember that because the position number start at 0, the last value is not at position 5, but at 4. In nearly all programming languages, getting the position of the last element in the list is obtained by determining the length of the list and deducting 1, which is what we're doing.

Now, we know what the first and last positions are when we start the algorithm. For our next line of code, we're going to create a *while* loop. A *while* loop takes a condition and keeps executing the code inside the loop until the condition evaluates to false. For our condition, we're going to tell Python to keep executing this loop until the value of *first* is less than or equal to the value of *last*. So,

```
while first <= last:
```

Well, why you may ask, why is this our condition? Well, let's work through this implementation and then a visualization should help. Inside the *while* loop we're going to calculate the midpoint of our list since that's the first step of binary search.

```
    midpoint = (first + last)//2
```

Now the two forward slashes here are what Python calls a floor division operator. What it does is it rounds down *midpoint* to the nearest whole number. So, let's say we have an eight-element array, where *first* is 0, *last* is 7. If we divided $(0 + 7)$ by 2, we would get 3.5. Now, 3.5 is not a valid index position, so we round that down to 3 using the floor division operator. Now we have a midpoint.

The next step of binary search is to evaluate whether the value at this midpoint is the same as the target we're looking for. So, we say

```
    if list[midpoint] == target:
        return midpoint
```

to find out if *list* and value at *midpoint* equals the *target*. Well, if it is, then we'll go ahead and return the *midpoint*. The return statement terminates our algorithm, and over here we're done. This is our best-case scenario. Next, we'll use the else-if, *elif* as follows.

```
    elif list[midpoint] < target
```

This checks if the value at midpoint is less than the target. Now, if the value is less than the target, then we don't care about any of the values lower than the midpoint. So, we redefine *first* to point to the value after the midpoint as follows.

```
        first = midpoint + 1
```

Now if the value at the midpoint is greater than the target, then we can discard the values after the midpoint and redefine *last* to point to the value prior to the midpoint. So, we say

```
    else:
        last = midpoint - 1
```

Now what if we had executed all this code and never hit a case where midpoint equal the target? Well, that would mean the list did not contain the target value. So, after the while loop, at the bottom, we'll return *None*.

```
return None
```

Here's the updated code.

```
def binary_search(list, target):
    first = 0
    last = len(list) - 1
```

```

while first <= last:
    midpoint = (first + last)//2
    if list[midpoint] == target:
        return midpoint
    elif list[midpoint] < target:
        first = midpoint + 1
    else:
        last = midpoint - 1

return None

```

We have several operations that make up our binary search algorithm. So, let's look at the runtime of each step. We start by assigning values to *first* and *last*. The value assigned to *last* involves a call to the *len* function to get the size of the list, but we already know this is a constant time operation in Python. So, both of these operations run in constant time.

Inside the *while* loop we have another value assignment, and this is a simple division operation. So again, the runtime is constant. In the next line of code, we're reading a value from the list and comparing the *midpoint* to the *target*. Both of these again are constant time operations.

The remainder of the code is just a series of comparisons and value assignments, and we know that these are all constant time operations as well. So, if all we have are a series of constant time operations, why does this algorithm have, in the worst case, a logarithmic runtime?

It's hard to evaluate by just looking at the code, but the *while* loop is what causes the runtime to grow. Even though all we're doing is a comparison operation, by redefining *first* and *last* in the last four steps of the code (*first* = *midpoint* + 1 and *last* = *midpoint* - 1), we're asking the algorithm to run as many times as it needs until *first* is equal to or greater than *last*.

Now, each time the loop does this, the size of the dataset (the size of the list) grows smaller by a certain factor until it approaches a single element, which is what results in the logarithmic runtime.

Now, just like with linear search, let's test that our algorithm works. But before we do that, we'll first go back to our previous *linear_search.py* code, copy the two *verify* cases, and paste that at the bottom of our current code (*binary_search.py*). The updated code is shown below.

```

def binary_search(list, target):
    first = 0
    last = len(list) - 1
    while first <= last:
        midpoint = (first + last)//2
        if list[midpoint] == target:
            return midpoint
        elif list[midpoint] < target:
            first = midpoint + 1
        else:

```

```

        last = midpoint - 1
    return None

def verify(index):
    if index is not None:
        print("Target found at index: ", index)
    else:
        print("Target not found in list")

numbers = [1,2,3,4,5,6,7,8,9,10]

result = binary_search(numbers,6)
verify(result)

```

The only thing we need to change now is instead of calling our code `linear_search`, this is going to be called `binary_search` (highlighted in yellow in the second to the last line). Don't forget to save your code. Select your `binary_search` code workspace (1), and then hit the run button (2).

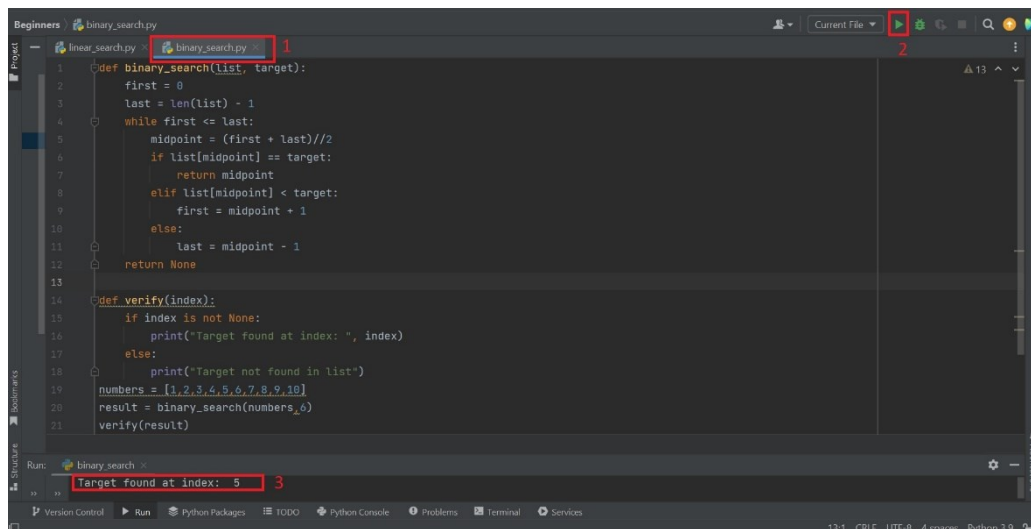
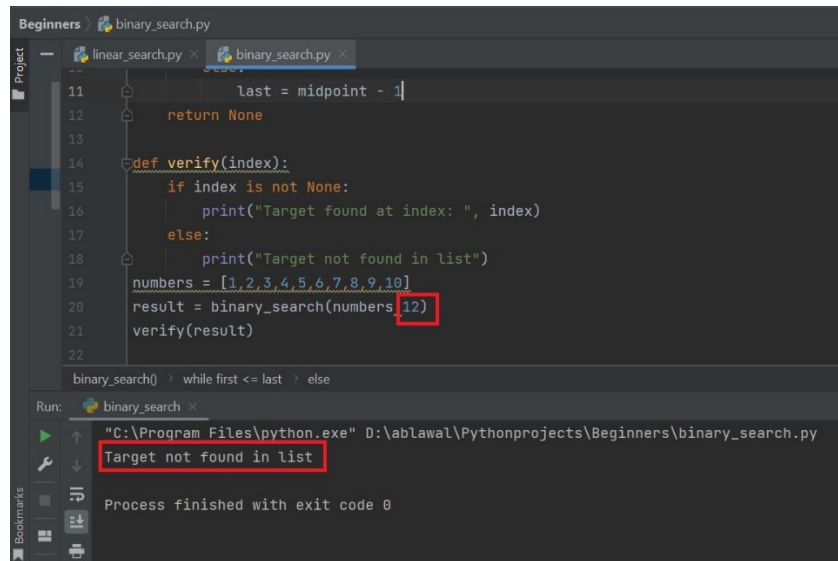


Figure 1.3.5: A search for the value 6 was performed by the `binary_search` code. The result displays “Target found at index: 5” as we expected

As shown in the screenshot in Figure 1.3.5, just like before, we get the same result **5** for target 6 in the `numbers` list:

```
Target found at index: 5
```

For a second test, let's search for the value **12** in the list. To do this, just go back to your code and change **6** to **12**. Then save your code and run it again. As you can see in Figure 1.3.6, the target **12** was not found in the list. The output of our script is what we expected.



```
11         last = midpoint - 1
12     return None
13
14     def verify(index):
15         if index is not None:
16             print("Target found at index: ", index)
17         else:
18             print("Target not found in list")
19     numbers = [1,2,3,4,5,6,7,8,9,10]
20     result = binary_search(numbers, 12)
21     verify(result)
22
```

Run: binary_search

"C:\Program Files\python.exe" D:\ablawal\Pythonprojects\Beginners\binary_search.py

Target not found in list

Process finished with exit code 0

Figure 1.3.6: A search for the value 12 was performed by the `binary_search` code. The result displays “Target not found in list” as we expected

Now note that an extremely important and distinction needs to be made here. The `numbers` list that we’ve defined for our test cases (`numbers = [1,2,3,4,5,6,7,8,9,10]`) has to be sorted. The basic logic of binary search relies on the fact that if the target is greater than the midpoint, then our potential values lie to the left or vice versa, since the values are sorted in ascending order. If the values are **unsorted**, our implementation of binary search may return `None`, even if the value exists in the list.

Now, you’ve written code to implement two search algorithms. How fun was that? Hopefully this course has shown you that it is not a topic to be afraid of and that algorithms like any other topic with code can be broken down and understood piece by piece.

Now we have a working implementation of binary search, but there’s actually more than one way to write it. So, in the next section, let’s write a second version.

1.3.3. Recursive Binary Search in Code

Create a new file. As always, name this file `recursive_binary_search.py`. We’re going to add our new implementation in a new tab on our Pycharm so that we don’t get rid of that first implementation we wrote. Let’s call this new function *recursive binary search*. Unlike our previous implementation, this version is going to behave slightly differently in that it won’t return the index value of the target element if it exists. Instead, it will just return a `True` value if it exists and a `False` if it doesn’t. Let’s begin.

```
def recursive_binary_search (list, target):
```

Like before, this is going to take a list. It accepts a *list* and a *target* to look for in that list. We’ll start the body of the function by considering what happens if an empty list is passed in. In that

case, we would return false. So, we would say,

```
if len(list) == 0:
    return False
```

This is one way to figure out if the list is empty. If *len(list)* is equal to zero, then we'll return *False*.

Now, you might be thinking that in the previous version of *binary_search*, we didn't care if the list was empty. Well, we actually did, but in a roundabout sort of way. In the previous version, our function had a loop, and that loop condition was true when *first* was less than or equal to *last*. So, as long as *first* is less than or equal to *last*, we continue the loop. Now, if we have an empty list, then *first* is greater than *last* and the loop will never execute. So, we return *None* at the bottom.

This is the same logic we're implementing here; we're just doing it in a slightly different way. If the list is not empty, we'll implement an *else* clause. Here, we'll calculate the *midpoint* by dividing the length of the list by 2 and rounding down using the floor division. Again, there's no use of *first* and *last* here, so we'll say

```
else:
    midpoint = (len(list))//2
```

Now, if this value at the midpoint is the same as the target, then we'll go ahead and return *True*.

```
if list[midpoint] == target:
    return True
```

So far, this is more or less the same except for the value that we're returning. If this is in the case, let's implement an *else* clause. Here, we have two situations.

```
else:
    if list[midpoint] < target:
```

First, if the value at the midpoint is less than the target, then we're going to do something new. We're going to call this function again - this recursive binary search function that we're in the process of defining. We're going to call it again and give it the portion of the list that we want to focus on.

In the previous version of binary search, we moved the first value to point to the value after the midpoint. Now here, we're going to create a new list using what is called a slice operation and create a sublist that starts at *midpoint* +1 and goes all the way to the end. We're going to specify the same target as a search target, and when this function call is done, we'll return the value. The return is important.

```
return recursive_binary_search(list[midpoint + 1:], target)
```

As you can see in the above line, we've called this function, *recursive_binary_search* again. This function takes a list, and here we're going to use that subscript notation to perform a slice operation by using two indexes, a **start** and an **end**. The start index is *midpoint* + 1 and the end index is the colon (:), which is a python syntactic "sugar", so to speak. If I don't specify an end index, Python knows to just go all the way to the end. So, the above line contains our new list that we're working with and we need the *target*.

Now we have another *else* case. This is a scenario where the value at the midpoint is greater than the target, which means we only care about the values in the list from the start going up to the midpoint. Now, in this case as well, we're going to call the binary search function again and specify a new list to work with. This time, the list is going to start at the beginning and then go all the way up to the midpoint. So, it looks the same.

```
else:
    return recursive_binary_search(list[:midpoint], target)
```

The colon we put just before *midpoint* without a start index tells Python to start at the beginning and go all the way up to the midpoint. The target here is the same, and this is our new binary search function.

Before we see if this works, we'll need to define a *verify* function. We're not going to copy the previous one because we're not returning *None* or an integer here. So,

```
def verify(result)
    print("Target found: ", result)
```

We wrote the above line because we'll verify the result that we pass in. This is just going to say *True* or *False* (whether we found it or not).

Like before, we need a numbers list. and we'll do something like 1, 2, 3, 4, ... all the way up to 8. Also, we need to test this out. So, we'll call our recursive binary search function and pass in the numbers list and the target here's 12. We're also going to verify this, so, we write

```
numbers = [1,2,3,4,5,6,7,8]
result = recursive_binary_search(numbers, 12)
verify(result)
```

Here's the complete code.

```
def recursive_binary_search (list, target):
    if len(list) == 0:
        return False
    else:
        midpoint = (len(list))//2

        if list[midpoint] == target:
            return True
```

```

else:
    if list[midpoint] < target:
        return recursive_binary_search(list[midpoint + 1:], target)
    else:
        return recursive_binary_search(list[:midpoint], target)

def verify(result):
    print("Target found: ", result)

numbers = [1,2,3,4,5,6,7,8]
result = recursive_binary_search(numbers, 12)
verify(result)

```

Now, make sure you save your code before you run it. As shown in the screenshot in Figure 1.3.7, the search works correctly by displaying *False* since 12 is not in the list.

Target found: False

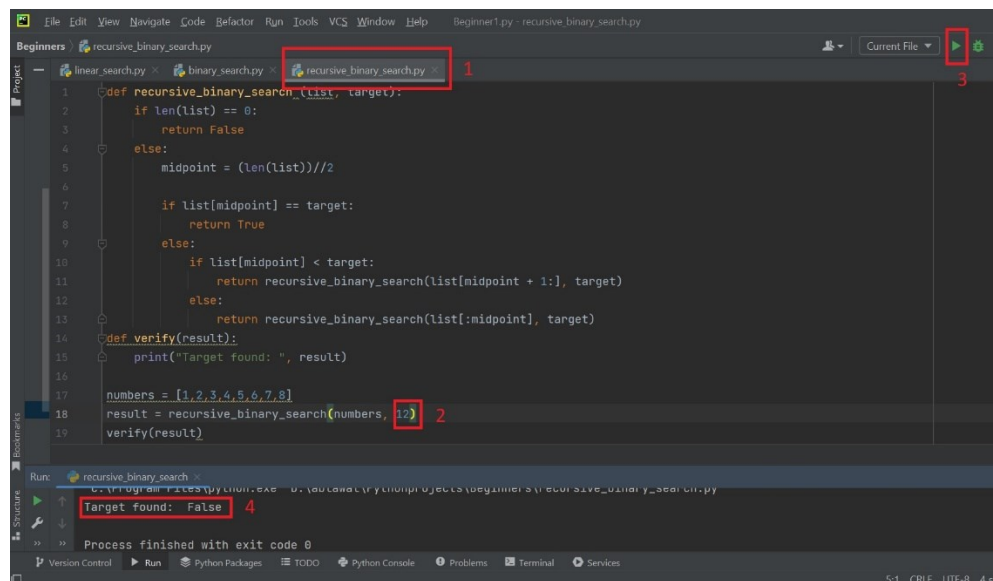


Figure 1.3.7: A search for the value 12 was performed by the `recursive_binary_search` code. The result displays “Target found: False” as we expected

For a second test, let's search for the value **6** in the list. To do this, just go back to your code and change **12** to **6**. Then save your code and run it again. The target **6** was found in the list. So, Python gives the response which is *True*:

Target found: True

So, the output of our script is what we expected.

While we can't verify the index position of the target value, which is a modification to how our algorithm works, we *can* guarantee by running across all valid inputs, that search works as intended.

Now, why write a different search algorithm or different binary search algorithm here, and what's the difference between these two implementations anyway? The difference lies in these four lines of code that you see below (copied from our complete code).

```
if list[midpoint] < target:
    return recursive_binary_search(list[midpoint + 1:], target)
else:
    return recursive_binary_search(list[:midpoint], target)
```

We did something unusual here. Now, before we get into this, a small word of advice: this is a confusing topic, and people get confused by it all the time. Don't worry, that doesn't make you any less of a programmer. In fact, I have trouble with it often and always look it up, including when I made this course. This version of binary search is a recursive binary search. A recursive function is one that calls itself. This is hard for people to grasp sometimes because there's few easy analogies that make sense, but you can think of it and sort this way.

Let's say you have this book that contains answers to multiplication problems. You're working on a problem and you look up an answer in the book. The answer for your problem says add 10 to the answer for problem 52. Hmm! Okay, so you look up problem 52, and there it says add 12 to the answer for problem 85. Well, then you go and look up the answer to problem 85, and finally, instead of redirecting you somewhere else, that answer says 10.

So, you take that 10 and then you go back to problem 52. Because remember the answer for problem 52 was to add 12 to the answer for problem 85. So, you take that 10 and then you now have the answer to problem 85. So, you add 10 to 12 to get 22. Then you go back to your original problem where it's said to add 10 to the answer for problem 52. So, you add 10 to 22 and you get 32 to end up with your final answer.

That's a weird way of doing it, but this is an example of **recursion**. The solution to your first lookup in the book was the value obtained by another lookup in the same book, which was followed by yet another lookup in the same book. The book told you to check the book until you arrived at some base value. Our function works in a similar manner.

Now, like I said at the beginning, this is pretty complicated, so you should not be concerned if this doesn't click. Honestly, this is not one thing that you're going to walk away with, knowing fully how to understand recursion after your first try. I'm really not lying when I say *I* have a pretty hard time with recursion.

Now, before we move on, I do want to point out one thing. Even though the implementation of recursion is harder to understand, it is easier in this case to understand how we arrive at the logarithmic runtime since we keep calling the function with smaller lists. Please take a break here and attempt the next practice exercise. In the subsequent section, we'll talk a bit more about recursion and why it matters.

1.3.4. Practice Exercise 4

Q1. What is the precondition for the binary search algorithm? Choose the correct answer below.

- A. The target value must be in the middle of the list
- B. The data must be sorted
- C. The data must be unsorted
- D. There are no preconditions

Q2. Which of the following two functions is a recursive function? Choose the correct answer below.

Function 1.

```
def sum(list):  
    sum = 0  
    for i in range(0, len(list)):  
        sum += list[i]  
    return sum
```

Function 2.

```
def sum_values(list):  
    if len(list) == 1:  
        return list[0]  
    else:  
        return list[0] + sum_values(list[1:])
```

- A. sum
- B. sum_values

Q3. Inspecting the code for the linear search algorithm (which I've included below for convenience), there is no single step that has a time complexity greater than $O(1)$. Why then do we say that the algorithm has a linear runtime? Choose the correct answer below.

- A. In the best case the algorithm has to compare every item in the list
- B. In the average case the algorithm has to compare every item in the list
- C. In the worst case the algorithm has to compare every item in the list

1.3.5. Answers to Practice Exercise 4

- 1. B
- 2. B
- 3. C

1.4. Recursion and Space Complexity

We've seen how the runtime of algorithm can vary depending on the implementation and the size of the input. But there's one measure of efficiency we haven't explored yet. All algorithms run on computers and computers don't have infinite memory. Over the next few sections, we'll take a look at the memory cost of an algorithm as the input size grows, also known as space complexity.

1.4.1. Recursive Functions

In the last section, we wrote a version of binary search that uses a concept called recursion. Recursion might be a new concept for you, so let's formalize how we use it.

A recursive function is one that calls itself.

In our previous example, the recursive binary search function called itself inside the body of the function. Here it is again.

```
return recursive_binary_search(list[midpoint + 1:], target)
```

When writing a recursive function, you always need a stopping condition, and typically we start the body of the recursive function with this stopping condition. It's common to call this stopping condition the **base case**. In our recursive binary search function, we had two stopping conditions. The first was what the function should return if an empty list is passed in.

It seems weird to evaluate an empty list because you wouldn't expect to run search on an empty list. But if you look at how our function works, recursive binary search keeps calling itself, and with each call to itself, the size of the list is cut in half.

If we searched for a target that didn't exist in the list, then the function would keep halving itself until it got to an empty list. Consider a three-element list with numbers 1, 2, 3, where we're searching for a target of 4. On the first pass, the midpoint is 2, so the function would call itself with the list 3.

On the next pass, the midpoint is 0 and the target is still greater, so the function would call itself this time passing in an empty list because an index of $0 + 1$ in a single element list doesn't exist. When we have an empty list, this means that after searching through the list, the value wasn't found. This is why we define an empty list as a stopping condition or a base case that returns false.

If it's not an empty list, then we have an entirely different set of instructions we want to execute. First, we obtain the midpoint of the list. Once we have the midpoint, we can introduce our next base case or stopping condition. If the value at the midpoint is the same as the target, then we return true. With these two stopping conditions, we've covered all possible paths of logic through the search algorithm. You can either find the value or you don't. Once you have the base cases,

the rest of the implementation of the recursive function is to call the function on smaller sum lists until we hit one of these base cases. The number of times a recursive function calls itself is called **recursive depth**.

Now, the reason I bring all of this up is because if after you start learning about algorithms, you decide you want to go off and do your own research, you may start to see a lot of algorithms implemented using recursion. The way we implemented binary search the first time is called an **iterative solution**. Now, when you see the word iterative, it generally means the solution was implemented using a loop structure of some kind.

A recursive solution on the other hand, is one that involves a set of stopping conditions and a function that calls itself. Computer scientists and computer science textbooks, particularly from back in the day, favor and are written in what are called **functional languages**. In functional languages, we try to avoid changing data that is given to a function. In our first version of binary search (section 1.3.2), we created *first* and *last* variables using a list and then modified *first* and *last* as we needed to arrive at a solution.

Functional languages don't like to do this. All this modification of variables prefer a solution using recursion. A language like Python, which is what we're using, is the opposite, and doesn't like recursion. In fact, Python has a maximum recursion depth after which a function will halt execution. **Python prefers an iterative solution**.

Now, I mentioned all of this for two reasons. If you decide that you want to learn how to implement the algorithm in a language of your choice that is not Python, then you might see a recursive solution as the best implementation in that particular language. I'm an iOS developer for example, and I work with a language called Swift. Swift is different from Python in that it doesn't care about recursion depth, and doesn't need tricks where it doesn't even matter how many times your function calls itself. So, if you want to see this in Swift code, then you need to know how recursion works. Well, now you have some idea.

The second reason I bring it up is actually way more important, and to find out, move onto the next section.

1.4.2. Space Complexity

At the beginning of this book, I mentioned that there were two ways of measuring the efficiency of an algorithm. The first was **time complexity**, or how the runtime of an algorithm grows as n grows larger. The second is **space complexity**. We took a pretty long route to build up this example, but now we're in a good place to discuss space complexity.

Space complexity is a measure of how much working storage or extra storage is needed as a particular algorithm grows. We don't think about it much these days, but every single thing we do on a computer takes up space and memory. In the early days of computing, considering memory usage was of paramount importance because memory was limited and really expensive.

These days were spoiled. Our devices are rich with memory. This is okay when we write everyday code because most of us are not dealing with enormously large data sets. When we write algorithms, however, we need to think about this because we want to design our algorithms to perform as efficiently as it can as the size of the data set n grows really large. Like time complexity, space complexity is measured in the worst-case scenario using Big O notation. Since you are now familiar with the different kinds of complexities, let's dive right into an example.

In our iterative implementation of binary search, the first one we wrote in section 1.3.2, that uses a *while* loop. Let's look at what happens to our memory usage as n gets large. Let's bring back that function.

```
def binary_search(list, target):
    first = 0
    last = len(list) - 1
    while first <= last:
        midpoint = (first + last)//2
        if list[midpoint] == target:
            return midpoint
        elif list[midpoint] < target:
            first = midpoint + 1
        else:
            last = midpoint - 1
    return None

def verify(index):
    if index is not None:
        print("Target found at index: ", index)
    else:
        print("Target not found in list")

numbers = [1,2,3,4,5,6,7,8,9,10]

result = binary_search(numbers,6)
verify(result)
```

Let's say we start off with a list of 10 elements. Now inspecting the code, we see that our solution relies heavily on these two variables, *first* and *last*. *first* points to the start of the list and *last* to the end. When we eliminate a set of values, we don't actually create a sublist. Instead, we just redefine *first* and *last* (highlighted yellow in the code above), to point to a different section of the list.

Since the algorithm only considers the values between *first* and *last*, when determining the midpoint by redefining *first* and *last*, as the algorithm proceeds, we can find a solution using just the original list. This means that for any value of n , the space complexity of the iterative version of binary search is constant, or that the iterative version of binary search takes **constant space**. Remember that we would write this as $O(1)$. This might seem confusing because as n grows, we

need more storage to account for that larger list size. Now, this is true, but that storage is not what space complexity cares about measuring. We care about what *additional* storage is needed as the algorithm runs and tries to find a solution.

If we assume something simple, say that for a given size of a list represented by a value n , it takes n amount of space to store it, whatever that means. Then for the iterative version of binary search, regardless of how large the list is at the start, middle and end of the algorithm process, the amount of storage required does not get larger than n . This is why we consider it to run in constant space.

Now, this is an entirely different story with the recursive version, however. In the recursive version of binary search, we don't make use of variables to keep track of which portion of the list we're working with. Instead, we create new lists every time with a subset of values or sublists with every recursive function call.

Let's assume we have a list of size n , and in the worst-case scenario, the target element is the last in the list. Calling the recursive implementation of binary search on this list and target would lead to a scenario like this. The function would call itself and create a new list that goes from the midpoint to the end of the list. Since we're discarding half the values, the size of the sublist is $n/2$. This function will keep calling itself creating a new sublist that's half the size of the current one until it arrives at a single element list and a stopping condition.

This pattern where the size of the sublist is reduced by a factor on each execution of the algorithmic logic we've seen before. Do you remember where? This is exactly how binary search works. It discards half the values every time until it finds a solution. Now, we know that because of this pattern, the running time of binary search is logarithmic. In fact, this space complexity of the recursive version of binary search is the same.

If we start out with a memory allocation of size n that matches the list, on each function call of recursive binary search, we need to allocate additional memory of size $n/2$, $n/4$ and so on until we have a sublist that is either empty or contains a single value. Because of this, we say that the recursive version of the binary search algorithm runs in logarithmic time with a Big O of $O(\log n)$. This is illustrated in Figure 1.4.1.

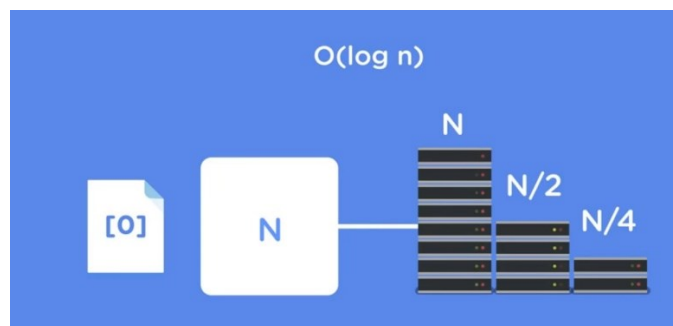


Figure 1.4.1: Illustration of the space complexity of the recursive version of binary search

Now, there's an important caveat here. This totally depends on the language. Remember how I said that a programming language like Swift can do some tricks to where recursion depth doesn't matter? The same concept applies here. If you care to read more about this concept, it's called **tail optimization**. It's called tail optimization because, if you think of a function as having a head and a tail, if the recursive function call is the last line of code in the function as it is in section 1.3.3, we call this tail recursion since it's the last part of the function that calls itself.

Now, the trick that Swift does to reduce the amount of space, and therefore computing overhead, to keep track of this recursive calls is called **tail call optimization** or **tail call elimination**. It's one of those things that you'll see thrown around a lot in algorithm discussions but may not always be relevant to *you*.

Now, what if any of this is relevant to *us*? While Python does not implement tail call optimization, so the recursive version of binary search takes logarithmic space. If we had to choose between the two implementations, given that time complexity or runtime of both versions, the iterative and the recursive version are the same, we should definitely go with the iterative implementation in Python since it runs in constant space.

Whew! Okay, that was a lot. But with all of this, we've now established two important ways to distinguish between algorithms that handle the same task and determine which one we should use.

1.4.3. A Recap of What You Learned

We've arrived at what I think is a good spot to take a long break and let all of these new concepts sink in. But before you go off to take your practice exercise, let's take a few minutes to recap everything we've learned so far.

While we did implement two algorithms in this course, in actual code, much of what we learned here was conceptual and will serve as building blocks for everything we're going to learn in the future. So, let's list all of it out.

1. Algorithmic Thinking

The first thing we learned about and arguably the most important was algorithmic thinking. Algorithmic thinking is an approach to problem solving that involves breaking a problem down into a clearly defined input and output, along with a distinct set of steps that solves the problem by going from input to output.

Algorithmic thinking is not something you develop overnight by taking one course, so don't worry if you're thinking, I still don't truly know how to apply what I learned here. Algorithmic thinking sinks in after you go through several examples in a similar fashion to what we did today. It also helps to apply these concepts in the context of a real example, which is another thing we will strive to do moving forward.

Regardless, it is important to keep in mind that the main goal here is not to learn how to implement a specific data structure or algorithm off the top of your head. I'll be honest, I had to look up a couple code snippets for a few of the algorithms myself in writing this course. But in going through this, you now know that binary search exists and can apply it to a problem where you need a faster search algorithm. Unlike most courses where you can immediately apply what you have learned to build something cool, learning about algorithms and data structures will pay off more in the long run.

2. How to Define and Implement Algorithms

The second thing we learned about is how to define and implement algorithms. We've gone over these guidelines several times (see Figure 1.1.8), so I won't bore you here again at the end. But I will remind you that if you're often confused about how to effectively break down a problem in code to something more manageable, following those algorithm guidelines is a good place to start.

3. Big O and Runtimes

Next, we learned about Big O and measuring the time complexity of algorithms. This is a mildly complicated topic, but once you've abstracted the math away, it isn't as hazy a topic as it seems. Now, don't get me wrong, the math is pretty important, but only for those designing and analyzing algorithms. *Our* goal is more about how to understand and evaluate algorithms.

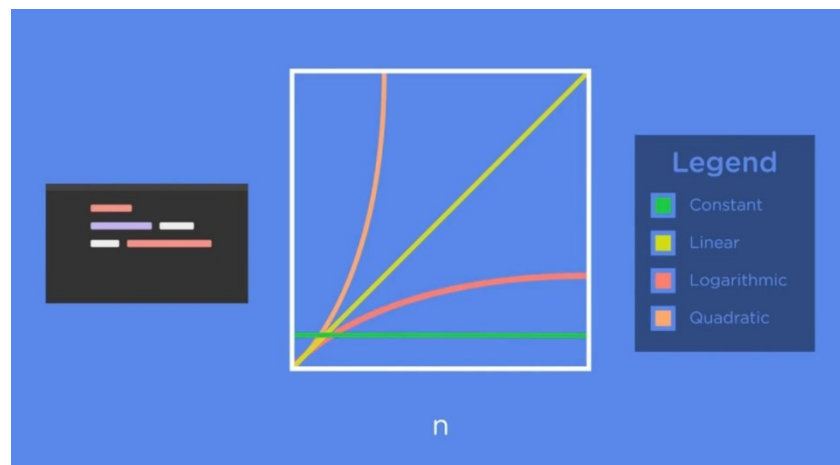


Figure 1.4.2: Graph of runtimes of different algorithms

We learned about common runtimes like constant, linear, logarithmic, and quadratic runtimes. These are all fairly new concepts, but in time you will immediately be able to distinguish the runtime of an algorithm based on the code you write and have an understanding of where it sits on an efficiency scale.

You'll also in due time internalize runtimes of popular algorithms, like the fact that binary search runs in logarithmic time and constant space, and be able to recommend alternative algorithms for

a given problem. All in all, over time, the number of tools in your tool belt will increase.

4. Important Search Algorithms

Next, we learned about two important search algorithms and the situations in which we select one over the other. We also implemented these algorithms in code so that you got a chance to see them work. We did this in Python, but if you are more familiar with a different language, you should try your hand at implementing it yourself. It's a really good exercise to go through.

5. Recursion

Finally, we learned about an important concept and a way of writing algorithmic code through recursion. Recursion is a tricky thing and, depending on the language you write code with, you may run into it more than others. It is also good to be aware of because as we saw in our implementation of binary search, whether recursion was used or not affected the amount of space we used.

Don't worry if you don't fully understand how to write recursive functions. I don't truly know either. The good part is you can always look these things up and understand how other people do it. Anytime you encounter recursion in *my* courses moving forward, you'll get a full explanation of how and why the function is doing what it's doing. That brings us to the end of part one of this course.

I'll stress again that the goal of part one of this course was to get you prepared for learning about more specific algorithms, by introducing you to some of the tools and concepts you'll need moving forward. So, if you are sitting there thinking, I still don't know how to write many algorithms or how to use algorithmic thinking, that's okay. We'll get there. Just stick with it. As always, have fun and happy coding!

All Books in the Series

Volume 1: Introduction to Algorithms & Data Structures 1

Volume 2: Introduction to Algorithms & Data Structures 2

Volume 3: Introduction to Algorithms & Data Structures 3

1.4.4. Practice Exercise 5

Q1. Space complexity is a measure of how much total storage an algorithm uses. Choose the correct answer below.

- A. True
- B. False

Q2. What is recursion depth? Choose the correct answer below.

- A. The number of times a recursive function calls itself
- B. The number of operations a recursive function needs to execute
- C. The number of lines of code in a recursive function

Q3. At minimum every recursive function must include which of the following. Choose the correct answer below.

- A. A loop construct
- B. A base case or stopping condition
- C. Tail recursion

Q4. Which of the following statements are true? Choose the correct answer below.

- A. A recursive function implements its logic using a loop construct while an iterative function calls itself
- B. An iterative function implements its logic using a loop construct while a recursive function calls itself

1.4.5. Answers to Practice Exercise 5

- 1. B
- 2. A
- 3. B
- 4. B

1.5. Download Training Resources & Get Further Help

Here's the link you can use to download the codes we wrote, screenshots used in this book, more practice exercises and other training resources:

<https://ln5.sync.com/dl/3448f5f90/s6wvzr5w-fwdezt2r-bp78yxma-uwnwvmcs>

Alternatively, you can use this short BIT.LY link:

<http://bit.ly/3zFSnWV>

These resources are updated regularly to provide further help. If you need to contact me for anything, or if you want to share your codes with me, use my email address below. I'll get back to you very quickly. I promise.

Cheers,

Ojula Technology Innovations

OjulaTech@gmail.com

www.ojulaweb.com