Algorithms Every Programmer Should Know

Aniket Wattamwar

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Algorithms Every Programmer Should Know

Version 1

For more information on this and other Manning titles go to manning.com.

welcome

Thanks for purchasing the MEAP of *Algorithms Every Programmer Should Know*. I am thrilled you are here, and your early feedback is going to be critical in shaping the final version of this book.

This book was written for developers who are tired of the traditional way algorithms are taught. If you have ever opened a classic computer science textbook only to be hit with a wall of Greek letters, mathematical proofs, and complex equations, you know exactly what I mean. Algorithms are the backbone of software engineering, system architecture, and technical interviews, yet they are almost always presented in the most intimidating way possible.

I wrote this book to fix that.

Instead of lecturing you with formal theorems, this book isolates the concepts from the code. You will learn through the conversations, debates, and realizations of three characters as they dissect algorithms by asking questions and solving an example. We build the intuition first, using real-world analogies, and only drop into Python once the underlying mechanics make complete sense.

The book is structured to take you from foundational data structures straight into the architectural algorithms that power production systems. You don't need a math degree to read this, but you do need to be comfortable with basic programming concepts.

Because you are reading this in MEAP, you have a direct line to me. I want to know what clicks, what feels confusing, and where the narrative needs more depth. Please share your thoughts, critiques, and questions in the liveBook Discussion forum. Your feedback is what will make this book exceptional.

—Aniket Wattamwar

brief contents

1 The Language of Algorithms

This chapter covers

- Introduction of the characters
- Understanding the basics of Algorithms
- Analyzing Algorithms
- Learning Fundamental & Technical concepts of algorithms

Algorithms over the years have been part of our lives in ways that we never anticipated. They are the fundamental blocks that drive the entire modern computing and now the AI ecosystem as well. All the apps we use and websites we visit and interact with, algorithms live at the core of it. We don't realize it as a user.

The book focuses on the foundational algorithms that, as a developer or programmer, you would have to learn at some point in your software journey. Most readers here would know how to code with basics like loops, functions, and conditions, but algorithms are a bit deeper. The intention is to develop a strong intuitive understanding of algorithms that form the core of software. Ideas that quietly power the real systems regardless of the programming language or framework.

Why should we care about algorithms? Well, they impact performance, reliability, and scalability. The idea of a good algorithm in a system drives the business, user experience, and convenience. We will look at practical examples, implement an example, walk through edge cases, and understand tradeoffs, efficiency over simplicity, and where to use which.

As a software developer, having the wisdom to know that every problem does not need a clever and complex solution. In real-world systems, maintainability and correctness matter more than theoretical optimality.

The goal of the book is not to overwhelm you with complex equations, technical words, theorems, and mathematics and jump right into code but to help you think algorithmically. It is written for students, early-career software engineers, and self-taught programmers who want to understand the nature of the algorithm, visualize it before implementing it, and know why and how these things work. Understanding this allows you to know the tradeoff and make decisions based on the use case. By the end of the book and going through the chapters with example runs, you should be able to recognize the nature and algorithmic patterns and be able to explain them in plain language.

The book is structured in a unique way. Instead of formal definitions and proofs, each chapter begins with a scenario, conversation, and questions inspired by real school life. The dialogue is intentional. It mirrors how people actually learn difficult concepts by asking, clarifying, thinking, and explaining. This approach retains the intuition of algorithms for a longer period of time in your mind.

1.1 Setting the Scene

Our journey in learning algorithms all started at the turn of the new semester.

After wrapping up the first term filled with course overloads, jet lag, and cultural chaos, I finally had a moment to breathe. Maya, Dev, and I had arrived in California just a few months ago, chasing our master’s dreams like thousands of others. Different backgrounds, same destination. Dev had worked in corporate, developing enterprise applications in Java for a few years before he came for his master's. Maya and I both studied electronics but worked in software development after our first job, and now all of us are pursuing masters in computer science.

Classes were long, but the professors made them interesting. With time my roommates and I connected over studies and how everyone wanted to be good at coding and become solid developers.

And somewhere in the middle of settling into this new life, we started finding comfort in the small, ordinary routines, cooking together on weekends, arguing about whose turn it was to take out the trash, sharing stories from home, and laughing about the tiny cultural surprises we encountered every day. These little moments made the apartment feel less like temporary student housing and more like a home we were building together. And within that comfort, studying didn’t feel like a burden anymore, it felt like an extension of our shared journey, a thing we were learning to carry as a team.

We’d barely survived the first term, and yet we signed up for one of the most notorious courses in the program: Algorithms.

Honestly? I wasn’t scared of it. I was excited. The curriculum was structured with algorithms, assignments, projects, quizzes, and of course midterms and finals. It seemed overwhelming, but the syllabus included everything that was necessary for understanding the fundamentals.

But here's the thing, when you're sharing a two-bedroom apartment with new people who are equally enthusiastic about technology, things tend to develop. The living room was our place to discuss anything from technology to chores. Whiteboards, laptops, takeout boxes, and chore charts stuck on the fridge.

What started as casual group study soon became a ritual. After long days of lectures and part-time jobs, we'd come back, throw our bags to the floor, reheat leftovers, and dive deep into discussing algorithms, assignments, or projects. It wasn't just about solving problems, it gave us confidence in understanding these complex algorithms. Maya would often take charge with her highlighters and diagrams, while Dev would question everything until he found a corner case that broke it all. Me? I guess I was the bridge, trying to keep the sanity intact.

This book is a retelling of our journey of understanding, questioning, and learning in the simplest way. Not in the dry, formula-filled, chalkboard-smudged kind of way, but in a way we lived it.

Each chapter is built around a moment, an afternoon, a late-night debate, or a pre-exam panic. Inside those moments, we unpacked the most important algorithms we had to learn: from classic sorting and searching to the world of graphs, dynamic programming, and beyond. We didn't just study them, we argued, explained, re-explained, and occasionally even acted them out.

And just like us, you won't need anything fancy. Just a curious mind and the willingness to see algorithms not as abstract puzzles, but as stories waiting to be understood.

Welcome to our spring semester.

It's going to be fun. I promise.

1.2 Basics of Algorithms

Algorithms, in the most fundamental terms, mean to solve a particular problem by following a series of steps. In computer science, algorithms are the heart of any computer program. The steps taken by the algorithm are expected to be generalized over a range of inputs, and the problem must be well specified with expected output. The same problem can be solved with different algorithms. For example, the problem of sorting a list of unsorted numbers. This problem is solved by algorithms like insertion sort, bubble sort, and selection sort. These algorithms output the same, but the series of steps taken to get the output vary. Any algorithm we design should have properties that would classify it as a good algorithm like correctness, easy verifiability, and easy implementation. This section covers the basics and the need for algorithms in computer science.

The semester began 2 weeks ago. Maya, Dev, and I attended our first algorithms class, which was brief and only included an overview of the syllabus. We got to the second class, and the professor went over the fundamentals of algorithms, which we had learned in our previous course during the sophomore year. Before the next class, we decided to revisit a few that had been taught. Dev also suggested explaining them to each other and sharing additional insights from our prior experience. It was a usual afternoon, I was at my desk staring at my laptop reading a technical article when Dev walked in.

"Aniket! Are you busy?" he asked.

"I was just reading the article about the Content Delivery Network (CDN) outage that happened last week. Why? What happened?" I asked. "Is Maya in her room? I was thinking we should discuss a few things before our algorithms class tonight." "Yes, she's in her room. Go ahead and call her, she brought this up earlier." I said, scratching my head.

Dev sat on the beanbag and shouted, "Maya!!" She walked in from her room, laptop in hand, as if she already knew what we called her for.

"I am ready when you guys are!" she announced.

1.2.1 Algorithms and why do we need them?

In real life, problems are just not about sorting numbers and searching values but vary significantly. DNA and RNA sequences, network engineering, the aviation industry, school and university software, and manufacturing industries are a few of the areas where algorithms play a vital role in solving problems. In the later chapters, we will see how different algorithms are applied to real-world applications.

Dev had already opened his laptop. "Should we talk about time and space complexity of algorithms?" Dev asked us.

"Let's take a step back and start from the beginning rather. Maybe like what algorithms are and how they work and why we need them?" I said. "Fair. But isn't it obvious? I mean, an algorithm is just steps we perform to complete a task or solve a problem, right?" Maya chipped in. "That's true. But the implementation of the steps is dependent on inputs and the particular problem to be solved."

"Exactly," I said. "So it's more about getting the answer within constraints like time, memory, cost, and scale, maybe!"

"Right. When we think about algorithms, we need to make sure the solution is generalized for all the inputs we provide. But there is always a trade-off in these constraints you mentioned, Aniket. There isn't a universal best and fastest algorithm. When we optimize something, there is some constraint we sacrifice." Maya nodded. "That's how it works. And by definition it should also provide the correct answer within the specified time."

"How do you guys come up with a solution to a problem?" I asked.

"For me, when I think about solving and coming up with a solution, it's always brute force…" Maya clarified. "…And that is exactly the first step we should take while solving a problem with an algorithm. The next step would be making it optimized, efficient, and fast," Dev said.

He paused and continued, "Google search makes sure that millions of people can use it simultaneously without crashing. I mean, the search algorithm for it is implemented at such a huge scale."

"That's just searching. There are so many other problems like mathematical computations, sorting, electronic payments, navigation, networks, e-commerce, and so much more. All of them are pretty efficient as well." Maya said.

Dev's eyes lit up. "Efficiency of the algorithm! I mean different algorithms solving the same problem but their efficiency differs a lot."

He picked up the marker and went to the whiteboard. He continued. “Let’s understand with examples of sorting problems.” He wrote a few numbers on the board. “I remember Insertion Sort, Bubble Sort, and Selection Sort," I said.

“Correct. All of them perform sorting, but their implementation is different!” said Dev.

1.3 Analysis of Algorithms

The steps we discussed to arrive at the output for a particular problem should be analyzed in the sense of how we measure how good or bad the steps taken were. We need a system that would help us identify the efficiency of algorithms before we implement them. When we start implementing algorithms with programming languages, they would consume space and memory to compute these steps. Let's understand how we can analyze an algorithm and how it helps us make efficient and better decisions in this section.

“We can define the efficiency and analysis of an algorithm in terms of speed, input, and memory, right? How fast does an algorithm take to give a result? If it takes longer, either we change the implementation of the algorithm or use a different one that solves the same problem. In this case, searching is the task.” Maya said, looking up from the laptop.

“So how do we calculate the efficiency?” I asked.

“It’s not that difficult,” said Dev. “If we want to know how good and efficient it is, we look at the worst condition it has to go through while solving the task. So for searching from an array of numbers like these, if we do not have the number in the array, what is going to happen?” asked Dev.

“Well, we will have to go through all the numbers first to know if it even existed or not.” I said.

“That’s correct. It means you had to go through all the N numbers given if N is the length of the array. This is called the worst-case analysis of this particular linear search algorithm.” Dev took a pause.

“Often it’s asked, why take the worst case and not the average case? Well, a failure in your algorithm, being a part of a system, occurs at edge cases. Considering them it’s logical to keep the worst case as the metric to track.”

Maya continued. “So if the input changes, our efficiency also changes. For a very large input and the number to be searched is at the end, we practically have to go through all numbers and use our system's memory.”

“So for the best case scenario, the number we are looking for is the first number itself. That would be the best case analysis," I said.

“Correct. So the worst case is the upper bound and the best case is the lower bound of the algorithm. And we use the Big O Notation as a way to express the analysis in terms of Time and Space Complexity, which I was talking about.” Dev said, while writing the same on the board.

He continued. "Based on the changes in the input, we can know a lot about the behavior of the algorithm like the order of growth and its efficiency, and do comparisons with different algorithms. Oftentimes engineers like ourselves will ignore the growing input value or optimize too early. It's important to understand when and where the optimization should be performed. Misusing doesn't make it better. I have seen really clean and good code but wrong implementation."

"Do we have anything for a lower bound limit, a notation for that?" Maya asked

"Yes," Dev confirmed. "It's called Big Omega Notation."

1.3.1 Time and Space Complexity

Time and space complexity are used to determine the analysis of algorithms. The time taken by an algorithm changes with the change in input. For a few algorithms, the running time or the time complexity changes exponentially with the change in input. Space complexity refers to the amount of memory algorithms required to execute. In the case of using additional data structures, more memory is consumed with respect to input. Analysis of algorithms, keeping the time and space complexity in mind, helps in selecting efficient algorithms where inputs are large and constraints are tight.

"Aniket, you said you knew about linear and binary search. Let's discuss the time and space complexities of those then." Maya confirmed. "Yes." I said and went towards the board, and Dev handed me the marker. "Before we analyze that, I'll share how to find the time and space complexity and their types, then it would be easier for any algorithm we discuss," I said.

He wrote on the board.

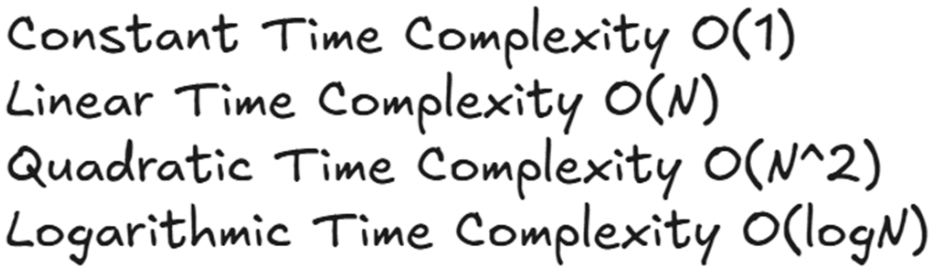

Figure 1.1 List of categories of time complexity used in algorithm analysis, including constant time O(1), linear time O(N), quadratic time O(N^2), and logarithmic time O(logN). N represents the increasing input size, highlighting the performance differences in time complexities.

“We can say time complexity is constant if the efficiency remains the same even when the input changes. In the simplest of terms, constant time is seen when there are no loops or recursive functions in our code.” I started explaining

“Few examples of constant time complexity would be swapping two numbers or getting the number in a contiguous array at a particular index. Because even if the input changes, it will still take O(1) running time.” I took a pause.

“That makes sense," Maya said.

“Then we have linear time complexity. This is where the linear search algorithm fits perfectly. We did discuss it, right? To find the element in the array, we would have to iterate through all the numbers and compare it with the one we are looking for. Here, we would be using a loop to iterate through the array, making the time complexity O(N), where N is the length of the array.” I explained

“The efficiency and the time would change with the input length,” Dev pitched in. “Higher N means we would search for a longer time.”

I continued. “Correct. Next we have quadratic time complexity, meaning the execution time grows proportionately to the square of the input changes. So if the input is N, the execution will take N*N. We see this in sorting algorithms like insertion and bubble sort. It has nested loops, so every element is compared with every other element in the array.”

“Lastly, we have logarithmic. This is very interesting, and binary search is the perfect example of this time complexity. If we need to look for a number in a sorted array, then we do not go linearly, we look at the middle element. If that number is higher or lower than our intended number, we decide which way to check.”

“Ahh! That is very clever. Basically, we are not searching one part of the array altogether. And we shouldn’t either since the middle number tells us which part to look for.” Maya said.

“Precisely. We repeat this in the entire array till we find the target number, and every time we eliminate half of the array giving us a time complexity of O(log N).” I completed it. “What about space complexity?” Maya asked.

Dev answered. “When we execute our algorithm, the total memory used by it changes with the input value. A poorly designed and executed algorithm will take up a huge amount of space in memory making it inefficient. Ideally, we optimize our algorithm keeping space and time complexity in mind."

I put the marker down and sat on my chair and leaned back. “I think that is what the professor taught us in the last class. Do you guys recall any other topics?"

“Phew!! That was a lot to take in.” Maya said with a sigh of relief. “Yeah! These are just the basics. I am excited to learn more about the type of algorithms," Dev said.

“Should we discuss data structures next?” I asked

“I need a break. Let’s continue in 30 minutes.” Maya announced and got up and walked towards the kitchen.

Apart from time complexities mentioned in figure 1.1. Cubic Time complexity O(n^3) and exponential time (2^n) exist, which are inefficient. While designing algorithms, the analysis discussed above guides making efficient choices in reducing the time and space complexity to the best possible way. Figure 1.2 displays how the time complexities vary with input.

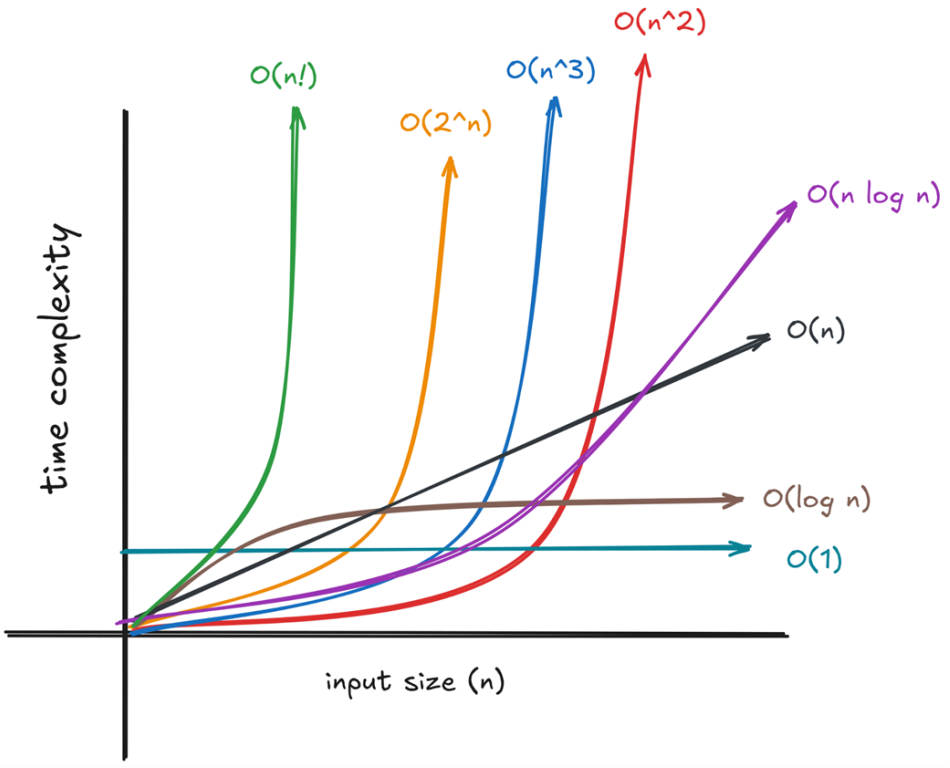

Figure 1.2 A graph showing the time complexities with varying inputs. O(1) is the constant time, while O(n!) shows the worst time. O(log n) is logarithmic, and O(n) is linear time. Quadratic and cubic are in between and work well for smaller values of n.

1.4 Data Structures

This section introduces data structures and how we can implement them. Algorithms are as good as the data structures we use to implement them. Powerful data structures reduce the time complexity giving us efficient algorithms. There are many data structures, but we will discuss the fundamental ones like HashMaps, linked lists, stacks, and queues. These data structures will allow you to develop more complex variations of data structures that are suited for the algorithm being designed and developed.

Dev was clearing the whiteboard and wrote "Data Structures" and "Hash Maps" below it and turned back.

“Alright. Algorithms decide how we solve problems,” he said. “But now let’s talk about where the data lives. An algorithm is only as fast as the structure it works on.”

I nodded. “That explains why the same algorithm behaves very differently depending on whether we use an array, set, or something else.” Maya leaned forward. “So if we choose the wrong one, even a good algorithm can become slow.” “Worse,” I added. “It can become unusable at scale.”

1.4.1 Hash Maps

A hash map is that data structure that stores key-value pairs. The keys in hash maps are always unique and are mapped to a value. The value can be anything based on the problem being solved. Hash maps make insertion, deletion, and retrieval faster in O(1) time complexity. The O(1) time complexity of a hash map is due to a well designed hash function. It minimizes the hash collisions, making the performance close to O(1). A poorly designed hash function for a hash map will have time complexity in the worst case as O(N). Combinations of other data structures are used with hash maps too. For example, the value for a particular key can be a list or a linked list as well.

Dev smiled and pointed at the whiteboard. “Exactly. Let’s start with one of the most useful ones - HashMaps.”

Dev drew a set of boxes and arrows on the board.

“A hash map stores data as key–value pairs,” he said. “Instead of searching linearly, we jump straight to where the data lives.” Maya frowned. “How does it know where to jump?”

“Hashing,” Dev replied. “We apply a hash function to the key. That function converts the key into an index.” I jumped in. “So if I give it the same key, I always get the same index.”

“Correct,” Dev said. “That’s the whole trick.” Maya glanced at her laptop. “Everyone says hash maps are fast. How fast are we talking?” Dev smiled and said, “O(1) for insert, search, and delete.” “That sounds almost too good,” I said. “What about space complexity?” Maya asked.

“Higher than arrays,” Dev replied. “We trade memory for speed.” I summarized, “So hash maps use extra space for buckets, pointers, and resizing, but in return we get near-constant time operations.”

“That’s the deal,” Dev said. “And most real systems happily take it.”

“So where do we actually use a hashmap?" I asked. “Anytime you hear, count how many times something appears," Dev said, “then hashmap should come to our mind.” Maya asked, “Like words?"

“It could be anything like characters in a string, words in a sentence or even clicks on a website. The key is the thing you’re counting. The value is the frequency," Dev said. “Each time we need the frequency, we use the same key and get the value directly," I said. “Exactly. It would be an efficient update and retrieval.” Dev said.

Maya again looked at her laptop. “In real systems, I can see here it’s used in counting page visits per user, tracking error occurrences in logs, and counting inventory items in warehouses.”

Dev paused and said, "There is a part that people might miss. So we can use hash maps not just to count but to remember state as well. So a scenario where we need to check if something happened before turning the maps to history with constant time complexity.”

“Where will this not work in our favor?” Maya asked. I said, "Do we need order or position?"

“Right. Hash maps don’t preserve order. So for any task or problem depending on sorting, traversal, or ranges, you’ll need some other data structure," Dev clarified. “Right now we store integers at the values. What other data types can we utilize as values?” I asked.

“That's very interesting actually. We can use multiple data types like arrays, strings, and even linked lists as the values in the hash map against a key.” Dev said and started writing on the board. “That increases the scope of implementation across so many tasks,” Maya said.

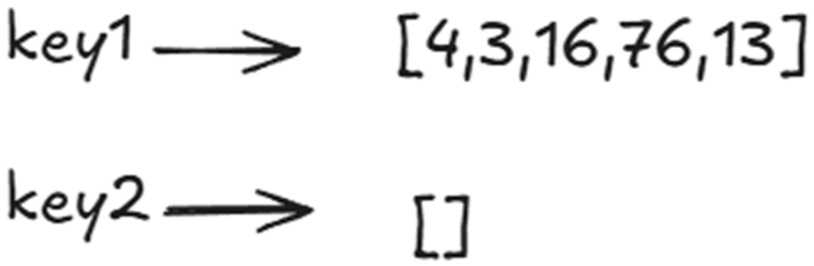

Figure 1.3 Illustration of a hash map where each key is associated with a list of values. Key1 maps to a list of numbers. Key2 maps to an empty list. The lists can be accessed via keys, which are unique in the hash map.

1.4.2 Linked List and Doubly Linked List

A linked list is a dynamic data structure where elements are stored as nodes. Each node points to the next node in the list. They are different from arrays, as elements in the arrays can be accessed directly, but in linked lists, traversal is necessary. Insertions and deletions are very efficient in linked lists as compared to arrays. Arrays store the elements in contiguous memory, to make an insertion or deletion, the elements have to shift. A doubly linked list is an extension of a linked list with the addition of another pointer pointing backwards. This allows the traversal to occur in both ways. As compared to a linked list, where traversal in one direction comes with a challenge of no reverse traversal, a doubly linked list solves this challenge. Although this added advantage of two-way traversals comes with the cost of extra memory usage.

"Now that you mentioned Linked List, in this data structure we don't have the speed as compared to a hash map." Maya said, looking confused. "That's right," I said and gestured to Dev to hand me the marker. "Looks like Aniket will share about linked lists now." Dev said, smiling and taking a seat.

I made a linked list on the board.

Figure 1.4 Illustration of a linked list with four nodes. Each node has data and a pointer pointing to the next node. The start of the linked list is marked as head. The end of the linked list is identified if the next pointer is null.

"Every problem might not need speed. Many problems need flexibility." I was assured.

Pointing at the board. "The linked list is like a chain where every element has two pieces of information, one is its own data or value, and the other is the pointer to the next element."

"This means that I cannot retrieve an element at position 6 the way we could do it for an array." Maya confirmed.

"No, you cannot. You start from the first always and check the data till you arrive at the data you want."

"How is this useful then?" Maya asked. "Its purpose is different. In a linked list, the insertions and deletions are faster now. In an array, adding an element at position 3 requires us to shift the elements towards the right. This is expensive," I explained.

Dev nodded. "Here, we just change the pointers then, right!" Maya raised an eyebrow. "The traversal is O(N), but then modifying anything is efficient," I added. "This data structure and its variations help in solving task scheduling problems or music player problems, where we play the next or remove a song and insert a song."

"Once we know how a linked list works, we modify it to a more complex data structure like a doubly linked list," I continued. "Doubly? Meaning we can traverse forward and backward?" Maya asked, looking at the board.

I was modifying the diagram. "That is exactly what we do. We store in one element the previous, the data, and the next value. This removes the bottleneck of moving in only one direction. Now we move either way."

"So we now have more control over the flow and navigation, it seems." Dev said, crossing his arms.

Figure 1.5 Illustration of a doubly linked list with 3 nodes. Each node has a previous pointer, data, and a next pointer. Doubly linked lists help in navigating both ways using the previous and next pointers.

Maya leaned back and summarized what she understood. "The real task is to understand which data structure fits perfectly for a given algorithm. Use a linked list when you know the direction and order are priorities. Doubly linked lists are used when backward traversal is required and the order of the elements is still needed. Both data structures take O(n) time complexity for searching and use more memory."

Listing 1.1 shows the implementation of a linked list in python. To create a node with data and the pointer, a class is created with actual data and the next pointer variable initialized. The linked list class is implements the node creation through the Node class.

Listing 1.1 Code implementation of linked list

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return

        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

    def display(self):
        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements))
```

Listing 1.2 is a doubly linked list with similar implementation but the pointers are in both directions as discussed in the chapter.

Listing 1.2 Code implementation of a doubly linked list

```
class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = DNode(data)
        if not self.head:
            self.head = new_node
            return

        current = self.head
        while current.next:
            current = current.next

        current.next = new_node
        new_node.prev = current

    def display_forward(self):
        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" <-> ".join(elements))
```

1.4.3 Stacks and Queues

Stacks and Queues are linear data structures where elements are added sequentially, but the stacks pop the last element first and the queues pop the first element first. These data structures perform common operations like undo/redo, function calls, and expression calls for stacks and scheduling and buffering for queues. This section covers the mechanism of these data structures and its implementation.

I turned back and started clearing the board again. “What’s next?” Maya asked. "Stacks and queues, I suppose," Dev confirmed. “Aren’t these data structures like arrays and lists, but slightly different?” Maya asked.

I continued. "Yes."

“In the simplest term, Stack is Last In, First Out (LIFO)! So what is put in the stack last is the one we can take out first. like a stack of pancakes. If we have a stack and I add numbers to it like 3, 4, 6, and 8 in that order, then 8 is the last one to go in, so I can take that out first before accessing the next," I said. “Isn’t it a bit restrictive then?" Dev asked.

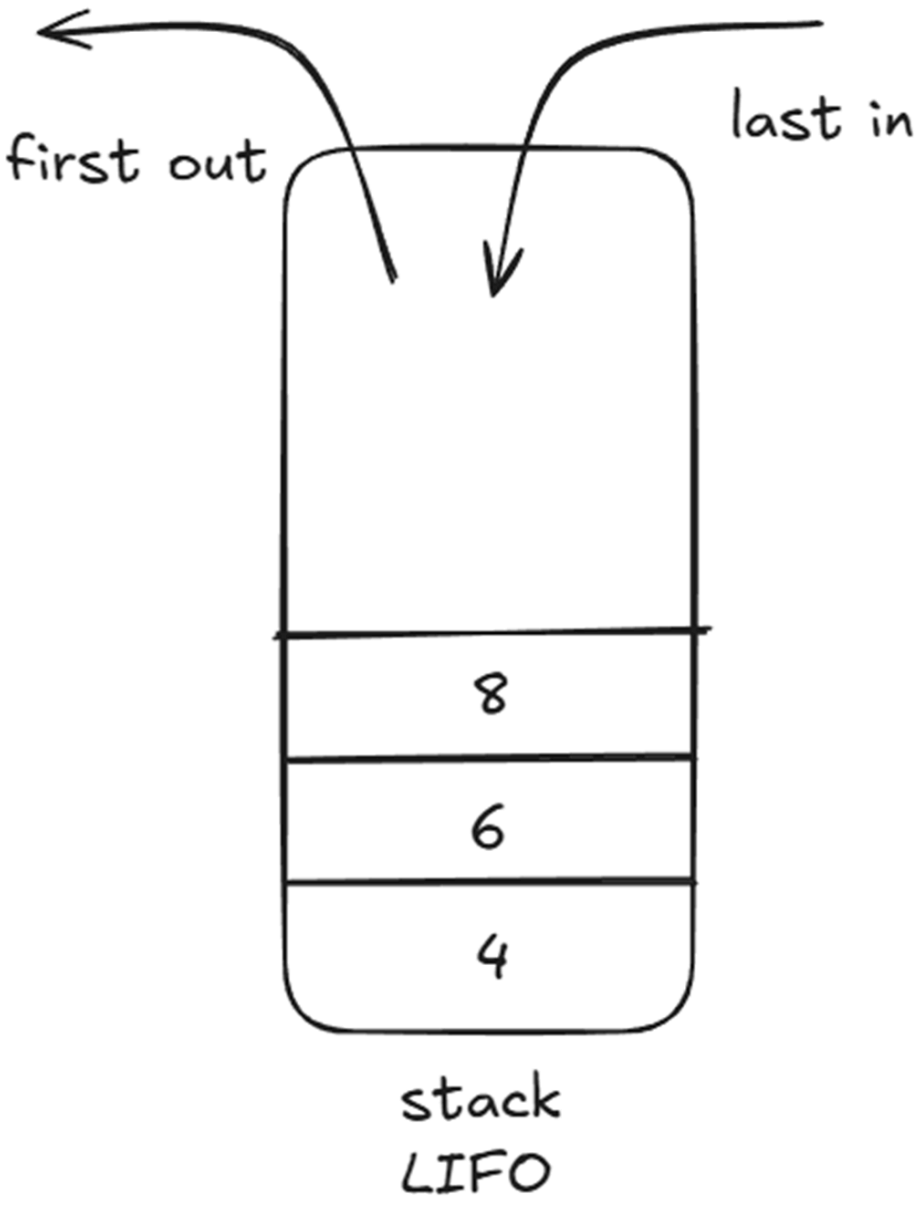

Figure 1.6 The stack data structure is Last In First Out(LIFO). Illustration of a stack with three numbers. The number 8 was inserted last and it will be popped first. The number 4 was inserted first, it will be popped only when all the numbers above the stack are popped first.

“Yes. But this restriction is useful. Think about it. A function call in a program. Every time it’s called, its local variables and parameters are pushed into a stack," I said. “Recursion?” Maya looked confused.

“Right. One of the best use cases. Once the function finishes, it pops off, and control is handed over to the function that called it.” Maya’s eyes widened. “That’s why we get stack overflow since we are not popping, only stacking. That actually makes the stack feel real.” Dev leaned forward. “The undo and redo operations work the same way.”

“Perfect examples," I said, “but operations are limited here. We push, pop, and peek.”

“What about the performance?” Dev asked. “For push and pop it would be O(1), because it’s always the element on the top. " I said,

"...And space complexity would be O(N) with N the number of elements on the stack.” Dev continued. “What about Queues ? I assume it would be similar but slightly different again.” Maya guessed. “Right. Similar to Stack but Queue are First In First out (FIFO). That element that goes in first comes out first as well.”

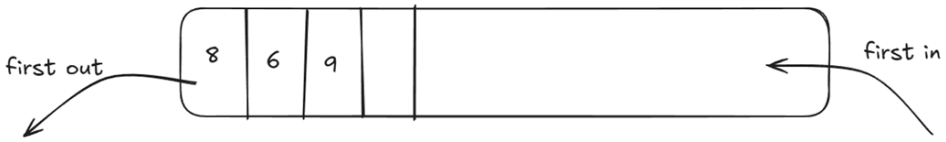

Figure 1.7 The queue data structure is First In First Out(FIFO). The number 8 illustrated in the image was inserted first, and it will be popped out first.

“Like ordering in Starbucks. Whoever ordered first gets their order ready first as well.” Maya said. “Correct. If we want to use queues in other algorithms like tree traversal or so. Queues are generally used. It’s called Breadth First Search in Trees or Graphs," I said.

“I think the time and space complexity would be the same, O(1) and O(N), right?” Maya said, thinking as she was speaking. Maya leaned back and shut her laptop. “So simple rules but massive impact and applications.” We all sat looking at the whiteboard. “That was a good discussion," I finally said.

“I am very excited for our next class.”

Listing 1.3 shows the Stack class, where the operations push, pop, and peek are implemented. The values that are added to the stack are stored in the items list.

Listing 1.3 Code implementation of a stack

```
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0
```

Listing 1.4 shows the Queue class, where the operations push, pop, and peek are implemented. It is important to note that we utilize a *deque* from the collections library. A deque is a doubly linked list under the hood from the collections python library and can perform operations in O(1) time complexity. The values that are added to the queue are stored in the items list.

Listing 1.4 Code implementation of a queue

```
from collections import deque

class Queue:
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.popleft()

    def front(self):
        if self.is_empty():
            raise IndexError("Front from empty queue")
        return self.items[0]

    def is_empty(self):
        return len(self.items) == 0
```

1.4.4 Trees

The data structures discussed above are linear, but in real world the data is not always linear. When that happens advanced data structures are used like Trees and Graphs. A *Tree* is a hierarchical data structure that organizes data top-down just like a computer file system.

"Think of a corporate org chart," I said, drawing a single circle at the top of the whiteboard. "This is the CEO. We call this the *root* of the tree."

"And the VPs reporting to them?" Maya asked.

"Children. Which also makes the root their parent," I said, drawing two arrows pointing downward. "In a tree, data strictly flows down. No loops. No one reports to two managers. The dead-ends at the very bottom, the individual contributors with no direct reports are the *Leaf* nodes. All parents and their children, along with the root, are also called the *nodes* of the tree. There is always one root, and every node descends from it.

"So, how many children a node of a tree can have?" Dev asked.

"We can have as many as we want, but having many children coming from a parent node and the hierarchy continuing, it becomes increasingly difficult to maintain. So tree algorithms are constrained to limit the child nodes they can have to maintain predictability." I took a pause.

"Keeping that in mind, we have different types of trees." I started writing them on the board.

- Binary Tree
- Full Binary Tree
- Perfect Binary Tree

"A binary tree, as the name says, consists of a maximum of two child nodes per parent node. They are called the left and right nodes. A *full binary tree* ensures every single node has either exactly zero children or exactly two. No single-child nodes. A *perfect binary tree* goes further, ensuring every leaf sits at the exact same depth. When the structure is perfectly symmetrical, our search math becomes incredibly fast and pred*ictable.*" I completed my explanation.

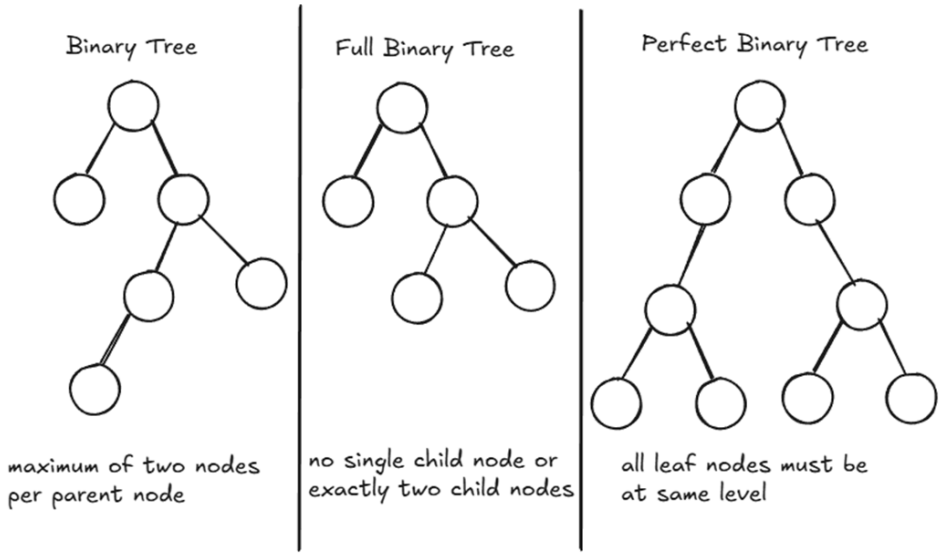

Figure 1.8 A comparison of common binary tree structural variations. Left: A standard binary tree where nodes have a maximum of two children. Middle: A full binary tree enforcing that nodes possess either exactly zero or exactly two child nodes. Right: A highly uniform perfect binary tree where all interior nodes contain two children and every leaf node aligns at the exact same depth level.

Dev looked at the board and said, "This is interesting! How do we find a node in this structure now? Do we also have push or pop operations here?"

"We don't push or pop the tree itself," I said. "The tree is just the map connecting nodes to other nodes. We use push and pop on our linear data structures that we discussed, like stacks and queues, to keep track of where we are while we explore it."

Maya leaned forward. "So the linear structures act like a GPS for the hierarchical ones." "Exactly," I said. "If you want to search a tree, we have two traversal techniques called Breadth First Search (BFS) and Depth First Search (DFS). DFS ensures we explore a complete path starting from a node to reaching a leaf node. Once we reach a leaf node, that is also the depth of the tree. BFS, on the other hand, will search layer by layer. A root node with 4 children, all 4 children will be searched before checking their children."

Dev pointed at the leaf node marked 8 on the board. "Let's say we start at the root node, 5, and we want to find 8. If we use Depth-First Search, how does the system actually navigate this?"

"Look at the arrows on the left diagram," I said. "DFS is aggressive. It doesn't care about the broad picture. It picks the very first path on the left and plunges completely to the bottom. It checks those leaves. When it realizes 8 isn't there, it moves back up to its parent and to the nearest intersection and plunges down the next available path. It prioritizes vertical depth, repeating this deep dive until it finally hits 8."

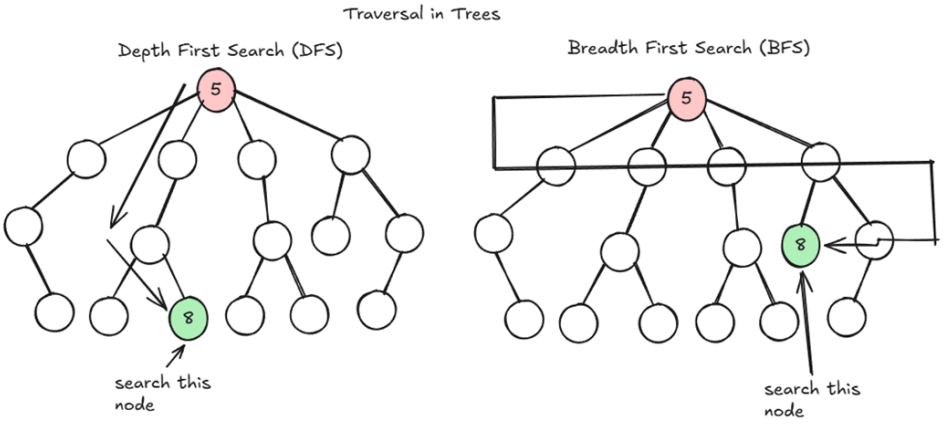

Figure 1.9 The two fundamental strategies for searching non-linear data structures. Left: Depth-First Search (DFS) goes deep down a single path to a leaf node. Right: Breadt-First Search (BFS) expands outward, exploring all the neighboring nodes layer by layer before descending to the next level.

"Breadth-First Search is the exact opposite," I said, tapping the right diagram. "Look at the horizontal lines. BFS refuses to go deep. From the root node 5, it first checks every single one of its immediate children. It scans the entire horizontal layer. Only when it confirms 8 isn't in that first layer does it drop down to the next level and start scanning horizontally again."

Maya studied the two distinct paths on the whiteboard. “Okay, the visual intuition makes sense. DFS goes deep, BFS goes wide. But a computer doesn't have intuition. If a system is in the middle of a massive network, how does it physically remember which branch to backtrack to or which horizontal layer it’s supposed to check next?”

“That's correct,” I replied. "You cannot traverse a hierarchical network without a linear map. You need the basic data structures we just discussed.”

“Stacks and Queues,” Maya realized.

“Exactly,” I confirmed. “To run a DFS, you use a stack. Last-In, First-Out. You push nodes onto the stack as you dive deep. When you hit a dead end, you pop the dead-end off the stack and backtrack to the previous node. It perfectly mirrors the physical action of reversing out of a maze.”

“Which means BFS uses a queue," Dev concluded.

“First-In, First-Out,” I nodded. “When you arrive at a node, you add all of its children to the back of the queue. Because it's a strict line, you are mathematically forced to evaluate every node in the current horizontal layer before you are allowed to touch the children in the layer below them.”

To put these traversal concepts in practice, we must understand the computational boundaries. For a tree containing V vertices or nodes, both BFS and DFS have a time complexity of O(V) because every single node must be checked once in the worst-case scenario. The space complexity for BFS, on the other hand, is O(W), where W is the maximum width of the tree, meaning the maximum number of nodes at a single depth level, as the queue must hold an entire layer of children. For DFS, the space complexity is the maximum depth of the tree, O(D)

Listing 1.5 shows the standard implementation of a binary tree with BFS and DFS as two functions. A node of the tree is implemented as a class with left and right nodes as their children.

Listing 1.5 Code implementation of a tree

```
from collections import deque

class TreeNode:
    def __init__(self, value): #A
        self.value = value
        self.left = None
        self.right = None

def tree_bfs(root):
    """
    Executes a Breadth-First Search traversal on a Binary Tree.
    Outputs the node values layer-by-layer (level-order).
    """
    if not root:
        return []

    queue = deque([root]) #B
    traversal_output = []

    while queue:

        current_node = queue.popleft() #C
        traversal_output.append(current_node.value)

        if current_node.left:
            queue.append(current_node.left) #D

        if current_node.right:
            queue.append(current_node.right) #E

    return traversal_output

def tree_dfs(root):
    """
    Executes an iterative Depth-First Search traversal on a Binary Tree.
    Outputs the node values along vertical paths before moving laterally.
    """
    if not root:
        return []

    stack = [root] #F
```

```
    traversal_output = []

    while stack:
        current_node = stack.pop() #G
        traversal_output.append(current_node.value)

        if current_node.right:
            stack.append(current_node.right) #H
        if current_node.left:
            stack.append(current_node.left)

    return traversal_output
```

#A Represents a single node within a Binary Tree.
#B Utilize a deque for O(1) FIFO queue performance
#C Pop the front node from our layer queue
#D Enqueue the left child if it exists
#E Enqueue the right child if it exists
#F Standard Python list used as a LIFO stack
#G Pop the most recently added node from the top of the stack
#H Push the right child first so the left child is processed first

1.4.5 Graphs

While trees follow hierarchy, real-world systems do not always follow hierarchy. To completely map open, interconnected networks where any data point can link to any other, we must transition to a structure known as a *graph.*

"So looking at graphs, they are essentially trees but with a massive catch, they can contain loops, or cycles, in the network," I said, drawing a new diagram on the board. "Mathematically, every tree is a graph, but not every graph is a tree. The moment you introduce a cycle or a lateral path, the tree-hierarchy collapses into a graph."

"Why do we need them?" Maya asked. "What's the advantage of a graph over a tree?"

"Trees are restricted to parent-child dynamics," Dev interjected, looking at the board. "A graph allows you to connect to multiple nodes. It solves problems where everything is interconnected, is what I understand."

"Exactly," I confirmed. "Graphs are the absolute engine behind modern logistics, social networks, and routing. Think about Google Maps. An intersection isn't a 'child' of another intersection, it's just a node connected by a road. Graphs allow us to solve complex real-world optimization problems like finding the absolute shortest path between two cities, modeling data flow through a telecommunications network, or detecting communities in a social network."

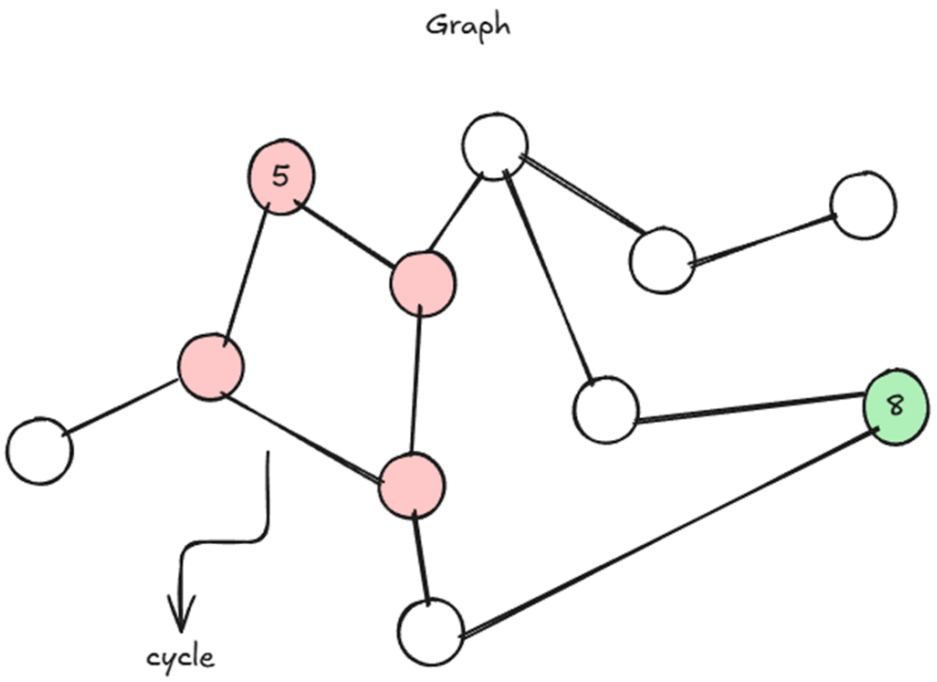

Figure 1.10 The transition from a tree to a relational graph. The structural tree rules are broken to form a cycle or a closed loop. One node can be connected to multiple nodes.

"But wait," Maya said, pointing to our previous traversal sketches. "If we run our standard depth-first search or breadth-first search on a graph that loops back on itself, won't the algorithm get stuck running in a circle forever?"

"It absolutely will," I said, redrawing our traversal models to include a closed loop. "Let's look at how DFS and BFS navigate a graph with a cycle."

"If we run a Depth-First Search (DFS) here," I said, tracing the arrows on the left graph, "the algorithm will plunge deep down a path just like it did in the tree. But when it encounters a loop, it will blindly follow it back to the start and loop forever. To fix this, we introduce a visited set, a mathematical checklist. Before DFS plunges to the next node, it checks the list. If it has been there before, it halts and backtracks."

"And Breadth-First Search (BFS)?" Dev asked, looking at the right graph. "Does it expand outwards?"

"Yes, it still ripples outward layer by layer using a Queue," I explained. "But just like DFS, if BFS doesn't use a Visited Set to block nodes it has already seen, the horizontal layers will eventually fold back on themselves, trapping the queue in an infinite execution loop."

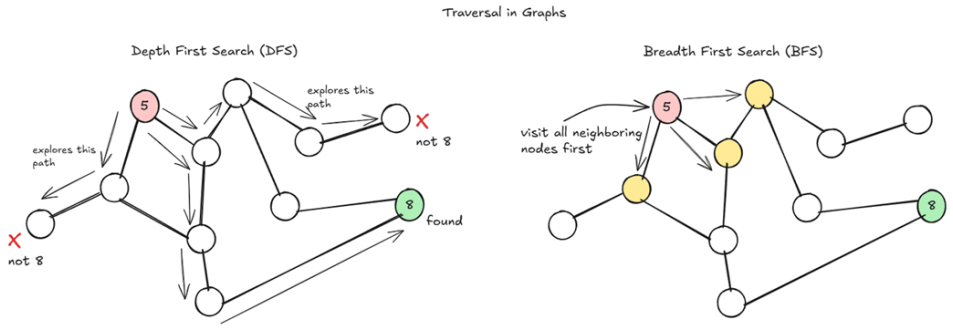

Figure 1.11 Traversal paths shown through a network with a cycle. Left: Depth-First Search (DFS) traces deep pathways until it detects dead-ends and mismatches. Right: A Breadth-First Search (BFS) expands to all neigboring or adjacent nodes. Both traversals use a visited set to track already visited nodes to avoid infinite loops.

Dev studied the diagram, then tapped his marker against his chin. "Wait a minute. In our tree diagrams, we always started at the Root. But you just said a graph has no hierarchy and no root. If there's no root node, where does the traversal actually begin?"

"In a graph, the starting point is completely arbitrary," I replied. "Because there is no natural entry point, the starting node, often called the source node, depends entirely on the problem you are trying to solve. If you are calculating a route on Google Maps, the source node is simply wherever the user is currently standing. You feed that specific node into the algorithm, and the traversal ripples outward from there."

"That makes sense for the logic," Maya said, opening her laptop. "But how do we physically write this in code? A tree uses simple object pointers like node.left and node.right. How do you represent a web of random, cyclical connections in a clean Python data structure?"

"In production, you choose between two classic memory blueprints," I said, turning back to the board. "The Adjacency Matrix and the Adjacency List. An *Adjacency Matrix* is a two-dimensional array where both the rows and columns represent your vertices. If Node A connects to Node B, you place a 1 at the intersection of row A and column B. If they aren't connected, it's a 0. An *Adjacency List*, on the other hand, is a key value structure and stores the graph as a hash map, where the keys are your vertices and the value of each key is a dynamic list containing only the neighbors it actually connects to."

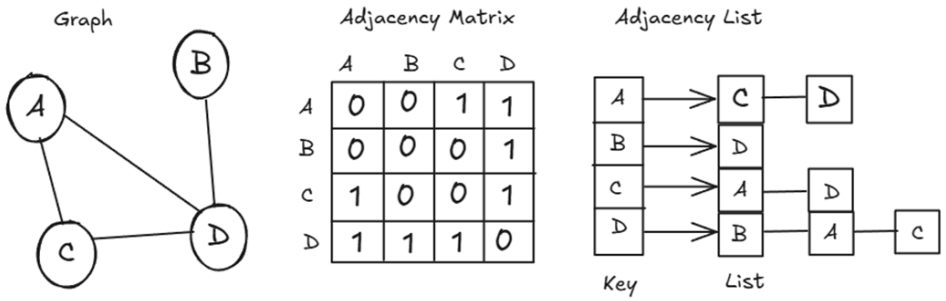

Figure 1.12 The two primary data structures for graph traversals. Left: Simple graph representation with four nodes connected to each other. Middle: An adjacency matrix maps connections from one node to another, denoting them by the value 1 in the matrix and otherwise 0. Right: An adjacency list showing a key-value dictionary connecting to neighboring nodes only.

"How do we handle the direction of the connections? If I follow someone on a social platform, but they don't follow me back, how does that look in our list?"

"That brings us to the two fundamental types of graphs," I replied, erasing a corner of the board to make room. "Undirected Graphs and Directed Graphs."

I used the previous graph example and drew directions on it again. "In an undirected graph, the edges are two-way streets. If Node A connects to Node B, the relationship is entirely mutual. Think of Facebook: if you are friends with someone, they are automatically friends with you. In your adjacency list, you have to add B to A's list, and add A to B's list."

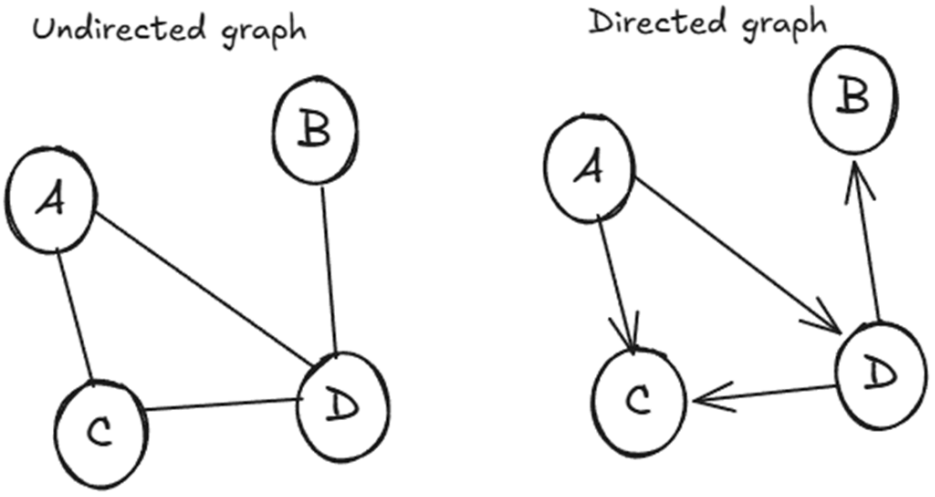

Figure 1.13 Two structural variations of a graph. Left: An Undirected Graph maps mutual, bidirectional connections. Right: A Directed Graph restricts travel paths to be unidirectional, forcing algorithms to explore a path based on direction.

"And a directed graph handles the one-way connections?" Maya asked.

"Exactly," I said, drawing an explicit arrow pointing from one node to another. "In a directed graph, or digraph, relationships have a specific orientation. This is Twitter or Instagram. You can follow a celebrity, so your node points to theirs, but their node does not point back to you unless they explicitly follow you. In the adjacency list, you only add the destination node to the source node's list."

Dev tapped his marker against his notebook. "So for a directed graph where A points to B, B's list stays completely empty unless B points somewhere else."

"Precisely," I nodded. "And this distinction fundamentally changes how your traversals behave. If you run BFS or DFS on an undirected graph, you can move back and forth along any edge. But in a directed graph, your traversal is strictly forced to obey the direction of the arrows.

When traversing a general graph, the time complexity for BFS and DFS is O(V+E), where V is represents the total number of vertices and E represents the total number of edges or connections. The space complexity is O(V) because the algorithm is required to maintain a dedicated visited set to track and block cyclic infinite loops, which can grow and hold every vertex of the graph in the worst-case scenario.

Listing 1.6 provides the code implementation of a graph with BFS and DFS functions with cycle protection.

Listing 1.6 Code implementation of a graph

```
def graph_dfs(graph, start_node):
    """
    Executes an iterative Depth-First Search traversal on a Graph
    represented as an Adjacency List.

    Args:
        graph (dict): A dictionary mapping nodes to lists of their neighbors.
        start_node: The chosen arbitrary source node to begin traversal.
    """
    if start_node not in graph:
        return []

    stack = [start_node] #A
    visited = set() #B
    traversal_output = []

    while stack:
        current_node = stack.pop()


        if current_node not in visited:
            visited.add(current_node)
            traversal_output.append(current_node) #C

            neighbors = graph.get(current_node, []) #D

            for neighbor in neighbors:
                if neighbor not in visited:
                    stack.append(neighbor) #E

    return traversal_output

from collections import deque

def graph_bfs(graph, start_node):
    """
    Executes an iterative Breadth-First Search traversal on a Graph
    represented as an Adjacency List.

    Args:
```

```
        graph (dict): A dictionary mapping nodes to lists of their neighbors.
        start_node: The chosen arbitrary source node to begin traversal.
    """
    if start_node not in graph:
        return []

    queue = deque([start_node]) #F

    visited = {start_node} #G
    traversal_output = []

    while queue:

        current_node = queue.popleft() #H
        traversal_output.append(current_node)

        neighbors = graph.get(current_node, []) #I

        for neighbor in neighbors: #J
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor) #K

    return traversal_output
```

#A An explicit LIFO stack to drive deep vertical exploration
#B A hash set providing O(1) lookups to track visited nodes and prevent cycles
#C If the node has not been processed yet, lock it in
#D Retrieve neighbors from the adjacency list
#E Push unvisited neighbors onto the stack
#F Utilize a deque for O(1) FIFO layer tracking
#G Protect against cyclical infinite loops with an O(1) tracking set
#H Dequeue the oldest node from the front of the line
#I Retrieve neighbors from our adjacency list mapping
#J Evaluate neighbors layer by layer
#K Append to the back of the queue to prioritize current layer nodes

The above-mentioned data structures are fundamental, and more complex data structures do exist, like red-black trees and tries, to name a few. However, these are created keeping the above as the basics. One should understand them and know how to implement them. Hash maps have O(1) time complexity, linked lists have O(N) time complexity for traversing, and stacks and queues have O(1) time complexities for add and pop operations. Trees and graphs have O(V) and O(V+E) time complexities, respectively.

1.5 Summary

- Algorithms govern the world of computer science. They are the fundamental blocks of software development. The design of algorithms should provide a general solution to inputs.
- An algorithm is a step-by-step procedure to be followed to arrive at a solution of a specified given problem
- Algorithms are needed, as they solve complex problems in multiple industries like healthcare, manufacturing, e-commerce, and more. Real-world systems rely on well-developed and scaled algorithms.
- Efficiency matters because slow and memory-heavy solutions break. Worst-case analysis is used generally to understand algorithm behavior, as it is the upper bound of performance. Best, worst, and Average time complexities are used for analysis of algorithms
- Common time complexities are constant time, linear time, quadratic time, and logarithmic time. Input N to these analyses shows the change in behavior of the output of the complexities.
- Data structures used influence algorithm performance. Arrays, linked lists, doubly linked lists, hash maps, stacks, and queues are a few data structures.
- Hashmaps store key-value pairs. They perform operations like insert, delete, and retrieval in O(1) time. They do not maintain order.
- Linked lists and doubly linked lists store data with pointers pointing forward and backward. This helps in traversal, and insertions and deletions are faster. Stacks and queues perform add and pop operations.
- Trees are hierarchical data structures for top-down, non-linear dependencies. Trees have a root node with multiple nodes attached to it called children, and those child nodes again have multiple nodes branching out. A node with no children is called a leaf node. The path from the root node to the leaf node is called the depth of a tree.
- Tree algorithms leverage constraints on the structure for predictability. Based on these constraints, types of trees are binary trees, full binary trees, and perfect binary trees.
- Binary Trees limit parents to a maximum of two children. A full binary tree permits nodes to have exactly two or zero children. Perfect binary trees are entirely symmetrical with all leaf nodes at the same depth level.
- Two traversal techniques exist: Depth First Search (DFS) and Breadth First Search (BFS). Breadth-First Search (BFS) utilizes a FIFO queue to sweep horizontally, level-by-level, expanding outward, bounding space complexity to the tree's width (O(W)). Depth-First Search (DFS) utilizes a LIFO stack to aggressively plunge vertically down a single branch before backtracking, bounding space complexity to the tree's depth (O(D)). Both track at O(V) time.

- Graphs are a network of nodes connected to each other but do not have constraints like trees. Any node can link to any other and contains loops and cycles. Algorithms must integrate an explicit visited set to track traversed nodes in a graph to avoid infinite loops. Time complexity of a graph is O(V + E), where V is the total number of vertices and E is the total number of edges or connections.
- Graphs use an adjacency matrix, a 2D array where the rows and columns are the vertices of the graph, a connection between two nodes is denoted by 1 or else 0, and an adjacency list, a memory-efficient key-value dictionary, where the keys are the vertices and the values represent the keys it's connected to.
- Connections in graphs are bidirectional or directional, called undirected graphs or directed graphs, respectively.

2 Gale-Shapley Algorithm

This chapter covers

- Explaining what problem the algorithm solves, and why stability matters
- Demonstrating with an example how proposals are made and rejected until stability
- Analyzing the correctness, termination, optimality and proposer bias
- Applying the algorithm in real-world scenarios like college admissions

In the previous chapter, we looked at the core fundamentals of computer science and the components that drive today's technology. We looked at data structures like hash maps, linked lists, stacks, and queues and how they perform by understanding the space and time complexities.

The Gale-Shapley algorithm was published in 1962 by David Gale and Lloyd Shapley. It is a stable matching algorithm that pairs equal numbers of participants to their preferences, like matching students to universities or online dating. It is also called the Deferred Acceptance Algorithm (DA) or the Propose-and-Reject Algorithm. This algorithm addresses one of the most elegant problems in combinatorial optimization, i.e., how to pair two equal-sized sets of participants based on their preferences in a way that no two individuals would rather be with each other than with their assigned preferences.

This algorithm tackles stability. A matching is unstable if there is a pair of a proposer and a recipient who are not currently matched but would prefer each other over their current matches. The stability of the Gale-Shapley algorithm ensures that everyone is matched to a preference where no other preference is higher than their current choice and guarantees a matching always exists.

The Gale-Shapley algorithm iteratively operates with proposers like students making offers and the recipients like universities accepting them temporarily, hence called *deferred acceptance*, and rejects sequentially when a better preference is found in the next iteration. This approach is optimal for the proposer among all the stable matchings. This highlights the power of the algorithm as well as the bias, which might have critical implications when applied to real-world use cases where proposers manipulate the results in their favor.

2.1 The Stable Matching Problem

Now we will walk through how the algorithm works by using an example of students and universities. At the end of the example, we will have successfully matched all students to universities based on their preferences. The matching will be stable since every student is allotted a university based on the order they preferred.

2.1.1 One-Pass Matching

The following example demonstrates the direct matching of students to universities where universities are the proposers, which basically means universities will send invitations to students. The example is straightforward and shows the mechanics of the algorithm.

I was scrolling through Stack Overflow on a slow Saturday morning when Dev shuffled in wearing fuzzy slippers. Maya followed, holding a marker and a whiteboard that read "Stable Matching Problem" and a chai mug in her hand.

She turned and asked us, ‘Have you guys heard about the Gale-Shapley Algorithm?’

Dev looked up, confused, a clear indication to Maya that he had not, and the same for me.

“So imagine we have two groups, a set of students and a set of universities. Each student ranks universities by preference, and each university ranks students. The idea is to match them in a way where nobody has a reason to switch.”

Dev was confused and asked, “Wait, what do you mean by that?

“Like,” she continued, “no student should prefer a different university that also prefers them back more than their current student. That’s called a *stable match*.”

Maya had the example on the board already and started explaining. The university X prefers Alice, Bob, and then Carol in that specific order. Similarly, Y prefers Bob, Carol, and Alice. Z prefers Carol, Alice, and Bob.

Universities (Proposers):

- X: [Alice, Bob, Carol]
- Y: [Bob, Carol, Alice]
- Z: [Carol, Alice, Bob]

Students (Receivers):

- Alice: [X, Y, Z]
- Bob: [Y, Z, X]
- Carol: [Z, X, Y]

Initially, all universities and students are not matched to anyone.

"Our step 1 is to compute the proposals from universities," said Maya. She wrote on the board. "X proposes to Alice. Y proposes to Bob, and Z proposes to Carol."

"This is like inviting exceptional students to opt for their universities," I said. "Precisely," said Maya, "but the students also have their list of preferences."

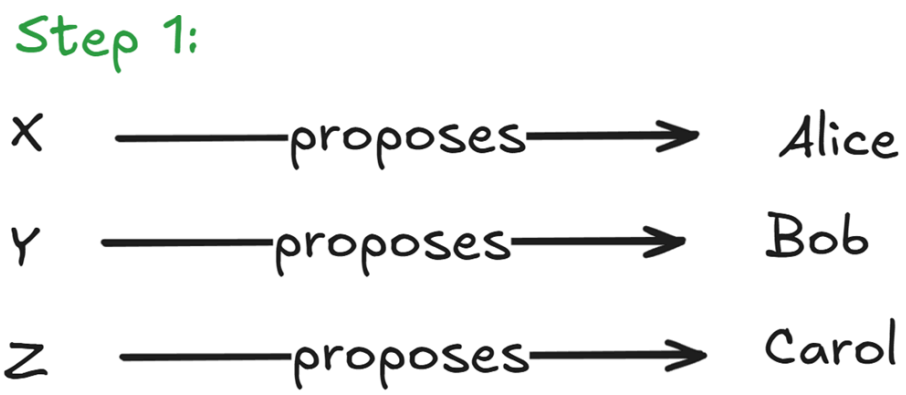

Figure 2.1 shows universities proposing their first preferences. X proposes to Alice, Y to Bob, and Z to Carol.

"So if the student has a different preference, they can reject the invitation?" asked Dev. "Correct. Let's see how that works in our example. Alice has a preference of X, Y, and then Z. Bob chooses Y, Z, and X, while Carol wants Z, X, and then Y," said Maya.

"So what's next? How can this be a stable match?" I asked.

"We move to step 2, where students have to decide whether to accept or reject the proposal from the university," said Maya.

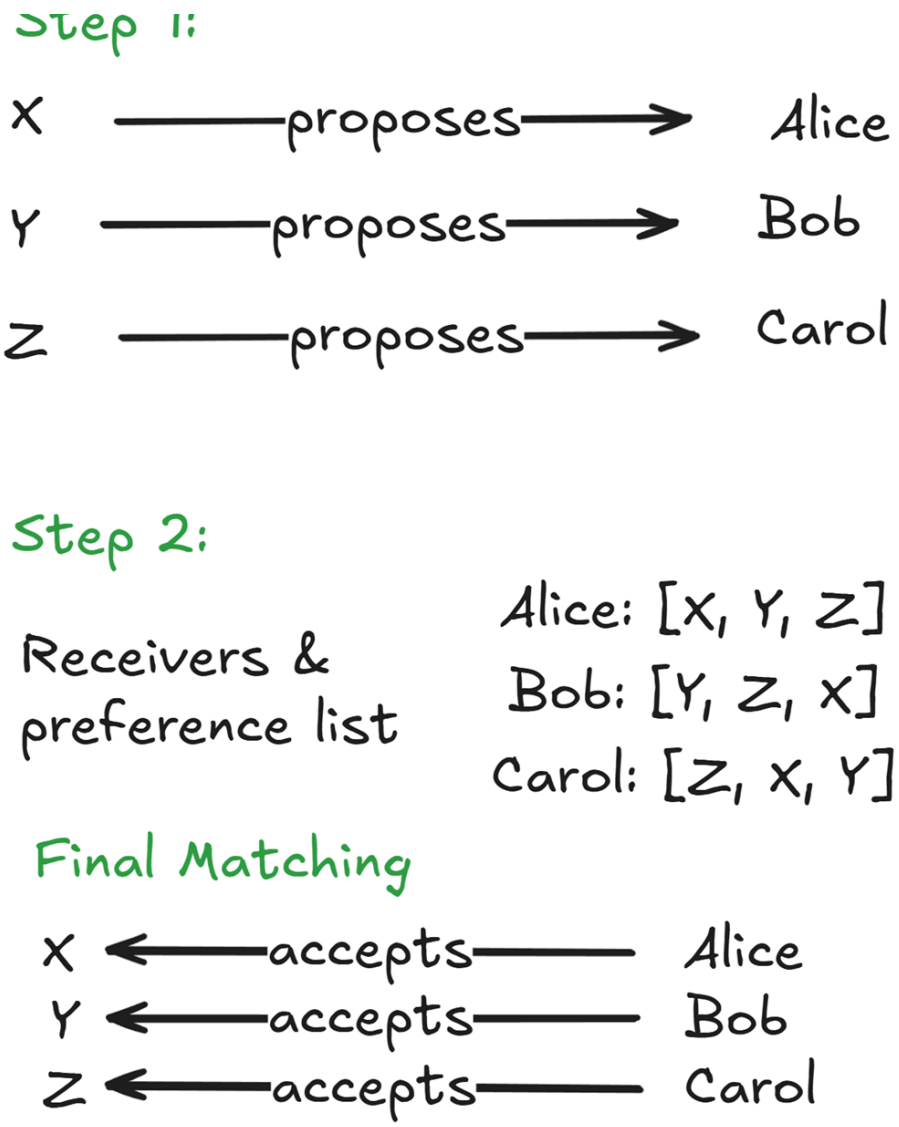

Figure 2.2 The receiver's preference list is checked, where Alice prefers X, Bob prefers Y, and Carol prefers Z. Final matching is completed once all students accept the proposal.

"Alice receives a proposal from X, and she accepts the offer since that is the only offer she has. Bob and Carol do the same. We can see everyone accepted and got the preference of their choice. This makes it a stable match," Maya concluded

This example walkthrough is very simple and ideal. The stable matching of students to universities was completed in one pass. In real scenarios, there would be rejections, switching, and advantages to the proposers. Here, the universities proposed and the students accepted, as the preference matched. Next, let's walk through an example of where the rejections appear and how they alter the final matching but still remain stable.

2.1.2 When preferences collide, rejections and rematching

In the example below, we start with students being the proposers and universities being the receivers. We change the preferences where the proposal doesn't always get an acceptance. This leads to rejections, and then we have to rematch them and still be able to keep the preference intact.

Students (Proposers):

- Alice: [Y, X, Z]
- Bob: [X, Y, Z]
- Carol: [X, Y, Z]

Universities (Receivers):

- X: [Alice, Bob, Carol]
- Y: [Bob, Alice, Carol]
- Z: [Carol, Alice, Bob]

"We start by students proposing to universities," she said. "Each round, they apply to their next preferred university. Universities only accept the best option they've received so far. The rest are politely rejected."

Maya wrote on the whiteboard.

- Alice → Y
- Bob → X
- Carol → X

"Now let's check who accepts whom," she said. "X has Alice. Y has Bob. Z has Carol. But there is a conflict." Dev raised a hand. "Yes! The university picks its top choice and rejects the rest."

She scribbled, "University X has Bob higher in preference. So, Carol is rejected."

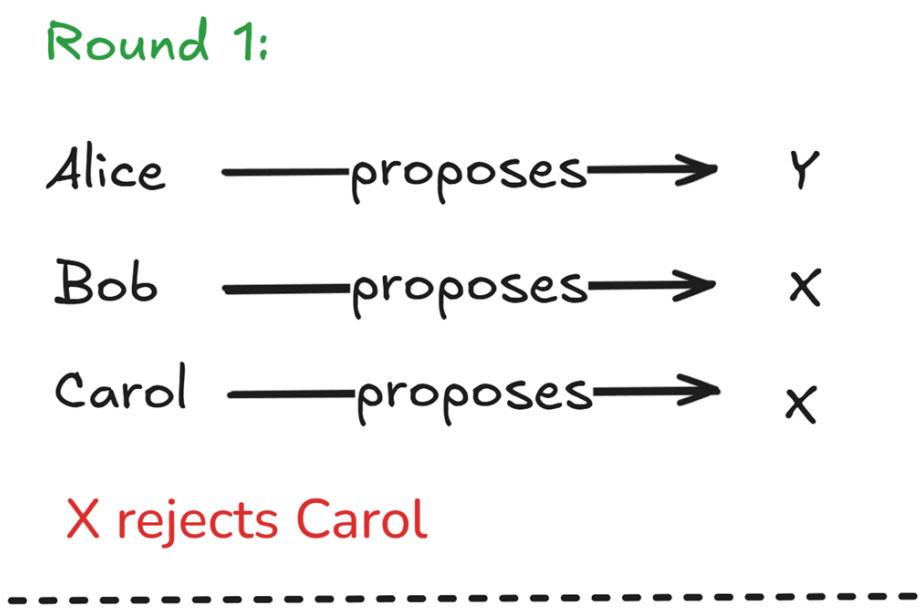

Figure 2.3 Round 1 where students are proposers. Alice, Bob, and Carol propose to their first choices from the list. A conflict is seen between Bob and Carol, where both students opt for university X. X prefers Bob over Carol and rejects Carol.

"We have to check what the next preference is for Carol," I said. "Correct! Carol proposes to University Y, her next preference." Maya confirmed.

"Hold on! That's another conflict since Alice is matched with Y already!" Dev said, confused.

"Yes! This is Round 2. We check the preference list of Y. Carol is the last preference for Y, and Carol is politely rejected again."

Maya updated the whiteboard:

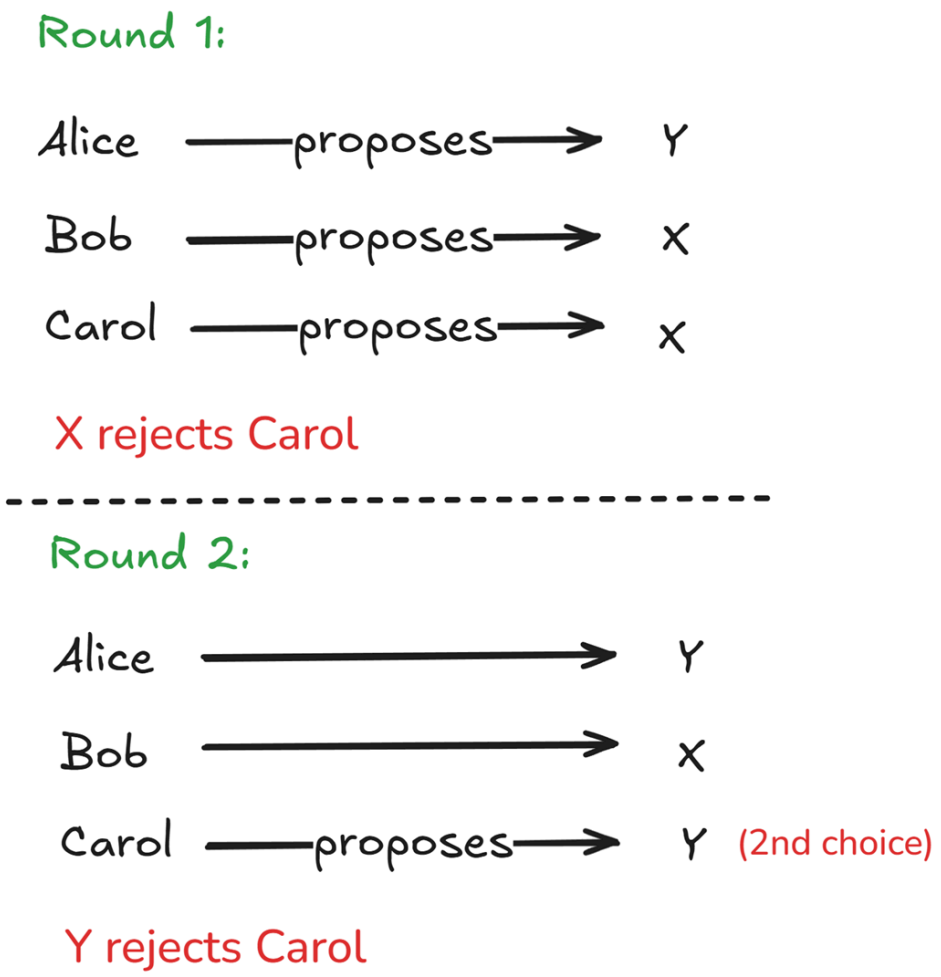

Figure 2.4 In Round 2, Carol proposes to her second choice while Alice and Bob keep their previous choices only. A conflict is seen again between Alice and Carol with university Y. Y prefers Alice over Carol, and Carol is politely rejected again.

"Oof," Dev said. "Carol isn't having a great time."

"She still has one option left: Z. Let's do Round 3." Maya said.

Proposals:

- Alice → Y
- Bob → X
- Carol → proposes to Z

In Z's preference list, Carol is the first choice. Z has no one yet, so it happily accepts Carol.

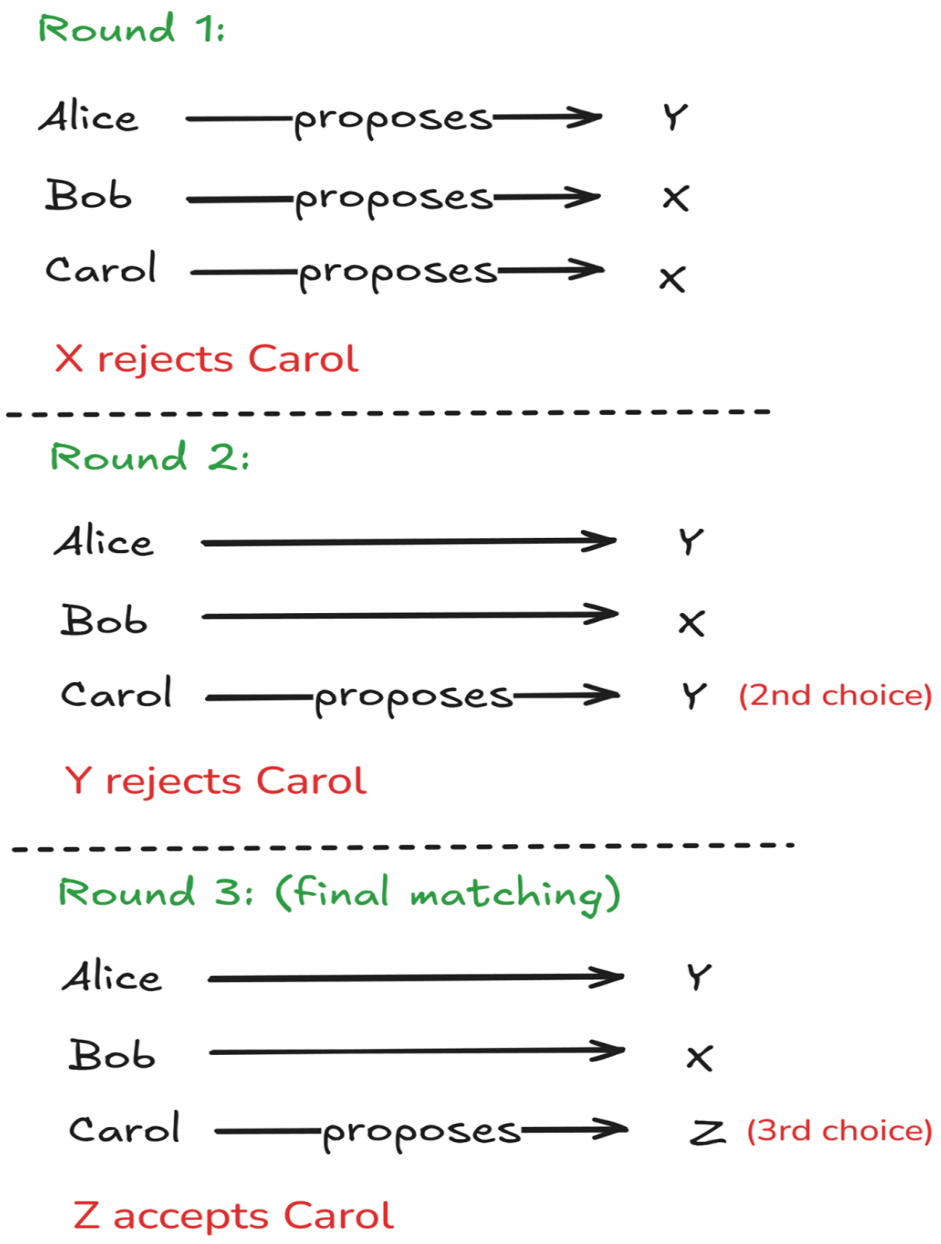

Figure 2.5 In Round 3, Carol is left with one choice, university Z. Z is not matched with anyone and prefers Carol too. Z accepts Carol, and matching is complete without conflict.

"Now every student has been accepted. No one has any preferred option left who would also want them more."

Dev looked up and said, "So now we need to check if this is stable."

- Alice → Y
- Bob → X
- Carol → Z

Let's verify:

- Does Alice prefer any university more than Y? No. She is matched with her first choice
- Bob is with X, his first choice too.
- Carol is with Z, her third choice. Does Carol prefer any university more than Z? Yes. But X and Y both rejected her. University Z ranks Carol as their first choice

"No one has an incentive to switch," I said. "That's stable."

"Exactly," Maya smiled. "We just implemented the Gale-Shapley algorithm." Dev nodded. "Clean. Deterministic. And actually kind of elegant." I grinned. "And just like in real life, sometimes you get your third choice... but at least you're not rejected."

"How does the final result differ if universities propose first?" I asked. Dev became thoughtful. "Let's keep the example as is and check it for ourselves."

Maya modified the universities as proposers.

Universities (Proposers):

- X: [Alice, Bob, Carol]
- Y: [Bob, Alice, Carol]
- Z: [Carol, Alice, Bob]

Students (Receivers):

- Alice: [Y, X, Z]
- Bob: [X, Y, Z]
- Carol: [X, Y, Z]

"Alright, so like before, initially students and universities are not assigned anything," Maya said.

She wrote on the board, Round 1.

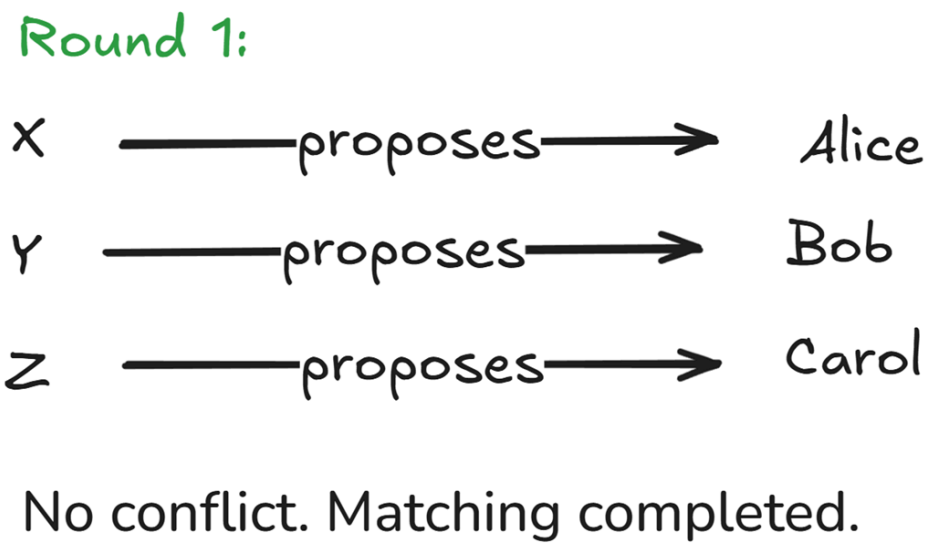

Figure 2.6 Proposers and receivers are interchanged. The matching completes in one pass itself. Universities get their first preference matched, but students do not control the match.

"Wait, there is no conflict. Does that mean this is the final matching?" I asked.

"Yes! But let's analyze what has happened. " Maya started explaining. "Alice preferred Y over X, and Bob preferred X over Y." Dev chipped in and continued, "But X and Y universities grabbed them first, and Alice and Bob had not received better offers from their first choices."

"Perfect explanation, Dev," said Maya.

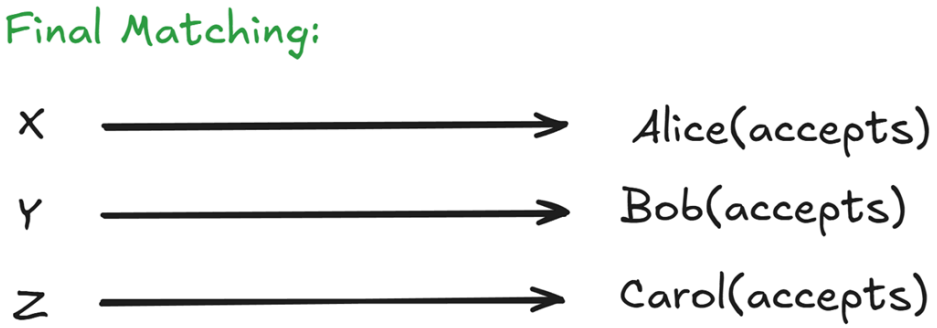

Figure 2.7 Students accept the proposals, as there are not better offers to choose from. In the final matching, students are matched to their second and third choices compared to when students were proposers.

Let’s verify:

- Does Alice prefer X over any other university? Yes. Alice prefers Y but cannot switch since universities are the proposers. So Alice got her second choice
- Bob got Y, while he preferred X, second choice again.
- Carol is with Z, her third choice.

“Interesting, and the matching is stable since nobody can switch," I said. “But the matching has changed significantly. When students were the proposers, Alice and Bob got their favorite choices...”

“...And when universities proposed first, Alice and Bob got their second choices. ” Dev completed my thought.”

Maya continued. “In one scenario, students are happy, and in the second scenario, universities got what they wanted even before students got a chance to choose. Students had to settle.”

In the entire walk-through of the example, we can note that the preference order is decreasing. It means once the proposal is rejected either by the university or students, the same preference is not considered again.

Listing 2.1 below shows the pseudocode that would translate to python code in listing 2.2.

Listing 2.1 Pseudocode for Gale-Shapley algorithm

```
Start with all students marked as free.
Keep an empty matching of universities to students.

Repeat while any students remain free:
    Take a free student and go through their universities in order of preference.

    For the next university the student hasn't proposed to:
        If the university has no match:
            Match the student with this university and mark them as matched.

        If the university already has a match:
            Check if the university prefers this new student over their current
match.
            If yes, replace the current match with this student and mark the old
match as free.
            If no, keep the current match and leave this student free.

When all students have been matched or exhausted their options, return the final
matching.
```

Listing 2.2 shows the code snippet based on the psuedudocode for Gale-Shapley algorithm. Pass the input like discussed to the function, and it will output the matches.

Listing 2.2 Code implementation of Gale-Shapley algorithm

```
def gale_shapley(students_prefs, universities_prefs):
   """
    Implements the Gale-Shapley stable marriage algorithm.

    Finds a stable matching between students and universities where no two
    participants would prefer each other over their current matches.

    Args:
        students_prefs (dict): Dictionary where keys are students and values
        are lists of universities in order of preference.
        universities_prefs (dict): Dictionary where keys are universities
        and values are lists of students in order of preference.

    Returns:
        dict: Dictionary mapping universities to matched students.
    """
    free_students = list(students_prefs.keys())
    matches = {}
    proposal_index = {student: 0 for student in students_prefs}

    while free_students:
        student = free_students[0]

        preferences = students_prefs[student]
        current_index = proposal_index[student]
        uni = preferences[current_index]

        proposal_index[student] += 1

        if uni not in matches:
            matches[uni] = student
            free_students.remove(student)
        else:
            current_match = matches[uni]
            uni_preference_list = universities_prefs[uni]

            if uni_preference_list.index(student) <
uni_preference_list.index(current_match):
                matches[uni] = student
                free_students.remove(student)
                free_students.append(current_match)
```

```
    return matches
```

2.1.3 Understanding the bias and complexity analysis

When students are the proposers, the students were allotted the universities that were the best for them, and that allotment never changes. It will only change if something better arrives. On the other hand, universities never got to choose and accept the proposal defensively.

When universities proposed first, they got the best possible student under the stable matching. The students do not control their stable match. This asymmetry is permanent and cannot be avoided. Basically, the proposer has the power and best outcome during the implementation. This is structurally biased towards the proposing side, but that is what makes it a theorem. It is stable and fair by definition, but not equal. In the following section of real-world scenarios, we will see an implementation called the National Resident Matching Program (NRMP), where students are the proposers and hence are protected from any manipulation of the result.

The time complexity of this algorithm in the worst case is O(N^2), where N is the number of participants (proposers and receivers). But how efficient is it? *Quadratic time complexity* in theory is excellent for computer science since it's deterministic and guarantees termination, and for this algorithm, it guarantees stable matching as well. During implementation, the value of N determines the efficiency of the algorithm. A higher value of N will use higher compute power compared to a lower value of N, as shown in table 2.1.

Table 2.1 Time Complexity Analysis for N ranging from 10^3 to 10^6

N (Participants)	Performance	Time Complexity
$N \leq 10^3$ (1,000)	Instant response	$O(10^6)$
$N \approx 10^4$ (10,000)	Fast (milliseconds)	$O(10^8)$
$N \approx 10^5$ (100,000)	Borderline but with optimization feasible	$O(10^{10})$
$N \geq 10^6$ (1,000,000)	Impractical without heavy hardware modifications	$O(10^{12})$

2.2 Real-World applications

Below are a few real-world applications where this algorithm is used. However, it is implemented based on the use case. The implementation we saw above is with only conflicts and rematching, but in the real world there are many other factors to be considered. The algorithms are designed to solve a problem.

2.2.1 Online dating and matchmaking

Online dating is also inspired by the same logic of stable matching. Although with time, it has evolved, the initial thought was very similar to the Gale-Shapley stable matching algorithm on how to match a set of participants based on preferences without unstable pairs.

In online dating, application users will explicitly express their preferences like age, interests, passion, job, location, and more. These preferences can be interpreted as ranked order lists that we saw in the example. The user who would initiate either a like or a message to the person would be the proposer, while there will be users as receivers accepting or rejecting this proposal. As mentioned, with time, the implementation has evolved where preferences are implicit as well, like swiping, liking, profile views, user activity, and more, making the algorithm go beyond its initial proposed framework.

Modern dating platforms like Bumble have added the additional feature of only women being able to message first, whereas other apps like Tinder do not have this. Stability of the matched pair still lies in the satisfaction of whether either of the preferences is violated, ensuring that matches are mutually acceptable.

2.2.2 Ride Sharing and Delivery Matching

Ride-sharing and delivery platforms such as Uber, Lyft, and food delivery services face a matching problem of pairing drivers with riders or delivery requests based on preferences and constraints.

In this context, drivers and riders (or delivery requests) can be treated as two sets of participants. Drivers may have preferences based on distance, expected fare, destination, or traffic conditions, while riders or delivery requests may implicitly prefer drivers based on estimated arrival time, ratings, or reliability. These preferences can be modeled as ranked ordered lists, even if they are computed algorithmically rather than explicitly stated.

The notion of stability in this scenario is modified where a driver or a rider is already matched, then there is no driver-rider pair who would mutually prefer others over the existing assignment. Rematching or reassigning in this business leads to cancellations, increased wait times, and poor customer feedback. Drivers and riders do have the option to cancel or modify the ride, but it often adds the cancellation charges. By minimizing these unstable matches, the system improves reliability and efficiency.

So, ride-sharing platforms can be considered matching algorithms in real-time, constraint-adapted, where stability is the key in reducing cancellations and ensuring a better experience.

2.2.3 Kolkata Paise Restaurant Problem (KPR Model)

KPR models a decentralized resource allocation scenario where many agents choose a limited set of resources based on preferences and previous outcomes. The fundamental formulation of this approach was customers and restaurants, where customers independently select restaurants while each restaurant has to serve only one customer. If more than one customer chooses a restaurant, then one is served and others are rejected.

Here customers have preferences generally over quality of food, service, and payoff (customer satisfaction), and restaurants implicitly serve customers rather than being idle. Compared with Gale-Shapley, stability matching here is the resource allocation is efficient, conflicts are minimized, and resources are used optimally.

The key difference between the Gale-Shapley and KPR models is that KPR does not rely explicitly on proposals and acceptances. Rather, the customers adapt based on repeated interaction and payoff values. Stability here is then defined as a state where customers do not overcrowd a particular restaurant but distribute evenly such that a customer has no incentive or improvement in payoff to go to a different restaurant. The rejection and rematching from Gale-Shapley are the payoffs getting updated, and customer switching is based on outcomes in KPR.

It can be clearly understood that KRP is an extension and modification of Gale-Shapley but decentralized, while stability is maintained by repetition and learning.

2.2.4 National Resident Matching Program (NRMP)

This nonprofit uses a variation of this algorithm to match the national medical talent to appropriate clinical training programs. They place efficient learners in US-based residency and fellowship training programs. They have their proprietary algorithm inspired by the stable matching algorithm.

In the NRMP system, applicants submit ordered lists of residency programs according to their preferences, while programs submit ranked lists of applicants based on their selection criteria. The matching process aims to produce a stable outcome in which no applicant–program pair would prefer each other over their assigned match. Such stability is critical, as unstable matches could lead to withdrawals, renegotiations, or unfair advantages for certain participants.

The algorithm used by the NRMP is a modified version of the applicant-proposing Gale–Shapley algorithm, which guarantees a stable matching that is optimal for applicants. This design choice reflects the program's objective of protecting applicants from strategic disadvantage while maintaining fairness and transparency across institutions.

2.3 Key Insights

Let's go through some key insights and properties to keep in mind about the Gale-Shapley algorithm while implementing.

2.3.1 Everyone gets matched

This algorithm guarantees that everyone is always matched when the number of participants proposing and the receivers are equal.

2.3.2 Stability over optimality

The matches are stable but do not necessarily mean they are an optimal match for everyone. The proposing side, as in the example above, tends to get more favorable outcomes than the receiving universities, this is an important insight while designing a system to decide who initiates the proposal.

2.3.3 Deferred acceptance principle

Matches are temporary, as seen in the example, until finalized. Proposals are either accepted or rejected so that instability is not aroused later, making it deterministic and fair.

2.3.4 Weighted and dynamic matching

The limitation of these algorithms is corrected with variations by adding costs and weights to the preferences to make the matches more reliable and adaptive. Weights add value to the preference, making it a more optimal match and not just stable. With the weights added, the preference list ordering is dynamic, which is closer to real-world use cases. In our example, the students can add weights like grades and costs like the distance of the university and perform the same algorithm.

2.4 Summary

- The Gale-Shapley algorithm performs steps to pair participants such that the matching is based on preferences and no pair wants someone other than their existing match.
- This algorithm is also called the Deferred Acceptance or Propose-and-Reject Algorithm.
- Each set of participants, like students and universities, has a ranked list of preferences in decreasing order. There are proposers and receivers who propose and accept/reject the proposals.
- Stable matching in this algorithm is defined as every participant in the matching cannot switch to a new participant based on their preference order.
- The algorithm favors the proposers who are matched with their best preference, while the receivers are matched with the preference they do not have control to change. This is by design and works best where the proposing side has to be favored in the result.
- The output guarantees termination, and every participant is matched at the end of the algorithm. The time complexity in worst case is O(N^2)
- Everyone always gets matched, and all matches are stable. Until finalized, matches are temporary and decided later, making it deterministic.
- Online dating, ride sharing, delivery platforms, and college admissions are a few real applications where the Gale-Shapley algorithm is used fundamentally.

3 Hungarian Algorithm

This chapter covers

- Solving the assignment problem with the Hungarian algorithm
- Constructing the cost matrix and executing the row-column reductions
- Analyzing the algorithmic optimization from O(n!) to O(n^3)

The *Hungarian algorithm*, also called the *Kuhns-Munkres algorithm,* is named after Harold W. Kuhn and James Munkres. This is a classic assignment problem to find an optimal one-to-one matching between two sets. This combinatorial optimization algorithm's purpose is to minimize the total cost or maximize profit by assigning n workers to n tasks.

If the number of workers and the number of tasks are equal, then it's a *balanced assignment problem*. After looking at a string matching algorithm in the previous chapter, in this chapter, we will see a balanced cost minimization assignment problem. The time complexity of the algorithm is O(n!), but it is further optimized to O(n^3) called polynomial time.

We use an n x n matrix where n is the size of the matrix with rows as workers and columns as tasks, and the matrix[i][j] has the cost we need to minimize to allocate the workers respective tasks.

3.1 Allocation problem in a nutshell

We are given a set of workers and tasks. Each worker can perform any task but incurs a fixed cost to complete the task based on distance, resources, or time. Our objective is to create a 1-1 mapping that optimizes the global cost.

Dev looked up and smiled. "Aniket," he said, "did you start working on the homework assignment that the professor gave us?"

"Not yet!" I said. "I just finished it. Phew!" Dev grinned.

”Already?? Wait… isn’t it like a matching or mapping algorithm?” I asked, confused. “Yep. It is. Do you guys want to go through the question together?” Dev said excitedly.

“Alright,” said Dev, and started cleaning the whiteboard.

Dev started. “Before we start solving the problem, let me show you guys a simple example I tried first. Let’s say we are given three chores, cleaning the kitchen, cleaning the floors, and cleaning the windows. So the algorithm of assigning a set of tasks and then a set of workers to do these tasks with a cost is now an allocation problem. Our job is to make sure we map these workers to tasks in a way where the cost is minimum.”

I said, “Wait… Wait… so one worker does one task, and it costs us something, and we have to make the pairing cheapest somehow?”

Figure 3.1 Cost matrix is created with the rows representing the workers and the columns representing the tasks. The values in the matrix denote the cost to complete the task. The cost can be replaced by other factors based on the use case like time taken, amount of work, or any other measurable value.

“Yes, correct,” said Dev.

I asked, “Don't we just loop through every possible pairing of the three workers and the three tasks? So if we have 3 workers with 3 tasks, we will have to do matching every time, calculate the cost, and again perform the same steps to compare the newer cost? Isn’t this a bit too much calculation?” I said, thinking about the way to find the minimum cost mapping.

Dev shook his head. "That's the brute-force trap. For 3 workers and 3 tasks, that's 3x2x1, meaning 3! (3 factorial), which is 6 permutations. Easy. But what if a company is matching 100 delivery drivers to 100 routes? The number of permutations is 100!. That is a number with 157 zeros. And that's why we have this algorithm, which does it in simple steps with row and column addition and subtraction. The question in the assignment is slightly different. But if we understand this example, we can solve that too then." Dev said while making a matrix on the whiteboard.

Dev continued, "Since we need equal workers and tasks, the Kunhes-Munkres algorithm requires a square cost matrix of 3 x 3."

"Our first step is row reduction," Dev explained. "We find the minimum cost in each row and subtract it from every element in that row. Alice's cheapest rate is 9. Bob's is 9. Carol's is 8. We subtract those from their respective rows."

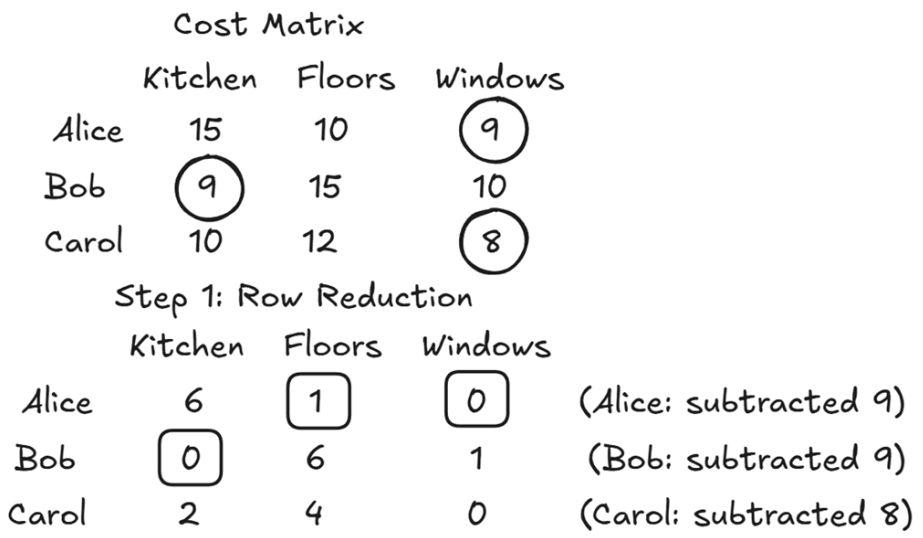

Figure 3.2 The circled values in the cost matrix are the minimum values in respective rows. Row reduction is performed, and each value in the row is subtracted from the minimum value in that row.

"Our second step is column reduction," Dev continued. "We do the exact same thing, but vertically. The minimums for the columns are 0, 1, and 0. We subtract those."

"Here is the magic trick," Dev said. "We draw the minimum number of straight lines horizontal or vertical, required to cover all the zeroes. If the minimum number of lines equals n (which is size of the matrix of min(n,m) for a rectangular matrix(n,m)), we have found our optimal assignment."

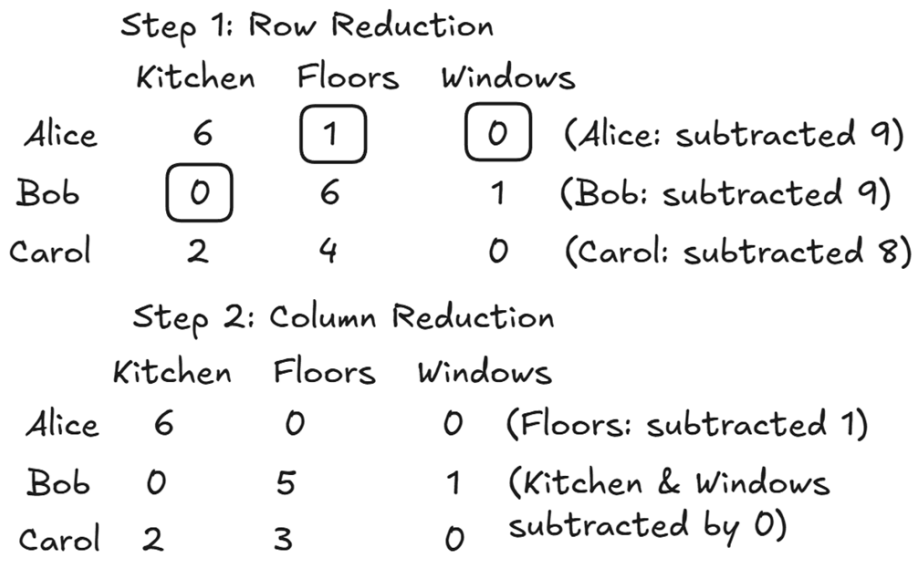

Figure 3.3 The squares in step 1 are the minimum in each column of the cost matrix after row reduction. These minimum values are used, and column reduction is performed. The column Floors is only modified and subtracted by 1.

Dev drew lines through Column 0, Row 0, and Column 2. "Three lines, equal to the size of the matrix" Maya noted. "So we are done?" "Exactly," Dev confirmed. "The zeroes indicate the absolute cheapest assignments without any overlap."

Bob gets the kitchen (Row 1, Col 0). Carol gets the Windows (Row 2, Col 2). Alice gets the Floors (Row 0, Col 1).

"Total cost: 9 + 8 + 10 = 27 dollars," I calculated. "That's the mathematically cheapest way to get the apartment clean."

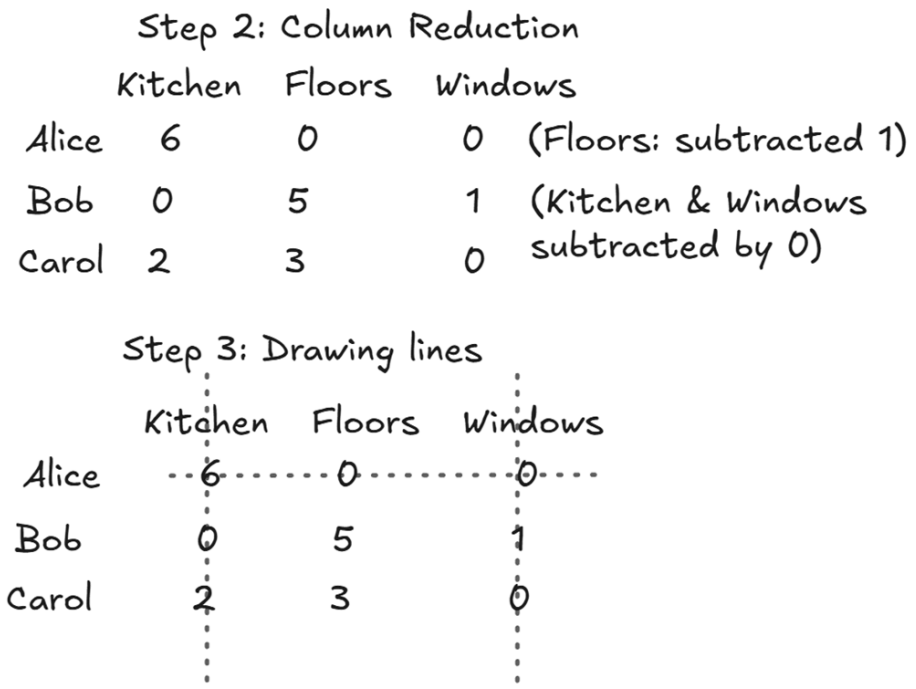

Figure 3.4 In our step 3, we start drawing lines across the rows and columns that contain 0. Lines across column 0, column 2 and row 1 cover all the zeroes.

The example we saw is the easy and happy path. In the real world, the algorithm rarely resolves on the first try. We continue with another example, which resembles engineering tasks to be assigned to a team of workers.

Dec continued by drawing a larger grid on the board. "This is our assignment question. Imagine we are managing a 4-person engineering team, and we have 4 independent software modules to build. Each developer has to give an estimate how many hours it would take them to build a specific module. Developer 0, for example, is great at databases and estimated a total of 8 hours to complete module 2. But for the same developer, it takes 19 hours to develop the UI module 0."

He filled the grid. "The numbers inside the matrix represent the time taken by each developer to develop the respective modules. The goal is to minimize the total hours and assign one module to each developer."

Cost Matrix

	Mod 0	Mod 1	Mod 2	Mod 3
Dev 0	10	19	8	15
Dev 1	10	18	7	17
Dev 2	13	16	9	14
Dev 3	12	19	8	18

Figure 3.5 The cost matrix consists of four developers from 0 to 3 and four modules, Mod 0 to Mod 3. The values within the matrix represent the number of hours required for the developer to complete the module.

“Let’s start the row reduction again. "Aniket, can you tell us how to do this step?" Dev said. I looked at the cost matrix on the board and said, “We have to find the lowest value for each row representing the developer time and subtract it across the row.

For developer 0, its module 2 is 8 hours. For developer 1, it's 7 hours, developer 2, it's 9 hours and for developer 3, it’s 8 hours.” I wrote the new matrix on the board after row reduction.

“That is correct, Aniket,” Dev said and looked at Maya. “Do you want to try the column reduction, Maya?”

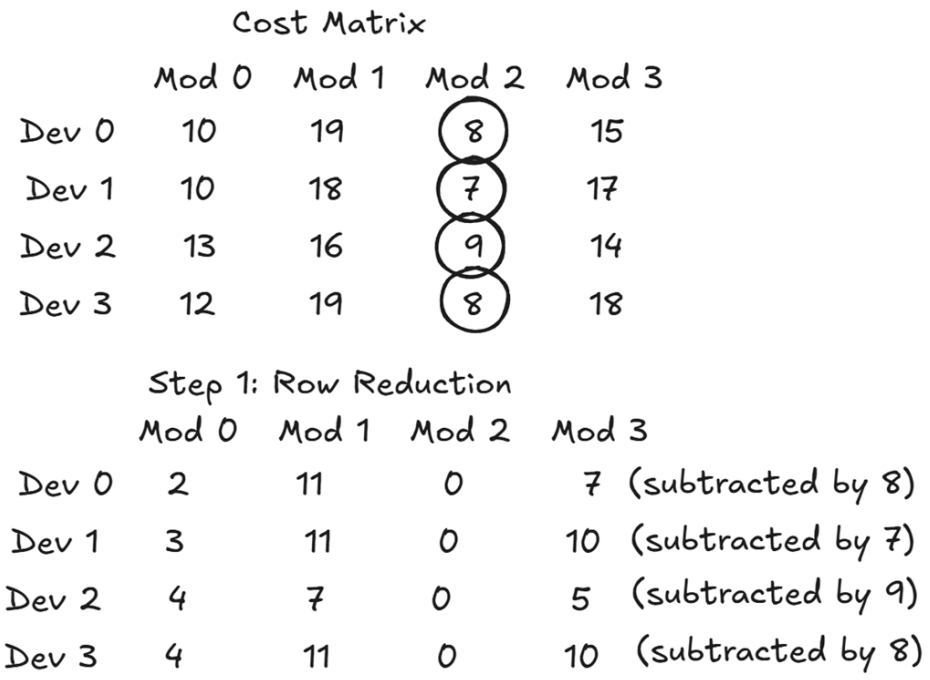

Figure 3.6 The circled numbers in the matrix are the lowest values in each row to be subtracted across the row. The row reduction is performed and shown in the step 1 as the modified matrix.

"Yes. Looking at the columns, the minimum values representing the modules are 2, 7, 0 and 5. Subtracting these values gives us the column reduction matrix." Maya wrote the matrix on the board. "That's perfect. Now we start drawing lines the way we did before," Dev said. "There are zeroes in column 2 and row 2, so we draw two lines across them. We also see a zero in column 0 and draw a straight line there too."

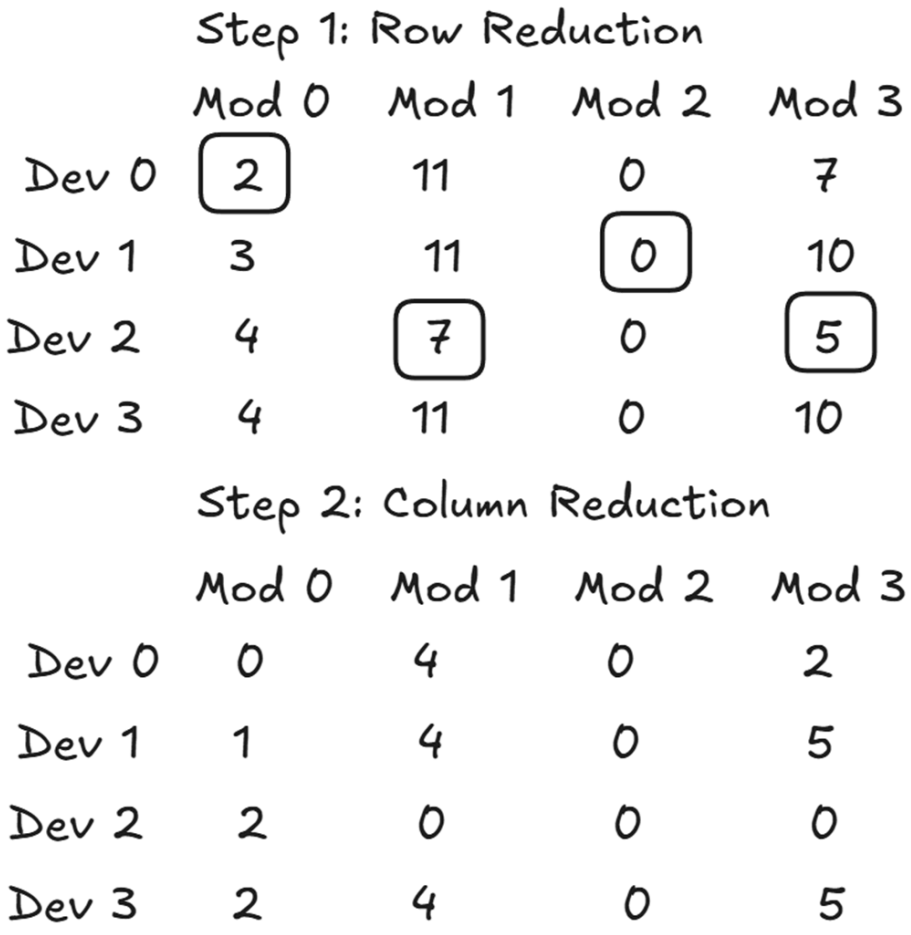

Figure 3.7 The marked square boxes in step 1 are the lowest value in each column to perform column reduction. The values are subtracted across each column and the resultant modified matrix is shown in step 2.

"Only three lines? One less than the size of the matrix" I asked. "This is where it gets interesting. We cannot make a 1-1 assignment now to allocate developers the required modules, since there are 4 developers. This is the bottleneck of the Hungarian algorithm," Dev explained.

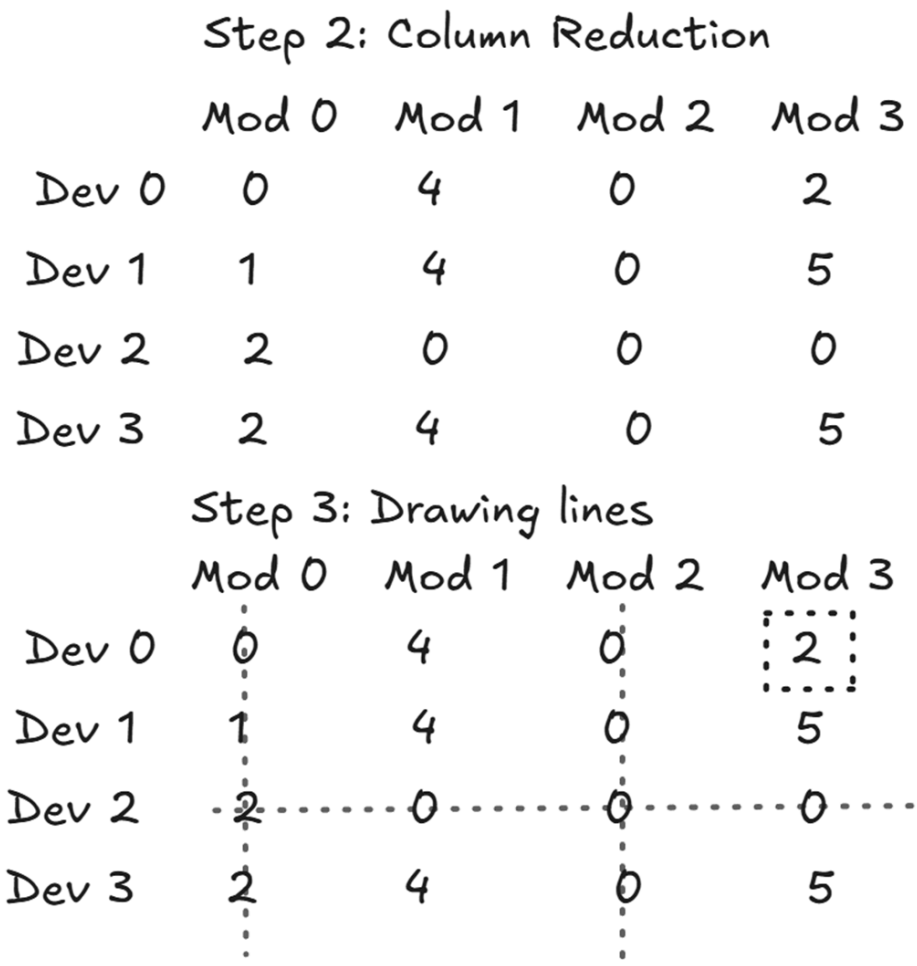

Figure 3.8 Lines are drawn across the matrix in step 3, covering column 0, column 2, and row 2. But only 3 lines are drawn to cover all zeros and do not match the total developer value of 4. A 1-1 assignment is not possible at this stage.

"So, what do we do now? Does the algorithm not work in this scenario?" Maya asked. "It does," Dev clarified. "We need to shift the weights of the values mathematically so that new zeroes are exposed, but they won't ruin the relative cost matrix ratio."

Maya leaned forward. "How do we shift the zeroes?" "Three strict rules," Dev said, writing them next to the matrix.

- Find the smallest uncovered number.
- Subtract it from all uncovered numbers.
- Add it to any number where two lines intersect.

“Look at the numbers our lines didn't touch,” Dev pointed out. “The smallest uncovered number is 2 (at Row 0, Column 3). We subtract 2 from all uncovered blocks, and we add 2 to the intersections at [2,0] and [2,2].

“Let's try drawing lines now,” Dev said. He drew a horizontal line across Row 0, a horizontal line across Row 2, and a vertical line down Column 2. "Still three lines," I said, frustrated. "It didn't equal 4." "Which means the matrix is fighting us," Dev said, not missing a beat. "So we hit it again. Same rules. What is the smallest uncovered number now?"

I scanned the remaining numbers. "It's 1 (at Row 1, Column 0)." "Do the shift," Dev instructed. I subtracted 1 from the uncovered blocks and added 1 to the intersections at [0,2] and [2,2].

"Now draw the lines, Aniket," Dev said. I looked at the board. The zeroes had completely scattered. To cover them all, I was forced to draw lines across Column 0, Column 2, Column 3, and Row 2. “Four lines,” I said, dropping the marker. “It finally equals n.” “Optimal state achieved,” Dev said. “We now have enough independent zeroes to make a flawless 1-to-1 mapping.”

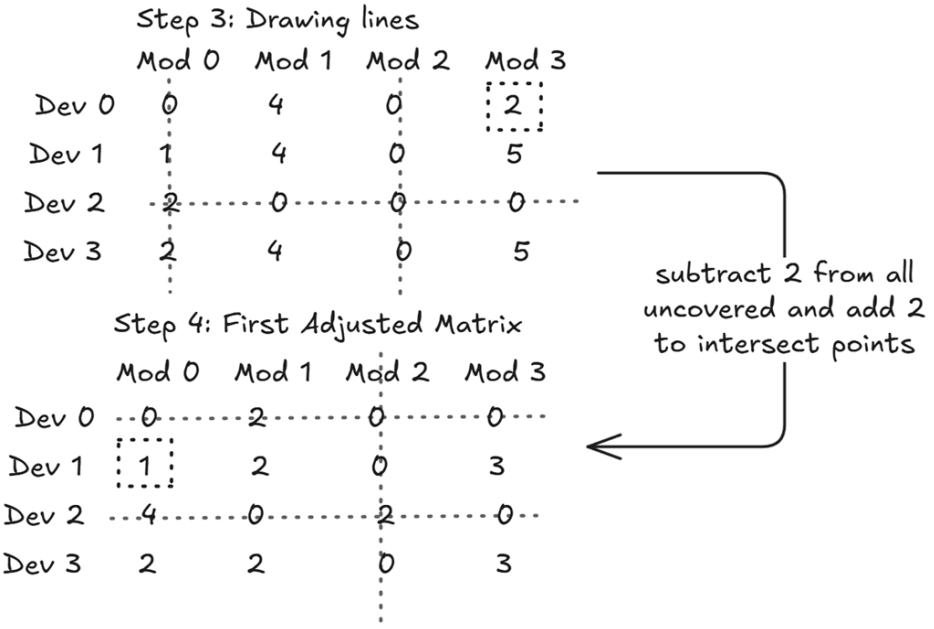

Figure 3.9 The dotted box in step 3 is the value of the smallest number uncovered. This number is subtracted from all the uncovered values. The same value is added to the intersection of the lines i.e. is [2,0] and [2,2]. The lines are drawn again covering row 0, row 2, and column 2. The lines still are still 3 and do not equal to the number of developers which is 4.

"Okay, let's map the developers to the modules. We look for rows or columns that only have one zero first to force our hand." "Look at Dev 3," Maya pointed out. "They only have one zero, located in Mod 2. So Dev 3 must take Module 2. Because Dev 3 took Module 2, Dev 1 can no longer use their zero in Column 2. That leaves Dev 1 with only one option: the zero in Column 0. Dev 1 takes Module 0. Because Dev 1 took Module 0, Dev 0 can no longer use it. Dev 0 is forced to use their other zero in Column 3. Dev 0 takes Module 3. That leaves Dev 2 with the final zero in Column 1. Dev 2 takes Module 1."

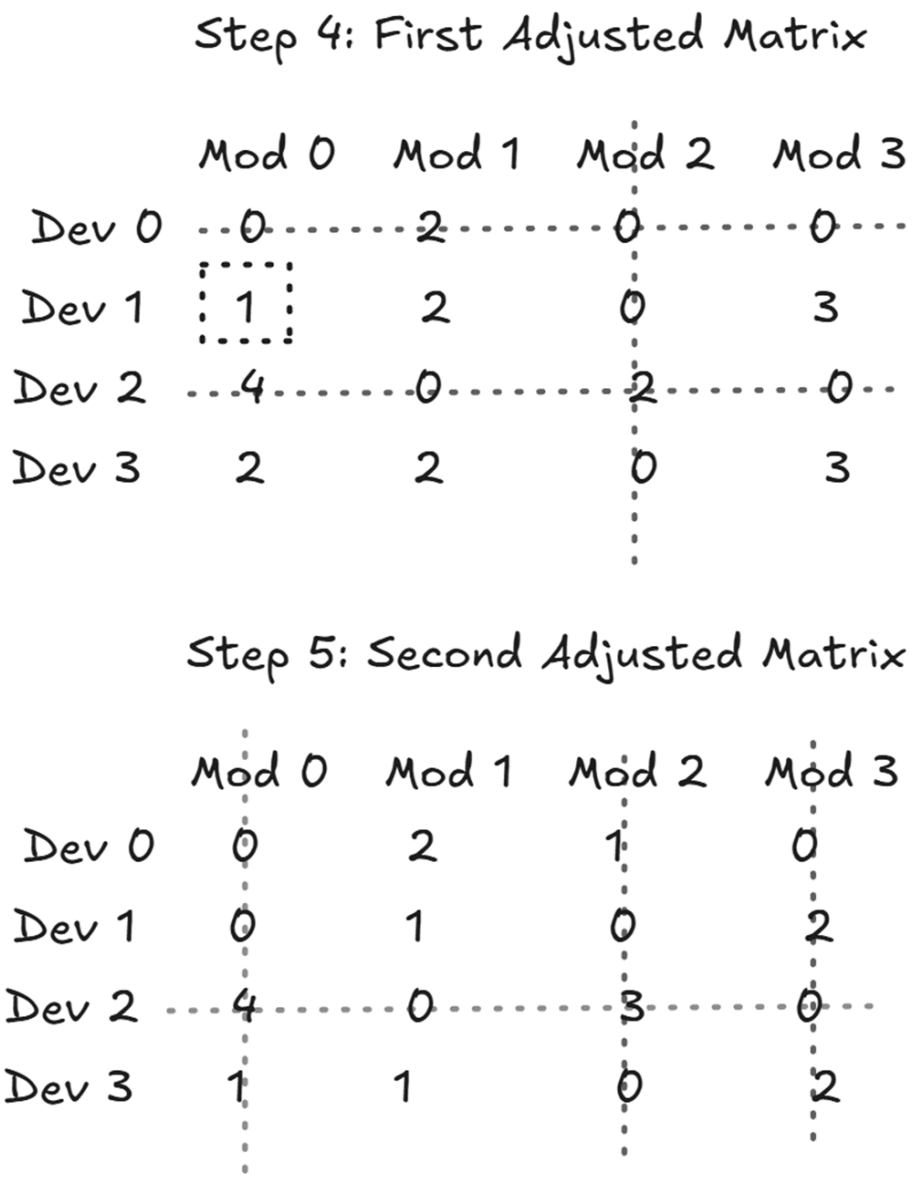

Figure 3.10 The same step is repeated again. The dotted box in step 4, is the smallest uncovered value. This value is subtracted from the matrix and added to the intersection at [0,2] and [2,2]. The lines are drawn again across column 0, column 2, column 3, and row 2. The lines equal 4 now, and a 1-1 assignment is possible.

"Perfect," I said. "Now, what is the actual cost? We have to look at the original cost matrix." Maya cross-referenced the assignments with the original 4x4 matrix.

Developer 0 maps to Module 3: 15 hours

Developer 1 maps to Module 0: 10 hours

Developer 2 maps to Module 1: 16 hours

Developer 3 maps to Module 2: 8 hours

"Total cost," Maya said, tapping her pen against the paper. "49 hours. Out of all 24 possible combinations, 49 hours is the absolute mathematical minimum it will take to build this software."

I sat back, finally understanding it. "So we just bypassed calculating 24 combinations by doing some basic addition and subtraction."

Listing 3.1 is the pseudocode, using the inbuilt libraries like scipy and without it as well. The python implementation although is using scipy.

Listing 3.1 Pseudocode for Hungarian algorithm

```
Option 1: Use an efficient solver if available like scipy
    Convert the cost matrix into the solver format.
    Ask the solver to find the minimum-cost pairing of workers to tasks.
    The solver returns one worker-task match per worker.
    Aggregate the chosen pairs and compute the total cost.
    Return the optimal assignment and total cost.

Option 2: Solve without a solver
    Start with no assignment and the best cost unknown.
    Generate every possible way to assign workers to distinct tasks.
    For each candidate assignment:
        Add the cost of each worker-task pair in the assignment.
        Compare the calculated cost with the best known cost.
        If this assignment is cheaper, remember it as the new best.
    After testing all possibilities, return the lowest-cost assignment and its
cost.
```

Listing 3.1 shows the implementation using numpy and scipy python libraries. The implementation takes input as the cost matrix.

Listing 3.2 Code implementation of Hungarian algorithm

The task of matching workers to jobs to minimize total cost is formally known as the *Linear Sum Assignment Problem (LSAP)*. While several algorithms can solve an LSAP, the historic gold standard for understanding its mechanics is the Hungarian Algorithm. However, the scipy code implementation shown below under the hood uses an algorithm called Jonker-Volgenant algorithm, which is efficient in memory caching and real world hardware.

```
Alice is assigned to Floors (Cost: 10)
Bob is assigned to Kitchen (Cost: 9)
Carol is assigned to Windows (Cost: 8)
Total Minimum Cost: 27

import numpy as np
from scipy.optimize import linear_sum_assignment

def hungarian(cost_matrix, workers, tasks):
"""
    Solves the assignment problem using the Hungarian algorithm.

    Uses scipy's linear_sum_assignment if available
    (efficient implementation),
    otherwise falls back to brute force method.

    Args:
        cost_matrix (list of lists): 2D list where cost_matrix[i][j]
        is the cost
                                    of assigning worker i to task j.
        workers (list): List of worker names/identifiers.
        tasks (list): List of task names/identifiers.

    Returns:
        tuple: (matches, total_cost) where matches is list of
        (worker_idx, task_idx)
               pairs and total_cost is the minimum total cost.
    """

    cost_array = np.array(cost_matrix)
    row_ind, col_ind = linear_sum_assignment(cost_array)

    total_cost = 0
    print("--- O(n^3) Solution ---")
```

```
for worker_idx, task_idx in zip(row_ind, col_ind):
    cost = cost_matrix[worker_idx][task_idx]
    total_cost += cost
    print(f"{workers[worker_idx]} is assigned to
    {tasks[task_idx]} (Cost: {cost})")

print(f"Total Minimum Cost: {total_cost}")
```

3.2 Real-World applications

The Hungarian algorithm is a combinatorial optimization algorithm for finding the optimal solution. While the algorithm performs 1-1 matching optimally, in real scenarios and in a production environment, the implementation differs a lot where the workers and tasks are not always equal. However, the algorithm is the foundational engine that powers most modern matching systems. Let's look at some applications to better understand its importance.

CREW SCHEDULING IN AIRLINES

Airlines start losing money when flights are delayed or cancelled by crew unavailability. This creates poor customer experience. The scheduling of resources like crew, pilots, and other airline staff to appropriate tasks is crucial. Allocation algorithms like the Hungarian algorithm tend to perform well here. A cost matrix can be created for types of workers, like pilots, flight attendants, and ground staff, with their respective tasks with values within the matrix like airline rules, overtime pay, number of flights, leaves, and more.

This guarantees minimal operational costs for the assignment and allocation across the entire fleet. We avoid the greedy mistake of optimizing a situation for one flight but making it worse for other flights. The significant problem here is scaling since there are many airports, multiple cities, and thousands of staff with many responsibilities. The cost matrix is not one but tends to evolve frequently with various tasks changing quickly.

In complex systems like these, airlines engineer the foundational algorithm by using unbalanced matrix variation where the number of workers is higher than the number of tasks or flights to manage unexpected scenarios. Corporations customize the implementation and performance of these algorithms to cater to their businesses and output the maximum value.

JOB SCHEDULING IN DISTRIBUTED CLOUD SYSTEMS

Let's look at a real application of the Hungarian algorithm and its variation in an engineering field. With many cloud providers emerging in the tech market along with Amazon Web Services (AWS) and Google Cloud, their hardware systems receive thousands and millions of requests and jobs per minute. These providers need to make sure the jobs are mapped to physical servers and respond back to the user with minimum time. Basically, requests mapping to physical servers is the actual allocation.

Here, a cost matrix can be created with the rows as the incoming requests or jobs and the columns as the available physical servers. The values in the matrix can be the costs of the available CPU hardware, network latency, memory, and request type. The advantage of such allocation makes sure the load is distributed evenly, costs are minimized, and utilization of the hardware is maximum. However, it doesn't take into account ad hoc failures in vast data centers. There is a need to perform recalculation of the assignment if a critical task is allocated but the server fails later.

3.3 Key Insights

Now that we have already seen the implementation of the Hungarian algorithm. Let’s dive into some nuances and insights that help us as engineers.

ALWAYS OPTIMAL AND DETERMINISTIC

The Hungarian algorithm gives an optimal minimum-cost matching solution. It is deterministic, as there is no overlap or variance and dependency. If the requirement is one-to-one pairing, this algorithm is the ideal choice.

SCALING CAN BE AN ISSUE

The time complexity is $O(n^3)$, and for larger n, this can be a problem. Time complexity is a framework to measure the performance of an algorithm based on the changes in the input. Recall the section on analyses of algorithm concepts from chapter 1. All workers have to be mapped to a task, whether they can do the task or not, maybe even with a fake cost. In real life, there are multiple factors to be considered, like worker fatigue and overlap due to unforeseen issues.

BIPARTITE GRAPH REPRESENTATION

"Bipartite" means consisting of two parts. In graph theory, a bipartite graph represents a network where nodes are divided into two sets, and every edge strictly connects a node from one set to another node in the second set. The cost matrix is just a representation of a graph with workers as one set, U, and tasks as set V, and the edge connecting them, E = edge(U,V), is the actual cost. The graph is then a bipartite graph since we can divide it into two independent sets. The bipartite graph for the first example shown above is shown in Figure 3.11.

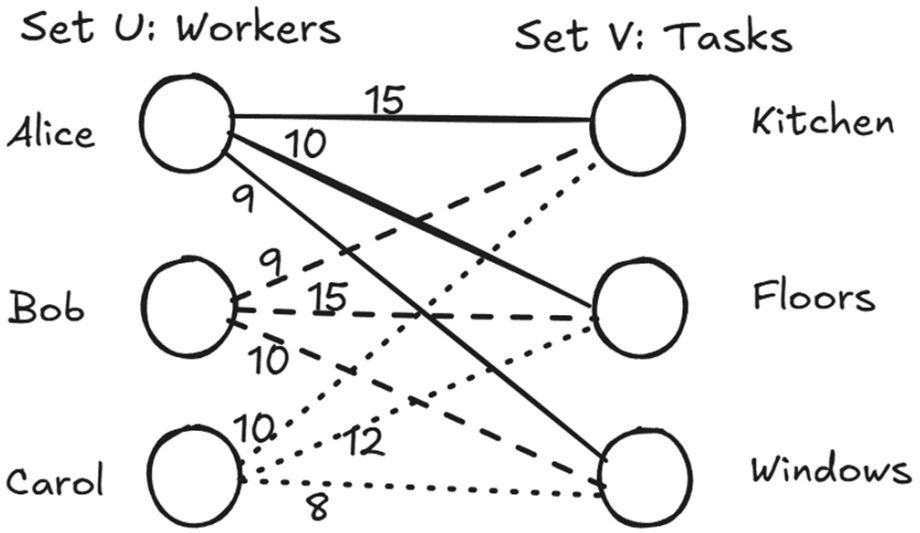

Figure 3.11 The bipartite graph shows two sets, U and V. Every edge from the vertex (Alice, Bob, Carol) is connected to another set V vertices (Kitchen, Floors, Windows) with the value of the edge as the costs.

3.4 Summary

- The Hungarian algorithm is an allocation problem to map an equal number of workers to tasks. The objective is to minimize the cost to obtain the optimal mapping.
- If the algorithm is not used, then manual combinations yield factorial time complexity. At enterprise scale, with 100 workers and 100 tasks, this creates an incomputable number of combinations.
- The cost matrix is created where the rows are the workers and the columns are the tasks. With the implementation of the Hungarian algorithm, the time complexity reduces to $O(n^3)$ from the factorial O(n!).
- The algorithm performs row reduction and column reduction. The algorithm subtracts the minimum value from every row, followed by subtracting the minimum value from every column. This safely alters the relative weights to expose zero-cost assignment opportunities.
- To verify if a complete assignment is possible, the algorithm requires drawing the absolute minimum number of straight lines to cover all zeroes in the matrix. If the minimum lines equal n, an optimal state is reached.
- If the lines drawn are strictly less than n, the matrix is bottlenecked. The algorithm forces a mathematical shift by locating the smallest uncovered number, subtracting it from all uncovered cells, and adding it to any cell where two lines intersect. This process loops until the lines equal n.

- Once the optimal state is achieved, assignments are locked in by selecting independent zeroes (starting with rows/columns that only possess a single zero). These mappings are cross-referenced with the initial cost matrix to calculate the final minimum cost.
- Real-world applications like crew scheduling use the Hungarian algorithm for the assignment of crew and staff to related tasks. Job scheduling in distributed systems is another application to map the incoming jobs or requests to physical servers.

3.5 References

1. Harold W. Kuhn, "The Hungarian Method for the assignment problem", Naval Research Logistics Quarterly, 2: 83–97, 1953. Kuhn's original publication.
2. YouTube -https://youtu.be/aocsXmWwAZQ?si=SFakJwJVqDFkJwUa
3. Algorithm in action -https://www.hungarianalgorithm.com/solve.php

4 Rabin–Karp Algorithm

This chapter covers

- Explaining a string matching algorithm to search a pattern within a given text
- Constructing robust hash functions to prevent collisions
- Applying rolling hashes to bypass the O(nxm) brute-force computational bottleneck.
- Analyzing algorithmic complexity, worst-case degradation, and performance.

In the previous chapters, we implemented and examined the Gale-Shapley Algorithm. The Gale-Shapley algorithm performed matching between two equal sets of participants. The matching should be stable based on the preferences provided. We covered the real-world applications of the Gale-Shapley algorithm along with its efficiency, tradeoffs, and key insight. This chapter focuses on one of the most widely used and discussed string-matching algorithms, called the Rabin-Karp algorithm.

The *Rabin–Karp algorithm*, created by Richard M. Karp and Michael O. Rabin in 1987, is a string-matching algorithm that matches a pattern inside a large string in a fast and efficient way. Instead of comparing every character of the pattern with every character of the text, the algorithm uses a hash function to convert the pattern into a numerical value. It then computes the hash values of different windows in the large input string and simply compares numbers instead of doing character-by-character matching. This is what makes the algorithm efficient, numeric comparison is much faster than comparing multiple characters.

The traditional way of comparing two texts is to check each character of the text with the pattern. For a pattern with many characters, it becomes highly inefficient. This is because there is a possibility that the last character of the pattern does not match the last character of the text. We have utilized an enormous amount of computation just to conclude that the pattern is not part of the text only after looking at the last character.

There are different types of hash functions that can be used, and we will see one such example below. A commonly used idea is the *rolling hash function*, which allows us to slide across the string without recalculating the entire hash value every time. It updates the hash using the previous hash in constant time. For example, a hash function can convert "Manning" into some numeric value like 837 based on its logic. However, different strings can sometimes end up with the same hash value, which is called a collision. A well-designed hash function reduces the chances of these collisions and improves accuracy.

On average, the Rabin-Karp algorithm is highly efficient. However, its performance starts to degrade when a weak hash function generates too many collisions. When continuous collisions occur, the algorithm has to verify each character within the window. Furthermore, the worst-case time complexity plummets to O(nxm), which is the mathematical product of text length and pattern length. The *time complexity* of the algorithm is a framework to evaluate how the algorithm scales at runtime when the volume of input data increases. The best and average time complexity of Rabin-Karp is O(n+m), where n is the length of the text and m is the length of the pattern. This is achieved because rolling hash takes O(1) at each n position.

Table 4.1 Comparison table of various string matching algorithms

Algorithm	Advantages	Disadvantages	Use cases
Rabin Karp	Converts strings to fast numeric hashes, allowing O(1) window sliding.	Degrades to catastrophic O(n x m) if weak hashes resulting in constant collisions	Plagiarism detection and bioinformatics
Knuth-Morris-Pratt	Guarantees deterministic O(n) time by creating Longest Prefix Suffix array	Wastes O(m) memory and provides zero speed advantage if the pattern lacks internal repetition.	Real time data streaming and log analysis
Boyer Moore	Achieves sub-linear performance by scanning right-to-left and skipping massive chunks of mismatched text.	Highly complex to implement the "Bad Character" tables, and degrades on highly repetitive text.	Text Editors and Integrated Development Environment (IDE)
Aho-Corasick	Constructs a state-machine to search for thousands of different patterns simultaneously in a single O(n) pass	Imposes massive memory overhead to build and store the deterministic finite states	Mass virus scanning against massive signature databases

Despite this, Rabin-Karp is widely used because of its simplicity and power. It is especially good when we need to search for multiple patterns at once, detect repeated substrings, or scan massive text files quickly. With improved hashing techniques, it remains a practical and elegant solution for real-world string search tasks.

4.1 String matching algorithm

Now we will cover the Rabin-Karp algorithm step-by-step through a practical example. We will trace search patterns against a single text string to expose the internal mechanics of how Rabin-Karp searches for and matches data.

The room was dim, and I was staring at my laptop screen. Maya asked, “What are we thinking about? More dynamic programming?”

I chuckled, “Actually… string matching. Ever heard of Rabin-Karp?”

Maya looked up. “Oh, the one with hashing. I never fully got how that works.” Dev sat on the couch. "Alright, Aniket, looks like you're the professor tonight.”

"Honestly, I don't know what you mean by hashing." Dev clarified. "That's alright. Let's start with the basics," I said.

I took the whiteboard marker and wrote the text, 'Alice and Joe baked cookies' For simplicity, I'm not showing the spaces.

I continued, "You want to know if a pattern, 'COOL' exists in it." Maya said, "We should compare each character of the pattern with the text."

"Correct! We compare the A from the text to C in the pattern. "Since they are not the same, we slide the pattern by one character towards the right," I started explaining. "If you look at it, it only makes sense to compare the next character if the first character matches, which happens after the word 'baked' where we see 'C' in the text and 'C' in the pattern."

Dev could understand this and chipped in. "We go to the next characters till we find a mismatch, in this case the characters K and L."

"Perfect! But, think about it, we had to compare C, O, and another O before we concluded that the pattern is not correct. For longer patterns, computation becomes heavy. Can we think about a better way to complete our task of finding the pattern? This is where Rabin and Karp developed their algorithm." I took a pause.

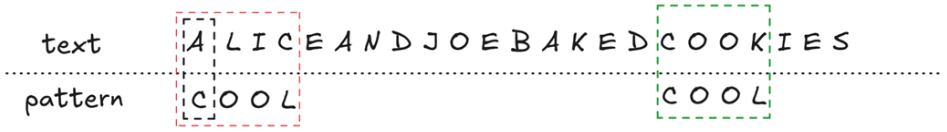

Figure 4.1 The pattern COOL is to be matched with the text "ALICEANDJOEBAKEDCOOKIES". Each character is checked one by one. The first box shows that A and C are compared. With no successful match, we move the pattern forward by one character and continue comparing. The second box shows three characters matched, and the last character doesn't, concluding the pattern is not in the text only after checking the first three characters.

"Instead of comparing character by character, can we assign a numeric value to the entire pattern?" Maya asked. "Yes. Let's change our pattern and use the Rabin-Karp algorithm, say 'JOE' is to be searched for in the text," I said.

Dev, looking at the board, said, "So, how do we assign a number to this pattern?"

I grinned. "We use a *hash function*. To define it in simple terms, it is a mathematical equation that can convert any data into any other form based on how we define the function."

"Interesting! So we take the pattern 'JOE' and convert it to a numeric value with the help of a hash function."

"That is exactly what we will do, Dev," I said, "but we will slightly modify it. We assign each character a number, and we can do this in any way possible, most of the time we use the ASCII character itself. For simplicity, we number the English alphabet from 1 to 26 and keep all of them uppercase," I said.

Table 4.2 Character mapping to numeric values

Character	Value
A	1
B	2
C	3
….	…
Z	26

So now, let's calculate the hash for the pattern 'JOE':"

J = 10

O = 13

E = 5

Hash("Joe") = 10 + 13 + 5 = 28

Maya scribbled the same hash function for the first three letters of the text: "Ali"

A = 1

L = 12

I = 9

Hash("Ali") = 1 + 12 + 9 = 22 ≠ 28

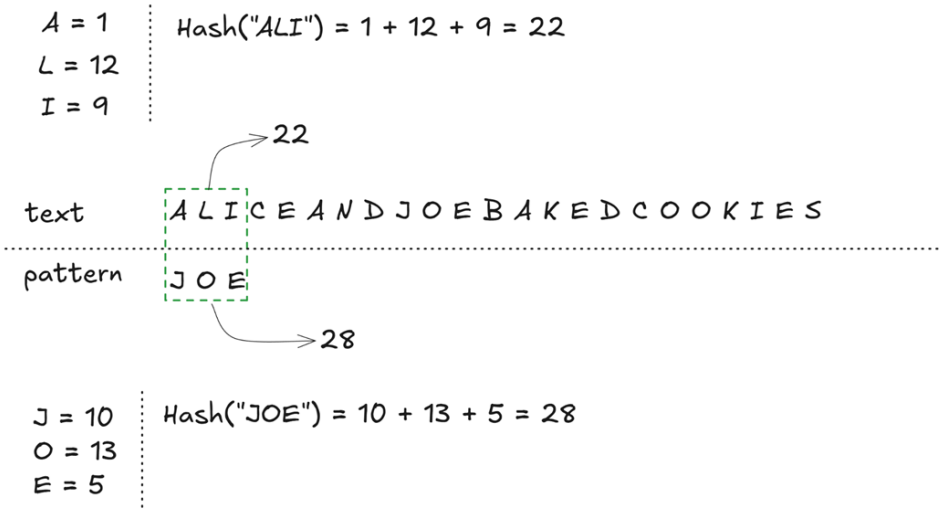

Figure 4.2 The pattern is JOE. Based on the character mapping table, the hash value of the pattern JOE is 28, and that of ALI is 22. Compare the entire values instead of characters.

"So we move on," said Dev, "and shift our window by one character, which means we calculate for 'LIC'?"

"Exactly," I said.

"But wait. If you have a pattern that's a thousand characters long, are you really summing up a thousand numbers every single time you move the window one space?" Dev leaned forward and asked.

I smiled. "Now here's the cool trick. Instead of recalculating the entire sum every time, we just remove the first character from the window and add the next one to the window when we slide. This is called a rolling hash."

Hash("LIC") = Hash("ALI") - value('A') + value('C')

= 22 - 1 + 3 = 24

The hash value of A is 1, so we subtract it from the previously calculated hash value. The hash value of C is 3, so we add it to the hash value. The final hash of the pattern LIC is 24.

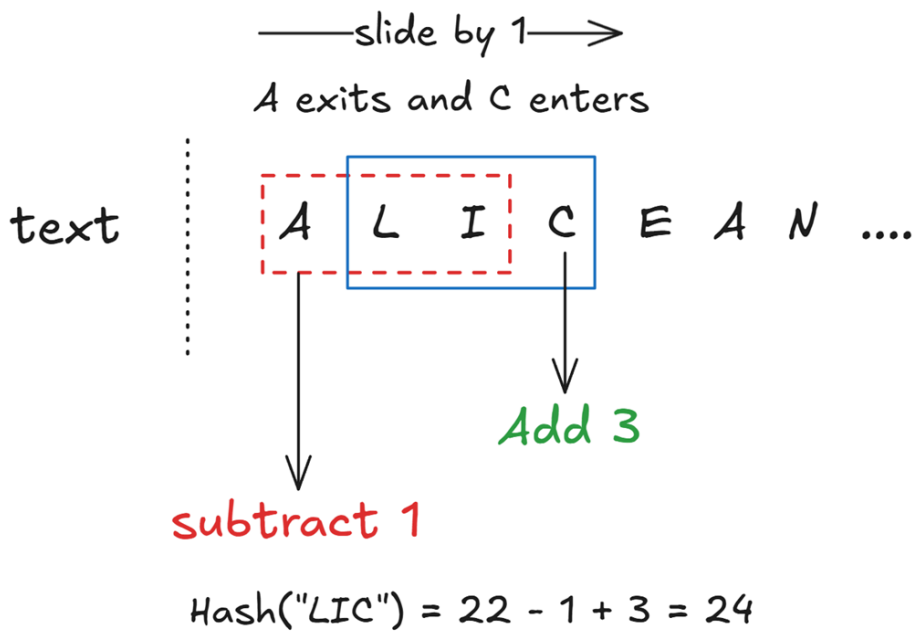

Figure 4.3 A sliding window of size equal to the size of the pattern. The window slides by one character. The character A exits from the left, and the character C enters.

In this way we were able to compare the pattern character by character with the text and decide if the pattern exists or not. Further, we used a simple hash function to convert our pattern to a numeric value. If a window's hash matches the pattern's numeric value, we perform a character-by-character check to verify the match and rule out a hash collision.

DEFINITION A *substring* is a contiguous non-empty sequence of characters within a string.

COLLISIONS AND IMPROVED HASH FUNCTION

Hash functions and the process of hashing are not perfect. Different text patterns and substrings can yield the same hash value. This gives rise to *collisions* or *spurious hits.* Collisions occur when a poorly defined hash function yields the same values for multiple substrings or patterns. Collisions cause invalid matches which are called spurious hits. In case of a collision, the next logical step is to check the characters of the pattern one by one again. To make an efficient algorithm, it is necessary to have a hash function that is defined to generate unique values almost all the time.

Few bad hash functions examples:

1. H(S) = sum(S[i]) - sum of all characters so "ACT" and "CAT" have same hash value
2. H(S) = S[1] - based on the first character, so "Apple" and "Axe" have the same hash value.

Good hash functions example:

1. *H(S) = sum(S[i]. R^{m-i}) mod Q* - standard rolling hash function

Maya frowned. "But what if two different substrings end up with the same hash?" Maya asked.

"Here's where the collision problem hits. Suppose another substring, "OJE," also sums to 28. That means we'd have to do a character-by-character comparison even though the strings are different."

"So can we fix it," Dev said, "by making a better hash?"

"Yes. That's correct. Instead of just adding values, we multiply by powers of 10 based on position. That way, 'JOE' and 'OJE' don't clash."

Now, even if the characters are the same but in a different order, the hash is different!

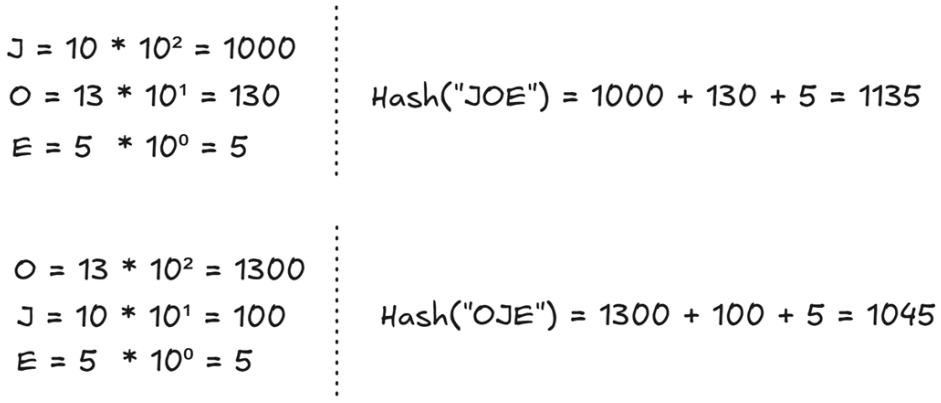

Figure 4.4 Two substrings previously with a simple hash function yield the same hash value. When the hash function is changed using powers of 10, the same two substrings yield different hash values. This reduces the problem of collision.

Collisions have a direct impact on the performance and efficiency of the Rabin–Karp algorithm, as they must pause the numeric comparison and perform the slow character-by-character check.

ROLLING HASH WITH BASE-10 POWERS

We discussed a trick in the previous section to avoid calculating the hash values repeatedly by using rolling hash. Rolling hash is a powerful technique that boosts the performance by calculating only the values that are necessary. We have modified our hash function to use base-10 powers to minimize collisions and increase efficiency with rolling hash.

Maya blinked. "Wait… How do we slide the rolling hash with base-10 powers towards the right when a mismatch happens?"

I jotted the formula on the whiteboard.

New_Hash = (Old_Hash - Old_Char * 10^2) * 10 + New_Char * 1

"So if looking at the first three characters, 'ALI' = 229 (A=1*100, L=12*10, I=9*1), we remove 'A' , multiply the result by 10, and add the new char's value multiplied by 1."

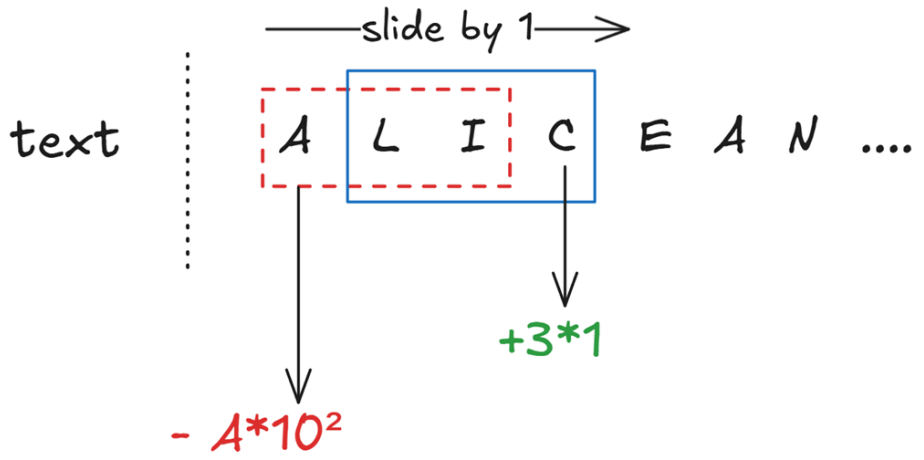

Figure 4.5 Rolling hash working with an improved hash function. The new value of the hash is obtained by subtracting the previous hash value with the character exiting the window multiplied by 10 and adding the new character. Calculation of the new hash is performed with efficiency.

Hash("LIC") = (Hash("ALI") - A*10²) * 10 + C*1

Hash("LIC") = (229 - 100) * 10 + 3 = 1290 + 3 = 1293

Maya lit up. "Neat! So we keep sliding this window until a hash matches the pattern hash."

"But we still might have a collision," said Dev. "So whenever we have a hash match, we double-check by comparing actual characters." I said, "Eventually, we reach the pattern window 'JOE' and the hash matches."

Maya smiled. "So that's how Rabin-Karp helps us solve string matching efficiently."

Listing 4.1 is the pseudocode for the Rabin-Karp algorithm. It would be translated into a Python implementation shown in Listing 4.2

Listing 4.1 Pseudocode for Rabin-Karp algorithm

```
Turn each character into a number based on its position in the alphabet.

Compute a hash value for the pattern.
Compute a hash value for the first window of the text (same length as pattern).

Slide a window through the text:
    For each position, check if the window hash matches the pattern hash.
        If they match, verify that the actual characters are the same (to avoid
false matches).
        If the characters match, record this position.

    Move to the next window position by:
        Removing the contribution of the leftmost character,
        Multiplying by the base value,
        Adding the contribution of the new rightmost character.

Continue until every window has been checked.
Return all positions where the pattern was found.
Listing 4.2 is the code snippet that will run the example we discussed. A
function is created that takes the text and the pattern as input parameters, and
we perform the rolling hash. Below line of code explains the rolling hash shown
in Figure 4.5.
window_hash = (window_hash - old_char_val * h_multiplier)*base+new_char_val
```

Listing 4.2 Code implementation of Rabin-Karp algorithm

```
def char_to_int(c):
   """
    Converts a character to an integer value for hashing.
    Maps 'a' to 1, 'b' to 2, ..., 'z' to 26.
    Args:
        c (str): A single character.

    Returns:
        int: The integer value of the character.
    """
    return ord(c) - ord('a') + 1

def rabin_karp(text, pattern):
   """
    Implements the Rabin-Karp string searching algorithm.

    Uses rolling hash to efficiently find all occurrences of a pattern
    in a text by comparing hash values instead of characters directly.

    Args:
        text (str): The text to search in.
        pattern (str): The pattern to search for.

    Returns:
        list: List of starting indices where the pattern is found.
    """
    n = len(text)
    m = len(pattern)
    base = 10  #A

    if m == 0 or n < m:
        return []

    pattern_hash = 0
    window_hash = 0
    h_multiplier = base ** (m - 1)  # Precompute multiplier for rolling hash

    for i in range(m):
        pattern_hash = pattern_hash * base + char_to_int(pattern[i])
        window_hash = window_hash * base + char_to_int(text[i])

    matches = []
```

```
    for i in range(n - m + 1):
        if window_hash == pattern_hash: #B
            if text[i:i+m] == pattern:
                matches.append(i)
        if i < n - m: #C
            old_char_val = char_to_int(text[i])
            new_char_val = char_to_int(text[i + m])
            window_hash=(window_hash - old_char_val*h_multiplier)*base +
            new_char_val

    return matches
```

#A Base for hashing (can be any number, 10 is simple)
#B Check if hashes match, then verify characters to avoid collisions
#C Update rolling hash for next window (if not at end)

4.2 Real world applications

The elegance of Rabin-Karp translates into computational power when applied to real systems where data has to be handled in gigabytes and terabytes. Transitioning this algorithm from a theoretical whiteboard concept into a distributed production environment demands strict architectural engineering.

4.2.1 Plagiarism Detection

Plagiarism detection is one of the most common and practical use cases for string matching, especially in academic and publishing environments. The goal is to quickly scan a submitted document (assignments, papers, or research articles) against a vast database of existing texts to quantify similarity and identify potential source material.

- Verbatim Matching (Rabin-Karp): The core mechanism employs the Rabin-Karp algorithm to efficiently detect exact or near-exact matches. The algorithm scales well because it uses fast hashing (fingerprinting) of text segments to filter non-matching documents, drastically reducing the number of full-text comparisons required.
- Scaling the System: Achieving high-speed detection across millions or billions of documents requires advanced engineering:
    - Data Corpus: Massive databases are continuously updated via web crawlers and indexing to incorporate new documents into the search corpus.
    - Distributed Architecture: Distributed storage is necessary for high-speed lookups and retrieval.

- Parallel Processing: For parallel processing, tasks are broken down and distributed across multiple servers using frameworks like MapReduce or Apache Spark, allowing chunks of the document to be compared simultaneously and the results aggregated into a final similarity score.

- Advanced Detection: Plagiarism is not solely detected via exact matching. Systems integrate semantic matching and Natural Language Processing (NLP) techniques, often involving extensive preprocessing steps, to identify paraphrasing and conceptual similarity.

4.2.2 BioInformatics

Bioinformatics is a field where biology, computer science, mathematics, and statistics work together to analyze and interpret large-scale biological data and sequences. Identifying DNA, RNA, or protein subsequences within massive genomic datasets is another critical practical use of sequence matching. In this domain, sequences are treated as strings with specialized alphabets:

- DNA: A DNA subsequence consists of a 4-character alphabet: Adenine (A), Thymine (T), Cytosine (C), and Guanine (G).
- RNA: A RNA subsequence consists of the same bases as that of DNA, but Uracil (U) replaces Thymine (T)
- Proteins: A protein subsequence is a 20-character alphabet corresponding to the 20 amino acids.

The primary task is to find specific functional patterns, or motifs, for example the DNA motif TGGAGGGAG, within a whole-genome dataset that may contain millions or billions of base pairs. The Rabin-Karp algorithm is highly efficient for these exact-match queries in linear sequences.

THE CORE PRINCIPLES OF HOMOLOGY

The effort to match and align these biological sequences is foundational, as sequence similarity implies shared function, structure, and evolutionary origin (homology). This matching facilitates key achievements:

- Functional Prediction: Matching a DNA or protein subsequence to a known functional element, like an enzyme in another organism, allows researchers to confidently predict the function of the new sequence.
- Homology and Evolution: When a human gene sequence aligns with that of another species, it establishes homology, suggesting descent from a common ancestor and providing insights into evolutionary history.

- Epidemiology and Tracking: In public health, quickly matching the genome of a new virus or bacteria to known strains is crucial for tracking its origin, rate of spread, and potential resistance to treatments.

These applications underpin major fields, including drug discovery, personalized medicine, and pathogen surveillance.

4.3 References

- The original paper of this algorithm is https://courses.csail.mit.edu/6.006/spring11/rec/rec06.pdf.
- Practice Problem of string matching specifically for Bioinformatics here - https://rosalind.info/problems/list-view/
- Plagiarism detection application in academic environment - https://www.ioinformatic.org/index.php/JAIEA/article/view/644
- YouTube Video explaining the example: https://www.youtube.com/watch?v=vTlsnAWp5-U&list=PL_drzcX_ki75Hif00uAhycjlqDNi0ZVlC

4.4 Summary

- Rabin-Karp algorithm is a string matching algorithm that utilizes a rolling hash function to perform efficient string searches in text
- Character-by-character matching is highly inefficient and computationally expensive with O(nxm) time complexity.
- The hash function in Rabin-Karp converts the pattern into a numeric value for comparison of the entire pattern as a single unit. Comparing integers is significantly faster than iterating through and comparing each individual character of the string.
- Simple addition-based hash functions are fundamentally flawed because anagrams, such as 'ACT' and 'CAT', will produce the exact same numeric value, creating a high volume of false positives.
- A collision or a spurious hit means that different substrings generate the same hash value. Collisions degrade the performance, and frequent collisions indicate a poorly defined hash function.
- A poorly defined hash function will generate frequent collisions, forcing constant character verification and degrading the algorithm's performance from its average linear time to its worst-case scenario of O(n x m).
- Collisions can be reduced by developing a better hash function that generates unique hash values for multiple substrings. A rolling hash function with base 10 powers means each character has a different value in cases such as "ACT" and "CAT".
- To ensure accuracy and rule out false positives from collisions, the algorithm must strictly perform a character-by-character verification every time a hash match occurs.

- Sliding window is a technique to avoid recalculating the entire hash value by sliding the window by one space but at the same time subtracting the value exiting the window and adding the new value entering the window.
- The rolling hash mechanism calculates the hash of the next text window in O(1) time.
- Plagiarism and bioinformatics are two key domains where Rabin-Karp algorithms work well. Text like research papers and assignments for plagiarism and DNA/RNA sequences for bioinformatics help in identifying exact or near-exact matches.
- Scaling the system requires core engineering, parallel processing, and distributed architecture approaches.
- The average time complexity of the Rabin-Karp algorithm is O(n+m), where n is text length and m is pattern length, while the worst-case time complexity is O(nxm).

5 Knuth-Morris-Pratt Algorithm

This chapter covers

- Critiquing the computational bottleneck of backtracking in naive string searches
- Constructing the LPS (Longest Prefix Suffix) array in O(m) time
- Executing the deterministic two-pointer shift formula to achieve strictly O(n)
- Learning real-world application, memory overhead for real-time data streaming

Previously, we used the Rabin-Karp algorithm to match patterns or substring within long texts. It uses a hash function as its core approach in finding the pattern. Creating hash functions and making sure to minimize the collisions. This chapter covers another way of finding patterns and substrings in text without the hash function.

In 1977, the foundational string-matching algorithm was formally proposed by Donald E. Knuth, James H. Morris, and Vaughan R. Pratt in their paper, "Fast Pattern Matching in Strings." It is widely recognized as the *Knuth-Morris-Pratt string matching algorithm, or KMP* algorithm.

Before KMP, standard string search methods were fundamentally inefficient due to backtracking. The naive approach is to use two pointers and start comparing the characters of the text and the pattern. At any given point in time, there is a mismatch, the pointer resets and starts checking the pattern again. This led to a significant performance bottleneck, particularly when searching highly repetitive texts.

DEFINITION *Backtracking*: An inefficient process of moving a search pointer backward in the text after a mismatch, wasting significant time re-evaluating characters that are already scanned.

Consider the challenging case of attempting to match the pattern P = AAAAAB against a very long input text T beginning with AAAA…. When the algorithm encountered the first mismatch (at the sixth character), we needed to reset the pointers back and start from the beginning, and that required considerable, wasted effort. The first five character matches are discarded, but they have information that can be utilized. In chapter 1, we looked at calculating time complexity and representing the worst case in Big-O notation. Time complexity is defined as a method to show how the execution of an algorithm changes with the change in input. For a standard naive approach, in the worst-case scenario, this repetitive re-examination of text characters caused the search time to degrade rapidly toward quadratic time complexity, O(nm), where n is the text length and m is the pattern length.

The core conceptual leap of the KMP algorithm was the realization that a mismatch is not a failure but a source of information. When a mismatch occurs, the sequence of previously matched characters holds valuable structural data that should not be disregarded.

KMP eliminates backtracking by performing a linear-time preprocessing step on the pattern itself. This process generates an auxiliary data structure, commonly called the *LPS (Longest Proper Prefix, which is also a Suffix) Array*.

- The LPS Array: The LPS array acts as an intelligent lookup table. When a mismatch occurs at a specific point in the pattern, the array instantly determines the optimal position for the pattern to be shifted forward. This shift is calculated based on the longest common border between the pattern's prefix (the matched part) and its suffix (the part just examined).
- Optimal Performance: By using the LPS array to instruct the shift, the algorithm guarantees that the pointer for the main input pointer i never moves backward. Every character in the text is inspected at most once.

DEFINITION **Proper Prefix: A contiguous sequence of characters located at the exact beginning of a string that is strictly shorter than the entire string itself (e.g., "aab" is a proper prefix of "aabad"). Proper Suffix: A contiguous sequence of characters located at the exact end of a string that is strictly shorter than the entire string itself (e.g., "bad" is a proper suffix of "aabad").**

5.1 Fast pattern matching

In the last chapter, we used the Rabin-Karp algorithm to find patterns in a simple sentence. In this chapter, we will use text that is more random, repetitive, and complex.

A few days had passed since we discussed the Rabin-Karp algorithm.

Dev poured himself some filter coffee. “Tsk tsk. After our discussion the other night, I was reading about different string matching algorithms and came across KMP!,Knuth-Morris-Pratt Algorithm”

Dev sipped his coffee. “But unlike Rabin-Karp, KMP doesn’t rely on hashing or rolling windows. It’s all about skipping redundant comparisons smartly. Super efficient and elegant.”

Maya leaned in. “Alright. Enlighten us.” I interrupted, "Wait! Let me freshen up and join the discussion.”

30 minutes later…

Dev grabbed a whiteboard marker and cleaned their Rabin-Karp discussion. “Okay, suppose we have this text ‘aabadeaabadcaab’ indexed from 0 to 14 and the pattern ‘aabadc’.”

“The length of the text is 15, and the length of the pattern is 6. The pattern exists in the text starting at index 6 till index 11, enclosed in the box. Now, let me introduce you to KMP’s technique: the LPS array. LPS stands for *Longest Prefix* which is also *Suffix*. It helps us know where to resume in the pattern when a mismatch occurs, without restarting from scratch and the text pointer also remains at the same position.”

I was a bit confused. “So we don’t backtrack the text?”

“Exactly,” Dev said. “We just move the pattern smartly. But first, let’s build the LPS array for the pattern.”

We defined proper prefixes and suffixes, and we need to find the redundancy in the string. Let's take an example string like 'XYZXY.' Below are the proper prefix and proper suffixes.

Proper Prefix: [X], [XY], [XYZ], [XYZX]

Proper Suffix: [Y], [XY], [ZXY], [YZXY]

We can identify the longest sequence as [XY], meaning that block of length 2 is repeated. The same is applied to other strings like binary stream 10101. Here, the repeating block is 101 of length 3.

5.1.1 Building the LPS array

The core engine of KMP is the LPS array, which is constructed in O(m) time, where m is the length of the pattern. This auxiliary data structure maps redundancies of the pattern text. The LPS array acts as a deterministic lookup table. When a mismatch occurs, the LPS array is checked to see how many characters the pattern can safely slide forward. In this way, we do not have to reset our pointer as we do in our naive approach.

Dev wrote down the pattern again and wrote the LPS array below it.

pattern - aabadc

LPS array - 010100

Maya stopped him. “Hold on, Dev. You can't just write the numbers down. How is the computer actually calculating that?”

Dev grinned. “Fair point. We use two variables. A variable length keeps track of the longest *prefix-suffix* match, and variable i iterates through the pattern starting at index 1. LPS[0] is always 0 because a single character has no proper prefix.” He wrote on the board.

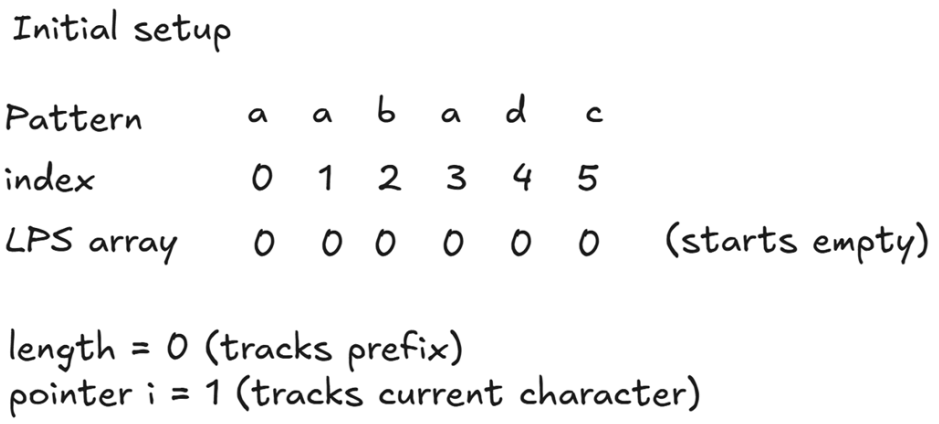

Figure 5.1 To create the LPS array, we start with an empty array of the same length as that of the pattern. We initialize a variable length as 0, and an iterator i starts with 1, which will track the current character.

"So, we need to check the pattern[length] with the pattern[i] and variable i or the pattern forward?" I asked. "Yes. And that's how we get the O(m), as we iterate the length of the pattern to create the LPS array. Let me show step by step for each iteration," Dev said.

Dev started making a table to show the step-by-step formation of the LPS array.

Table 5.1 Step-by-Step LPS array construction for "aabadc"

Index (i)	Current char (pattern[i])	Current prefix (length)	Compare against (pattern[length])	Match/Mismatch Action	LPS array
0	a	0	N/A (Base case)	lps[0] = 0.	[0, 0, 0, 0, 0, 0]
1	a	0	pattern[0]=('a')	Match. Increment length to 1. Record lps[1] = 1.	[0, 1, 0, 0, 0, 0]
2	b	1	pattern[1]=('a')	Mismatch. Fallback to length becomes lps[0] Compare again 'b' vs pattern[0] ('a'). Mismatch. Record lps[2] = 0.	[0, 1, 0, 0, 0, 0]
3	a	0	pattern[0]=('a')	Match. Increment length to 1. Record lps[3] = 1.	[0, 1, 0, 1, 0, 0]
4	d	1	pattern[1]=('a')	Mismatch. Fallback to length becomes lps[0]. Compare again: 'd' vs pattern[0] ('a'). Mismatch. Record lps[4] = 0.	[0, 1, 0, 1, 0, 0]
5	c	0	pattern[0]=('a')	Mismatch. No fallback possible (length is already 0). Record lps[5] = 0.	[0, 1, 0, 1, 0, 0]

“Got it?” Dev asked, looking at me and Maya.

Maya nodded slowly. “So this LPS array is like a lookup for how far we go back and reset the pointer instead of starting again from 0 when a mismatch hits?”

“Correct! So, there is a mismatch, and we check the array, and it tells us to move back 2 positions only and start comparing again. In the end, the final LPS array is [0,1,0,1,0,0]."

“Here’s where KMP helps,” he said. “We use two pointers: `i` for text and `j` for pattern. We move `i` forward always, but `j` is smart, it uses LPS to decide whether to reset or not.”

He walked through the match with the text ‘aabadc’

At i = 0, j = 0 → match ‘a’

i = 1, j = 1 → match ‘a’

i = 2, j = 2 → match ‘b’

i = 3, j = 3 → match ‘a’

i = 4, j = 4 → match ‘d’

i = 5, j = 5 → mismatch (‘e’ != ‘c’)

“Instead of resetting i to 0, the pointer i stays exactly at 5. We consult the LPS array constructed above for the character before the mismatch, LPS[4]. Since LPS[4] = 0, the new pattern pointer becomes j = 0. The pattern physically shifts so pattern[0] aligns with text[5]. The naive approach we discussed would have started the comparison with i = 1 and j = 0.” Dev completed the explanation.

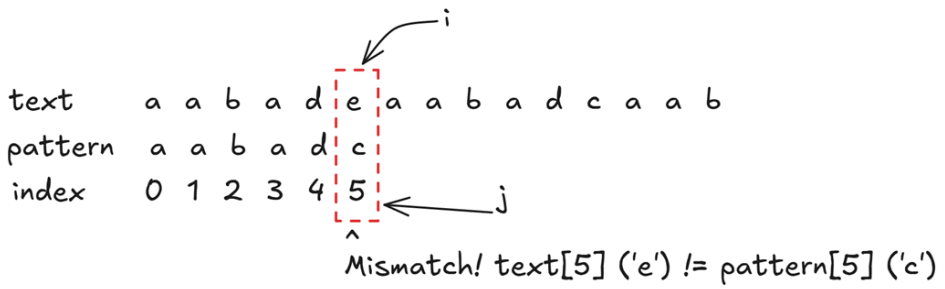

Figure 5.2 The KMP match starts with two pointers, i and j, starting at 0. If it’s a match with text and pattern, we increment i and j by 1 and so on. At index 6, enclosed with a box there is a mismatch of character ‘e’ with ‘c’

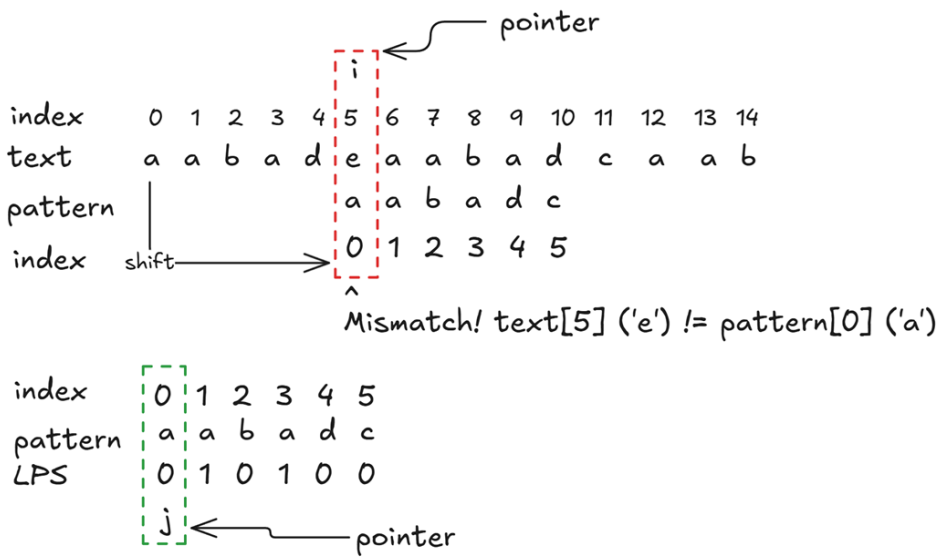

Figure 5.3 The pattern shifts to the right, and the comparison now takes place with 'e' and 'a'. A mismatch occurs again, but the value of j is already at 0, looking at the LPS array. So now i is incremented by 1 to 6.

"And we go on like that and utilize the LPS array," Dev said. "Eventually, we hit a full match at index 6, where the substring 'aabadc' starts."

My eyes widened. "That was easy. So no recomputing hash, just one clean pass using LPS?"

"Yup. Linear time: O(n + m). Rabin-Karp algorithm's worst-case time complexity can get messy as we have to create hash functions. On the other hand, KMP doesn't need hash functions, only the LPS array. Iterating the text and the pattern is the only operation we perform, making the algorithm deterministic."

Maya leaned back. "Well done, Dev. This one's structured, smooth, and intelligent." Dev grinned. "Exactly what I was going for. Now, let's try one with many mismatches so we can understand it better."

He erased the warm-up and wrote the full example in big, clear letters.

The above example covered a straightforward pattern

THE KMP SHIFT EXECUTION

A better and more complex example will clarify the working of the KMP algorithm. The next example has a pattern that has near matches. The LPS array is utilized frequently by resetting the pointer few positions back instead of starting from the beginning. The pointer utilized that iterates over the text never resets.

I stood up and said, “Let me try to create the LPS array.” Dev handed me the marker. I walked up to the board and wrote the pattern with indices 0 to 7 below it. I created an empty array below the indices with dashes where the final values of the LPS array will be filled.

Index: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22

Text: a b c x a b c d a b x a b c d a b c d a b c y

Pattern: a b c d a b c y

“Alright,” I said, “Index 0 is always 0. A single character has no proper prefix. So, LPS[0] = 0. The pointer j starts at 0 on the LPS array.” Maya leaned on her elbows. “And another pointer i, starts at 1.”

“Exactly. So, i = 1. The character is ‘b’. I compare it to pattern[length], which is pattern[0], the letter 'a'. Mismatch. So LPS[1] = 0. Same thing for ‘c’ at index 2 and ‘d’ at index 3. They don't match 'a'. So LPS[2] and LPS[3] are both 0.”

I quickly filled in the first four slots of the array. LPS: [0, 0, 0, 0, _, _, _, _]

“Now,” I said, pointing at index 4. “At i = 4, the character is ‘a’. I compare it to pattern[length], which is pattern[0] (‘a’). It’s a match! So I increment my prefix length to 1 and record it.”

LPS[4] = 1.

I paused and looked at Dev. Dev nodded in approval. “Keep going. Don't break the chain.”

“At i = 5, the character is ‘b’. I compare it to pattern[length], which is now pattern[1] (‘b’). Another match. The prefix a b matches the suffix a b. I increment length to 2.”

LPS[5] = 2

“At i = 6, the character is ‘c’. I compare it to pattern[2] (‘c’). Match again. The length goes up to 3.”

LPS[6] = 3

I stepped back, looking at the board. “And now the final character. At i = 7, we have ‘y’. My length pointer is at 3, so I compare ‘y’ with pattern[3] (‘d’). Mismatch.” Maya pointed her pen at the board. “So what happens to your length pointer? Do you just drop it to zero?”

“Not immediately,” I said, remembering the previous example. “I check the fallback. My current length is 3, so I look at the LPS value of the character right before it, LPS[length - 1], which is LPS[2]. Since LPS[2] is 0, my length drops to 0. I do one final check, does ‘y’ match pattern[0] (‘a’)? No. So the final value is 0.”

Final LPS Array,

Pattern: a b c d a b c y

Index: 0 1 2 3 4 5 6 7

LPS: 0 0 0 0 1 2 3 0

Dev had a genuine smile on his face and said, "Flawless execution. Now we have the blueprint to search the text without ever resetting the text pointer to 0."

Dev started writing on the board.

i = 0, j = 0

Text[i] = 'a', Pattern[j] = 'a'

i = 1, j = 1

Text[i] = 'b', Pattern[j] = 'b'

i = 2, j = 2

Text[i] = 'c', Pattern[j] = 'c'

i = 3, j = 3

Text[i] = 'x', Pattern[j] = 'd' → mismatch

Dev paused. "Here's where LPS is checked. Pattern[j=3] mismatches with text[i=3]. LPS[2] is 0, so instead of starting from pattern[0], we just set j = LPS[2] = 0. Because there is no increment of i yet."

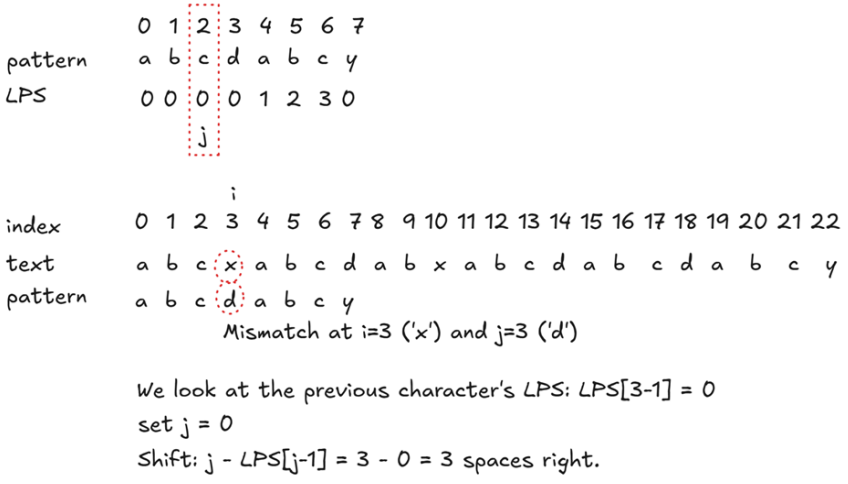

Figure 5.4 The text and the pattern match perfectly till i = 3. At i = 3, the character is 'x' and j = 3 is 'd', a mismatch occurs. In the LPS array, we look at the previous LPS[j-1] and set that value to j. The right shift of the pattern can be calculated as j - LPS[j-1], which is 3 - 0 spaces towards the right.

"Now the i = 3 is checked with j = 0, meaning we compare 'x' with 'a'?" I asked. "...And that is not a match either," Maya said, looking at the board. "Yes, you both are correct. At this moment we increment i by 1," Dev said

"From a diagram perspective this is equivalent to shifting the pattern towards the right," he said.

"It looks like the comparison starts with i = 4 and j = 0," I said, "and all characters match till i = 10 and j = 6, where 'x' and 'c' are compared and a mismatch occurs."

"That is absolutely correct," Dev said. "The next step will be to check the LPS array, in this case LPS[5], and set that value to j. We will not change the value of i, i stays the same, and we do not need to reset it. As for the value of j, the LPS array indicates to set it to 2. This means the pattern 'a b' is already compared, and we should skip this and begin comparing again from j = 2." Dev completed his explanation and looked at us.

Dev looked impressed, smiled, and said, "I think you both understand the algorithm. Maya is right. We shift by 4. LPS only tells us which part of the pattern is already checked, that's it."

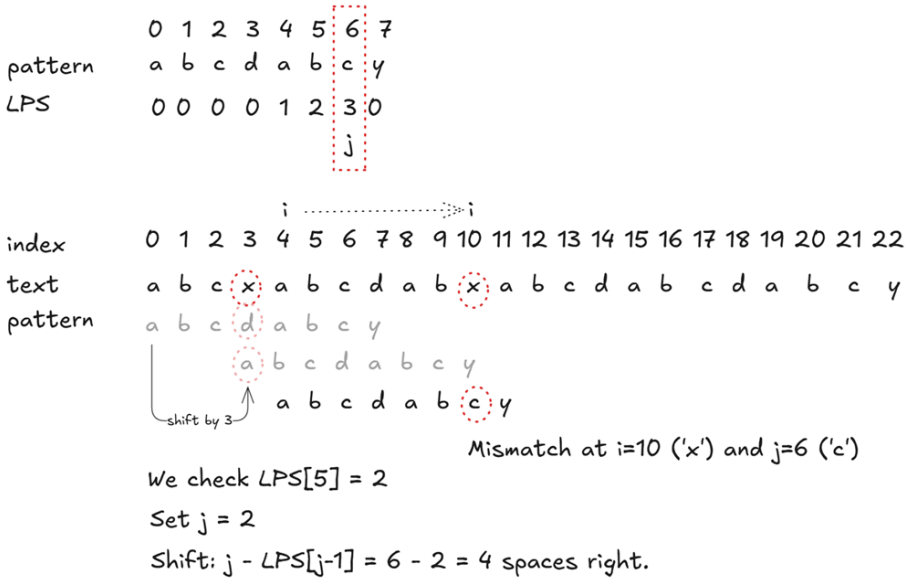

Figure 5.5 The shift by 3 characters is completed. Now i = 3 and j = 0, a mismatch occurs, but j is already 0, so increment i by 1, and j is still 0. The next comparison is with i = 4 and j = 0, which is a match. Proceeding further, i increments until 10 and j increments until 6, a mismatch between 'x' and 'c' occurs. Previous LPS[6-1] is set to j. The visual shift is going to be j - LPS[j-1], which is 6 -2 = 4 spaces towards the right.

"Even if we move 4 spaces to the right, our comparison is still a mismatch." I stood and walked towards the board.

j = 2, i = 10

text[i=10] = 'x', pattern[j=2] = 'c' → 'x' ≠ 'c'

j = LPS[1] = 0

"This is where we increment i now since j = 0." I said.

Dev said, "With that, i becomes 11 and j = 0, and we start comparing again."

i = 11, j = 0

text[11] = 'a', pattern[0] = 'a' → match, increment i and j by 1

text[12] = 'b', pattern[1] = 'b' → match, increment i and j by 1

text[13] = 'c', pattern[2] = 'b' → match, increment i and j by 1

text[14] = 'd', pattern[3] = 'c' → match, increment i and j by 1

text[15] = 'a', pattern[4] = 'a' → match, increment i and j by 1

text[16] = 'b', pattern[5] = 'b' → match, increment i and j by 1

text[17] = 'c', pattern[6] = 'c' → match, increment i and j by 1

text[18] = 'd', pattern[7] = 'y', mismatch 'd' ≠ 'y'

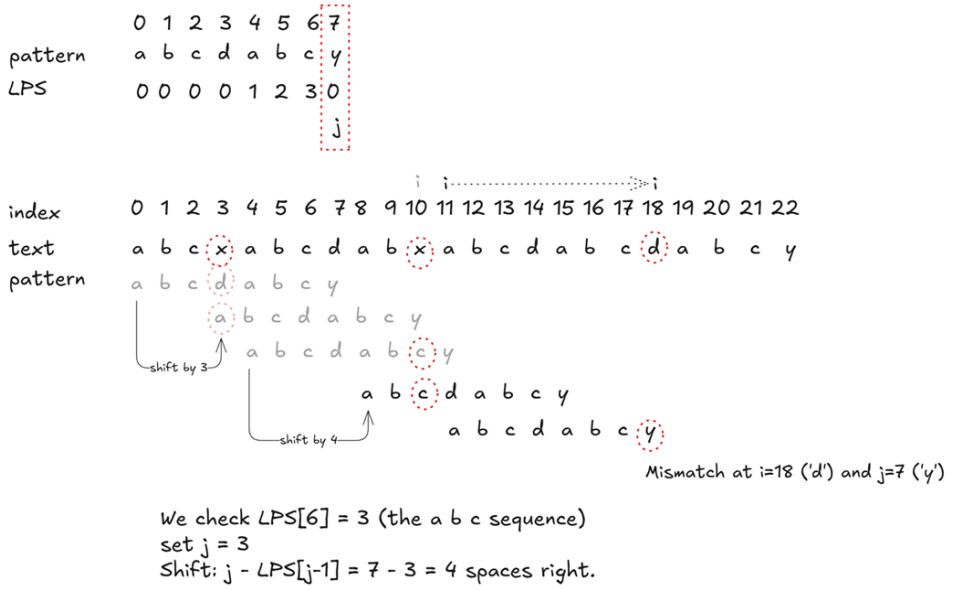

Figure 5.6 A comparison is done with 'x' and 'c', which is a mismatch. Since, LPS[1] is 0, we increment i by 1. The pointer i = 11 and j = 0. The pointer i increases up to 18 for character 'd', and pointer j increments till 7 for character 'y'. Mismatch occurs here, and LPS[7-1] = 3 is set to the pointer j. Ths shift is 7 - 3 = 4 spaces towards the right.

"Oh, that was the last character," Maya said. Dev continued, "Yes. LPS[7-1] = 3, and j is now 3. The sequence a b c is already compared, we skip these comparisons."

"And the shift is going to be 7 - 3 = 4 towards the right," I said, pointing at index 15 on the board.

"Oh, now I see it clearly. This is our last shift by 4, and we can match our pattern within the text. Aniket is right. At index 15 we can see the pattern matches completely till the end of the text," Maya said.

Maya was leaning forward by now, nodding. "The way LPS lets us jump in the pattern without going back in the text is… efficient."

I chimed in, "We're never repeating characters in the text, each is scanned once. That's why it's linear."

Dev smiled. "Exactly. The time complexity is O(n + m). That's why KMP is so powerful, it precomputes the repetitive pattern in the LPS array to avoid redundant comparison. We do not always reset the j pointer on the LPS array at the beginning."

He stepped back from the whiteboard and kept the marker on the table.

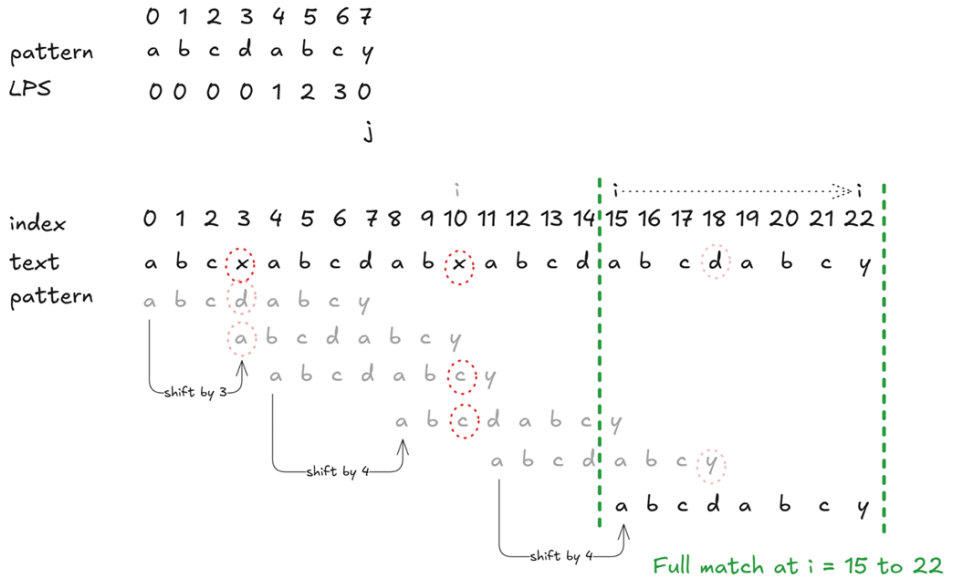

Figure 5.7 This is the final check where the full match occurs. We have moved 4 spaces towards the right, matching the pointer and j = 3. The sequence a b c is already compared, and pointer i = 18 with j = 3, so the comparison starts again. The comparison ends with all characters of the text matched with the pattern. The pattern is found from pointer i = 15 to 22 in the text.

The example walkthrough showed many mismatches, and the LPS array was utilized at every stage. There were a total of 5 mismatches before we found the pattern. Every time we skipped a few comparisons by looking at the LPS array. On the other hand, we never reset the pointer i that iterates over the text, it always moves forward, making it linear. The time complexity of the algorithm is O(n + m), where n is the length of the text and m is the length of the pattern.

Listing 5.1 shows the pseudocode for KMP algorithm. The pseudocode shows how to calculate the lookup table or the LPS array and then use it for searching the pattern. Listing 5.2 is the python implementation based on the pseudocode.

Listing 5.1 Pseudocode for KMP algorithm

```
First, analyze the pattern to build a lookup table:
    For each position in the pattern, find the longest portion that is both:
        - A prefix (starts at the beginning)
        - A suffix (ends at this position)
    Store these lengths in a table.

Then search through the text:
    Starting from the beginning of both the text and pattern:
        Compare characters one by one.

        When characters match:
            Advance both pointers.

        When you've matched the entire pattern:
            Record this match position.
            Use the lookup table to decide where to resume searching.

        When characters don't match:
            If you had partial matches before this mismatch,
                use the lookup table to skip ahead instead of going back in the
text.
            If no partial matches, just move forward in the text.

Return all positions where the pattern was found.
```

Listing 4.2 is the code definition to show the working of the KMP algorithm. Passing the input and pattern will yield the index at which the pattern is found.

Listing 5.2 Code implementation of KMP algorithm

```
def compute_lps(pattern):
"""
    Computes the Longest Prefix Suffix (LPS) array for the KMP algorithm.

    The LPS array stores the length of the longest proper prefix that is
    Also a suffix for each substring of the pattern. This helps skip
    Unnecessary comparisons during pattern matching.

    Args:
        pattern (str): The pattern string to analyze.

    Returns:
        list: LPS array where lps[i] is the longest proper
        prefix-suffix length
              for pattern[0..i].
    """

    m = len(pattern)
    lps = [0] * m
    length = 0
    i = 1

    while i < m:
        if pattern[i] == pattern[length]:
            length += 1
            lps[i] = length
            i += 1
        else:
            if length != 0:
                length = lps[length - 1]
            else:
                lps[i] = 0
                i += 1
    return lps

def kmp_search(text, pattern):
"""
    Searches for all occurrences of a pattern in text using KMP algorithm.

    Uses the LPS array to efficiently skip characters that can't match,
    avoiding backtracking in the text.
```

```
    Args:
        text (str): The text to search in.
        pattern (str): The pattern to search for.

    Returns:
        list: List of starting indices where the pattern is found.
    """

    n = len(text)
    m = len(pattern)

    if m == 0 or n < m:
        return []

    lps = compute_lps(pattern)
    print(f"Calculated LPS Array: {lps}\n")

    matches = []
    i = 0  #A
    j = 0  #B

    while i < n:
        if pattern[j] == text[i]:
            i += 1
            j += 1

        if j == m:
            matches.append(i - j)
            j = lps[j - 1]
        elif i < n and pattern[j] != text[i]:
            if j != 0:
                j = lps[j - 1]  #C
            else:
                i += 1

    return matches
```

#A index for text
#B index for pattern
#C Use LPS to skip characters

5.2 Real-World applications

The KMP algorithm has advantages in production environments compared to other string matching algorithms. KMP never backtracks or resets its pointer but is a strictly forward-moving linear traversal algorithm while iterating over the input text. Other advanced algorithms skip character comparisons, making them faster and reliant on static files and environments. However, use cases where continuous ephemeral data streams are present, KMP becomes the foundational algorithm to be implemented. Let's look at similar real-world applications.

5.2.1 Log analysis and real-time data streaming

Modern software applications, systems, and architectures are developed to produce lots of log data to debug, analyze, and improve the software. They generate millions of data points, which is not static but continuous and dynamic.

Let's say we wish to look at a specific error type in our live data stream. Using standard, naive, or even advanced string matching algorithms like Boyer-Moore (we will learn about this algorithm in the next chapter) fails here, since their implementation is different. In live data streams, the data points you have already passed through the buffer and been discarded. Even if we wish to look backward, we need massive caching in our RAM, creating a bottleneck.

In KMP, the text pointer never decrements, the system only needs to hold a single character of the incoming data stream in memory at any given time. The LPS array constructed for the specific error type is responsible and handles the internal logic of comparison and pattern shift. This implementation is fast and efficient and processes lots of incoming data points in real-time.

5.2.2 Intrusion Detection Systems (IDS)

Intrusion Detection Systems (IDS) are used to monitor network traffic that can potentially be a threat to the system, like a virus. It checks for specific patterns or signatures in the incoming data streams. When the IDS boots up, the threat signature patterns are preprocessed to generate the LPS array in O(m) time, and the incoming traffic is scanned continuously against these preprocessed signatures. As raw bytes stream into the network, the KMP algorithm inspects them sequentially. If a partial threat signature matches but ultimately fails (a mismatch), the LPS array instantly shifts the internal pattern pointer. Because KMP guarantees that no byte of network traffic ever needs to be re-evaluated, the IDS engine never stalls.

Modern IDS engines might not use KMP directly, but variations like the Aho-Corasick algorithm are implemented, but the foundation remains the same.

5.3 Key insights

The Knuth-Morris-Pratt algorithm is a highly specialized tool designed to solve a specific mechanical flaw. To deploy it effectively in production, we must understand the mathematical trade-offs it demands.

5.3.1 The cost of preprocessing and memory overhead

Before a single character of the text is scanned, the algorithm requires O(m) time and O(m) auxiliary space to construct the LPS array (where m is the length of the pattern).

- The Latency Trade-off: In medical systems where the search pattern changes dynamically and is incredibly long (e.g., massive DNA/genomic sequences), this preprocessing step introduces a mandatory latency bottleneck. The search phase cannot begin until the array is fully compiled.
- The Space Trade-off: While allocating memory for a 10-character search string is trivial, allocating a massive integer array in memory-constrained environments (like embedded systems or IoT devices) can be prohibitive. The algorithm buys time at the strict expense of space.

5.3.2 Separating complexity

By analyzing the pattern separately and mapping its internal symmetries, KMP guarantees that the actual scanning phase will strictly take O(n) time. This decoupling allows the text pointer to move relentlessly forward, making it the mathematically optimal approach for continuous, real-time data streaming where historical backtracking is not preferred.

5.3.3 The “No Redundancy" trap

The most critical blind spot for developers implementing KMP is failing to analyze the nature of their text and the pattern. KMP derives its entire performance advantage from repetition. Let’s say we have a pattern “abcdef” where the LPS array is [0,0,0,0,0,0], consisting of all 0s.

When a mismatch occurs in this scenario, the LPS array tells the pointer to reset to zero every single time. The algorithm effectively degrades, mimicking the exact behavior of the naive brute-force approach. However, because KMP still spent time and memory building the LPS array, it will actually perform marginally worse in practice than a simpler algorithm. As engineers, we must process the data corpus, if the patterns and text lack repetition, KMP is the wrong architectural choice. But for few applications in the software development cycle, like log analysis, the KMP algorithm performs better since most logs contain repetitive texts and patterns like error codes, response codes, and custom-defined terms as well.

5.4 Summary

- The Knuth-Morris-Pratt Algorithm (KMP) introduces the concept where a mismatch is not a failure but a source of information. Naive and brute force algorithms reset the text pointer to the start when a mismatch occurs
- The time complexity without KMP is O(nm), where n is the length of the text and m is the length of the pattern.
- KMP makes sure the text pointer only moves forward in linear time.
- The search pattern is preprocessed and stored in auxiliary space called the Longest Prefix which is also the Suffix (LPS) array. This operation is done in O(m) time.
- For every index in the pattern, the LPS array calculates the length of the longest contiguous sequence of characters at the beginning of the string that perfectly matches the sequence at the end.
- During a mismatch at pattern index j, the algorithm avoids a full reset. It consults the LPS array (LPS[j-1]) to determine exactly how many characters of the prefix are guaranteed to still match the text. This makes it deterministic.
- The physical distance the pattern slides to the right during a mismatch is mathematically defined by, j - LPS[j-1]
- By decoupling the pattern analysis from the text scanning, the actual search phase executes in strictly O(n) time, bringing the total algorithmic time complexity to O(n + m).
- KMP is the foundational algorithm for continuous, real-time data streaming, live log analysis, and line-speed network Intrusion Detection Systems (IDS).
- KMP derives its power exclusively from pattern repetition. If there is no repetition in the pattern or the text, the LPS array is not useful, and the approach falls to the naive approach with worse time complexity. This is because the LPS array without repetition consists of only zeroes. Every mismatch reset the pointer to 0, making it naive.

5.5 References

- The original paper of this algorithm is https://www.cs.jhu.edu/~misha/Spring23/Knuth77.pdf.
- Practice Problems of string matching specifically for bioinformatics here - https://rosalind.info/problems/list-view/
- YouTube video - https://www.youtube.com/playlist?list=PL_drzcX_ki75Hif00uAhycjlqDNi0ZVlC

6 Horspool's Algorithm

This chapter covers

- **Constructing the Bad Character Shift table**
- **Executing right-to-left pattern scanning to bypassing redundant character evaluation**
- **Evaluating how the Horspool heuristic allows for sub-linear performance**

Every string matching algorithm we covered in the previous chapters, like the Rabin-Karp and Knuth-Morris-Pratt algorithms (KMP), operates on the fundamental assumption of scanning the input text and the pattern from left to right. The Boyer-Moore-Horspool algorithm, commonly known as the Horspool algorithm, removes that assumption. Instead of scanning from left to right, we scan from right to left. And instead of focusing on matching characters, it derives its massive speed advantage from focusing on mismatches.

Horspool operates on a simple heuristic: if you are looking for a word in a massive document, and the character you are currently looking at is nowhere to be found in your search pattern, why slide the pattern forward by just one space? You should skip the pattern past that invalid character entirely.

To achieve this, a table is created called the Bad Character Shift Table, the algorithm mathematically pre-calculates exactly how far it can safely jump whenever a mismatch occurs. In practice, this allows the algorithm to skip analyzing massive chunks of text, resulting in sub-linear search times that easily outpace KMP in standard text processing.

6.1 The paradigm shift: Right to left scanning

I was sprawled on the carpet with my notepad.

I glanced up and said, "Okay, so we've covered brute force, Rabin-Karp with the hash trick, and then KMP, which used the Longest Prefix Suffix (LPS) array. I think I'm finally starting to see how these string-matching algorithms work. Are there any algorithms that rival KMP?"

"The *Boyer-Moore-Horspool Algorithm*," Maya said, closing her laptop. "It's a shifting algorithm like KMP, but the logic is reversed. We align the pattern, but we scan backwards."

Dev stopped chewing. "Backwards? Like, checking the last character of the pattern first?"

"Exactly," Maya said, walking over to the whiteboard. "Let's look at a baseline example."

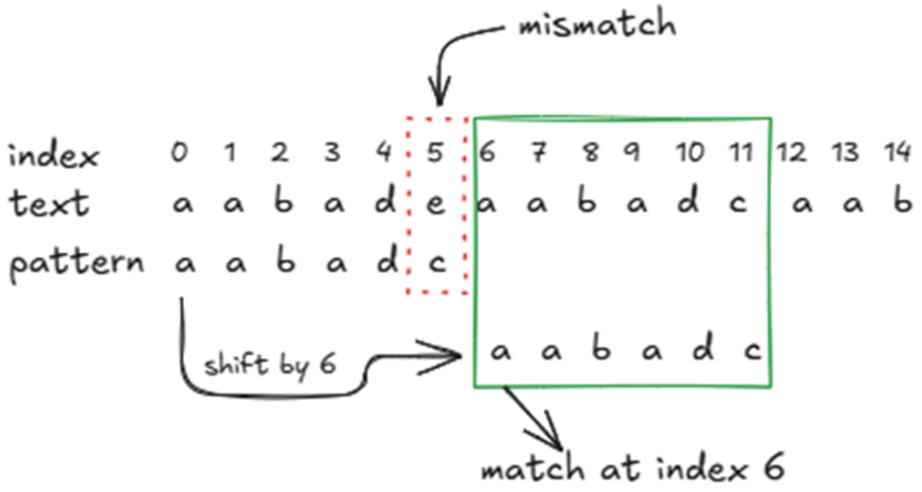

Figure 6.1 Baseline example for text and pattern. The matching starts from behind, and 'e' is not equal to 'c'. A mismatch is already found. We move by 6, and the pattern is found.

Maya drew an arrow pointing to the end of the first window. "We start at the right end of the pattern. We compare the text character 'e' with the pattern character 'c'."

"Mismatch," I said.

"Right. Now, here is the Horspool trick," Maya explained. "Look at the text character that caused the mismatch: 'e'. Is the letter 'e' anywhere inside our pattern?"

I scanned the pattern "aabadc". "No."

"So why would we shift the pattern by one space? If we shift by one, 'e' is still in our window. It will just cause another mismatch." Maya drew a massive, sweeping arrow forward. "Because 'e' isn't in our pattern, there is zero risk of skipping over a valid match. We shift the entire pattern to the right completely past the 'e'."

Dev raised an eyebrow. "That's a 6-character jump from a single mismatch."

"Exactly," Maya smiled. “But what happens if the mismatched character is in our pattern? To handle that, we have to do some math before the search even begins.”

BAD CHARACTER SHIFT TABLE

When a mismatched character in the text does exist within the search pattern, the algorithm cannot blindly skip forward by the full pattern length, or it risks sliding right past a valid match. It must shift the pattern just enough to align the text character with its matching character in the pattern.

To do this instantly in O(1) time complexity, Horspool builds a Bad Character Shift Table before the search begins. It uses a strict formula to map every character in the alphabet to a specific integer shift value:

Shift = pattern length - 1 - index

Maya wrote the pattern text on the whiteboard to show the construction of the bad character table.

pattern = “aabadc”

“Let’s manually construct the shift table for our pattern, ‘aabadc’,” Maya said. “Before we calculate anything, we establish the default shift value for any character that does not appear in our pattern. That default value is simply the total length of the pattern itself, which in this case is 6, because ‘aabadc’ has exactly six characters.”

Dev nodded. “That makes sense. So if we encounter a completely random character like ‘z’ during the search, we instantly shift the entire pattern forward by its full length 6.”

“Exactly,” Maya confirmed. “Now, for the characters inside the pattern, we apply our formula to calculate their custom shift values. We take the pattern length, subtract 1, and then subtract the index of the character we are currently looking at.”

Maya pointed at the first character of the pattern at index 0. ”We start with the first ‘a’. The pattern length is 6, the formula constant is 1, and the current index is 0. So we get Shift = 6 - 1 - 0 = 5. This means initially the character ‘a’ is assigned a shift value of 5.”

“Wait,” Dev interrupted. “Does every ‘a’ in the pattern become 5? And do we shift 5 every single time we hit an ‘a’ during the text search?"

“No,” Maya said. “The next character is also an ‘a’ at index 1. The formula now gives us 6 - 1 - 1 = 4. Because ‘a’ is already in our table, we overwrite the old value 5 with our new, smaller shift value of 4. This continues till the pattern ends.”

To see how this process plays out mechanically across the entire length of the pattern, table 6.1 demonstrates the step-by-step process. A *bad character table* is a preprocessing step implemented as a fixed-size array or a hash map that maps characters to the maximum safe distance the search window can slide forward when a mismatch occurs.

During an active text search phase, matching is performed right to left. When a mismatch occurs, that text character is labeled as a ‘bad character.' Instead of shifting the pattern forward blindly, the algorithm queries this table using the mismatched character as the lookup key.

Table 6.1 offers actions that are defined as follows. *Default Shift* is the baseline initialization. Any character not a part of the pattern has the default shift value equal to the length of the pattern. Another action called *Add a new character,* when a character is encountered for the first time during the left-to-right scan of the pattern, its initial shift value is calculated via the formula (length - 1 - index) and inserted into the table. Next is *overwrite previous character,* if a character already exists in the table from an earlier index, the algorithm calculates its new, smaller shift value and overwrites the older entry. This ensures the rightmost occurrence is always preserved in the table.

Table 6.1 Step-by-step bad character table construction for 'aabadc'

Index (i)	character (pattern[i])	Shift = Length - 1 - index	Action taken	Current table state
initial	-	-	Default shift	[Default: 6]
0	a	6-1-0=5	Add new character	[a: 5]
1	a	6-1-1=4	Overwrite previous 'a'	[a: 4]
2	b	6-1-2=3	Add new character	[a: 4, b: 3]
3	a	6-1-3=2	Overwrite previous 'a'	[a: 2, b: 3]
4	d	6-1-4=1	Add new character	[a: 2, b: 3, d: 1]
5	c	-	Ignore final character	[a: 2, b: 3, d: 1]

Dev looked at the table and asked, "Why are we ignoring the last character of the pattern?"

"Because a shift value of zero is a fatal error in a search loop," Maya explained. "If the algorithm hits a mismatch, queries the table, and receives a shift value of zero, the search window stays exactly where it is. It won't move forward by even a single character. The entire engine freezes instantly, trapping your code in an infinite loop."

"Right," I added. "Think back to the example we looked at earlier where our pattern collided with the text, and we hit a mismatch between 'e' and 'c'. That shift value of 6 was derived directly from this blueprint. Because the letter 'e' was completely absent from our bad character table, the algorithm automatically fell back to the default shift value of 6, clearing the hurdle cleanly."

Table 6.2 is the final bad character table for the pattern 'aabadc'. It has three unique characters and one default shift value. These three characters are the only characters of triggering a shift, since they are present in the pattern.

Table 6.2 Final Bad Character Table for 'aabadc'

Character	Value
d	1
a	2
b	3
default	6

"This is quite interesting. Can we construct another bad character shift table for another pattern?" I asked. Maya started clearing the board and said, "I actually have another complex example that we should try solving."

6.2 Handling internal redundancy

We saw how Horspool's algorithm functions with the use of a bad character shift table. Now let's see what happens with another example when repetition is present in the text and the pattern.

BUILDING THE SHIFT TABLE FOR REDUNDANT PATTERN

Maya wrote another pattern on the whiteboard.

Pattern = "abcdabcy"

"Eight characters," Dev said. "So our default maximum shift is 8."

"Right," Maya said. "Now, we build the table. Let's look at the first character, 'a'. It appears twice: at index 0 and index 4."

I did the math in my head. "Using the formula 8 - 1 - index... the first 'a' gives us a shift of 7. The second 'a' gives us a shift of 3." Maya tapped the board. "Which one do we keep?"

"The second 'a' with value 3 because we preserve the rightmost occurrence in the table," Dev answered immediately. "If you shift by 7, you might overshoot the second 'a' and miss a valid match. You have to be conservative. The second 'a' overwrites the 7 with a shift value of 3."

"Exactly," Maya said. "The exact same logic applies to the character 'b' at index 1 and index 5, as well as 'c' at index 2 and index 6. The rightmost calculation always wins. And we skip the final character because if 'y' were assigned a shift value via the formula, it would equal 0."

Table 6.3 outlines this step-by-step bad character table construction completely.

Table 6.3 Step-by-step bad character table construction for 'abcdabcy'

Index (i)	character (pattern[i])	Shift = Length - 1 - index	Action taken	Current table state
initial	-	-	Default shift	[Default: 8]
0	a	8-1-0=7	Add new character	[a: 7]
1	b	8-1-1=6	Add new character	[a: 7, b: 6]
2	c	8-1-2=5	Add new character	[a: 7, b: 6, c: 5]
3	d	8-1-3=4	Add new character	[a: 7, b: 6, c: 5, d: 4]
4	a	8-1-4=3	Overwrite previous 'a'	[a: 3, b: 6, c: 5, d: 4]
5	b	8-1-5=2	Overwrite previous 'b'	[a: 3, b: 2, c: 5, d: 4]
6	c	8-1-6=1	Overwrite previous 'c'	[a: 3, b: 2, c: 1, d: 4]
7	y	-	Ignore final character	[a: 3, b: 2, c: 1, d: 4]

Table 6.4 displays this polished, production-ready dictionary. This is the exact data structure that sits in memory during the active search phase that resolves mismatches in O(1) time complexity.

Table 6.4 Final Bad Character Table for 'abcdabcy'

Character	Value
d	4
a	3
b	2
c	1
default	8

EXECUTING THE HORSPOOL ALGORITHM

We have constructed the bad character table for the pattern 'abcdabcy'. This pattern has to be searched in a long input text and utilize the bad character table efficiently from Table 6.4. The matching starts from right-to-left in Horspool's algorithm.

Maya wrote a 24-character input text string on the board. This input text is expected to have the pattern 'abcdabcy'.

Text: a b c x a b c d a b x a b c d a b c y

Pattern: a b c d a b c y

“Scan right to left,” Maya directed.

I looked at the alignment on the board. “The pattern is 8 characters long, so our initial search window spans from index 0 to index 7 of the text. Since Horspool dictates that we check for a match from right to left, we start our comparison at the very end of this window, index 7. That means we compare ‘d’ in the text to ‘y’ in the pattern. It's an immediate mismatch.”

The window here is the alignment of the text and the pattern, as shown in figure 6.2. The comparison happens in this window from right to left.

“Exactly,” Maya said. “And this brings us to the core rule of the algorithm. What is our bad character here, and why?”

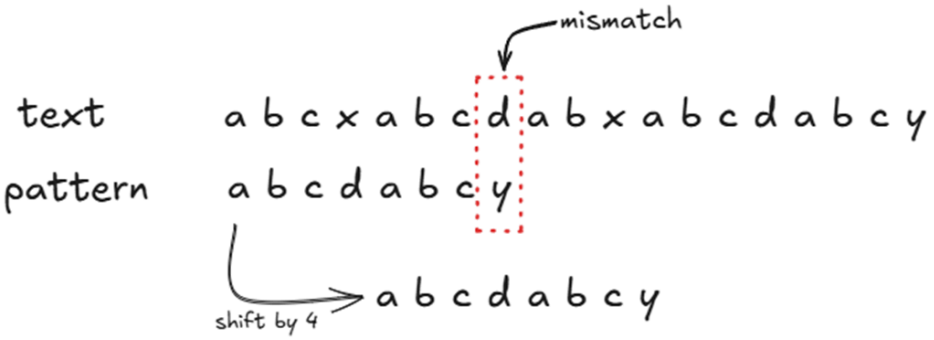

Figure 6.2 The letter ‘d’ in the text is compared with ‘y’ in the pattern as scanning is done from right to left. It’s a mismatch. So, checking the bad character table for the letter ‘d’ from table 6.4, the corresponding shift value is 4. The pattern is shifted by 4, and successfully 4 character comparisons were skipped.

“The bad character is ‘d’, the character residing in the text,” I replied. “Even though the mismatch occurred at the pattern's ‘y’, it is the unexpected character in the text string that has blocked our path. We look up the text character because it is the actual obstacle we have to clear or align with.”

“Perfect,” Maya said, nodding. “Now look at our bad character table we constructed above. What is the designated shift value for ‘d’?”

“Four.” I said, looking at the table 6.4

Maya slid the pattern exactly four spaces to the right, perfectly aligning the 'd' in the pattern with the 'd' in the text.

Dev leaned against the wall, staring at the board. “That’s brilliant. The table acts like a mathematical anchor. It pulls the pattern forward until the bad character finally lines up with its matching counterpart.”

Figure 6.3 demonstrates the execution steps through various states to find the final match.

Step 1 to Step 2: The initial shift of 4 positions brings us to a new text window ending in a. A right-to-left check yields an immediate mismatch, and looking up the text character "a" dictates a shift of 3.

Step 2 to Step 3: Moving forward by 3 positions sets up the next window. The right-to-left check reveals another mismatch against the text character d. The table dictates a shift of 4, sliding the pattern rightward again.

Step 3 to Step 4: This final shift of 4 positions brings the pattern into full alignment with the end of the text string, where every character matches perfectly, completing our search.

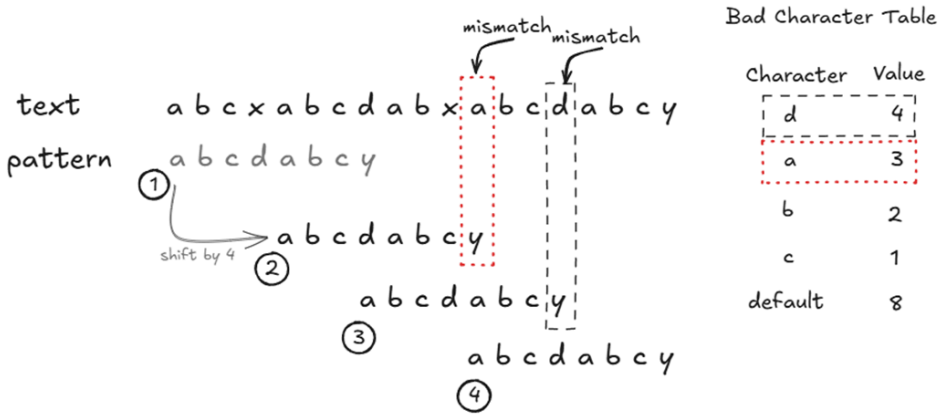

Figure 6.3 The matching is completed in step 4 in the figure. At step 2, the mismatch occurs and the letter is 'a'. Checking the bad character table, the value of 'a' is 3, and we move forward by 3. At step 3, the mismatch occurs again, and the character is 'd' with value 4 from the bad character table, and we move forward by 4. Step 4 is the final step where the pattern is matched.

Our new text window was abcdabxa. 'But we aren't done," Maya said, tapping the end of the new window. "Scan right to left again. We compare the text character 'a' to the pattern character 'y'."

"Mismatch," I said. "The bad character in the text is 'a'."

"Check the table," she ordered.

"The shift for 'a' is 3," I replied.

Maya erased the box and jumped the pattern 3 spaces forward. "Next window. Text character is 'd'', pattern character is 'y'."

"Mismatch again," Dev chimed in. "Bad character is 'd'. The table says shift by 4."

Maya jumped the pattern 4 spaces. "Next, the text is 'd' and the pattern is 'y'."

"Another mismatch on 'd'," I said, catching the rhythm. "Shift by 4 again."

Maya made the final slide. The pattern was now sitting completely flush against the end of the 24-character string.

“Now check it,” she said, stepping back.

I looked at the end of the window. "Text 'y' matches pattern 'y'."

I moved left. "Text 'c' matches pattern 'c'."

"b matches b."

"a matches a."

"d matches d."

"c matches c."

"b matches b."

"a matches a."

I dropped my notepad. "Full match at index 15."

Maya capped her marker. “We just searched a 24-character string, and we only had to test alignments at exactly five index points: 0, 4, 7, 11, and 15. We completely bypassed checking 19 different starting positions because the bad character table mathematically proved they were dead ends.”

Listing 6.1 is the pseudocode for the Horspool algorithm, showing the flow of first creating the bad character table and then using it for lookup when a mismatch happens.

Listing 6.1 Pseudocode for Boyer-Moore-Horspool algorithm

```
First, create a shift table for the pattern:
    For each character in the pattern (except the last one),
    record how many positions to shift the pattern if that character causes a
mismatch.

Then search the text:
    Starting from the beginning of the text:
        Compare the pattern against the text window from right to left.

        If all characters match:
            Record this position as a match.
            Shift the pattern forward by one position.

        If a mismatch occurs:
            Look at the character in the text where the mismatch happened.
            Shift the pattern by the amount stored in the table for that
character.
            If the character is not in the table, shift by the full pattern
length.

    Continue until you can no longer fit the pattern in the remaining text.

Return all positions where matches were found.
```

Listing 6.2 shows the implementation of the algorithm in Python. There are two functions, one creates the bad character table, and the main function executes the table and does the comparison of the input text and the pattern.

Listing 6.2 Code implementation of Boyer-Moore-Horspool algorithm

```
def build_bad_char_table(pattern):
    """
    Constructs the bad character shift table for Horspool's algorithm.

    Rule: Shift = Length - 1 - index
    Rule: Ignore the final character. Overwrite with rightmost occurrences.
    """
    m = len(pattern)
    bad_char_table = {}

    for i in range(m - 1):
        bad_char_table[pattern[i]] = m - 1 - i

    return bad_char_table


def horspool_search(text, pattern):
    """
    Executes the Boyer-Moore-Horspool search utilizing right-to-left scanning.

    Args:
        text (str): The text to search within.
        pattern (str): The pattern to find.

    Returns:
        list: Starting indices of each match in the text.
    """
    n = len(text)
    m = len(pattern)

    if m == 0 or n < m:
        return []

    bad_char_table = build_bad_char_table(pattern)
    matches = []
    i = 0

    while i <= n - m:
        j = m - 1

        while j >= 0 and pattern[j] == text[i + j]:
            j -= 1
```

```
        if j < 0:
            matches.append(i)
            shift = bad_char_table.get(text[i + m - 1], m)
        else:
            bad_char = text[i + m - 1]
            shift = bad_char_table.get(bad_char, m)

        i += shift

    return matches
```

6.3 Real-World applications

To understand where Boyer-Moore-Horspool belongs in a production environment, we must look at what it was designed to exploit: large alphabets and static text. Unlike Rabin-Karp, which excels at searching for multiple patterns simultaneously, or KMP, which is built for continuous data streams where you cannot look backward, Horspool is engineered to aggressively skip through text that is already sitting in memory.

RELATIONAL DATABASES

When a developer runs a SQL query using the `strpos()` function in PostgreSQL to find the starting index of a substring within a massive text column, the database engine frequently relies on a Boyer-Moore-Horspool variant. Database columns often hold massive blocks of natural language, such as user comments or article bodies. Because natural language utilizes a large alphabet, the probability of encountering a "bad character" that does not exist in the search pattern is incredibly high. By taking a microscopic fraction of a millisecond to build the bad character table in memory, the engine can aggressively skip through the text data, minimizing CPU cycles and keeping read latencies incredibly low.

THE V8 JAVASCRIPT ENGINE

Building a bad character table requires memory allocation. If you are searching for a two-letter string, the time spent building the table outweighs the benefits of skipping. V8 handles this by checking the length of the search pattern. For very short strings, it executes a heavily optimized linear search. However, if the pattern exceeds a specific length threshold, V8 dynamically switches to a Boyer-Moore-Horspool implementation. The engine calculates that the overhead of building the shift table is instantly paid off by the massive right-to-left skipping capability.

6.4 Key insights

Maya's whiteboard demonstration exposes that the algorithm actively wants to fail, and it wants to fail as early as possible. By evaluating characters in the text from right to left, Horspool turns mismatches into a strategic advantage. When we understand this mechanism, several critical nuances about the algorithm's performance become clear.

THE SUB-LINEAR ADVANTAGE

In standard algorithms like Rabin-Karp or KMP, you generally have to inspect every character in the text at least once, resulting in linear time complexity. Horspool breaks this barrier. Because it can skip up to the entire length of the pattern in a single move, it frequently completely bypasses massive chunks of the text. In the average case, its time complexity is O(n/m), where n is the text length and m is the pattern length. Time complexity measures how long it takes for the algorithm to finish when the input changes. Counter-intuitively, this means that the longer your search pattern is, the faster the algorithm runs, because the maximum possible jump distance increases.

THE DEPENDENCY ON ALPHABET SIZE

Horspool's speed is entirely dependent on the probability of encountering a "bad character" that is not in the pattern. The default skip we have seen before is the length of the pattern. Therefore, the algorithm thrives on data with a large, diverse alphabet (like standard English text or source code, which uses 256 ASCII characters). In these environments, mismatches almost always trigger a maximum-length skip.

THE BINARY AND REPETITIVE PATTERN EDGE CASE

Conversely, if you apply Horspool to an environment with a tiny alphabet, such as a binary stream (only 0s and 1s) or a DNA sequence (A, C, G, T), the algorithm's performance collapses. We have seen that Rabin-Karp performs well in bioinformatics. Horspool fails because there are so few unique characters, the mismatched text character is almost always present somewhere inside the pattern. The algorithm is forced to make tiny, conservative shifts of 1 or 2 spaces, degrading its speed back to O(n x m) in the worst case.

6.5 Summary

- The Boyer-Moore-Horspool algorithm represents a fundamental paradigm shift in string matching, abandoning the left-to-right scanning tradition to achieve sub-linear search speeds
- Unlike naive algorithms, Rabin-Karp, or KMP, Horspool physically aligns the pattern with the text but executes its character-by-character comparisons in reverse, starting from the rightmost character of the search window and moving left.

- The algorithm derives its speed not from tracking what matches but by heavily penalizing what does not. It assumes that if a mismatched text character does not exist anywhere in the search pattern, the entire pattern can be safely skipped past that character.
- To instantly calculate these skips in O(1) time, Horspool pre-computes a mathematical shift table before the search begins.
- Every unique character in the pattern is assigned a shift distance using the equation Shift = Length - 1 - index.
- If a character repeats within the pattern, its shift value is continuously overwritten. The table must always store the value of the rightmost occurrence. This mathematically guarantees the algorithm will make the smallest, safest possible shift, completely preventing it from jumping over a valid match.
- The very last character of the search pattern is strictly excluded from the table's construction. If evaluated, it would yield a shift value of 0, creating a fatal infinite loop in the search logic. By ignoring it, the minimum possible shift becomes 1.
- Any text character that does not exist in the shift table is instantly assigned the default maximum shift distance: the total length of the pattern.
- In optimal conditions, Horspool does not evaluate every character in the text. It leaps over massive blocks of data, resulting in an average-case time complexity of O(n/m). Counter-intuitively, the longer the search pattern, the faster the algorithm performs.
- The algorithm thrives on text with massive, diverse alphabets (e.g., standard English, source code, server logs). A large alphabet ensures that mismatches frequently trigger the maximum default shift.
- Because of its aggressive skipping capabilities in static memory, variants of Boyer-Moore-Horspool are the standard underlying engines for relational database substring queries, V8 JavaScript engine operations on long strings, and common desktop search utilities

6.6 References

1. Horspool, R. N. (1980). "Practical fast searching in strings". Software: Practice and Experience. 10 (6): 501–506. CiteSeerX 10.1.1.63.3421. doi:10.1002/spe.4380100608. S2CID 6618295.
2. YouTube: https://www.youtube.com/watch?si=SMZJ2fEtDejKr6u0&v=isK0Qu3XHvo&feature=youtu.be
3. Relation databases application: https://github.com/postgres/postgres/blob/master/src/backend/utils/adt/varlena.c
4. V8 Engine: https://github.com/v8/v8/blob/main/src/strings/string-search.h
5. Code implementation: https://github.com/aniketwattamwar/Algorithms-Every-Programmer-Should-Know