

# Grokking Algorithm

## Chapter 1 : Introduction

Algorithm : algorithm is a set of instructions for accomplishing a task

Binary Search : is an algorithm; its input is a sorted list of elements

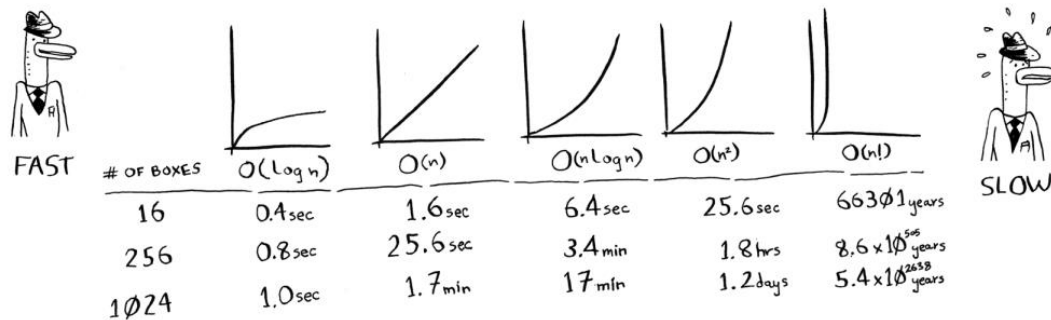


- In general, for any list of  $n$ , binary search will take  **$\log_2 n$**  steps to run in the worst case
- Binary search only works when your list is in sorted order
- the maximum number of guesses is the same as the size of the list. This is called linear time.
- Binary search runs in logarithmic time (log time)

### Big O notation:

- Its relation between Increase amount of time as the list size increases.
- Big O doesn't tell you the speed in seconds. Big O notation lets you compare the number of operations. It tells you how fast the algorithm grows.

## Common Big O run time



- $O(\log n)$ , also known as log time. Example: Binary search.
- $O(n)$ , also known as linear time. Example: Simple search.
- $O(n * \log n)$ . Example: A fast sorting algorithm, like quicksort.
- $O(n^2)$ . Example: A slow sorting algorithm, like selection sort.
- $O(n!)$ . Example: A really slow algorithm, like the traveling salesperson.
- $O(\log n)$  is faster than  $O(n)$ , but it gets a lot faster as the list of items you're searching grows.

## Notes:

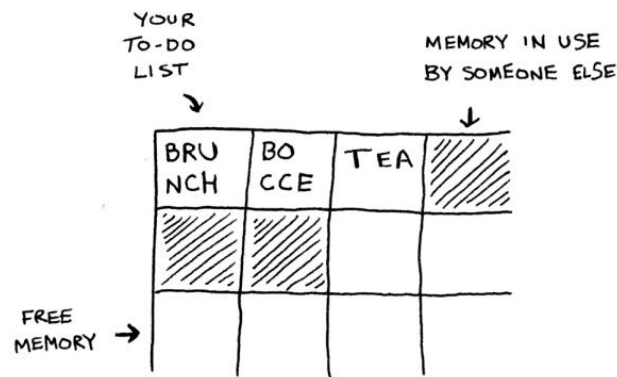
- Algorithm speed isn't measured in seconds, but in growth of the number of operations.
- Instead, we talk about how quickly the run time of an algorithm increases as the size of the input increases.
- Run time of algorithms is expressed in Big O notation

## Chapter 2:

### Array V.S Linked List

#### How Array Works :

Using an array means all your tasks are stored contiguously (right next to each other) in memory



#### Cons :

- You may not need the extra slots that you asked for, and then that memory will be wasted. You aren't using it, but no one else can use it either.
- You may add more than 10 items to your task list and have to move anyway

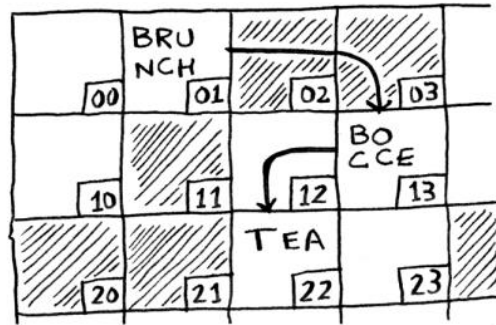


#### Pros :

- Arrays are different. You know the address for every item in your array
- Arrays are great if you want to read random elements, because you can look up any element in your array instantly

## How Linked List Works :

Each item stores the address of the next item in the list. A bunch of random memory addresses are linked together



## Cons:

- Linked lists are great if you're going to read all the items one at a time: you can read one item, follow the address to the next item, and so on. But if you're going to keep jumping around, linked lists are terrible.
- With a linked list, the elements aren't next to each other, so you can't instantly calculate the position of the fifth element in memory—you have to go to the first element to get the address to the second element

## Pros :

Linked lists are great if you're going to read all the items one at a time: you can read one item, follow the address to the next item, and so on. But if you're going to keep jumping around, linked lists are terrible.

	ARRAYS	LISTS
READING	$O(1)$	$O(n)$
INSERTION	$O(n)$	$O(1)$

$O(n)$  = LINEAR TIME  
 $O(1)$  = CONSTANT TIME

### Diff of Array and Linked List :

- Arrays have fast reads and slow inserts
- Linked lists have slow reads and fast inserts.

Exp :

**2.1** Suppose you're building an app to keep track of your finances.

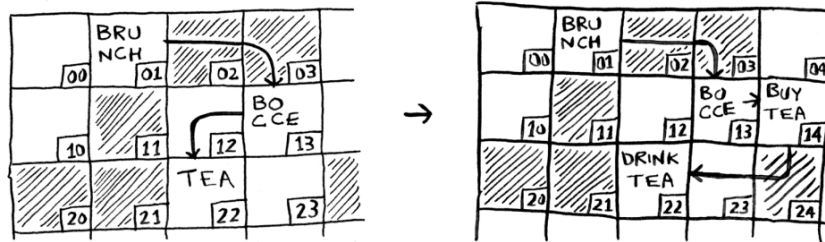


Every day, you write down everything you spent money on. At the end of the month, you review your expenses and sum up how much you spent. So, you have lots of inserts and a few reads. Should you use an array or a list?

*Answer:* In this case, you're adding expenses to the list every day and reading all the expenses once a month. Arrays have fast reads and slow inserts. Linked lists have slow reads and fast inserts. Because you'll be inserting more often than reading, it makes sense to use a linked list. Also, linked lists have slow reads only if you're accessing random elements in the list. Because you're reading *every* element in the list, linked lists will do well on *reads* too. So a linked list is a good solution to this problem.

## Inserting into the middle of a list:

What's better if you want to insert elements in the middle: arrays or lists? With lists, it's as easy as changing what the previous element points to.



But for arrays, you have to shift all the rest of the elements down.



And if there's no space, you might have to copy everything to a new location! Lists are better if you want to insert elements into the middle.

## Deletions :

What if you want to delete an element? Again, lists are better, because you just need to change what the previous element points to. With arrays, everything needs to be moved up when you delete an element.

	ARRAYS	LISTS
READING	$O(1)$	$O(n)$
INSERTION	$O(n)$	$O(1)$
DELETION	$O(n)$	$O(1)$

*2 run times for common operations on arrays and*

It's worth mentioning that insertions and deletions are  $O(1)$  time only if you can instantly access the element to be deleted. It's a common practice to keep track of the first and last items in a linked list, so it would take only  $O(1)$  time to delete those

### **Types of Accesses:**

There are two different types of access: **random access** and **sequential access**.

- 1- **Sequential access** means reading the elements one by one, starting at the first element. Linked lists can only do sequential access. If you want to read the 10th element of a linked list, you have to read the first 9 elements and follow the links to the 10th element.
- 2- **Random access** means you can jump directly to the 10th element.

## Chapter 3 : Recursion

Recursion is where a function calls itself

Exp :

```
def look_for_key(box):  
    for item in box:  
        if item.is_a_box():  
            look_for_key(item) ←..... Recursion!  
        elif item.is_a_key():  
            print "found the key!"
```

*3 Looking for a Key in Box*

There's no performance benefit to using recursion; in fact, loops are sometimes better for performance. I like this quote by Leigh Caldwell on Stack Overflow: "Loops may achieve a performance gain for your program. Recursion may achieve a performance gain for your programmer. Choose which is more important in your situation!"

### Base case and Recursive case :

When you write a recursive function, you have to tell it when to stop recursing. That's why every recursive function has two parts:

the base case, and the recursive case.

**The recursive case** is when the function calls itself.

**The base case** is when the function doesn't call itself again ... so it doesn't go into an infinite loop.

### Stack:

- Every time you make a function call, your computer saves the values for all the variables for that call in memory called Call Stack
- stack has two operations: push and pop.
- Recursion uses Call Stack
- All function calls go onto the call stack.
- the call stack can get very large, which takes up a lot of memory.



## Chapter 4 :

Divide & conquer: D&C isn't a simple algorithm that you can apply to a problem. Instead, it's a way to think about a problem.

***D&C algorithms are recursive algorithms.***

here's how D&C works:

1. Figure out a simple case as the base case. [when you stop the recursive calls]
2. Figure out how to reduce your problem and get to the base case

When you're writing a recursive function involving an array, the base case is often an empty array or an array with one element.

**Why would I do this recursively if I can do it easily with a loop?**

Functional programming languages like Haskell don't have loops, so you have to use recursion.

## Quicksort:

[Faster than Selection Sort & Depend on D&C]

Quicksort is unique because its speed depends on the pivot you choose.

In the worst case, the stack size is  $O(n)$

In the best case, the stack size is  $O(\log n)$ .

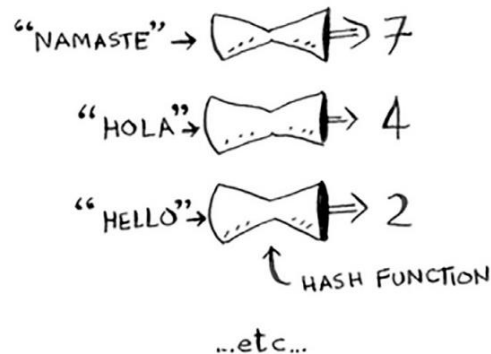
## Merge sort vs. quicksort

Quicksort has a smaller constant than merge sort. So if they're both  $O(n \log n)$  time, quicksort is faster. And quicksort is faster in practice because it hits the average case way more often than the worst case

## Chapter 5 : Hash Table

### Hash Function :

A hash function is a function where you put in a string and you get back a number.



requirements for a hash function:

- It needs to be consistent. For example, suppose you put in “apple” and get back “4”. Every time you put in “apple”, you should get “4” back. Without this, your hash table won’t work.
- It should map different words to different numbers. For example, a hash function is no good if it always returns “1” for any word you put in. In the best case, every different word should map to a different number

### What is Hash Table :

Put a hash function and an array together, and you get a data structure called a hash table. A hash table is the first data structure you’ll learn that has some extra logic behind it. Arrays and lists map straight to memory, but hash tables are smarter. They use a hash function to intelligently figure out where to store elements. Hash tables are probably the most useful complex data structure you’ll learn. They’re also known as hash maps, maps, dictionaries, and associative arrays. And hash tables are fast! Remember our discussion of arrays and linked lists back in chapter 2? You can get an item from an array instantly. And hash tables use an array to store the data, so they’re equally fast. You’ll probably never have to implement hash tables yourself. Any good language will have an implementation for hash tables. Python has hash tables; they’re called dictionaries.

A hash table has keys and values. In the book hash, the names of produce are the keys, and their prices are the values. A hash table maps keys to values.

## hashes are good for

- Modeling relationships from one thing to another thing
- Filtering out duplicates
- Caching/memorizing data instead of making your server do work

## compare hash tables to arrays and lists

	HASH TABLES (AVERAGE)	HASH TABLES (WORST)	ARRAYS	LINKED LISTS
SEARCH	$O(1)$	$O(n)$	$O(1)$	$O(n)$
INSERT	$O(1)$	$O(n)$	$O(n)$	$O(1)$
DELETE	$O(1)$	$O(n)$	$O(n)$	$O(1)$

**Hash table pros :**

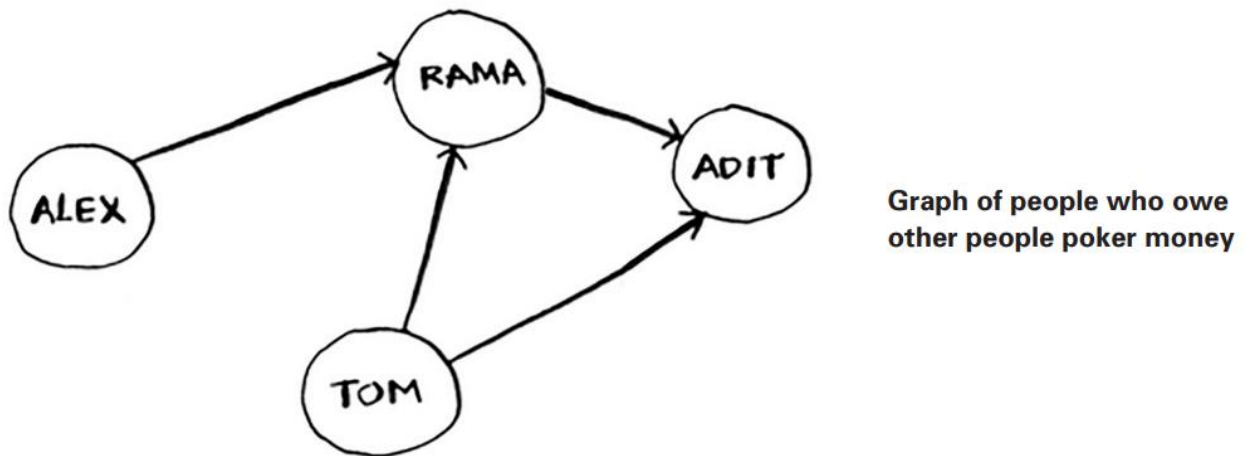
- You can make a hash table by combining a hash function with an array.
- Collisions are bad. You need a hash function that minimizes collisions.
- Hash tables have really fast search, insert, and delete.
- Hash tables are good for modeling relationships from one item to another item.
- Once your load factor is greater than .07, it's time to resize your hash table.
- Hash tables are used for caching data (for example, with a web server).
- Hash tables are great for catching duplicates.

## Chapter 6 : Breadth-first search

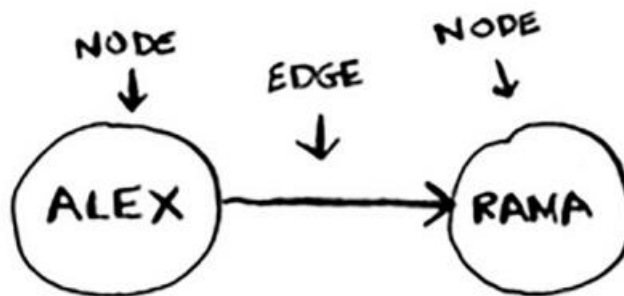
The algorithm to solve a shortest-path problem is called breadth-first search.

### What is a graph?

A graph models a set of connections



Each graph is made up of nodes and edges



Graphs are made up of nodes and edges. A node can be directly connected to many other nodes. Those nodes are called its neighbors. In this graph, Rama is Alex's neighbor. Adit isn't Alex's neighbor, because they aren't directly connected. But Adit is Rama's and Tom's neighbor. Graphs are a way to model how different things are connected to one another. Now let's see breadth-first search in action

## Breadth-first search

Breadth first search is a different kind of search algorithm: one that runs on graphs. It can help answer two types of questions:

- Question type 1: Is there a path from node A to node B?
- Question type 2: What is the shortest path from node A to node B?

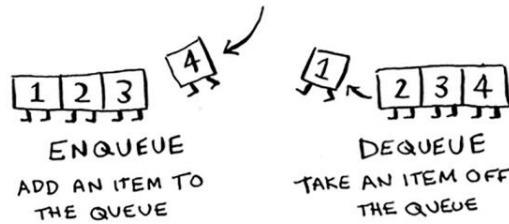
**Notice that this only works if you search people in the same order in which they're added.**

What happens if you search Anuj before Claire, and they're both mango sellers? Well, Anuj is a second-degree contact, and Claire is a first-degree contact. **You end up with a mango seller who isn't the closest to you in your network. So you need to search people in the order that they're added. There's a data structure for this: it's called a queue**

## Queues :

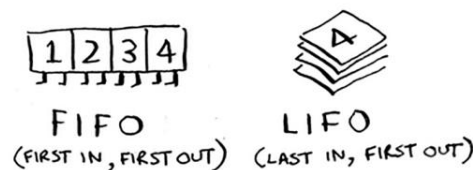
A queue works exactly like it does in real life.

A queue works the same way. Queues are similar to stacks. **You can't access random elements in the queue.** Instead, there are two only operations, enqueue and dequeue



If you enqueue two items to the list, the first item you added will be dequeued before the second item. You can use this for your search list! People who are added to the list first will be dequeued and searched first.

The queue is called a *FIFO* data structure: First In, First Out. In contrast, a stack is a *LIFO* data structure: Last In, First Out.



## How to implement Graph :

A graph consists of several nodes. And each node is connected to neighboring nodes. How do you express a relationship like “you -> bob”? Luckily, you know a data structure that lets you express relationships: a hash table! Remember, a hash table allows you to map a key to a value. In this case, you want to map a node to all of its neighbors.

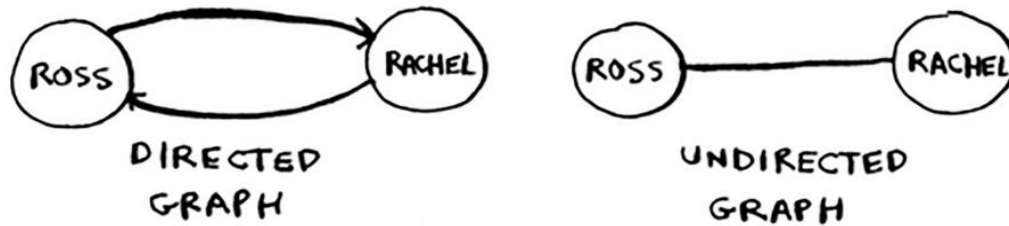
```
graph = {}
graph["you"] = ["alice", "bob", "claire"]
```

### directed graph :

- the relationship is only one way
- **A directed graph has arrows**, and the relationship follows the direction of the arrow (rama -> adit means “rama owes adit money”)

### undirected graph :

- **doesn't have any arrows**, and both nodes are each other's neighbors
- Undirected graphs don't have arrows, and the relationship goes both ways (ross - rachel means “ross dated rachel and rachel dated ross”)

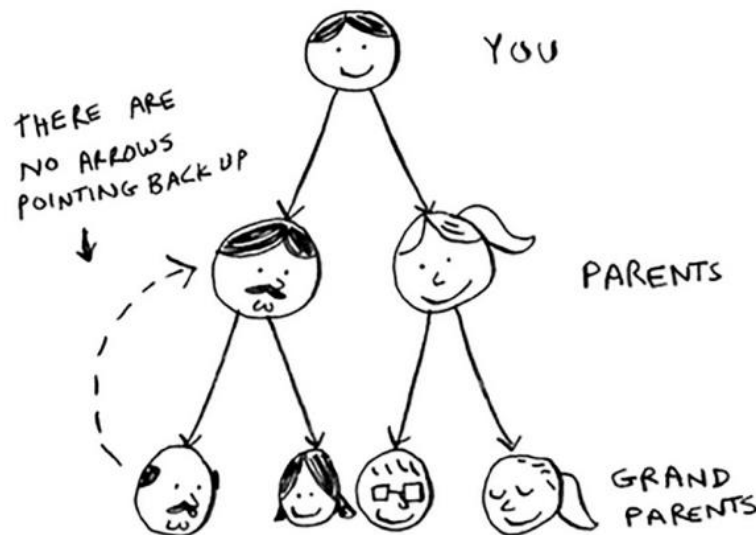


Breadth-first search takes  $O(\text{number of people} + \text{number of edges})$ , and it's more commonly written as  $O(V+E)$  (V for number of Nodes, E for number of edges).

### Important when implementing breadth First search

- You need to check people in the order they were added to the search list, so the search list needs to be a queue. Otherwise, you won't get the shortest path.
- Once you check someone, make sure you don't check them again. Otherwise, you might end up in an infinite loop

**Tree** : A tree is a special type of graph, **where no edges ever point back**.



Trees are a subset of graphs. So a tree is always a graph, but a graph may or may not be a tree

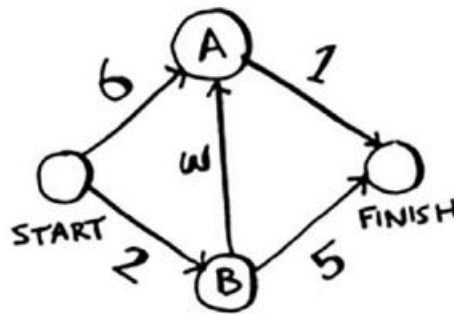


## Chapter 8 : Dijkstra's Algorithm

Breadth first Algorithm: find shortest path

Dijkstra's Algorithm: find fastest path

There are four steps to Dijkstra's algorithm:

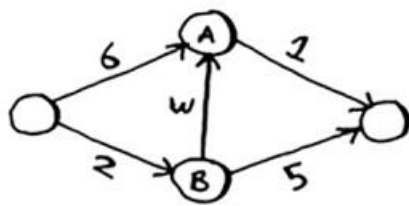


1. Find the cheapest node. This is the node you can get to in the least amount of time.
2. Check whether there's a cheaper path to the neighbors of this node. If so, update their costs.
3. Repeat until you've done this for every node in the graph.
4. Calculate the final path.

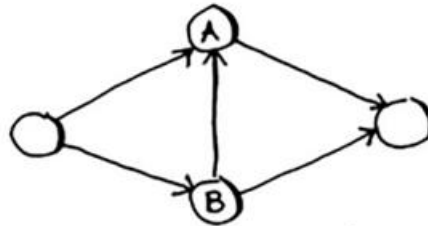
### Diff between Breath First & Dijkstra :

breadth-first search to find the shortest path between two points. Back then, “shortest path” meant the path with **the fewest segments**. But in Dijkstra's algorithm, you assign a number or weight to each segment. Then Dijkstra's algorithm finds the path with the **smallest total weight**.

## Weighted vs Unweighted Graphs :



WEIGHTED GRAPH



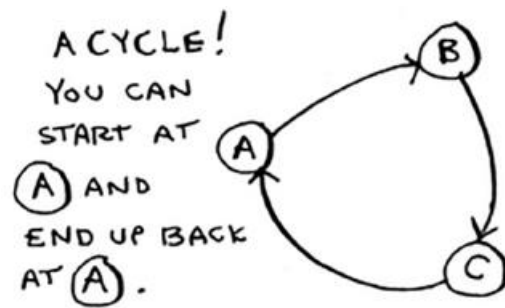
UNWEIGHTED GRAPH

## Usage of breadth-first search & Dijkstra's algorithm :

To calculate the shortest path in an unweighted graph, use breadth-first search. To calculate the shortest path in a weighted graph, use Dijkstra's algorithm

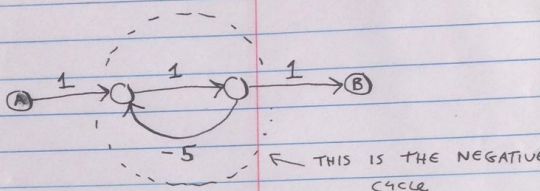
## Cycle in Graphs :

it means you can start at a node, travel around, and end up at the same node.



Dijkstra's algorithm works even if there is a cycle, as long as it is a positive weight cycle.

GRAPH WITH A NEGATIVE CYCLE:



COST FROM A TO B IF YOU TAKE THE CYCLE  $\infty$  TIMES:

$$0 \xrightarrow{1} \rightarrow \rightarrow 0 = 3$$

IF YOU TAKE THE CYCLE 1 TIME:

$$0 \xrightarrow{1} \rightarrow \rightarrow 0 = 1 + 1 - 5 + 1 + 1 = -1$$

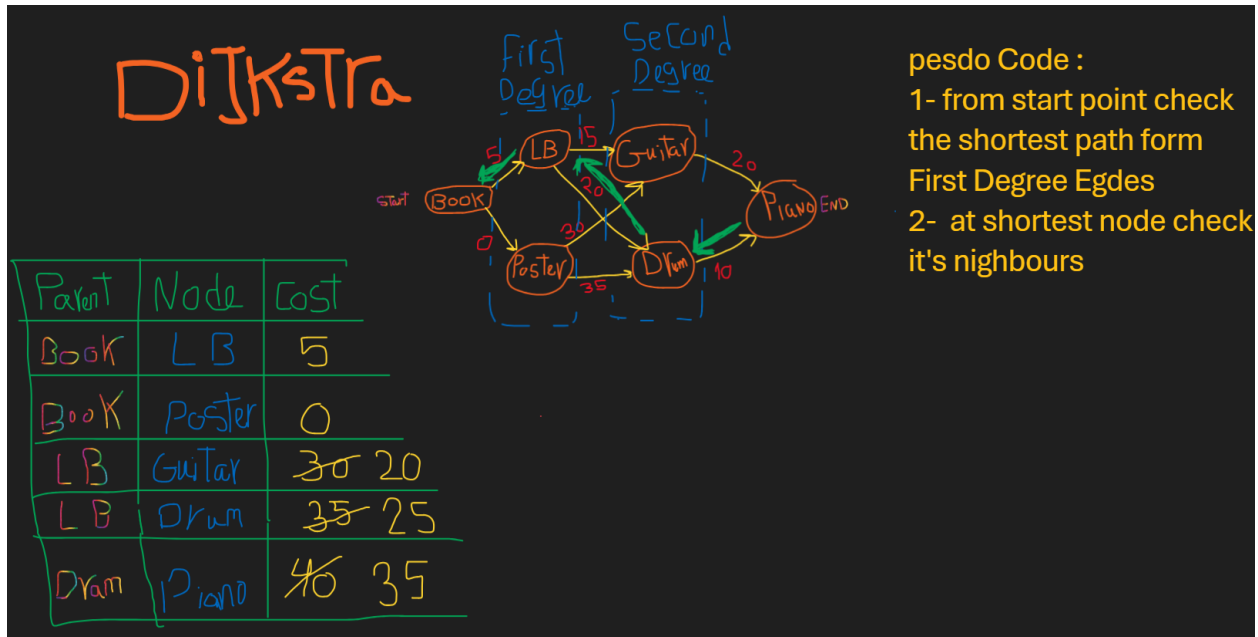
IF YOU TAKE THE CYCLE 2 TIMES:

$$0 \xrightarrow{1} \rightarrow \rightarrow 0 = 1 + 1 - 5 + 1 - 5 + 1 + 1 = -5$$

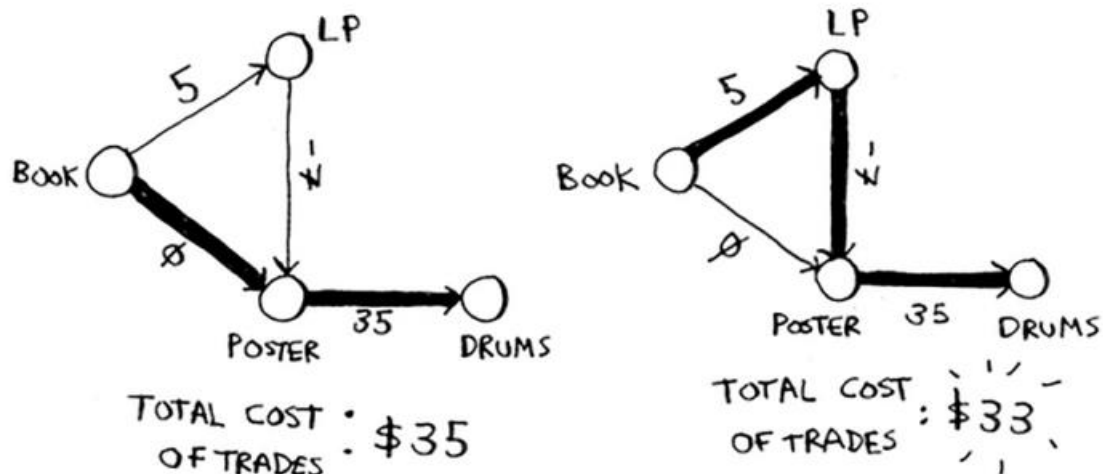
IF YOU TAKE THE CYCLE 3 TIMES:

$$0 \xrightarrow{1} \rightarrow \rightarrow 0 = 1 + 1 - 5 + 1 - 5 + 1 - 5 + 1 + 1 = -9$$

Example:



**Negative Weight Edges :**



If you run Dijkstra's algorithm on this graph, Rama will take the wrong path. He'll take the longer path. You can't use Dijkstra's algorithm if you have negative-weight edges. Negative-weight edges break the algorithm

**the cost of a node is how long it takes to get to that node from the start**

## Chapter 8 : Greedy Algorithm

هو الجورزم بيستخدم لحل أي مشكلة و بيديك ناتج تقريبي و بيعتمد انه بيبدأ من نقطة و يروح لاقرب نقطة بتتطبق شروط معينة

**Greedy Algorithm :** A greedy algorithm is simple: at each step, pick the optimal move.

greedy algorithms used to solve Optimization problem :

Greedy Algorithm Terms:

**Optimization problem:** Problem either require maximum or minimum result

**Feasible Solution :** solution that meet constrains

**Optimal solution :** solution that achieving the objective of problem either require maximum or minimum result

Solutions of optimization problem :

- 1- Greedy algorithm
- 2- Dynamic programming
- 3- Bianch And Bound

in computer science, NP (short for "nondeterministic polynomial time") refers to a class of problems that share specific characteristics related to their solvability.

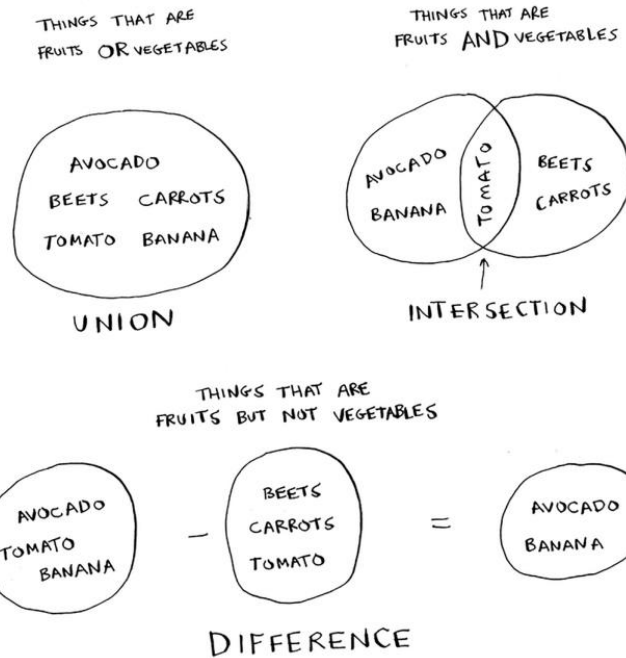
### **Examples of NP Problems:**

- **Traveling Salesman Problem (TSP):** Given a list of cities and the distances between them, find the shortest possible route that visits each city exactly once and returns to the starting city.
- **Knapsack Problem:** Given a list of items with weights and values, fill a knapsack with the highest overall value without exceeding its weight limit.
- **Vertex Cover Problem:** In a graph, find the smallest set of vertices that covers all edges (at least one vertex from the set is connected to each edge).

## Sets:

A set is like a list, except that each item can show up only once in a set. Sets can't have duplicates

Here are some things you can do with sets.



- A set union means “combine both sets.”
- A set intersection means “find the items that show up in both sets” (in this case, just the tomato).
- A set difference means “subtract the items in one set from the items in the other set.”

## Chapter 9 : Dynamic Programming

Dynamic programming starts by solving subproblems and builds up to solving the big problem to find optimal set

Knapsack problem :

Dynamic Programming  
Knapsack:-

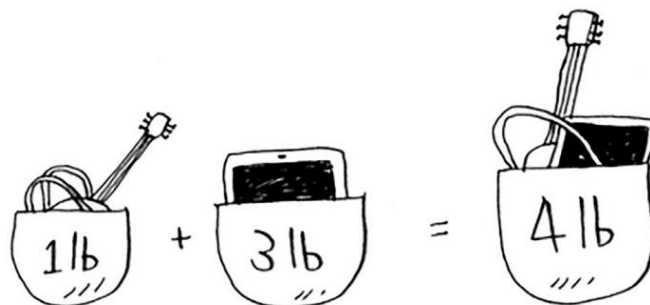
	1	2	3	4
Guitar	1500 G	1500 G	1500 G	1500 G
STERO	1500 G	1500 G	1500 G	3000 G
LAPTOP	1500 G	1500 G	2000 L	3500 L.G

ANSWER

Why calculating max values for smaller knapsacks ?

When you have space left over, you can use the answers to those subproblems to figure out what will fit in that space. It's better to take the laptop + the guitar for \$3,500.

You combined the solutions to two subproblems to solve the bigger problem





## Handling items that depend on each other

- Dynamic programming is powerful because it can solve subproblems and use those answers to solve the big problem. Dynamic programming only works when each subproblem is discrete—when it doesn't depend on other subproblems

## Is it possible that the solution will require more than two sub-knapsacks ?



- Dynamic programming is useful when you're trying to optimize something given a constraint. In the knapsack problem, you had to maximize the value of the goods you stole, constrained by the size of the knapsack.
- You can use dynamic programming when the problem can be broken into discrete subproblems, and they don't depend on each other.
- Every dynamic-programming solution involves a grid
- The values in the cells are usually what you're trying to optimize. For the knapsack problem, the values were the value of the goods.
- Each cell is a subproblem, so think about how you can divide your problem into subproblems. That will help you figure out what the axes are
- here's no single formula for calculating a dynamic-programming solution

## Longest common substring:

1. IF THE LETTERS  
DONT MATCH,  
THE VALUE IS  
ZERO.

	H	I	S	H
F	0	0	0	0
I	0	1	0	0
S	0	0	2	0
H	0	0	0	3

2. IF THEY DO MATCH,  
THIS VALUE IS  
VALUE OF TOP-LEFT NEIGHBOR + 1

the final solution may not be in the last cell! For the knapsack problem, this last cell always had the final solution. But for the longest common substring, the solution is the largest number in the grid—and it may not be the last cell.

### **So is dynamic programming ever really used? Yes:**

- Biologists use the longest common subsequence to find similarities in DNA strands. They can use this to tell how similar two animals or two diseases are. The longest common subsequence is being used to find a cure for multiple sclerosis.
- Have you ever used diff (like git diff)? Diff tells you the differences between two files, and it uses dynamic programming to do so.
- We talked about string similarity. Levenshtein distance measures how similar two strings are, and it uses dynamic programming. Levenshtein distance is used for everything from spell-check to figuring out whether a user is uploading copyrighted data.
- Have you ever used an app that does word wrap, like Microsoft Word? How does it figure out where to wrap so that the line length stays consistent? Dynamic programming!

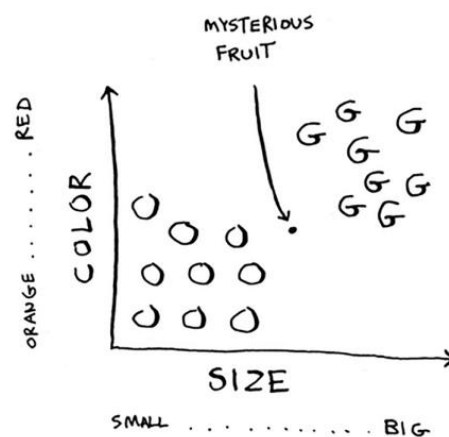
## Chapter 10 : K-nearest Neighbors [KNN]

KNN Algorithm: used to classify objects.

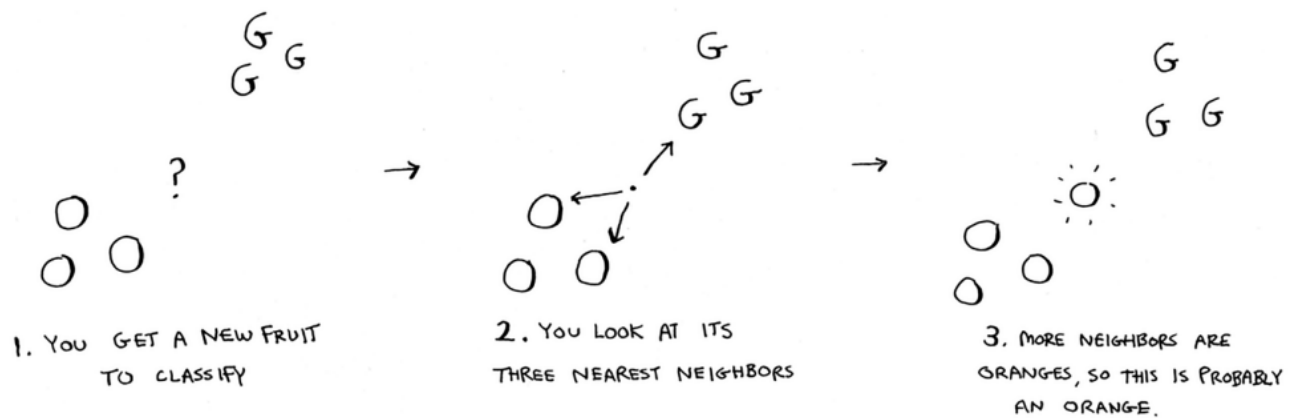
These are the two basic things you'll do with KNN—classification and regression:

- Classification = categorization into a group
- Regression = predicting a response (like a number)

### Classification:



look at the neighbors of this spot. Take a look at the three closest neighbors of this spot.



## How do you figure out how similar two objects are?

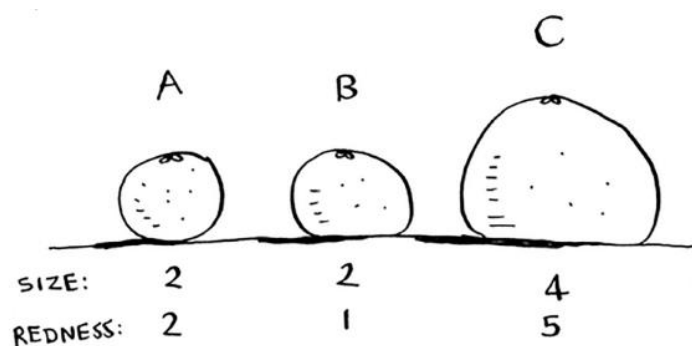
**Feature extraction:** capture the essential characteristics of the data that are most relevant to the task at hand,

Or means converting an item (like a fruit or a user) into a list of numbers that can be compared.

**Picking good features is an important part of a successful KNN algorithm.**

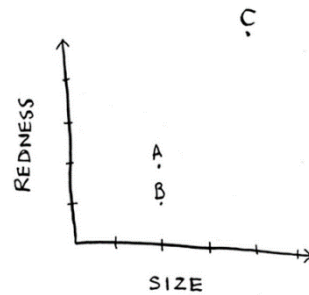
When you're working with KNN, it's really important to pick the right features to compare against. Picking the right features means

- Features that directly correlate to the movies you're trying to recommend
- Features that don't have a bias (for example, if you ask the users to only rate comedy movies, that doesn't tell you whether they like action movies)



In the grapefruit example, you compared fruit based on how big they are and how red they are. Size and color are the features you're comparing. Now suppose you have three fruit. You can extract the features.




Then you can graph the three fruit.



From the graph, you can tell visually that fruits A and B are similar. Let's measure how close they are. To find the distance between two points, you use the **Pythagorean formula**.

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

So, you need to convert each user to a set of coordinates

	 PRIYANKA	 JUSTIN	 MORPHEUS
COMEDY	3	4	2
ACTION	4	3	5
DRAMA	4	5	1
HORROR	1	1	3
ROMANCE	4	5	1

A mathematician would say, instead of calculating the distance in two dimensions, you're now calculating the distance in five dimensions. But the **distance formula remains the same**.

$$\sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 + (d_1 - d_2)^2 + (e_1 - e_2)^2}$$

**What does distance mean when you have five numbers?**

The distance tells you how similar those sets of numbers are.

**Regression:** try to predict value by get Average form closet neighbors values

Exp :

Suppose you're trying to guess a rating for Pitch Perfect. Well, how did Justin, JC, Joey, Lance, and Chris rate it?

You could take the average of their ratings and get **4.2 stars**.

JUSTIN : 5  
JC : 4  
JOEY : 4  
LANCE : 5  
CHRIS : 3

**You can calculate distance by**

- 1- Distance Formula
- 2- Cosine similarity

## Introduction to machine learning

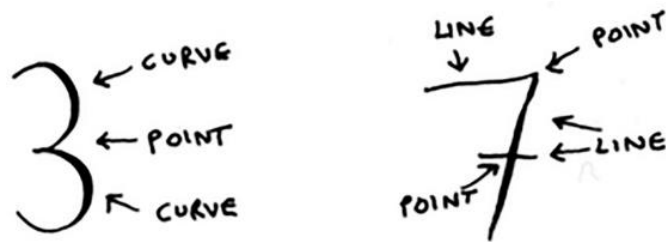
KNN is a really useful algorithm, and it's your introduction to the magical world of machine learning! Machine learning is all about making your computer more intelligent. You already saw one example of machine learning: building a recommendations system. Let's look at some other examples

## OCR:

stands for optical character recognition. It means you can take a photo of a page of text, and your computer will automatically read the text for you. Google uses OCR to digitize books.

How would you figure out what number this is? You can use **KNN** for this:

1. Go through a lot of images of numbers, and extract features of those numbers.
2. When you get a new image, extract the features of that image, and see what its nearest neighbors are!



Feature extraction is a lot more complicated in OCR than the fruit example. But it's important to understand that even complex technologies build on simple ideas, like KNN. You could use the same ideas for speech recognition or face recognition.

When you upload a photo to Facebook, sometimes it's smart enough to tag people in the photo automatically. That's machine learning in action! The first step of OCR, where you go through images of numbers and extract features, is called training. Most machine-learning algorithms have a training step: before your computer can do the task, it must be trained. The next example involves spam filters, and it has a training step.



## Exp of machine learning

### spam filter:

Spam filters use another simple algorithm called the Naive Bayes classifier. First, you train your Naive Bayes classifier on some data.

SUBJECT	SPAM?
"RESET YOUR PASSWORD"	NOT SPAM
"YOU HAVE WON 1 MILLION DOLLARS"	SPAM
"SEND ME YOUR PASSWORD"	SPAM
"NIGERIAN PRINCE SENDS YOU 10 MILLION DOLLARS"	SPAM
"HAPPY BIRTHDAY"	NOT SPAM

Suppose you get an email with the subject "collect your million dollars now!" Is it spam? You can break this sentence into words. Then, for each word, see what the probability is for that word to show up in a spam email. For example, in this very simple model, the word million only appears in spam emails. **Naive Bayes figures out the probability** that something is likely to be spam.

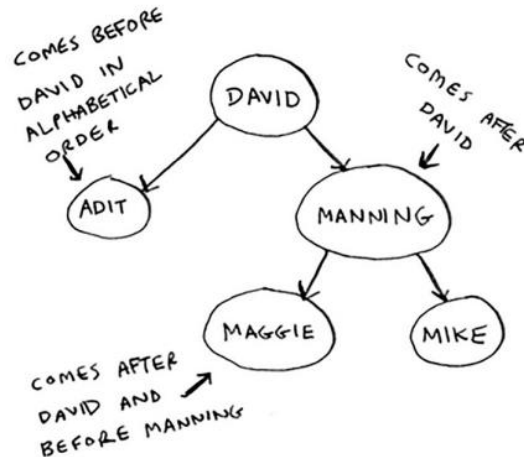
- There's no guaranteed way to use past numbers to predict future performance. Predicting the future is hard, and it's almost impossible when there are so many variables involved.
- Feature extraction means converting an item (like a fruit or a user) into a list of numbers that can be compared.
- If you look at fewer neighbors, there's a bigger chance that the results will be skewed. A good rule of thumb is, if you have  $N$  users, you should look at  $\sqrt{N}$  neighbors.

## Chapter 11 : Where To Go Next

### What is the problem with Binary Search ?

you have to re-sort the array, because binary search only works with sorted arrays

That's the idea behind the binary search tree data structure

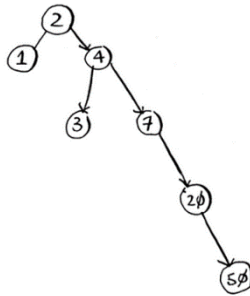


Searching for an element in a binary search tree takes  $O(\log n)$  time on average and  $O(n)$  time in the worst case. Searching a sorted array takes  $O(\log n)$  time in the worst case, so you might think a sorted array is better. But a binary search tree is a lot faster for insertions and deletions on average.

	ARRAY	BINARY SEARCH TREE
SEARCH	$O(\log n)$	$O(\log n)$
INSERT	$O(n)$	$O(\log n)$
DELETE	$O(n)$	$O(\log n)$

Disadvantage of Binary Search Tree:

- 1- you don't get random access. You can't say, "Give me the fifth element of this tree."
- 2- you have an imbalanced tree like the one shown next



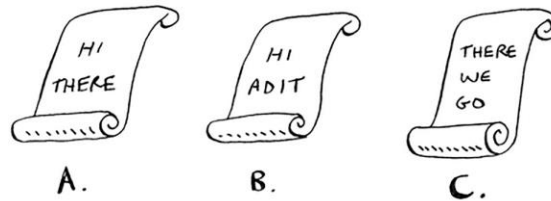
This tree doesn't have very good performance, because it isn't balanced.

### **special type of binary trees:**

- 1- B-trees
- 2- Red-black trees
- 3- Heaps
- 4- Splay trees

## inverted index:

Here's a very simplified version of how a search engine works. Suppose you have three web pages with this simple content.



The keys of the hash table are the words, and the values tell you what pages each word appears on. Now suppose a user searches for hi. Let's see what pages hi shows up on.



HI	A, B
THERE	A, C
ADIT	B
WE	C
GO	C

Inverted index data structure it's commonly used to build search engines.

## The Fourier transform:

- The Fourier transform is a powerful and versatile mathematical tool. It can analyze complex signals (like music) by breaking them down into their basic components (frequencies). This allows us to manipulate and understand the signal in various ways. Examples include:
- Boosting or hiding specific frequencies: This can be used to adjust bass or treble in music.
- Compressing signals: By identifying and discarding less important frequencies, the Fourier transform helps compress audio files (like MP3) and images (like JPG).
- Analyzing other signals: The technique is used beyond audio and images, for tasks like earthquake prediction or DNA analysis.
- You can use it to build an app like Shazam, which guesses what song is playing.

## Parallel algorithms:

- laptops and computers ship with multiple cores. To make your algorithms faster, you need to change them to run in parallel across all the cores at once!
- Here's a simple example. The best you can do with a sorting algorithm is roughly  $O(n \log n)$ . It's well known that you can't sort an array in  $O(n)$  time—unless you use a parallel algorithm! There's a parallel version of quicksort that will sort an array in  $O(n)$  time.
- Parallel algorithms are hard to design. And it's also hard to make sure they work correctly and to figure out what type of speed boost you'll see.
- Overhead of managing the parallelism—Suppose you have to sort an array of 1,000 items. How do you divide this task among the two cores? Do you give each core 500 items to sort and then merge the two sorted arrays into one big sorted array? Merging the two arrays takes time.
- Load balancing—Suppose you have 10 tasks to do, so you give each core 5 tasks. But core A gets all the easy tasks, so it's done in 10 seconds, whereas core B gets all the hard tasks, so it takes a minute. That means core A was sitting idle for 50 seconds while core B was doing all the work! How do you distribute the work evenly so both cores are working equally hard?

## MapReduce:

special type of parallel algorithm: the distributed algorithm

It's fine to run a parallel algorithm on your laptop if you need two to four cores, but what if you need hundreds of cores? Then you can write your algorithm to run across multiple machines. The MapReduce algorithm is a popular distributed algorithm. You can use it through the popular open source tool Apache Hadoop

### **Why are distributed algorithms useful?**

Suppose you have a table with billions or trillions of rows, and you want to run a complicated SQL query on it. You can't run it on MySQL, because it struggles after a few billion rows. Use MapReduce through Hadoop!

Or suppose you have to process a long list of jobs. Each job takes 10 seconds to process, and you need to process 1 million jobs like this. If you do this on one machine, it will take you months! If you could run it across 100 machines, you might be done in a few days.

Distributed algorithms are great when you have a lot of work to do and want to speed up the time required to do it. MapReduce in particular is built up from two simple ideas: the map function and the reduce function.

## The map function:

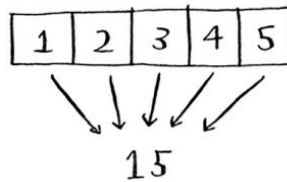
The map function is simple: it takes an array and applies the same function to each item in the array.

Wouldn't it be great if you had 100 machines, and map could automatically spread out the work across all of them? Then you would be downloading 100 pages at a time, and the work would go a lot faster! This is the idea behind the "map" in MapReduce.

## The reduce function:

The reduce function confuses people sometimes. The idea is that you "reduce" a whole list of items down to one item.

With `reduce`, you transform an array to a single item.



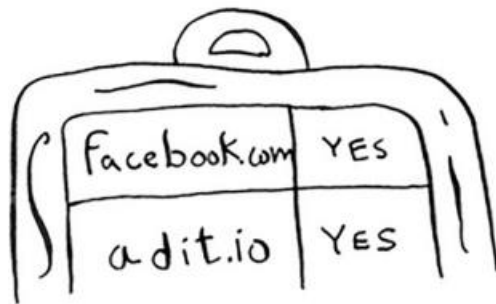
MapReduce uses these two simple concepts to run queries about data across multiple machines. When you have a large dataset (billions of rows), MapReduce can give you an answer in minutes where a traditional database might take hours.

## Bloom filters:

suppose you're Google, and you're crawling web pages. You only want to crawl a web page if you haven't crawled it already. So you need to figure out whether this page has been crawled before

Or suppose you're running bit.ly, which is a URL shortener. You don't want to redirect users to malicious websites. You have a set of URLs that are considered malicious. Now you need to figure out whether you're redirecting the user to a URL in that set. All of these examples have the same problem. **You have a very large set Now you have a new item, and you want to see whether it belongs in that set.**

You could do this quickly with a hash. For example, suppose Google has a big hash in which the keys are all the pages it has crawled.



The average lookup time for hash tables is  $O(1)$

Except that this hash needs to be huge. Google indexes trillions of web pages. If this hash has all the URLs that Google has indexed, it will take up a lot of space

**Bloom filters** offer a solution. Bloom filters are probabilistic data structures. They give you an answer that could be wrong but is probably correct. Instead of a hash, you can ask your bloom filter if you've crawled this URL before. A hash table would give you an accurate answer. A bloom filter will give you an answer that's probably correct:

- False positives are possible. Google might say, "You've already crawled this site," even though you haven't.
- False negatives aren't possible. If the bloom filter says, "You haven't crawled this site," then you definitely haven't crawled this site. Bloom filters are great because they take up very little space. A hash table would have to store every URL crawled by Google, but a bloom filter doesn't have to do that.



## HyperLogLog:

Along the same lines is another algorithm called HyperLogLog.

Suppose Google wants to count the number of unique searches performed by its users. Or suppose Amazon wants to count the number of unique items that users looked at today. Answering these questions takes a lot of space!

With Google, you'd have to keep a log of all the unique searches. When a user searches for something, you have to see whether it's already in the log. If not, you have to add it to the log. Even for a single day, this log would be massive!

**HyperLogLog approximates the number of unique elements in a set.** Just like bloom filters, it won't give you an exact answer, but it comes very close and uses only a fraction of the memory a task like this would otherwise take. If you have a lot of data and are satisfied with approximate answers, check out probabilistic algorithms!

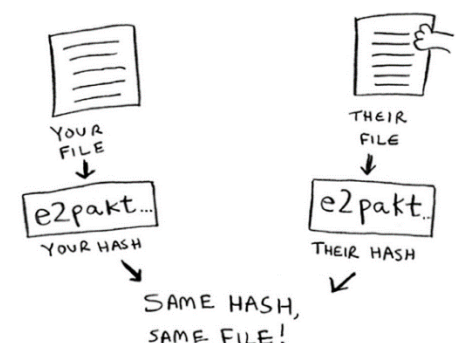
## The SHA algorithms:

Another hash function is a secure hash algorithm (SHA) function. Given a string, SHA gives you a hash for that string.

The terminology can be a little confusing here. SHA is a hash function. It generates a hash, which is just a short string. The hash function for hash tables went from string to array index, whereas SHA goes from string to string. SHA generates a different hash for every string.

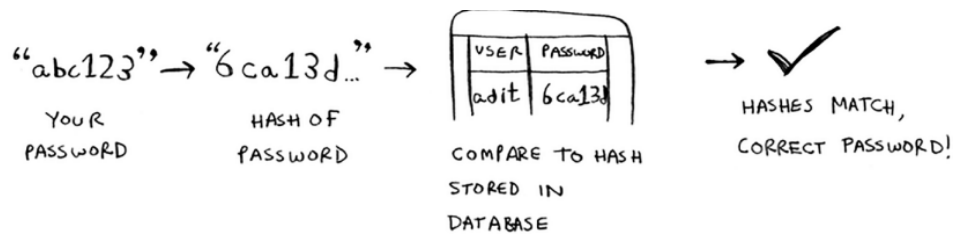
"hello"  $\Rightarrow$  2cf24db...  
"algorithm"  $\Rightarrow$  b1eb2ec..  
"password"  $\Rightarrow$  5e88489...

You can use SHA to tell whether two files are the same. This is useful when you have very large files. Suppose you have a 4 GB file. You want to check whether your friend has the same large file. You don't have to try to email them your large file. Instead, you can both calculate the SHA hash and compare it.



## Checking passwords

SHA is also useful when you want to compare strings without revealing what the original string was. For example, suppose Gmail gets hacked, and the attacker steals all the passwords! Is your password out in the open? No, it isn't. Google doesn't store the original password, only the SHA hash of the password! When you type in your password, Google hashes it and checks it against the hash in its database.



So it's only comparing hashes—it doesn't have to store your password! SHA is used very commonly to hash passwords like this. It's a one-way hash. You can get the hash of a string

abc123 → 6ca13d

But you can't get the original string from the hash

? ← 6ca13d

That means if an attacker gets the SHA hashes from Gmail, they can't convert those hashes back to the original passwords! You can convert a password to a hash, but not vice versa. SHA is actually a family of algorithms: SHA-0, SHA-1, SHA-2, and SHA-3. As of this writing, SHA-0 and SHA-1 have some weaknesses. If you're using an SHA algorithm for password hashing, use SHA-2 or SHA-3. The gold standard for password-hashing functions is currently bcrypt.

## Locality-sensitive

hashing SHA has another important feature: it's locality insensitive. Suppose you have a string, and you generate a hash for it.

dog → cd6357

If you change just one character of the string and regenerate the hash, it's totally different!

dot → e392da

This is good because an attacker can't compare hashes to see whether they're close to cracking a password.

Sometimes, you want the opposite: you want a locality-sensitive hash function. That's where Simhash comes in. If you make a small change to a string, Simhash generates a hash that's only a little different. This allows you to compare hashes and see how similar two strings are, which is pretty useful!

- Google uses Simhash to detect duplicates while crawling the web.
- A teacher could use Simhash to see whether a student was copying an essay from the web
- Scribd allows users to upload documents or books to share with others. But Scribd doesn't want users uploading copyrighted content! The site could use Simhash to check whether an upload is similar to a Harry Potter book and, if so, reject it automatically. Simhash is useful when you want to check for similar items.

## Diffie-Hellman key exchange:

The Diffie-Hellman algorithm deserves a mention here, because it solves an age-old problem in an elegant way. How do you encrypt a message so it can only be read by the person you sent the message to?

Diffie-Hellman has two keys: a public key and a private key. The public key is exactly that: public. You can post it on your website, email it to friends, or do anything you want with it. You don't have to hide it. When someone wants to send you a message, they encrypt it using the public key. An encrypted message can only be decrypted using the private key. As long as you're the only person with the private key, only you will be able to decrypt this message!

## Linear programming:

Linear programming is used to maximize something given some constraints. For example, suppose your company makes two products, shirts and totes. Shirts need 1 meter of fabric and 5 buttons. Totes need 2 meters of fabric and 2 buttons. You have 11 meters of fabric and 20 buttons. You make \$2 per shirt and \$3 per tote. How many shirts and totes should you make to maximize your profit? Here you're trying to maximize profit, and you're constrained by the amount of materials you have.

You might be thinking, "You've talked about a lot of optimization topics in this book. How are they related to linear programming?" All the graph algorithms can be done through linear programming instead. Linear programming is a much more general framework, and graph problems are a subset of that. I hope your mind is blown! Linear programming uses the Simplex algorithm. It's a complex algorithm, which is why I didn't include it in this book. If you're interested in optimization, look up linear programming.