Gentleman Programming Book

> A clean programmer is the best kind of programmer
>
> *- by Alan Buscaglia*

Chapter 1

Clean Agile

Chapter 2

Communication First and Foremost

Chapter 3

Hexagonal Architecture

Chapter 4

GoLang

Chapter 5

NVIM Gentleman Guide

Chapter 6

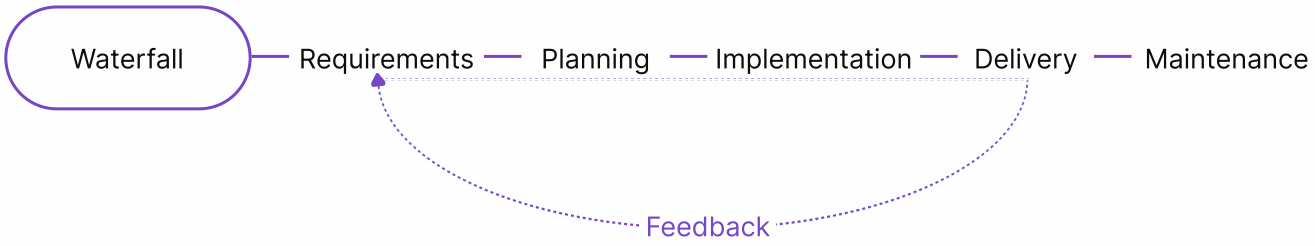
Algorithms the Gentleman Way

Clean Agile

Fantastic! Everything is agile today, all companies love agile, the entire world does agile, but ...are they?

Problems of waterfall:

Let's talk about waterfall, yeah, the bad boy in town, the one everybody hates. Waterfall in The main idea is:



- We get the requirements, what we want to do, the stakeholders needs.
- You plan how to do it, often comes with an analysis and design of the solution.
- You implement it, creating working software.
- You deliver the software and wait for feedback, creating documentation during the process.
- Maintenance of the working solution.

Once we deliver the solution, we ask for feedback and if we need to touch anything, we start the process all over again.

This is awesome as long as the requirements are super stable and we know they will not change during implementation, something that in the real world is practically impossible as they are ALWAYS changing.

If we were a fabric we wouldn't have any of these issues, as we know the specific required materials to create something, we put them in the machine, and the result is always going to be the same.

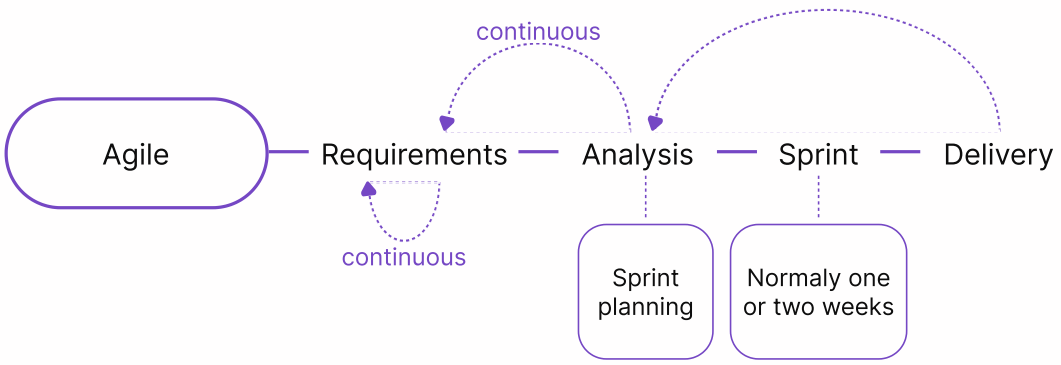
But we work with software solutions to meet people's needs, and these are always changing, evolving. Here comes the biggest problem with the waterfall methodology. We have to wait until the end of the implementation to receive feedback and then start the whole process again, so what if the user needs have changed in the meantime ? We've just wasted a lot of useful time.

One great analogy is the airplane's pilot one, you would like to be informed as soon as possible if there's any problem with the plane and not wait until the engine fails, or even worse, the airplane crashes to receive a notification.

Why agile?

Now here's the thing, a project it's an always evolving succession of events, for example, the analysis never ends ! so getting feedback as soon as possible is the main key of the Agile methodology. Here, we search for the stakeholders involvement in the whole process, delivering minimal amounts of functionalities waiting for a, let's hope positive, response; and if it's a negative one, no problems at all, we can attack them as soon as possible without having to wait for the end of the world to do so.

So how companies usually use agile is the following:



- We continuously get requirements, as we work on small functionalities we can choose what are the most critical needs and implement a plan of action to deliver small parts that can fulfill this end.
- We continuously do an analysis of the requirements to prepare future work. The amount will correspond to the timeframe we already decided according to the needs.
- Now with everything prepared we are comfortable to start working on our tasks inside a sprint. It represents the time frame we decided in which we compromise ourselves to deliver a certain amount of work and it can be variable depending on the need.
- We deliver the functionalities and continue with the process again. The main difference is that we start the work with stakeholders feedback from the previous iteration.

We can fail to deliver inside an adjusted timeframe, and that's not a problem at first, as we measure the team and recollect feedback to adjust to the next sprint. After some iterations we can estimate the correct amount of work the team can deliver in a certain context.

Why you think you are doing agile but in reality...you don't

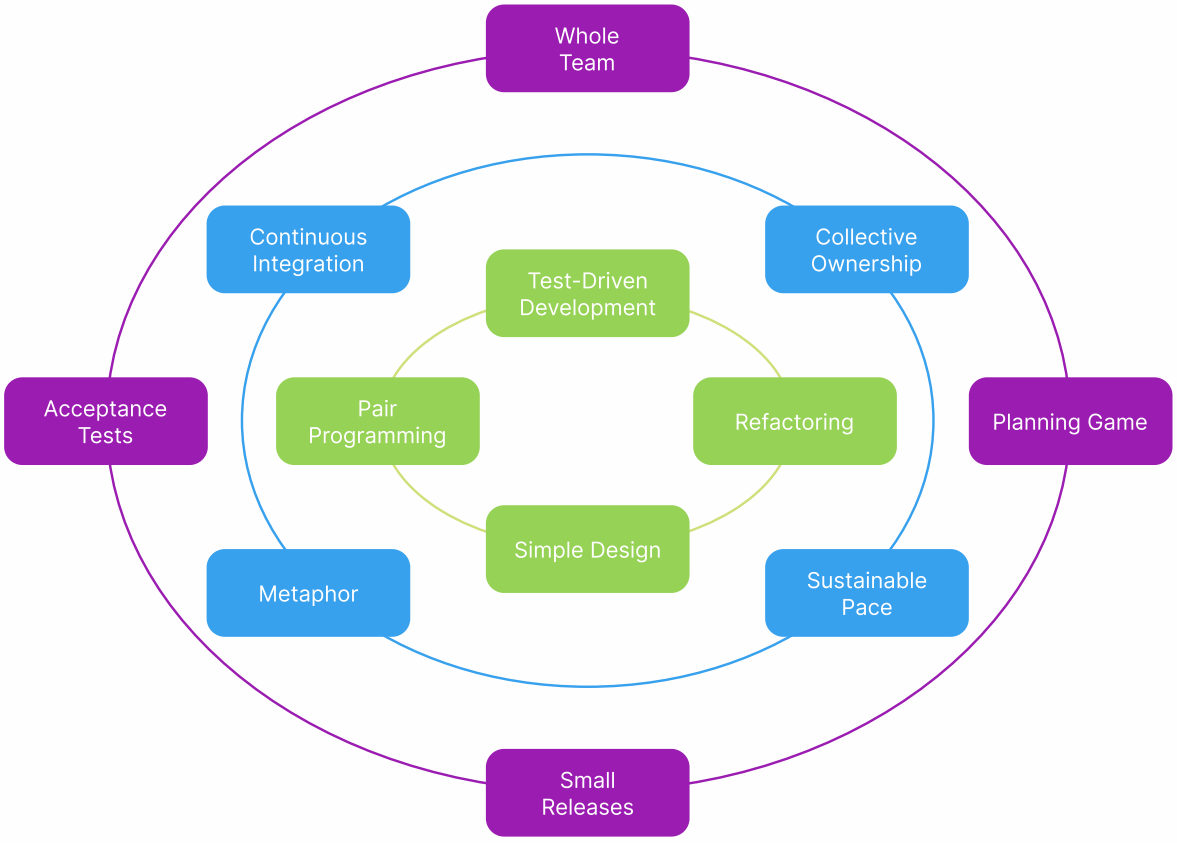
This is normal, you think you are doing agile because you have dailies and that makes the team agile, but in fact that is just one ceremony from many. You can't define being or not agile by the ceremonies that take place inside the project, as it's more of a way of thinking.

You may be working on sprints, using scrum, having retros, and all those amazing things but you also may be working in huge features, not delivering each sprint, not accepting any change until you finish your work or even being owner of your knowledge and not sharing it with the team. If you find yourself in any of these last items...ey...you are not doing agile.

Extreme Programming

This is an amazing practice that according to R.C Martin, co-founder of the agile manifesto and author of Clean Agile, is the true essence of it.

It consists of the organization of practices by three rings called the "Circle of Life". Each ring represents a different aspect of project development:



The "Outer Ring" represents the aspect of business, it contains all business focused practices which when put together creates the perfect environment for project development.

- **Planning Game:** grabbing the project and breaking it down into smaller pieces for better understanding and organization. Features, stories, tasks, etc.
- **Small Releases:** here comes what I was saying about attacking a whole functionality at once and how that could bring a waterfall approach to something that should be agile. We should always try to identify and prioritize the smaller pieces of value and work around them, being the work we want to deliver as soon as possible. The smaller the piece, the faster we will receive feedback and act accordingly.

- **Acceptance Tests:** now this is an easy one to understand but difficult to implement, we need to work in what we consider as a team as done, what are the requirements to really say something has been completely done, or at least on the agreed limitations. One recommendation is to think again in the minimum value we want to, and can, attack with the provided team. If we grab something really big it will be difficult to implement as we need to consider too many things, resulting in missed requirements, vague definitions and miscommunication. Consider creating functionalities that have a clear start and end, the result needs to be something that provides value on its own.
- **Whole team:** inside a proper project team, each member provides a certain functionality, we have our front-ends, back-ends, designers, product owners, project managers, etc. The main problem is always the same, how to communicate the work when it's so different and at the same time, dependent on each other (we will come back to this point in a little while).

The "Middle Ring" represents the aspect of the team, containing all team focused practices to improve team collaboration and interaction:

- **Sustainable Pace:** if you ask, when do you want this done ? the result will always be...well, as soon as possible ! and of course that's really difficult to do so, no because the team can't do it, most of them can, the problem is doing it every single time maintaining the same pace, it's impossible. Your team will be burned out at the third or fourth iteration and then no work will be done, the delivery speed will be greatly reduced. Again, think in small, contained functionalities, that can be delivered at a comfortable speed.
- **Collective Ownership:** how many times do you have to ask your pairs what the product owner is talking about in a meeting because you don't have the correct amount of context ? I'm not talking about those times you are gaming during the daily, but the vortex sucking all information that should be shared with the team and no one seems to know what's going on because no one ever told them. This is a super known issue in companies, the information happens at private quarters so only the people that were in the conversation know what's going on, and later on, try to communicate as best as possible the result with the rest of the team but they create a broken phone game in the process. The project needs to have a communication strategy to attack this kind of situation.
- **Continuous Integration:** as programmers we should be commiting code as much as possible, having the limit of not leaving a single day without new changes in the repository. There's always the possibility of working together in a functionality and missing communication efforts with each other. For example, one creates a method that does exactly the same as another, that was already created by a pair but didn't reach out to notify about it. Commiting changes as fast as possible will deliver feedback to your teammates and keep them updated to always be working on the "latest changes". If we wait to deliver new code after the functionality is done, we will enter waterfall territory all over again.
- **Metaphor:** If we will be working together in a project, we should all understand the context around in the same way, having definitions of each item. Having the same exact name to describe a certain item will bring a higher level of team understanding and leave out the confusion that could bring referring to the exact same item in more than one way. You could think of this as if each project is a different country, there are some of them that communicate in the exact same languages but they use different metaphors. For example, the United States and Londong both use English as their main language, but to represent being upset, American English uses "Disappointed" and British English uses "Gutted".

The "Inner Ring" represents the technical aspect, containing all practices related to improving technical work.

- **Pair Programming:** Sitting with each other to resolve a problem is not only going to make reaching a solution faster, but at the same time you are sharing your point of view between your pairs and also gaining theirs, with also the benefit of reaching a middle ground and creating a set of conventions that the team will follow after. Communication is key when working in a team, and having the possibility to work together to resolve a problem will bring feedback and context around the implementation.
- **Simple Design:** Here we go again, work small. We have already talked about this one but let's bring a little tip, let's say that we want to bring a certain functionality that represents a pretty big challenge to the team, we should always search for a way to provide the same amount of value by giving a much easier alternative. Sometimes we challenge ourselves and deliver a really complex but beautiful proposition of value, but the problem is that maybe that proposition goes nowhere because requirements change and we may find out that in reality the user doesn't want it, that's why also working as simple as possible is the way to go. You can always provide a simple but elegant solution to find the proper value and then iterate on something better.
- **Refactoring:** we all love the phrase "if it works, don't touch it", but that's not the correct mindset as we will enter in a spiral of legacy code by reusing no longer maintainable code. We need to refactor as much as possible. Technical debt is pretty much unavoidable, we always generate some bad quality code because of deadline's time constraints by implementing the fast, but not so correct, solution. One good way of dealing with it is to use part of the start or end of the sprint, according to the priority, to refactor the code, also this could be done using pair programming to use the benefits previously described.
- **Test-Driven Development:** We will talk about it later on, but we can define it as a process where we write our tests before even coding a single line. The main idea is that the requirements of the task define the tests we want to do and in result guide what we code.It can be a great ally when refactoring as we will understand later.

TDD

EVERYONE hates doing tests, for example clients hate PAYING companies for their Front End devs "wasting" time doing tests, and in the end... money. So why we, the wasters of time and money, should want to implement testing right?

Well, there are some things in my mind that can make up for all that hate:

- Code quality
- Code maintenance
- Coding Speed (yeah, you are reading this item correctly)

Understandable right ? you write tests so they pass the use cases, to write them you need to be organized because if not... it will be impossible to test your code. But there are things to consider, how do we write tests in a way that really increases the quality of our code in any meaningful way ?

First let's see what code quality means, and then I will tell you my take on what code quality means to ME.

If you look for an answer this is the one you may find:

"A quality code is one that is clear, simple, well tested, bug-free, refactored, documented, and performant"

Now, the measure of quality goes by the company requirements and the key points are usually reliability, maintainability, testability, portability, and reusability. It's really difficult to create code with 100% of quality, even Walter White couldn't create meth with more than 99.1% of purity; development problems, deadlines and other context and time consuming situations will arise endangering your code quality.

You can't write readable, maintainable, testable, portable, and reusable code if you are being rushed to finish a 4 point story task in just a morning (I really hope that's not your case, and if it is...you got this)

So here comes my take on what code quality is for me. Doing it, it's a mix of doing your best with the current tools, good practices and experience, against the existing context boundaries to create the cleanest code possible. My recommendation to all my students is to reach the objective first and then, if you have time, use it to improve the quality as high as possible. It is better to deliver an ugly thing than an incomplete, but beautiful, functionality. The quality of your code will increase with your experience along the way, as you gain more of it, you will know the best steps to reach an objective in the least amount of time and with the best practices.

Quality code also relates to the level of communication you can provide to your teammates or anyone in a simple glance. It's easy to see a code and say..wow, this is great ! and also say...wow, what a mess ! So when you code, you need to think that you are not the only one working on it, even if you are working alone as a single dev army, that will help a lot.

So let me give you some tools to write better code, first let's open your mind a little.

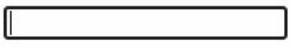
Atomic design, Front End point of view

Separate your code in the minimum piece of logic as possible, the smaller the code the easier to test. This also brings more benefits, like reusage of the code, better maintenance and even better performance; as the code gets smaller and better organized and depending on the language/framework we use, we could end in less processing cycles.

Maintenance will be greatly improved, as we are coding small pieces of work, each one with the loosest coupling and highest cohesion as possible, we can track and modify the code with the minimum number of problems.

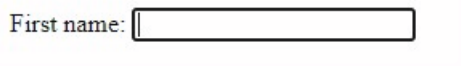
Let me show you how to think atomically, and how you can reach a complete app starting from a small input.

First we have our input:



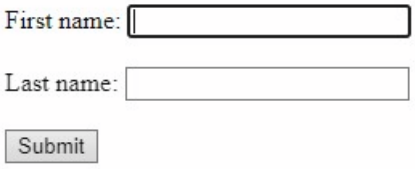
simple stuff, now that is what we call an Atom, the minimum piece of logic as possible. If you code it atomically you can reuse this input everywhere in your app and, later on, if you need to modify its behavior or its looks you just modify one little atom with the result of having an impact on the whole application.

Now, let's say you add a label to that input:

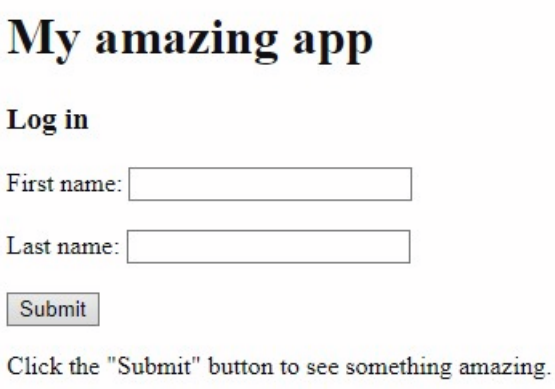


Congrats ! Now you have what it's called a Molecule, the mix between atoms, in this case a label and an input. We can continue going forward and reducing granularity.

We can use the input with the label inside a Form creating an Organism, the mix between molecules:



If we mix Organisms, we will get a **Template**:



And a collection of templates creates our Page, and then using the same logic, our App.

Using this way of thinking will result in your code being really maintainable, easy to browse to track errors and more than anything...easy to test !

If you write anything other than an Atom, it would be really difficult to test anything, as the logic would be of high coupling and therefore impossible to separate enough so that you can check specific cases.

One example would be testing a high coupled code, to validate just a simple thing one would need to start including one piece of code...and then another...and another, and after you finish you will see that you included almost the whole code because there were just too many dependencies from one place into another.

And that's the key to include a mvp (most valuable player) in all of this.

Functional Programming

So functional programming it's a paradigm that specifies ways of coding in a way that we divide our logic into declarative, with no side effects methods. Again...think atomically.

When we start learning how to code we normally do it in an Imperative way, where the priority is the objective and not the way we reach it. Even if it's faster than functional programming, which it is, it can bring a lot of headaches apart from leaving aside all the benefits from the other one.

Let's write a comparison in Javascript.

Imperative way of searching an element inside an array:

```
var exampleArray = [
    { name: 'Alan', age: 26 },
    { name: 'Axel', age: 23 },
];

function searchObject(name) {
    var foundObject = null;

    var index = 0;

    while (!foundObject && exampleArray.length > index) {
        if (exampleArray[index].name === name) {
            foundObject = exampleArray[index];
        }

        index += 1;
    }

    return foundObject;
}

console.log(searchObject('Alan'));

// { name: 'Alan', age: 26 }
```

And now the functional way of reaching the same objective:

```
const result = exampleArrayMap.find(element => element.name === name);

console.log(result);
```

Is not only shorter, it's also scalable. The map method we are applying into the array is a declarative one from ECMAScript, that means that every single time we run the method using the same parameters, we will always get the same result. We also won't modify anything outside the method, that's what's called side effects, the method returns a new array with the elements that comply with the condition.

So if we create methods representing the minimal units of logic as possible, we can reuse working and tested code across the app and maintain it if needed. Again...think atomically.

Now that we know the way of thinking to create high quality and easy maintenance code, let's go into what an User Story is.

🕶 User Story and TDD

What a title right ? Everyone knows about user stories, how to define them, what we need to do with them, but no one follows the same way of writing one or even its structure.

A user story is the explanation of a feature from an user point of view. It normally looks something like this:

What? As an user (who), I want to have the possibility of...(what) to...(why)

How? (use cases) 1- step 1 2- step 2 3- step 3 ...

So as you can see, we define who...the user, what he wants to do...the functionality, why we want this functionality...which while writing it we can even discover that it doesn't even make sense creating it because the objective is not clear, and how we will create it...the use cases.

The use cases represent the number of requirements that we need to fulfill to clearly say that a user story is done, they normally tell the story of the happy path to follow. There are also places where the entities related to the user story and the corner cases (sad path) are also described inside them and I believe that's a really good practice, but the same as when writing high quality code...we need to identify the boundaries of our context to see how can we write the content as specific as possible without transforming our task into a really difficult to follow and time consuming document.

Now TDD, Test Driven Development, it's a process where we define our tests before even coding a single line, so...how do we test something that's not even created right ? Well, that's the magic of it, you can grab your use cases and define what you need to reach each one of them, create tests around them, make them fail, and then fix them to pass..simple as that.

The main idea behind TDD is to think about:

- What do you want to do?.
- What are the core requirements?
- Write tests that will fail around them.
- Create your code knowing what you want to do.

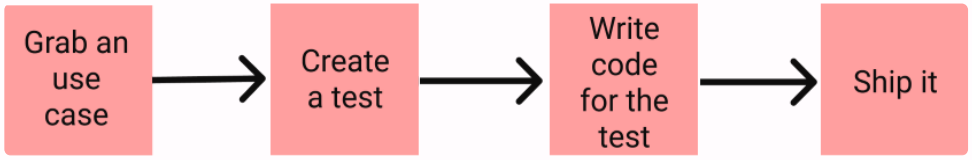
- Make the test pass.

If you think testing is time consuming, well it is, but because you may have previously done it in the classic bad way of first coding everything and then trying to test your code. Remember what we were talking about code quality, good practices, etc ? Well, those are the main elements that will help you test your code and if you are not implementing them correctly we will end in an impossible to separate and test functionality.

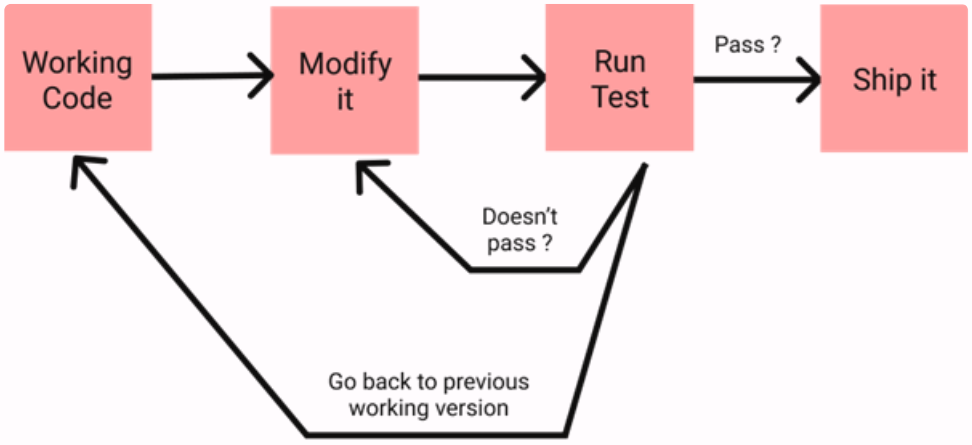
That's why coding knowing what you want to test, using functional programming and an atomic way of thinking can be so beneficial, because you will be creating logic, pinpointing the requirements and in the end... increasing the coding speed.

So here it is, testing also helps increase the coding speed, as you write more manageable code, it's easier to modify a requirement (use case) of your functionality as you have it identified by a test that will tell you if your refactor went correctly. It also reduces the possibility of bugs, so less time fixing problems later on.

TDD flow:



Here's a TDD flow on how to improve code quality without breaking anything:



Communication First and Foremost

We are reaching a new era! Remote work is coming strong and as the commodities increase also problems communicating between distributed teams.

Companies are not grounded to a certain location

Companies are not grounded to their office's location as they now play between the whole world's rules, so we need to change the mindset to understand this new paradigm. A great example is giving a job offer to a candidate, if we consider a salary limited to the candidate's location we run the risk of it being declined as he may have received offers from all over the globe, more tempting and better paid.

This concept may induce problems at any organizational level, people will compare themselves and what they do to professionals from all over the world and may think that they are not being offered the same level of benefits or that they just are not being paid enough. So how do you deal with this problem? making them feel part of something awesome, helping them overcome blockers to personal growth and most important of them all, keeping them learning new stuff.

Not everything that shines is gold goes the phrase, and we can apply the same concept between a company and its employees. People do not always search for money! knowledge is one of the greatest values one can provide as I always preach the following: “Knowledge first, money second; The more you know, the more someone is willing to pay you for it”

Coordinating across time zones

Being a distributed first company is not an easy task, managing work synergy across individuals that are not even in the same time zone can result in a bigger challenge than anticipated.

I recommend learning how to work asynchronously, it is just putting the puzzle pieces together, but the challenge is to find which are those pieces. Along with my professional experience, I detected that the most important and also difficult piece is generating a balanced amount of context among the team.

The way I have been doing it over the years is by understanding that communication is key and secondary to none, your team needs to be in the same boat or it will not work.

Second, you need to know your sources of truth, a place where you can check to clear doubts and search for context because it's always up to date. From my experience I found that there are two different kinds:

- One represents the reality of each business logic, a great example is the usage of Notion or Confluence as sources of truth, where we detail what we expect, why we are doing what we are doing, requirements, corner cases, etc.
- There's also the need for a second type related to workloads and what the team has agreed to do, and is one you already know, the created tickets that the team will complete over the passage of time.

We need two different types because while the first one gives us all the context we may need, we also have the necessity to put a stop at some point and decide when to start the implementation process. This last one details what the team has agreed on doing with the correct amount of context so they can provide enough value and don't be blocked in the process, and we need it so we can continue working on improving and evolving.

Let's think about what happens when having doubts about a certain task, a lot of people would create a comment inside the ticket, leave a message inviting communication over a chat channel, etc. This usually ends in a lot of meaningful chatter, confusion and more doubts, as we can't fully understand the intention of the text. How to fix it? Just sort the order of the elements in a different way:

- Ask for a call, talk with each other and adapt in a way that needs are fulfilled on both sides.
- Document the results over the related ticket and leave a history that can be tracked for future similar situations.
- It's ok to use chat channels for simple subjects but move into a meeting as soon as you see that the conversation is going nowhere.
- Meetings are not meant to add context or new information, please update the source of truth and share it as a first choice.
- Meetings are not meant to add context or new information, please update the source of truth and share it as a first choice.
- Include the related source of truth link inside the tickets, but also try to add all the needed information from it directly into the ticket to leave a definition of what has been agreed at the moment of creation.

There are some other easy ways we improve this mindset

We can start by over-communicating decisions across all geographies, this will result in people understanding what's going on and the whys.

Minimizing the friction in setting up a work environment, having documentation and some sort of guidelines will improve new additions to the team processes of getting to know the way they are expected to work and getting up to the required level to start working.

Clearly define the definition of done, this one relates to the need of having a workload source of truth, what are the acceptance criteria we need to complete to move a task as "done". Also, remember what I said before, a feature needs to have a start and an end of its own in a way that can provide value while being independent.

Using some of the concepts already provided in the previous chapters, for example doing pair programming or code reviews helps distribute knowledge between offices, helps generate a structure between global teams and minimizes the amount of collaboration required.

Creating Rapport

Reaching an accepted level of affinity between distributed teams can prove to be challenging but there are some things we can do to increase our possibilities:

- Communicate even minute details until both offices find a healthy groove.
- Communicate decisions.
- Everyone needs to understand the decision and why it was made.
- Don't use emails as they are an easy way of losing information.
- Use a content management system, for example, a wiki.
- Create channels for individuals and teams to communicate and see updates, another great idea would be to create channels for a certain future and the involved people.
- Spend time creating a simple but effective "Getting Started" guide.

Hexagonal Architecture

Hexagonal architecture is widely used and for good reason. This architecture advocates for "separation of concerns," meaning that business logic is divided into different services or hexagons, which communicate with each other through adapters to the resources they need to fulfill their purposes.

Hexagon and its Actors

It is called hexagonal because its shape resembles a hexagon with a vertical line dividing it in half. The left half refers to primary actors, who initiate the action that starts the hexagon's operation. These actors do not communicate directly with the service; instead, they use an adapter. The right side represents secondary actors, who provide the resources needed for the hexagon to execute its internal logic.

Adapters are key components in hexagonal architecture as they mediate communication between two entities so that they can interact comfortably. For example, in a service that provides user information, the term "username" might be used to identify the user's name, while in another service, the term "userIdentifier" might be used for the same action. Here is where the adapter intervenes to perform a series of transformations, allowing the information to be used in the most convenient way for each entity. In essence, adapters facilitate the integration of different components of the system and interoperability between them.

When an adapter communicates with a primary actor, it is called a driver, while when communicating with a required resource, it is called driven. The drivens are on the right side of the vertical line in the hexagonal diagram, and it is important to note that they can also represent other services. In this case, communication between services should occur through the corresponding adapters, drivers for the service providing the resource and drivens for the service requesting it. Thus, a service can act as the primary actor for another service.

Ports and Resources

The next important concept to understand is that of ports. These indicate the limitations that both our service and adapters have and represent the different functionalities they must provide to primary and secondary actors to meet requests and provide necessary resources.

Types of Logic in a Service

To understand the different types of logic within a service, it is useful to distinguish between business logic, organizational logic, and use cases. An example illustrating this distinction is an application that manages user bank accounts and must allow registration of users over 18 years old.

- **Business Logic:** This is logic that comes from the product and is not affected by external changes. In this example, the requirement that users must be over 18 years old does not stem from a technical limitation but from a specific need of the application being developed. Also, the need to create a user record is business logic, as it is a specific requirement of the application.

- **Organizational Logic:** It is similar to business logic but is reused in more than one project within the same organization. For example, the methodology used to validate and register credit cards in our application could be organizational logic used in several projects within the same company.
- **Use Cases:** These are cases that have a technical limitation and can change if the application's use changes. For example, the requirements for registering a user might not be a use case, as there is no technical limitation to validate the fields of a form. However, the arrangement of the error message, its color, size, etc. can affect the application's use, making these elements use cases. The position and form of displaying fields on the screen can also be a use case, as it affects the application's SEO if there is a change in the content layout shift.

Application Example: The Hexagonal Pizza Shop

One of the most well-known examples is that of a pizzeria where a person wants to place an order. To do this, they will look at the menu with different options, tell the cashier their order, the cashier will communicate the order to the kitchen, the kitchen will perform the necessary procedures to meet the requirement, and return the completed order to the cashier for delivery to the buyer. If we think carefully about each entity in the example, we can find that the buyer is the main actor, the menu with different options is the port, the cashier is the adapter, and the kitchen is our service.

The consumer will order a product by looking at the menu and can only order what they see on it. At the same time, that same order, which may have a catchy name for the public, probably has a simpler name for the kitchen to increase process efficiency. The cashier knows this nomenclature and is responsible for managing proper communication between the consumer and the kitchen. What is a margarita pizza for one person is number 53 for another.

A part we don't see is that the kitchen itself needs resources to complete the order. This means that both the cheese, tomato, and other ingredients must be requested to manage orders. For this, there is probably someone in charge between the restaurant and a raw material supplier. Again, what is tomato for one person is product ABC for another. Here we see the clear example of a secondary actor, the supplier, communicating through an intermediary, the person in charge, to provide the necessary resources to our hexagon.

Recommended Steps for Working with Hexagonal Architecture with Application Example

- 1- Carefully Consider Requirements. An example illustrating this distinction is an application that manages user bank accounts and must allow the registration of users over 18 years old.
- 2- Identify the Business Logic You Need to Fulfill. In the example of the flight reservation system, business logic could include validating flight availability and assigning seats to passengers, as well as managing payments and issuing tickets.
- 3- Identify What Actions Your Hexagon Must Provide to Satisfy the Required Logic. In the example of the flight reservation system, the actions that the hexagon must provide could include: searching for available flights, reserving a flight, assigning seats to passengers, managing payments, and issuing tickets.

- 4- Identify the Resources Needed to Satisfy That Logic and Who Can Provide Them. In the example of the flight reservation system, the necessary resources could include: a database of flights and seats, an online payment provider, and a ticket issuance service. At this step, it is essential to identify who can provide these resources and how they will be integrated into the system.
- 5- Create the Necessary Ports for Drivers and Drivens. In the example of the flight reservation system, the necessary ports could include: a flight search port, a reservation port, a seat assignment port, a payment management port, and a ticket issuance port. It is important that these ports are designed to interact with drivers (user interfaces) and drivens (databases, payment providers, etc.) clearly and consistently.
- 6- Create Stub/Mocked Adapters to Immediately Satisfy Requests and Test Your Hexagon. In this step, adapters are created for drivers and drivens that mimic their real behavior but can be used to test the hexagon immediately without depending on external systems. For example, Stub/Mocked adapters could be created for the flights and seats database, the online payment provider, and the ticket issuance service.
- 7- Create the Necessary Tests That Must Pass Successfully for Your Hexagon to Satisfy the Request. In this step, tests are created to validate that the hexagon works as expected and satisfies the system requirements. For example, tests could be created to validate that available flights can be searched, a flight can be reserved, seats can be assigned to passengers, payments can be managed, and tickets can be issued successfully.
- 8- Create the Logic Inside Your Hexagon to Satisfy Use Cases. In this step, business logic is created inside the hexagon to satisfy the use cases identified in step 2. In the example of the flight reservation system, logic could be created to validate flight and seat availability, assign seats to passengers, and manage the payment and ticket issuance process. This logic should be designed in a way that is easily modifiable and scalable in the future if changes or improvements to the system are required. It is also important that this logic is separated from the specific logic of drivers and drivens to facilitate maintenance and evolution of the system in the future.

To work with hexagonal architecture, it is crucial to follow a methodical and careful process. First and foremost, it is essential to read the requirements carefully to think about the best way to solve them. It is crucial to recognize the business logic and necessary use cases to fulfill it, as you will remember from the previous explanation; this is essential to ensure that all needs are covered.

Once requirements and necessary resources have been identified, it is time to create our ports. To do this, we must recognize the essential methods that must be available for our primary and secondary actors. This will allow us to control access and exits to the service. By convention, port names should start with the word "For," followed by the action they must perform. For example, if we need a port to perform a registration action, we could call it "ForRegistering." We can also reduce the number of ports if we associate different related actions, such as "ForAuthenticating," which will provide registration and login actions.

Following these steps and paying careful attention to details, we can effectively work on hexagonal architecture and achieve optimal results in our projects.

Next, we need to create our driver and driven adapters. To do this, I recommend using the performer of the action as the adapter name, followed by the action itself. For example, in our case, we could call them Registerer or Authenticator, respectively.

The adapters, first and foremost, must be of the stub type and provide controlled information that can be used to satisfy business logic and tests. This way, we can close our hexagon and get it ready for implementation.

With our adapters complete, we proceed to use Test-Driven Development (TDD). We will create the necessary tests to meet the use cases, so we can verify the correct functioning of the logic when implementing it.

Our service must have the necessary entities for typing, satisfy the methods provided in the primary ports, and meet the use cases. The service is responsible for receiving a request, finding the necessary resources through secondary adapters, and using them to meet use cases and, therefore, business logic.

🕶 Code Example Following the Theme of a Banking Application:

```
// Class representing the hexagon
export class BankAccountService {
  private bankAccountPort: BankAccountPort;

  constructor(bankAccountPort: BankAccountPort) {
    this.bankAccountPort = bankAccountPort;
  }

  /**
   * Method to create a new bank account.
   * @param name - The name of the account holder.
   * @param age - The age of the account holder.
   * @throws AgeNotAllowedException if the age is not allowed.
   */
  public createBankAccount(name: string, age: number): void | AgeNotAllowedException {
    if (age >= 18) {
      const bankAccount = new BankAccount(name, age);
      this.bankAccountPort.saveBankAccount(bankAccount);
    } else {
      throw new AgeNotAllowedException("The minimum age to create a bank account is 18 years.");
    }
  }
}

// File bank-account-port.ts
// Interface defining the port to access the database of bank accounts
export interface BankAccountPort {
  saveBankAccount(bankAccount: BankAccount): void;
}

// File bank-account.ts
// Class representing the entity of a bank account
export class BankAccount {
  private name: string;
  private age: number;
```

```
  constructor(name: string, age: number) {
    this.name = name;
    this.age = age;
  }

  public getName(): string {
    return this.name;
  }

  public getAge(): number {
    return this.age;
  }
}

// File age-not-allowed-exception.ts
// Custom exception for when the age is not allowed
export class AgeNotAllowedException extends Error {
  constructor(message: string) {
    super(message);
    this.name = "AgeNotAllowedException";
  }
}

// File bank-account-repository.ts
// Class implementing the port to access the database of bank accounts
export class BankAccountRepository implements BankAccountPort {
  private bankAccounts: BankAccount[] = [];

  public saveBankAccount(bankAccount: BankAccount): void {
    this.bankAccounts.push(bankAccount);
  }
}

// File bank-account-controller.ts
// Class representing the driver for creating bank accounts
import { BankAccountService } from "./bank-account-service";
import { AgeNotAllowedException } from "./age-not-allowed-exception";

export class BankAccountController {
  private bankAccountService: BankAccountService;

  constructor(bankAccountService: BankAccountService) {
    this.bankAccountService = bankAccountService;
  }

  /**
   * Method to create a new bank account.
   * @param name - The name of the account holder.
   * @param age - The age of the account holder.
   */
  public createBankAccount(name: string, age: number): void {
    try {
```

```
      this.bankAccountService.createBankAccount(name, age);
      console.log("The bank account has been created successfully.");
    } catch (e) {
      if (e instanceof AgeNotAllowedException) {
        console.log(e.message);
      } else {
        console.log("An error occurred while creating the bank account.");
      }
    }
  }
}

// File main.ts
// Usage example
import { BankAccountRepository } from "./bank-account-repository";
import { BankAccountService } from "./bank-account-service";
import { BankAccountController } from "./bank-account-controller";

const bankAccountPort = new BankAccountRepository();
const bankAccountService = new BankAccountService(bankAccountPort);
const bankAccountController = new BankAccountController(bankAccountService);

bankAccountController.createBankAccount("John Doe", 20);
bankAccountController.createBank
```

How to GoLang

An AMAZING language created by Google in collaboration with Rob Pike, Ken Thomson, and Robert Griesemer.

Advantages:

- Fast, compiles directly into machine code without using an interpreter.
- Easy to learn, very good documentation, and many things are simplified.
- Scales very well, supports concurrent programming through "GoRoutines".
- Automatic garbage collector, automatic memory management.
- Included formatting engine, no need for third parties.
- No libraries required for testing or benchmarks because they are already included.
- Very little boilerplate for creating applications.
- Has an API for network programming, included as a standard library.
- VERY fast, in some benchmarks it is faster than backend applications made in Java and Rust.
- Built-in template system, GREAT for working with HTMX.

Recommended Structure:

- **ui** (frontend-related content in case of server-side rendering)
  - *html* (templates)
  - *static* (multimedia and style static content)
    - *assets*
    - *css*
- **internal** (content related to tools and reusable entities throughout the project)
  - *models*
  - *utils*
- **cmd**
  - *web* (contains the application logic)
    - *domain* (business logic)

- *routes* (available routes)

How Does GoLang Work?

GoLang uses a base file called go.mod, which will contain the main module that will be called the same as the project, and also the version of Go used. Then each file will have the extension ".go" to identify that it is a package belonging to the language.

But... what is a package? If you come from JavaScript you can think of it in the same way as an ES module since it is used to encapsulate related logic. But unlike ES modules, the package is identified by the lines of code "package packageName" in camel case the name of the package in question and it imports the location of the package. Different files containing logic belonging to the same package can be arranged separately BUT they must be under the same parent folder as this is of utmost importance to later import said package in different ones.

To import a different package is done through the word "import" plus the path to which the package belongs.

```
import "miProject/cmd/web/routes"
```

If you need more than one package at a time, it is not necessary to repeat the line of code since it can be grouped using "()" the various packages:

```
import (
    "miProject/cmd/web/routes"
    "miProject/internal/models"
)
```

It is worth mentioning that then Go will relate the final name of the path with the use of the package so in order to use logic contained in it will be done thinking of it as if it were an object, where each property represents a logical element of the package:

```
routes.MyRoute
```

Private and public package methods:

If the method starts with lowercase it is a private method, it cannot be accessed from outside the package itself.

```
func myFunction()
```

If the method starts with uppercase it is a public method, it can be accessed by importing the package from another.

```
func MyFunction()
```

Package scope

Let's see a Go file

```
package main - package name

import "fmt" - fmt package is imported, no path because it's Go's own

var number int = 2

func main() {
    i, j := 42, 2701 - local variables to the method, i with value 42 and j with value 2701

    fmt.Println(i) - using the "Println" method of the "fmt" package
}
```

You surely have noticed something, "number" has a type "int" preceding the value assignment, while "i" and "j" do not, this is because like Typescript, Go infers the type for those primitives. Let's see how to work with types.

Data Types

- **bool** = true / false
- **string** = string of characters
- **int, int8, int16, int32, int64, uint, uint8, uint16, uint32, uint64, uintptr** = integer numeric values with their limits, these are generally 32 bits on 32-bit systems and 64 for 64-bit systems. Integer should be used unless there is a specific reason to use a restricted value.
- **byte** === uint8
- **rune** === int32
- **float32, float64** = represents real numeric values
- **complex64, complex128** = complex numbers that have a real part and an imaginary part.

Structs

Represents a collection of properties, you can think of it as a Typescript interface, as it represents the contract that must be followed when creating a property. Important, if you want that property to be accessible outside the package, remember it should start with "uppercase".

```
type Person struct {
    Name string
    LastName string
    Age int
}

var person = Person {
    Name: "Gentleman",
    LastName: "Programming",
    Age: 31
}

fmt.Println(person.Name)
```

Another way:

```
var persona2 = Persona{"Gentleman", "Programming", 31}
```

Arrays

Now the fun begins, arrays are quite different from what we are used to as they MUST have the maximum number of elements they are going to contain inside:

```
var a [10]int - creates an array of 10 elements of type int

a[0] = "Gentleman"
```

Or also

```
var a = [2]int{2, 3}

fmt.Println(a) - [2 3]
```

If we need it to be dynamic we can talk about "slices". A "slice" is a portion of an existing array or a representation of a collection of elements of a certain type.

```
var primes = [6]int{2 ,3 ,5, 7, 11, 13}
var s []int = primes[1:4] - creates a slice using "primes" as a base from position 1 to 4

fmt.Println(s) - [3 5 7]

s = append(s , 14)
```

```
fmt.Println(s) - [3 5 7 14]

fmt.Println(primes) - [2 3 5 7 14 13]
```

You can also omit values for maximum and minimum ranges making them have default values:

```
var a [10]int

is the same as

a[0:10]
a[:10]
a[0:]
a[:]
```

Make Method

To create dynamic slices you can use the included "make" method, this will create an array filled with empty elements and return a slice referring to it. The "len" method can be used to see how many elements it currently contains and "cap" to see its capacity, that is, how many elements it can hold.

```
a := make([]int, 0, 5)  // len(a)=0 cap(a)=5
```

Pointers

If you come from Javascript...

this will take a bit, but let's see together the following example:

```
type ElementType struct {
    name string
}

var exampleElement = ElementType {
    name: "Gentleman",
}

func MyFunction(element ElementType) {
    ...
}

MyFunction(exampleElement)
```

```
Here you might think that we are working on the element "exampleElement", but it's quite the opposite.

So if we want to work with the same element passed as a parameter to the function, a pointer must be us

```go
var a = 1
```

It creates a memory space which inside it contains the value "1" and we create a reference to that memory space called "a". The difference with Javascript is that this reference is not passed to the method unless we have created a pointer to it!

```go
var p *int // pointer "p" that will reference a property of type "int"

i := 42
p = &i // create a direct pointer to the property "i"

// If we want to access the value referenced by the pointer "p", we use the pointer's name preceded by
fmt.Println(*p) // 42

*p = 21

fmt.Println(*p) // 21
```

Where this changes is if we point to a "struct", as it would be a bit cumbersome to do (*p).Property, it reduces to using it as if it were the struct itself:

```go
v := Person{"Gentleman"}
p := &v
p.Name = "Programming"
fmt.Println(v) // {Programming}
```

Default Values

In Go, when you declare a variable without explicitly assigning a value, it takes on a default value based on its type. Here's a table summarizing the defaults:

Default Values for Data Types:

- **bool**: `false`
- **string**: `""` (empty string)
- **Numeric Types**: `0`
- **array**: `nil` (uninitialized)
```

- **map**: `nil` (uninitialized)
- **slice**: `nil` (uninitialized)
- **pointer**: `nil` (uninitialized)
- **function**: `nil` (uninitialized)

Range Loop

The `range` loop is a powerful construct for iterating over sequences like slices, arrays, maps, and strings. It provides two components: the index (`i`) and the value (`v`) of each element. Here are three common variations:

- **Full Iteration:**

```
var arr = []int{5, 4, 3, 2, 1}

for i, v := range arr {
  fmt.Printf("index: %d, value: %d\n", i, v)
}
```

This approach iterates over both the index and value of each element in `arr`.

- **Ignoring Index:**

```
for _, v := range arr {
  fmt.Printf("value: %d\n", v)
}
```

The underscore (_) discards the index information, focusing only on the element values.

- **Ignoring Value:**

```
for i, _ := range arr {
  fmt.Printf("index: %d\n", i)
}
```

Similarly, you can use an underscore to skip the value and access only the indices.

Maps

Maps are unordered collections that associate unique keys (of any hashable type) with values. Go provides two ways to create and work with maps:

- **Using `make` Function:**

```
type Persona struct {
  DNI, Nombre string
}

var m map[string]Persona

func main() {
  m = make(map[string]Persona)
  m["123"] = Persona{"123", "pepe"}
  fmt.Println(m["123"])
}
```

- **Map Literal:**

```
type Persona struct {
  DNI, Nombre string
}

var m = map[string]Persona{
  "123": Persona{"123", "pepe"},
  "124": Persona{"124", "jorge"},
}

func main() {
  fmt.Println(m)
}
```

  Map literals offer a concise way to initialize maps with key-value pairs.

Mutating Maps

- **Insertion:**

```
m[key] = element
```

  Adds a new key-value pair to the map `m`.

- **Retrieval:**

```
element = m[key
```

Functions

Functions are reusable blocks of code that perform specific tasks. They are declared with the `func` keyword, followed by the function name, parameter list (if any), return type (if any), and the function body enclosed in curly braces.

Here's an example:

```
func greet(name string) string {
  return "Hello, " + name + "!"
}

func main() {
  message := greet("Golang")
  fmt.Println(message)
}
```

Function Values

Functions can be assigned to variables, allowing you to pass them around like any other value. This enables powerful techniques like higher-order functions.

Here's an example demonstrating how to pass a function as an argument and call it indirectly:

```
func CallCallback(callBack func(float64, float64) float64) float64 {
  return callBack(3, 4)
}

func hypot(x, y float64) float64 {
  return math.Sqrt(x*x + y*y)
}

func main() {
  fmt.Println(hypot(5, 12))
  fmt.Println(CallCallback(hypot))
}
```

Closures

Closures are a special type of function that captures variables from its enclosing environment. This allows the closure to access and manipulate these variables even after the enclosing function has returned.

Here's an example of a closure that creates an "adder" function with a persistent sum:

```
func adder() func(int) int {
  sum := 0

  return func(x int) int {
    sum += x
    return sum
  }
}

func main() {
  pos, neg := adder(), adder()

  for i := 0; i < 10; i++ {
    fmt.Println(
      pos(i),
      neg(-2*i),
    )
  }
}
```

Methods

Go doesn't have classes, but it allows defining methods on types (structs, interfaces). A method is a function associated with a type, taking a receiver argument (usually the type itself) that implicitly refers to the object the method is called on.

Here's an example of a `Persona` struct with a `Saludar` method:

```
type Persona struct {
  Nombre, Apellido string
}

func (p Persona) Saludar() string {
  return "Hola " + p.Nombre
}

func main() {
  p := Persona{"Pepe", "Perez"}
  fmt.Println(p.Saludar())
}
```

Methods can also be defined on non-struct types:

```
type Nombre string

func (n Nombre) Saludar() string {
  return "Hola " + string(n)
}

func main() {
  nombre := Nombre("Pepe")
  fmt.Println(nombre.Saludar())
}
```

Methods can accept pointers as receivers, enabling modifications to the original object:

```
type Persona struct {
  nombre, apellido string
}

func (p *Persona) cambiarNombre(n string) {
  p.nombre = n
}

func main() {
  p := Persona{"pepe", "perez"}
  p.cambiarNombre("juan")
  fmt.Println(p) // Output: {juan perez}

  pp := &Persona{"puntero", "persona"}
  pp.cambiarNombre("punteroNuevoNombre")
  fmt.Println(*pp) // Output: {punteroNuevoNombre persona}
}
```

Go automatically dereferences pointer receivers when necessary, so you don't always need to use the explicit * operator.

Interfaces

Interfaces define a set of methods that a type must implement. They provide a way to achieve polymorphism, allowing different types to be used interchangeably as long as they implement the required methods.

Here's an example of an `Interface` that defines two methods, `Saludar` and `Moverse`:

```
type Persona interface {
  Saludar() string
  Moverse() string
```

```
}

type Alumno struct {
  Nombre string
}

func (a Alumno) Saludar() string {
  return "Hola " + a.Nombre
}

func (a Alumno) Moverse() string {
  return "Estoy caminando"
}

func main() {
  var persona Persona = Alumno{
    "Pepe",
  }

  fmt.Println(persona.Saludar())
  fmt.Println(persona.Moverse())
}
```

Interface Values with Nil

Interface values can be `nil`, indicating that they don't hold a reference to any specific object. Here's an example demonstrating how to handle `nil` interface values:

```
type I interface {
  M()
}

type T struct {
  S string
}

func (t *T) M() {
  if t == nil {
    fmt.Println("<nil>")
    return
  }
  fmt.Println(t.S)
}

func main() {
  var i I

  var t *T
  i = t
```

```
  describe(i)
  i.M() // Output: <nil>

  i = &T{"hello"}
  describe(i)
  i.M() // Output: hello
}

func describe(i I) {
  fmt.Printf("(%v, %T)\n", i, i)
}
```

Empty Interfaces

If you don't know the specific methods an interface might require beforehand, you can create an empty interface using the `interface{}` type. This allows you to store any value in the interface, but you won't be able to call methods on it directly.

```
var i interface{}
```

Type Assertion

When we use an empty interface `go interface{}`, we may use any kind of type BUT, this also comes with problems. How do we know if the parameter of a method is of the expected type if if it's an empty interface ? Here's where Type Assertions come handy, as they provide the possiblity of testing if the empty interface is of the expected type.

```
t := i.(T)
```

This means that the interface value `i` holds the concrete type `T` and assigns the underlying `T` value to the variable `t`.

If `i` does not hold a `T`, this will trigger a panic.

You can test if the interfaces faclue holds a specific type by using a second parameter, just like we do with `err`:

```
t, ok := i.(T)
```

This will save true or false inside `ok`. If false, `t` will save a zero value inside and no panic will occur.

```go
func main() {
    var i interface{} = "hello"

    s := i.(string)
    fmt.Println(s) // hello

    s, ok := i.(string)
    fmt.Println(s, ok) // hello true

    f, ok := i.(float64)
    fmt.Println(f, ok) // 0 false

    f = i.(float64) // panic: interface conversion: interface {} is string, not float64
    fmt.Println(f) // nothing, it will panic before
  }
```

Type Switches

It provides the possiblity of doing more than one Type Assertion in series.

Just like a regular switch statemenet, but we use types instead of values, and the later ones will be compared against the type of the value held by the given interface value.

```go
switch v := i.(type) {
    case T:
    // if v has type T
    case S:
    // if v has type S
    default:
    // if v has neither type T or S, it will have the same type as "i"
  }
```

Acclaration: just like Type Assertions, we use a type as a parameter `go i.(T)`, but instead of using `T`, we need to use the keyword `type`.

This is great when executing different logics which depends on the type of the parameter:

```go
type Greeter interface {
    SayHello()
}

type Person struct {
    Name string
}

func (p Person) SayHello() {
```

```
    fmt.Printf("Hello, my name is %s!\n", p.Name)
}

type Number int

func (n Number) SayHello() {
    if n%2 == 0 {
        fmt.Printf("Hello, I'm an even number: %d!\n", n)
    } else {
        fmt.Printf("Hello, I'm an odd number: %d!\n", n)
    }
}

func main() {
    greeters := []Greeter{
        Person{"Alice"},
        Person{"Bob"},
        Number(3),
        Number(4),
    }

    for _, greeter := range greeters {
        switch value := greeter.(type) {
        case Person:
            value.SayHello()
        case Number:
            value.SayHello()
        }
    }
}
```

Stringers

It's a type that defines itself as a `string`, it's defined by the `fmt` package and it's used to print values.

```
type Person struct {
  Name string
  Age int
}

func (p Person) String() string {
  return fmt.Sprintf("%v (%v years)", p.Name, p.Age)
}

func main() {
    a := Person{"Gentleman Programming", 32}
    z := Person{"Alan Buscaglia", 32}
```

```
    fmt.Println(a, z) // Gentleman Programming (32 years) Alan Buscaglia (32 years)
```

Example using Stringers to modify the way we show an IpAdress when using fmt.Println :

```
type IPAddr [4]byte

func (ip IPAddr) String() string {
  str := ""

    for i, ipValue := range ip {
    str += fmt.Sprint(ipValue)

        if i < len(ip)-1 {
            str += "."
        }
  }

    return fmt.Sprintf("%s", str)
}

// TODO: Add a "String() string" method to IPAddr.

func main() {
    hosts := map[string]IPAddr{
        "loopback":  {127, 0, 0, 1},
        "googleDNS": {8, 8, 8, 8},
    }

    for _, ip := range hosts {
        fmt.Println(ip)
    }
}
```

Errors

To show erros, Go uses `error` values to express `error states`, and for this, the `error` type exists and it's similar to the `fmt.Stringer` interface:

```
type error interface {
  Error() string
}
```

Exactly as with `fmt.Stringer` the `fmt` package looks for the `error` interface when printing values. Normally methods return an `error` value and we should use it to manage what to do in case it's different to `nil`:

```
i, err := strconv.Atoi("42")

if err != nil {
  fmt.Println("couldn't convert number: %v\n", err)
  return
}

fmt.Println("Converted integer: ", i)
```

Readers

Another great interface which represents the read end of a stream of data, this data may be streamed over files, network connections, compressors, ciphers, etc.

And it has a **Read** method:

```
func (T) Read(b []byte) (n int, err error)
```

This method will populate the byte array with data and returns the number of bytes populated and an error value. It returns an `io.EOF` error when the stream ends.

```
func main() {
  data := "Gentleman Programming"


  // create a new io.Reader reading from data
  reader := strings.NewReader(data)

  // create a buffer to store the copied data
  var buffer strings.Builder

  // copy data from the reader to a buffer. io.Copy reads from the reader and writes to the writer unti
  n, err := io.Copy(&buffer, reader)

  if err != nil {
    fmt.Println("Error:", err)
  } else {
      fmt.Println("\n%d bytes copied successfully. \n", n)

      // access the data copied into the buffer
      fmt.Println("Copied Data:", buffer.String())
    }
}
```

Example, let's get a ciphered string and decode it !

```go
package main

import (
    "io"
    "os"
    "strings"
)

type rot13Reader struct {
    r io.Reader
}

func (rr *rot13Reader) Read(p []byte) (n int, err error) {
    n, err = rr.r.Read(p)
    for i := 0; i < n; i++ {
        if (p[i] >= 'A' && p[i] <= 'Z') || (p[i] >= 'a' && p[i] <= 'z') {
            if p[i] <= 'Z' {
        // p[i] - 'A' calculates the position of the current character relative to 'A', then we add 13
        // then we apply '%26' to ensure that the result is within the range of the alphabet (26 letter
        // and at the end we add 'A' wich converts the result back to the ASCII value of a letter
                p[i] = (p[i]-'A'+13)%26 + 'A'
            } else {
                p[i] = (p[i]-'a'+13)%26 + 'a'
            }
        }
    }
    return
}

func main() {
    s := strings.NewReader("Lbh penpxrq gur pbqr!")
    r := rot13Reader{s}
    io.Copy(os.Stdout, &r)
}
```

Images

The package `iamges` defines the `Image` interface, which is a powerful tool to work with images as you can create, manipulate, and decode various types of images such as PNG, JPEG, GIF, BMP, and more:

```go
package image

type Image interface {
  ColorModel() color.Model
  Bounds() Rectangle
  At(x, y int) color.Color
}
```

Let's create a black small image with a red pixel in the center:

```
package main

import (
    "image"
    "image/color"
    "image/png"
    "os"
)

func main() {
    // Create a new RGBA image with dimensions 100x100
    img := image.NewRGBA(image.Rectangle(0, 0, 100, 100))

    // Set all pixels to black
    for x := 0; x < 100; x++ {
        for y := 0; y < 100; y++ {
            img.Set(x, y, color.Black)
        }
    }

    // Set the pixel at the center to red
    img.Set(50, 50, color.RGBA{255, 0, 0, 255})

    // Create a PNG file to save the image
    file, err := os.Create("simple_image.png")
    if err != nil {
        panic(err)
    }
    defer file.Close()

    // Encode the image to PNG format and save it to the file
    err = png.Encode(file, img)
    if err != nil {
        panic(err)
    }

    println("Simple image generated successfully!")
}
```

GoRoutines

As we mentioned before, Go is a language that supports concurrent programming through "GoRoutines". A GoRoutine is a lightweight thread managed by the Go runtime, allowing you to run multiple functions concurrently.

BUT it's different than other languages, as it's a virtual thread that runs on a real thread, and it's managed by the Go runtime.

To execute a function as a GoRoutine, you just need to add the `go` keyword before the function call:

```
    go f(x, y, z)
```

This will run the function `f(x, y, z)` concurrently in a new GoRoutine. The parameters are evaluated at the time of the function call, so if they change later, the GoRoutine will use the updated values.

Let's see an example:

```
package main

import (
  "fmt"
)

func say(s string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
  }
}

func main() {
  // Launch a new goroutine to run the say function with "Hello"
  go say("Hello")

  // Print "World" 3 times in the main function
  for i := 0; i < 3; i++ {
    fmt.Println("Gentleman")
  }
}
```

When you run this code, the output won't necessarily be "Hello" followed by "Gentleman" three times each. This is because the goroutines are running concurrently. You might see "Hello" and "Gentleman" mixed together.

Goroutines are lightweight, so you can create thousands of them without any performance issues. They are managed by the Go runtime, which schedules them efficiently on real OS threads.

Another great feature is that the run in the same address space, so they can communicate with each other using channels sharing memory, but, this also needs to be managed and synchronized.

To do so we can use `channels`:

Channels

They will be our way to communicate between goroutines, they are typed and can be used to send and receive data with the channel operator <-:

```
ch <- v // Send v to channel ch.
v := <-ch // Receive from ch, and assign value to v.
```

The data will flow in the direction of the arrow, so if you want to send data to a channel, you should use the arrow pointing to the channel, and if you want to receive data from a channel, you should use the arrow pointing from the channel.

You can also create a channel with the use of the `make` function:

```
ch := make(chan int)
```

This will create a channel that will send and receive integers.

By default, sends and receives block until the other side is ready. This allows goroutines to synchronize without we having to manually manage that synchronization.

```
package main

import (
  "fmt"
)

func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Wait infinitely for messages on the channel (ensure all "Hello" are printed)
  for {
    msg := <-ch // Receive message from the channel
    fmt.Println("Received:", msg)
  }
```

```
  fmt.Println("Gentleman") // Print "Gentleman" after receiving all messages
}
```

Buffered Channels

All channels can be buffered, this means that they can hold a limited number of values without a corresponding receiver for those values.

When the channel is full, the sender will block until the receiver has received a value. This is extremely useful when you want to send multiple values and you don't want to loose them if the receiver is not ready.

```
ch := make(chan int, 100)
```

This will create a channel that can hold up to 100 integers.

If you send more than 100 values to the channel, the sender will block until the receiver has received some values.

```
func main() {
    ch := make(chan int, 2)
    ch <- 1
    ch <- 2
    ch <- 3 // fatal error: all goroutines are asleep - deadlock!

    fmt.Println(<-ch)
    fmt.Println(<-ch)
    fmt.Println(<-ch)
}
```

Range and Close

You can close a channel at any time ! a recommended time to close a channel is when you want to signal that no more values will be sent on it and the one to do it should be the sender, never the receiver as sending on a closed channel will cause a panic:

```
v, ok := <-ch
ok will be false if there are no more values to receive and the channel is closed.
```

To receive values from a chanel until it's closed you can use `range`:

```
for i:= range ch {
    fmt.Println(i)
```

```
}
```

Do we need to close them ? Not necessarily, only if the receiver needs to know that no more values will be sent, or if the sender needs to tell the receiver that it's done sending values, this way we will terminate the `range` loop.

Example:

```
func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
  close(ch) // Close the channel after sending messages
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Loop to receive and print messages until channel is closed
  for {
    msg, ok := <-ch // Receive message and check channel open state
    if !ok {
      break // Exit loop if channel is closed
    }
    fmt.Println("Received:", msg)
  }

  fmt.Println("Messages received. Exiting.")
}
```

GoRoutines Select

The `select` statement lets a goroutine wait on multiple communication operations. It blocks until one of its cases can run, then it executes that case.

It's useful when you want to wait on multiple channels and perform different actions based on which channel is ready.

```
func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
```

```
  // Close the channel after the loop finishes sending messages
  close(ch)
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Use select to handle messages from the channel or a timeout
  for {
    select {
    case msg, ok := <-ch: // Receive message and check channel open state
      if !ok {
        fmt.Println("Channel closed. Exiting.")
        break
      }
      fmt.Println("Received:", msg)
    case <-time.After(1 * time.Second): // Timeout after 1 second if no message received
      fmt.Println("Timeout waiting for message.")
      break
    }
  }
}
```

You can also use a `default` case in a `select` statement, this will run if no other case is ready:

```
func say(s string, ch chan string) {
  for i := 0; i < 3; i++ {
    fmt.Println(s)
    ch <- s // Send "Hello" to the channel after each print
  }
  close(ch) // Close the channel after sending messages
}

func main() {
  // Create a channel to hold strings
  ch := make(chan string)

  // Launch a new goroutine to run the say function
  go say("Hello", ch)

  // Use select with default case
  for {
    select {
    case msg, ok := <-ch:
      if !ok {
        fmt.Println("Channel closed. Exiting.")
```

```
        break
      }
      fmt.Println("Received:", msg)
    case <-time.After(1 * time.Second):
      fmt.Println("Timeout waiting for message.")
      break
    default:
      fmt.Println("Nothing to receive or timeout yet.")
    }
  }
}
```

Now let's do an exercise where we will check if two node trees have the same sequence of values:

```
package main

import (
    "fmt"
)

type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}

func isSameSequence(root1, root2 *TreeNode) bool {
    seq1 := make(map[int]bool)
    seq2 := make(map[int]bool)

    traverse(root1, seq1)
    traverse(root2, seq2)

    return equal(seq1, seq2)
}

func traverse(node *TreeNode, seq map[int]bool) {
    if node == nil {
        return
    }

    seq[node.Val] = true
    traverse(node.Left, seq)
    traverse(node.Right, seq)
}

func equal(seq1, seq2 map[int]bool) bool {
    if len(seq1) != len(seq2) {
        return false
    }
```

```
    for val := range seq1 {
        if !seq2[val] {
            return false
        }
    }

    return true
}

func main() {
    // Constructing the first binary tree
    root1 := &TreeNode{
        Val: 3,
        Left: &TreeNode{
            Val: 1,
            Left: &TreeNode{
                Val: 1,
            },
            Right: &TreeNode{
                Val: 2,
            },
        },
        Right: &TreeNode{
            Val: 8,
            Left: &TreeNode{
                Val: 5,
            },
            Right: &TreeNode{
                Val: 13,
            },
        },
    }

    // Constructing the second binary tree
    root2 := &TreeNode{
        Val: 8,
        Left: &TreeNode{
            Val: 3,
            Left: &TreeNode{
                Val: 1,
                Left: &TreeNode{
                    Val: 1,
                },
                Right: &TreeNode{
                    Val: 2,
                },
            },
            Right: &TreeNode{
                Val: 5,
            },
        },
        Right: &TreeNode{
```

```
                Val: 13,
                },
    }

    fmt.Println(isSameSequence(root1, root2)) // Output: true
}
```

Mutex

Something that we need to take care of when working with GoRoutines is the access to shared memory, were more than one GoRoutine can access the same memory space at the same time, this can lead to great conflicts.

This concept is called **`mutual exclusion`**, and it's solved by the use of **`mutexes`**, which are used to synchronize access to shared memory.

```
import ("sync")

var mu sync.Mutex
```

It has two methods, **`Lock`** and **`Unlock`**, which are used to protect the shared memory:

```
func safeIncrement() {
    mu.Lock() // lock the shared memory
    defer mu.Unlock() // unlock the shared memory when the function returns
    count++ // increment the shared memory
}
```

Here we are using the **`defer`** statement to ensure that the mutex is unlocked when the function returns, even if it panics.

```
package main

import (
    "fmt"
)

type TreeNode struct {
    Val   int
    Left  *TreeNode
    Right *TreeNode
}

type SequenceCollector struct {
    sequence map[int]bool
}
```

```go
func isSameSequence(root1, root2 *TreeNode) bool {
    seq1 := &SequenceCollector{sequence: make(map[int]bool)}
    seq2 := &SequenceCollector{sequence: make(map[int]bool)}

    traverse(root1, seq1)
    traverse(root2, seq2)

    return equal(seq1.sequence, seq2.sequence)
}

func traverse(node *TreeNode, seq *SequenceCollector) {
    if node == nil {
        return
    }

    seq.sequence[node.Val] = true

    traverse(node.Left, seq)
    traverse(node.Right, seq)
}

func equal(seq1, seq2 map[int]bool) bool {
    if len(seq1) != len(seq2) {
        return false
    }

    for val := range seq1 {
        if !seq2[val] {
            return false
        }
    }

    return true
}

func main() {
    // Construyendo el primer árbol binario
    root1 := &TreeNode{
        Val: 3,
        Left: &TreeNode{
            Val: 1,
            Left: &TreeNode{
                Val: 1,
            },
            Right: &TreeNode{
                Val: 2,
            },
        },
        Right: &TreeNode{
            Val: 8,
            Left: &TreeNode{
                Val: 5,
```

```
            },
            Right: &TreeNode{
                Val: 13,
            },
        },
    }

    // Construyendo el segundo árbol binario
    root2 := &TreeNode{
        Val: 8,
        Left: &TreeNode{
            Val: 3,
            Left: &TreeNode{
                Val: 1,
                Left: &TreeNode{
                    Val: 1,
                },
                Right: &TreeNode{
                    Val: 2,
                },
            },
            Right: &TreeNode{
                Val: 5,
            },
        },
        Right: &TreeNode{
            Val: 13,
        },
    }

    fmt.Println(isSameSequence(root1, root2)) // Salida: true
}
```

NVIM Gentleman Guide !

First of all...

What is NVIM ?

Vim was first invented on the 2nd of November 1991 by Bram Moolenaar. Vim is a highly configurable text editor built to enable efficient text editing.

It is an improved version of the vi editor distributed with most UNIX systems. Vim is often called a "programmer's editor," and so useful for programming that many consider it an entire IDE.

It's not just for programmers, though. Vim is perfect for all kinds of text editing, from composing email to editing configuration files.

So... what is NVIM then ?

NVIM is a fork of Vim, with a lot of new features, better performance, and a lot of new plugins and configurations.

It is a text editor that is highly configurable and can be used for programming, writing, and editing text files.

The most funny part is that it isn't even released yet ! as the time of writting this guide, NVIM is still in the beta phase 0.9.5. But it is stable enough to be used as a daily driver.

Why I love NVIM ?

EFficiency, Speed, and Customization.

Let's talk about me a little bit, Im a software engineer, and I spend most of my time writing code. I have tried a lot of text editors and IDEs, but I always come back to Vim. Why ? because it is fast, efficient, and highly customizable.

But Vim has its own problems, it is hard to configure, and it has a steep learning curve for beginners, but as a Dark Souls lover... I love the challenge. Once you master the beast, you will never go back.

Having the power of not leaving my keyboard, learn something new every day, play with new plugins, and create my own configurations is what makes me love Vim. It can run anywhere, on any platform, it's fast, it's light, you can share your config really easily, and it is open-source. And the best part... you look amazing while using it, a lot of people will ask you "what is that ?" and you will feel like a hacker in a movie, running commands and shortcuts and showing the power of your efficiency.

Ok, let's start this guide, I will show you how to install NVIM, configure it, and use it as a daily driver. I will show you how to install plugins, create your own configurations, and make it look amazing.

Let's start with the installation.

Installation

Previous Requirements

(Execute all commands using the system default terminal, we will change it later)

WINDOWS USERS:

First, let's install WSL (https://learn.microsoft.com/en-us/windows/wsl/install), this is a must-have for windows users, it will allow you to run a full Linux distribution on your windows machine and I recommend using the version 2 as it uses the whole machine resources.

```
wsl --install
wsl --set-default-version 2
```

As we are now running a full linux distribution on our windows machine, the next steps will be the same for all platforms, being windows, mac, or linux.

1- Install Homebrew, this is a package manager for macOS and Linux, it will allow you to install a lot of packages and tools easily and it's always up to date.

```
set install_script (curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)
```

Include HomeBrew Path in your shell profile

```
Change 'YourUserName' with the device username

(echo; echo 'eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"') >> /home/YourUserName/.bashrc
eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv)"
```

2- Install build-essential, this is a package that contains a list of essential packages for building software and we will need it to compile some plugins. This step is not needed for macOS users.

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential
```

3- Install NVIM, we will install NVIM using Homebrew, this will install the latest version of NVIM and all its dependencies.

```
brew install nvim
```

4- Install Node and NPM, needed for web development plugins and some language servers.

```
brew install node
brew install npm
```

5- Install GIT, we will need GIT to clone repositories.

```
brew install git
```

6- Install FISH, this is a shell that is highly customizable and has a lot of features, I recommend using it as your default shell. Some of it's amazing features are autocompletion and syntax highlighting.

```

```bash
brew install fish
// set as default:

which fish
// this will return a path, let's call it whichFishResultingPath

// add it as an available shell
echo whichFishResultingPath | sudo tee -a /etc/shells

// set it as default
sudo chsh -s wichFishResultingPath
```

7- Install Oh My Fish, this is a framework for fish shell, it will allow you to install themes, plugins, and configure your shell easily.

```
curl https://raw.githubusercontent.com/oh-my-fish/oh-my-fish/master/bin/install | fish
```

8- Install the following dependencies needed to execute LazyVim

```
brew install gcc
brew install fzf
brew install fd
brew install ripgrep
```
```

9- Install Zellij, this is a terminal multiplexer, it will allow you to split your terminal in multiple panes and execute multiple commands at the same time.

```
brew install zellij
```

10 - Install Wezterm, this is a terminal emulator, it is highly customizable and has a lot of features, I recommend using it as your default terminal. One of the strongest features is the GPU acceleration, it will make your terminal faster and more responsive and that it's written in lua, the same language that LAZYVIM uses.

```
https://wezfurlong.org/wezterm/index.html
```

11- Install Iosevka Term Nerd Font, this is a font that is highly customizable and has a lot of features, I recommend using it as your default font. It has a lot of ligatures and special characters that will make your terminal look amazing. A nerd font is a font that has a lot of special characters and ligatures that will make your terminal look amazing and it's needed to render icons.

```
https://github.com/ryanoasis/nerd-fonts/releases/download/v3.1.1/IosevkaTerm.zip
```

12- Now let me share my custom repository that contains all my configurations for NVIM, FISH, Wezterm, and Zellij.

```
https://github.com/Gentleman-Programming/Gentleman.Dots
```

Just follow the steps and you will have a fully customized and gentlemanized terminal and code editor. One last thing before continuin, we will use some plugins that are already configured inside the repository and are managed by LazyVim, an amazing package manager that will allow you to install and update plugins easily.

Now that we have everything... let's start learning how to configure NVIM !

Configuration

As we are using my custom repository, all the configurations are already done, but I will explain how to configure NVIM and how to install plugins and create your own configurations.

As previously said, we are using LazyVim, http://www.lazyvim.org/, an amazing package manager that will allow you to install and update plugins smoothly like butter, it also provides already configured plugins that will make your life easier.

But first, let's learn how to install plugins manually.

If you see the nvim folder structure you will find a "plugins" folder, it will contain a number of files representing each one of the installed plugins by our hand.

Each file will contain the plugin name and the repository URL. To install a plugin manually, you will need to create a new file inside the "plugins" folder and add the following content.

```
return {
    "repository-url",
}
```

And that's it, the next time you open NVIM, the plugin will be installed automatically.

To access a LazyVim management window, just open nvim using the command "nvim" at your terminal and then type the following command ":LazyVim", this will open a window with all the installed plugins and their status, you can install, update, and remove plugins using this window.

Now, to acess all extra already configured plugins provided by LazyVim, just type the following command ":LazyVimExtra", this will open a window with all the available plugins, you can install, update, and remove plugins using this window.

To install a new programming language, type ":MasonInstall" and select the language you want to install, this will install all the necessary plugins and configurations for that language and that's it, you are ready to go.

To stablish new keybindings, just open the "keymaps.lua" file inside the "lua" folder and add the following content.

```
vim.keymap.set('mode', 'whatDoYouWantToPress', 'WhatDoYouWantToDo')
```

The mode represents the mode you are going to use, it can be "n" for normal mode, "i" for insert mode, "v" for visual mode, and "c" for command mode. The second parameter represents the key you want to press, and the third parameter represents the action you want to do.

You shuld come back later to this part after we touch the basics of vim modes.

Basics

Nvim has 4 modes, Normal, Insert, Visual, and Command. Each one of them has its own purpose and shortcuts.

Normal Mode

At this mode, you can navigate through the text, delete, copy, paste, and execute commands. You can enter this mode by pressing the "ESC" key.

In resume, this is the mode you will spend most of your time and where we will move across our code.

Horizontal Movement

To navigate, we will NOT use the arrow keys, we will use the "h" key to move left, the "j" key to move down, the "k" key to move up, and the "l" key to move right. This is the most efficient way to navigate through the text and it will make you look like a pro.

I really recommend you to use the "hjkl" keys to navigate, it will make you more efficient and you will not need to move your hands from the home row. Efficiency is the name of the game.

To jump to the beginning of the line, use the "0" key, to jump to the end of the line, use the "$" key. To jump to the beginning of the file, use the "gg" keys, to jump to the end of the file, use the "G" key.

It's always the same, if you press a command you will do something, and if you press "Shift" while doing it, you will do the opposite.

To correctly move horizontally throug a line, you will need to use the "w" key to jump to the beginning of the next word, the "b" key to jump to the beginning of the previous word, the "e" key to jump to the end of the next word, and the "ge" key to jump to the end of the previous word.

You can also use the amazing "f" key to jump to a specific character in the line, just press "f" and then the character you want to jump to, and that's it, you are there. And I said before, if you use "Shift" while doing it, you will do the opposite, moving to the previous ocurrency.

You can also use the "s" key to search for a character, this is using a plugin called "Sneak", it will allow you to search for a character and jump to it after pressing the key that will appear next to all the ocurrences.

Vertical Movement

To navigate vertically, you can use the "Ctrl" key with the "u" key to move up half a page, the "Ctrl" key with the "d" key to move down half a page, the "Ctrl" key with the "b" key to move up a page, and the "Ctrl" key with the "f" key to move down a page. This is the most efficient way to navigate through the text as we don't know where that particular piece of logic is, so we can move pretty quickly this way and find what we are looking for.

Another great way of navigating vertically is using the "Shift" key with the " and " keys, this will allow you to jump to the next or previous paragraph, and this is one of the things that makes me love NVIM, if your code is clean and correctly indented, you will be able to jump through the code really quickly and find what you are looking for, IT'S TEACHING YOU TO WRITE CLEAN CODE !!

If you want to jump to a particular line, you can use the ":" key, this will open the command mode, and then you can type the line number you want to jump to, and that's it, you are there.

Visual Mode

This mode is used to select text, you can enter this mode by pressing the "v" key. You can use the same commands as the normal mode, but now you can select text. You can also use the "Shift" key with the "v" key to select the whole line.

Again we can use motions to select text, for example, if we want to select the next 10 lines, we can use the "10j" command, and if we want to select the next 10 words, we can use the "10w" command. And if we want to select the next 10 characters, we can use the "10l" command.

Block Visual Mode

This mode is used to select a block of text, you can enter this mode by pressing the "Ctrl" key with the "v" key. You can use the same commands as the normal mode, but now you can select a block of text.

Again we can use motions to select text, for example, if we want to select the next 10 lines, we can use the "10j" command, and if we want to select the next 10 words, we can use the "10w" command. And if we want to select the next 10 characters, we can use the "10l" command.

A block of text is a rectangle of text, and you can copy, paste, and delete it. You can also use the "Shift" key with the "I" key to insert text in a block, and the "Shift" key with the "A" key to append text in a block.

It's also useful to write a lot of lines at the same time, for example, if you want to write a comment in multiple lines, you can use the "Ctrl" key with the "v" key to select the lines you want to write the comment, and then use the "Shift" key with the "I" key to insert the comment, and that's it, you are done after you press the "ESC" key.

Visual Line Mode

This mode is used to select a line of text, you can enter this mode by pressing the "Shift" key with the "v" key. You can use the same commands as the normal mode, but now you can select a line of text.

Again we can use motions to select text, for example, if we want to select the next 10 lines, we can use the "10j" command, and if we want to select the next 10 words, we can use the "10w" command. And if we want to select the next 10 characters, we can use the "10l" command.

Insert Mode

This is the mode you will use to write text, you can enter this mode by pressing the "i" key. You can use the same commands as the normal mode, but now you can write text. You can also use the "Shift" key with the "i" key to insert text at the beginning of the line, and the "A" key to append text at the end of the line, and the same if you want to start writting at a specific character, you can use "i" to insert before the character and "a" to append after the character.

You can also use the "o" key to insert a new line below the current line, and the "O" key to insert a new line above the current line. Using the "Ctrl" key with the "w" key will delete the last word, and using the "Ctrl" key with the "u" key will delete the last line while being in insert mode.

Another useful command is the "Ctrl" key with the "n" key, this will autocomplete the text you are writing, and it's really useful when you are writing code. And if you want to exit the insert mode, you can use the "ESC" key.

Command Mode

This mode is used to execute commands, you can enter this mode by pressing the ":" key. Here is were we can exit NVIM !!! Just do ":q" and that's it ! if you have changes, first save them by using ":w" and if you want to force the exit ":q!".

Another cool thing is that you can do more than one command at once, for example, if you want to save and exit, you can use ":wq".

One cofiguration I recommend it's setting the number of lines to relative by doing ":set relativenumber", this will allow you to see the line number relative to the line you are in, and it's really helpful to know where you are in the file. You can do this by typing the following command ":set relativenumber", and we want this as we can move to a specific line really quickly by using a number and the direction we want to go to, for example, if we want to jump to the 10th line above us, we can use the "10k" command, and if we want to jump to the 10th line below us, we can use the "10j" command.

Nvim Motions

And this introduces the concept of "Motions" in NVIM, each command we type is a motion, and it's created by combining a number, a direction, and a command. For example, if we want to delete the next 10 lines, we can use the "10dd" command, and if we want to copy the next 10 lines, we can use the "10yy" command. This is the most efficient way to navigate through the text and and one of the strongest features of NVIM.

Now let's use what we have learned to delete, copy, and paste text.

To delete text, we can use the "d" key, and then the motion we want to use, for example, if we want to delete the next 10 lines, we can use the "10dd" command, and if we want to delete the next 10 words, we can use the "10dw" command. And if we want to delete the next 10 characters, we can use the "10dl" command, and if we want to delete the whole line we can use the "dd" command.

To copy text, we can use the "y" key, and then the motion we want to use, for example, if we want to copy the next 10 lines, we can use the "10yy" command, and if we want to copy the next 10 words, we can use the "10yw" command. And if we want to copy the next 10 characters, we can use the "10yl" command, and if we want to copy the whole line we can use the "yy" command.

To paste text, we can use the "p" key, this will paste the text after the cursor, and if we want to paste the text before the cursor, we can use the "P" key.

Registers

And now's the funny thing, have you seen what happens when we delete or copy text ? the text is saved in a register, and we can access it by using the "p" key, and we can access the last deleted text by using the "P" key. This is something a lot of beginners hate because they don't know what a register is or how to access it, so let me explain it to you.

A register is a place where the text is saved, and we can access it by using "Leader" (normally "Space") and the double quote key, sometimes we need to do "Leader" and double quote two times if you layout is International, and a panel will appear with all the registers, and you will see that the lates copied text is saved in the "0" register, so now that we know this, you can access it by using the "0p" command. And if you want to access the last deleted text, you can use the "1p" command.

Buffers

A buffer is a place where the text is saved, and you can access it by using the "Leader" key and "be", and a panel will appear with all the buffers, and you can navigate through them by using the "j" and "k" keys. You can also use the "d" key to delete a buffer.

One way of thinking of buffers is like tabs, you can have multiple buffers open at the same time, and you can navigate through them, each time you open a file a new buffer is created and saved into memory, and if you open the same buffer in two places at the same time you will see that if you change something in one buffer, it will change in the other buffer too.

There's a special command I created so you can clear all buffers but the current one for those special times where you have been coding for hours and the performance is a little bit slow, you can do "Leader" and "bq".

Marks

Marks are amazing, you can create a new mark by using the "m" key and then a letter, for example, if you want to create a new mark in the current line, you can use the "ma" command, and if you want to jump to that mark, you can use the "`a" command.

If you want to delete a mark, do ":delm keyOfTheMark", and to delete ALL the marks ":delm!" .Marks are saved in the current buffer, and you can use them to navigate through the text quickly.

Recordings

Now this is amazing and super useful, let's say we need to do an action multiples times and its super tedious to do so, what NVIM provides is a way to replicate a set of commands by creating a macro, you can start recording by using the "q" key and then a letter, for example, if you want to start recording a macro in the "a" register, you can use the "qa" command, and then you can do the actions you want to replicate, and then you can stop recording by using the "q" key.

To replay the macro, you can use the "@" key and then the letter, for example, if you want to replay the macro in the "a" register, you can use the "@a" command. This is super useful and will make you more efficient.

And again you can use motions with your recordings, for example, if you want to delete the next 10 lines and copy them, you can use the "qad10jyy" command, and then you can replay the macro by using the "@a"

command, and also you can replicate the macro multiple times by using the "10@a" command.

Algorithms the Gentleman Way

Big O Notation

"How code slows as data grows"

- Performance of an algorithm depends on the amount of data it is given.
- Number of steps needed to complete. Some machines run algorithms faster than others so we just take the number of steps needed.
- Ignore smaller operations, constants. O(N + 1) → O(N) where N represents the amount of data.

```
function sum(n: number): number {
  const sum = 0;

  for(let i = 0; i < n; i++) {
    sum += i;
  }

  return sum;
}

// if n equals 10, then O(N) is 10 steps, if in equals 100, then O(N) is 100 steps
```

Here we can see that O(N) is linear which means that the amount of steps depends on the number of data we are given.

```
function sayHi(n: string):  string{
  return `Hi ${n}`
}

// if n equals 10, then O(1) is 3 step... 3 ? YES 3 steps

// 1 - Create new string object to store the result (allocating memory for the new string)
// 2 - Concatenate the 'Hi' string with the result.
// 3 - Return the concatenated string.
```

But now we have O(1) as the amount of steps does not depend on the amount of data we are given, it will always be 1.

Needed previous knowledge

- The 'Log' of a number is the power to which the base must be raised to produce that number. For example, the log base 2 of 8 is 3 because 2^3 = 8.
- 'Linear' means that the number of steps grows linearly with the amount of data.
- The 'Quadratic' of a number is the square of that number. For example, the quadratic of 3 is 9 because 3^2 = 9.
- 'Exponential' of a number is the power of the base raised to that number. For example, the exponential of 2 to the power of 3 is 8 because 2^3 = 8.
- 'Factorial' of a number is the product of all positive integers less than or equal to that number. For example, the factorial of 3 is 6 because 3!!) - Factorial time - The number of steps grows factorially (brute force algorithms, those which try all possible solutions)

Example with N equal to 1000:

```bash
- O(1) - 1 step
- O(log N) - 10 steps
- O(N) - 1000 steps, a thousand steps
- O(N log N) - 10000 steps, a ten thousand steps
- O(N^2) - 1000000 steps, a million steps
- O(2^N) - 2^1000 steps
- O(N!) - 1000! steps, factorial of 1000
```

The main idea is that we want to avoid exponential and factorial time algorithms as they grow very fast and are not efficient at all, UNLESS we are sure that the amount of data we are given is very small as it can actually be faster than other algorithms.

Letter grade for Big O Notation, from best to worst, taking in consideration we are using a big dataset of data:

```
- O(1) - Constant time - A
- O(log N) - Logarithmic time - B
- O(N) - Linear time - C
- O(N log N) - Linearithmic time - D
- O(N^2) - Quadratic time - F
- O(2^N) - Exponential time - F
- O(N!) - Factorial time - F
```

Examples using code

O(1) - Constant time

```
function sayHi(n: string):  string{
  return `Hi ${n}`
}
```

Here's why it's O(1):

- The algorithm performs a constant amount of work, regardless of the size of the input.
- The number of steps needed to complete the algorithm does not depend on the input size.

Therefore, the time complexity of the algorithm is O(1) in all cases.

O(log N) - Logarithmic time

```
// having the following array that represents the numbers from 0 to 9 in order
const arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

// we want to find the index of a number in the sorted array
function binarySearch(arr: number[], target: number): number {
  // initialize left and right pointers
  let left = 0;
  let right = arr.length - 1;

  // while left is less or equal to right we keep searching for the target
  while (left <= right) {
    // get the middle of the array to compare with the target
      // we iterate using the middle of the array to find the target because we know the array is sor
    const mid = Math.floor((left + right) / 2); // middle index
    const midValue = arr[mid]; // middle value

    // if the middle value is the target, return the index
    if (midValue === target) {
      return mid;
    }

    // if the middle value is less than the target, we search the right side of the array by updating
    if (midValue < target) {
      left = mid + 1;
    } else {
      right = mid - 1;
    }
  }
  return -1; // target not found
}
```

In binary search, the algorithm continually divides the search interval in half until the target element is found or the search interval becomes empty. With each iteration, the algorithm discards half of the search space based on a comparison with the middle element of the current interval.

Here's why it's O(log N):

- In each iteration of the while loop, the search space is halved.
- This halving process continues until the search space is reduced to a single element or the target is found.
- Since the search space is halved with each iteration, the number of iterations required to reach the target element grows logarithmically with the size of the input array.

Thus, the time complexity of binary search is O(log N) on average.

O(N) - Linear time

```
function sum(n: number): number {
  const sum = 0;
  for(let i = 0; i < n; i++) {
    sum += i;
  }
  return sum;
}
```

Here's why it's O(N):

- The algorithm iterates over the input array once, performing a constant amount of work for each element.
- The number of iterations is directly proportional to the size of the input array.
- As the input size increases, the number of steps needed to complete the algorithm grows linearly.

Therefore, the time complexity of the algorithm is O(N) in the worst-case scenario.

O(N log N) - Linearithmic time

```
// having the following array
const arr = [5, 3, 8, 4, 2, 1, 9, 7, 6];

// we want to sort the array using the quick sort algorithm
function quickSort(arr: number[]): number[] {
  // first we check if the array has only one element or no elements
  if (arr.length <= 1) {
    return arr;
  }

  // we get the pivot as the last element of the array, the pivot is the element we are going to comp
  const pivot = arr[arr.length - 1];
```

```
  // we create two arrays, one for the elements less than the pivot and another for the elements grea
  const left = [];
  const right = [];

  // we iterate through the array and compare each element with the pivot
  for (let i = 0; i < arr.length - 1; i++) {
    // if the element is less than the pivot, we add it to the left array
    if (arr[i] < pivot) {
      left.push(arr[i]);
    } else {
      // if the element is greater than the pivot, we add it to the right array
      right.push(arr[i]);
    }
  }

  // we recursively call the quickSort function on the left and right arrays and concatenate the resu
  return [...quickSort(left), pivot, ...quickSort(right)];
}
```

Here's why it's O(N log N):

- The algorithm partitions the array into two subarrays based on a pivot element and recursively sorts these subarrays.
- Each partitioning step involves iterating over the entire array once, which takes O(N) time. However, the array is typically divided in a way that the size of the subarrays reduces with each recursive call. This results in a time complexity of O(N log N) on average.

O(N^2) - Quadratic time

```
// having the following array
const arr = [5, 3, 8, 4, 2, 1, 9, 7, 6];

// we want to sort the array using the bubble sort algorithm
function bubbleSort(arr: number[]): number[] {

  // we iterate through the array
  for (let i = 0; i < arr.length; i++) {

    // we iterate through the array again
    for (let j = 0; j < arr.length - 1; j++) {

      // we compare adjacent elements and swap them if they are in the wrong order
      if (arr[j] > arr[j + 1]) {

        // we swap the elements
        const temp = arr[j];
        arr[j] = arr[j + 1];
        arr[j + 1] = temp;
```

```
      }
    }
  }
  return arr;
}
```

Here's why it's O(N^2):

- Bubble sort works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order.
- In the worst-case scenario, where the array is in reverse sorted order, bubble sort will need to make N passes through the array, each pass requiring N-1 comparisons and swaps.
- This results in a total of N * (N-1) comparisons and swaps, which simplifies to O(N^2) in terms of time complexity.

O(2^N) - Exponential time

```
// we want to calculate the nth Fibonacci number using a recursive algorithm
function fibonacci(n: number): number {

  // we check if n is 0 or 1 as the base case of the recursion because the Fibonacci sequence starts
  if (n <= 1) {
    return n;
  }

  // we recursively call the fibonacci function to calculate the nth Fibonacci number
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Here's why it's O(2^N):

- In each recursive call to the fibonacci function, two additional recursive calls are made with n - 1 and n - 2 as arguments.
- This leads to an exponential growth in the number of recursive calls as n increases.
- Each level of recursion branches into two recursive calls, resulting in a binary tree-like structure of recursive calls.
- The number of function calls doubles with each level of recursion, leading to a total of 2^N function calls when calculating the nth Fibonacci number.

Therefore, the time complexity of the algorithm is O(2^N) in the worst-case scenario.

O(N!) - Factorial time

```
// having the following array
const arr = [1, 2, 3];

// we want to generate all permutations of a given array using a recursive algorithm
function permute(arr: number[]): number[][] {
  // base case: if the array has only one element, return it as a single permutation
  if (arr.length <= 1) {
    return [arr];
  }

  // initialize an empty array to store permutations
  const result: number[][] = [];

  // iterate over each element in the array
  for (let i = 0; i < arr.length; i++) {
    // generate all permutations of the array excluding the current element
    const rest = arr.slice(0, i).concat(arr.slice(i + 1));
    const permutations = permute(rest);

    // add the current element to the beginning of each permutation
    for (const perm of permutations) {
      result.push([arr[i], ...perm]);
    }
  }
  return result;
}
```

Here's why it's O(N!):

- In each recursive call to the permute function, the algorithm generates permutations by selecting each element of the array as the first element and then recursively generating permutations of the remaining elements.
- The number of permutations grows factorially with the size of the input array.
- For each element in the array, there are (N-1)! permutations of the remaining elements, where N is the number of elements in the array.
- Therefore, the total number of permutations is N * (N-1) * (N-2) * ... * 1, which is N factorial (N!).

Hence, the time complexity of the algorithm is O(N!) in the worst-case scenario.

Worst-case, Best-case, and Average-case Complexity

- The worst-case time complexity represents the maximum number of steps an algorithm takes to complete for a given input size. It provides an upper bound on the algorithm's performance. It is the most commonly used measure of time complexity in job interviews.

- The best-case time complexity represents the minimum number of steps an algorithm takes to complete for a given input size. It provides a lower bound on the algorithm's performance. It is less informative than the worst-case complexity and is rarely used in practice.
- The average-case time complexity represents the expected number of steps an algorithm takes to complete for a given input size, averaged over all possible inputs. It provides a more realistic estimate of an algorithm's performance than the worst-case complexity. However, calculating the average-case complexity can be challenging and is often avoided in favor of the worst-case complexity.

Space complexity

The space complexity of an algorithm is a measure of the amount of memory it requires to run as a function of the input size. It is typically expressed in terms of the maximum amount of memory used by the algorithm at any point during its execution.

It is important to distinguish between time complexity and space complexity, as an algorithm with good time complexity may have poor space complexity, and vice versa. For example, a recursive algorithm with exponential time complexity may also have exponential space complexity due to the recursive calls consuming memory.

But something to have in mind is that space complexity is not as important as time complexity, as memory is usually cheaper than processing power and in real life scenarios, we usually skip the space complexity analysis and focus on time complexity.

Imagine you're at a traditional Argentine barbecue, known as an "asado." You've got limited space on the grill (similar to limited memory in computing), and you want to optimize how much meat you can cook at once.

Now, let's compare the meat (or "carne") to the data in an algorithm. When you're cooking, you have to consider how much space each cut of meat takes up on the grill. Similarly, in computing, algorithms have to consider how much memory space they need to store and process data.

But here's the thing: at an asado, the most important factor is usually how quickly you can cook the meat and serve it to your guests. Similarly, in computing, the time it takes for an algorithm to run (time complexity) is often the most critical factor for performance.

So, while it's essential to be mindful of how much space (or "espacio") your algorithm uses, it's usually more exciting to focus on how efficiently it can solve a problem in terms of time.

Of course, in some situations, like if you're grilling on a tiny balcony or cooking for a huge crowd, space becomes more of a concern. Similarly, in computing, if you're working with limited memory resources or on a device with strict memory constraints, you'll need to pay closer attention to space complexity.

But overall, just like at an Argentine barbecue, the balance between time and space complexity is key to creating a delicious (or efficient) outcome!

However, let's talk about how you calculate the space complexity, or "cuánto espacio ocupas" in the case of our barbecue analogy. Just as you'd assess how much space each cut of meat takes up on the grill, in

computing, you need to consider how much memory each data structure or variable in your algorithm consumes.

Here's a basic approach to calculate space complexity:

- **Identify the Variables and Data Structures**: Look at the algorithm and identify all the variables and data structures it uses. These could be arrays, objects, or other types of variables.
- **Determine the Space Used by Each Variable**: For each variable or data structure, determine how much space it occupies in memory. For example, an array of integers will take up space proportional to the number of elements multiplied by the size of each integer.
- **Add Up the Space**: Once you've determined the space used by each variable, add them all up to get the total space used by the algorithm.
- **Consider Auxiliary Space**: Don't forget to account for any additional space used by auxiliary data structures or function calls. For example, if your algorithm uses recursion, you'll need to consider the space used by the call stack.
- **Express Space Complexity**: Finally, express the space complexity using Big O notation, just like you do with time complexity. For example, if the space used grows linearly with the size of the input, you'd express it as O(N). If it grows quadratically, you'd express it as O(N^2), and so on.

So, just as you carefully manage the space on your grill to fit as much meat as possible without overcrowding, in computing, you want to optimize the use of memory to efficiently store and process data. And just like finding the perfect balance of meat and space at an Argentine barbecue, finding the right balance of space complexity in your algorithm is key to creating a delicious (or efficient) outcome!

Example Time !!!

Let's use a simple algorithm to find the sum of elements in an array as an example for calculating space complexity.

```
function sumArray(arr: number[]): number {
    let sum = 0;  // Space used by the sum variable: O(1)

    for (let num of arr) {  // Space used by the loop variable: O(1)
        sum += num;  // Space used by temporary variable: O(1)
    }

    return sum;  // Space used by the return value: O(1)
}
```

In this example:

- We have one variable `sum` to store the sum of elements, which occupies a constant amount of space, denoted as O(1).

- We have a loop variable `num` that iterates through each element of the array. It also occupies a constant amount of space, O(1).
- Within the loop, we have a temporary variable to store the sum of each element with `sum`, which again occupies a constant amount of space, O(1).
- The return value of the function is the sum, which also occupies a constant amount of space, O(1).

Since each variable and data structure in this algorithm occupies a constant amount of space, the overall space complexity of this algorithm is O(1).

In summary, the space complexity of this algorithm is constant, regardless of the size of the input array.

Now let's consider an example where we create a new array to store the cumulative sum of elements from the input array. Here's the algorithm:

```
function cumulativeSum(arr: number[]): number[] {
    const result = []; // Space used by the result array: O(N), where N is the size of the input array
    let sum = 0;  // Space used by the sum variable: O(1)
    for (let num of arr) {  // Space used by the loop variable: O(1)
        sum += num;  // Space used by temporary variable: O(1)
        result.push(sum);  // Space used by the new element in the result array: O(1), but executed N t
    }
    return result;  // Space used by the return value (the result array): O(N)
}
```

In this example:

- We have a variable `result` to store the cumulative sum of elements, which grows linearly with the size of the input array `arr`. Each element added to `result` contributes to the space complexity. Therefore, the space used by `result` is O(N), where N is the size of the input array.
- We have a loop variable `num` that iterates through each element of the input array `arr`, which occupies a constant amount of space, O(1).
- Within the loop, we have a temporary variable `sum` to store the cumulative sum of elements, which occupies a constant amount of space, O(1).
- Inside the loop, we add a new element to the `result` array for each element in the input array. Each push operation adds an element to the array, so it also contributes to the space complexity. However, since it's executed N times (where N is the size of the input array), the space used by the push operations is O(N).
- The return value of the function is the `result` array, which occupies O(N) space.

Overall, the space complexity of this algorithm is O(N), where N is the size of the input array. This is because the space used by the `result` array grows linearly with the size of the input.

Arrays

When we talk about arrays, we usually think of ordered collections of elements, right? But in JavaScript, arrays are actually objects. So, what's a real array, you might ask? Well, a true array is a contiguous block of memory where each element takes up the same amount of space.

In a real array, accessing an element is super quick—it's a constant time operation, meaning it takes the same amount of time no matter how big the array is. Why? Because you can calculate exactly where each element is in memory.

Now, let's contrast that with JavaScript arrays. They're implemented as objects, where the indexes are the keys. So, when you access an element in a JavaScript array, you're actually accessing a property of an object. This means accessing elements isn't as snappy—it's a linear time operation because the JavaScript engine has to search through the object keys to find the right one.

To find the memory location of an element in a real array, you use a simple formula:

```
const index = base_address + offset * size_of_element;
```

Here, the **`index`** is what we usually call the index, but it's more like an offset. The **`base_address`** is the starting point of the array in memory, and **`size_of_element`** is, well, the size of each element.

With this formula, every time you search for an element, you're performing a constant time operation because it doesn't matter how big the array is—the math stays the same.

Now, let's illustrate this with some code:

```
// Let's create a new array with 5 elements
const a = [1, 2, 3, 4, 5];

// Calculate the total space the array occupies in memory
const totalSpace = array.length * 4; // assuming each element occupies 4 bytes

// Choose an index
const index = 3;

// Calculate the memory location of the element
const sizeOfEachElement = 4; // each element occupies 4 bytes
const baseAddress = array; // the reference to the array itself
const offset = index * sizeOfEachElement;
const memoryLocation = baseAddress + offset;

// Access the element at the calculated memory location
const elementAtIndex = memoryLocation;

console.log("Value at index", index, "is:", elementAtIndex); // Value at index 3 is: 4
```

In this example, we're simulating how a real array works under the hood. We calculate the memory location of an element using the index, and then we access that memory location to get the element.

Now, let's visualize this with Node.js, where we can peek into what's happening in memory:

```
// Create a new array buffer in Node.js
const buffer = new ArrayBuffer(16); // 16 bytes of memory

// Check the size of the buffer (in bytes)
console.log("buffer.byteLength: " + buffer.byteLength); // Output: buffer.byteLength: 16

// Now let's create a new Int8Array, a typed array of 8-bit signed integers
const int8Array = new Int8Array(buffer);

// Log the contents of the Int8Array (all zeros initially)
console.log(int8Array); // Output: Int8Array(16) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

// Each element in the Int8Array represents a single byte in the buffer

// Now let's change the value of the first element of the array to 1
int8Array[0] = 1;

// Log the contents of the Int8Array and buffer again
console.log(int8Array); // Output: Int8Array(16) [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
console.log(buffer); // Notice how the underlying buffer is also modified (first byte becomes 1)
// Output: Buffer after change: ArrayBuffer { [Uint8Contents]: <01 00 00 00 00 00 00 00 00 00 00 00 00

// TypedArrays provide a different view of the same memory

// Now let's create a 16-bit signed integer array (uses 2 bytes per element)
const int16Array = new Int16Array(buffer);

// Log the contents of the Int16Array (initial values based on buffer)
console.log(int16Array); // Output: Int16Array(8) [1, 0, 0, 0, 0, 0, 0, 0]

// The Int16Array has fewer elements because it uses more bytes per element, 2 bytes

// Again, let's change a value (third element) and see the effects
int16Array[2] = 4000;

// Log the contents of all three arrays and the buffer
console.log(int16Array); // Output: Int16Array(8) [1, 0, 4000, 0, 0, 0, 0, 0]
console.log(buffer); // Notice how multiple bytes in the buffer are modified (5th and 6th bytes)
// Output: ArrayBuffer { [Uint8Contents]: <01 00 00 00 a0 0f 00 00 00 00 00 00 00 00 00 00> } (contents
console.log(int8Array); // The view of the Int8Array is also affected  (5th and 6th bytes change)
// Output: Int8Array(16) [1, 0, 0, 0, -96, 15, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] (5th byte is -96, 6th byte

// Now let's create a 32-bit signed integer array (uses 4 bytes per element)
const int
```

```
32Array = new Int32Array(buffer);

// Log the contents of the Int32Array (initial values based on buffer)
console.log(int32Array); // Output: Int32Array(4) [1, 4000, 0, 0]

// Even fewer elements due to the larger size of each element, 4 bytes

// Let's change the value and observe the effects
int32Array[2] = 100000;

// Log the contents of all three arrays and the buffer
console.log(int32Array); // Output: Int32Array(4) [1, 4000, 100000, 0]
console.log(buffer); // ArrayBuffer { [Uint8Contents]: <01 00 00 00 a0 0f 00 00 10 27 00 00 00 00 00 00
console.log(int16Array); // Int16Array(8) [1, 0, 4000, 0, 10000, 0, 0, 0] (4th element is now 10000)
console.log(int8Array); // Int8Array(16) [1, 0, 0, 0, -96, 15, 0, 0, 16, 39, 0, 0, 0, 0, 0, 0] (9th, 10
```

In this Node.js example, we're creating a buffer of 16 bytes, which is like a block of memory. Then, we create different typed arrays to view this memory differently—as arrays of different types of integers. Modifying one of these arrays also changes the underlying buffer, showing how they're different views of the same memory.

Now, let's circle back to why we're seeing decimals. We'll need to understand two's complement representation first.

So, two's complement is like the magic trick we use in computing to handle both positive and negative numbers using the same binary system. Picture this: in our binary numbers, the first digit from the left (the big boss, you know?) decides if the number is positive or negative. If it's 0, it's positive, and if it's 1, it's negative.

Now, for the positive numbers, it's easy peasy. You just write them down in binary, as usual. For example, number 3 in 8-bit binary is 00000011. No problem there, right?

But when it comes to negative numbers, we need a trick. We take the positive number's binary representation, flip all the bits (0s become 1s and 1s become 0s), and then add 1 to the result. This gives us the two's complement of the negative number. Let's say we want to represent -3. First, we start with 3's binary representation, which is 00000011. Then, we flip all the bits to get 11111100, and finally, we add 1 to get 11111101. That's the two's complement of -3 in 8-bit binary.

Now, why do we do all this? Well, it's because in computing, we like things to be tidy and consistent. With two's complement, we can use the same rules for adding and subtracting both positive and negative numbers, without needing to worry about special cases. It keeps our math nice and clean, just like sipping on yerba mate on a sunny day in Buenos Aires.

Now we can see why we are seeing decimals !

When working with 8 bits, the values are being represented in decimal, as the range of an 8-bit signed integer is -128 to 127. If the value overflows, it wraps around the range:

When you're dealing with 8 bits, the values are shown in decimal because, you know, an 8-bit signed integer can only hold numbers from -128 to 127. It's like having a limited space in a crowded bus—you can only fit so many people!

Now, imagine you're trying to squeeze in the number 4000. In binary, that's `111110100000`. But here's the thing: our bus only goes up to 127. So, when you try to cram 4000 in there, it's like trying to fit a football team into a tiny car—it's just not gonna happen.

Now, when you try to jam 4000 into our 8-bit bus, it overflows. But instead of causing chaos, it does something pretty neat: it wraps around. You see, the bus route starts again from the beginning, like a never-ending loop.

This "wrapping around" is where things get interesting. The leftmost bit, the big boss of the number, flips its sign. So, instead of being positive, it becomes negative. It's like turning the bus around and going in the opposite direction!

Now, we use our previous trick called two's complement to figure out the new number. First, we flip all the bits of `111110100000` to get `000001011111`. Then, we add 1 to this flipped number, giving us `000001100000`.

And voilà! That binary number `000001100000` represents -96 in decimal. So, even though you tried to squeeze in 4000, our little bus handles it like a champ and tells you it's -96. That's the magic of overflow and wrap-around in an 8-bit world!

How to think?

When you're up to your eyeballs in code, understanding the concepts and applying them to crack those problems is the name of the game. My advice? Pour your heart and soul into commenting your code, explaining each step inside the method, and THEN dive into implementation.

That way, you'll know what needs doing and just need to figure out how to pull it off.

Example:

```
function sumArray(arr: number[]): number {
        // Initialize the sum variable to store the sum of elements
        let sum = 0;  // O(1)

        // Iterate over each element in the array
        for (let num of arr) {  // O(N)
                // Add the current element to the sum
                sum += num;  // O(1)
        }

        // Return the final sum
        return sum;  // O(1)
}
```

Linear Search

It's the bread and butter of algorithms, but do you really know how it struts its stuff?

Here's the scoop: we're sashaying through each element of a collection, asking if the element we're hunting down is the one staring us in the face. The JavaScript bigwig that uses this algorithm? The `indexOf` method, baby.

```
function indexOf(arr: number[], target: number): number {
    // Iterate over each element in the array
    for (let i = 0; i < arr.length; i++) {

        // Check if the current element is equal to the target
        if (arr[i] === target) {

            // If it is, return the index of the element
            return i;
        }
    }

    // If the target is nowhere to be found, return -1
    return -1;
}
```

So... what's the worst-case scenario here that'll give us the Big O Notation? That we come up empty-handed or that the element we're after is lounging at the end of the array, making this algorithm O(N). As N gets bigger, so does the complexity - it's like watching your waistline after devouring a mountain of alfajores.

Binary Search

Now, this bad boy right here is the cream of the crop, the bee's knees, and one of the heavy hitters in professional interviews and coding showdowns. Let's unpack when and how to unleash its power:

The name of the game? Divide and conquer, baby! Instead of playing tag with each element of the array, we slice that sucker in half and ask, "Hey, are you on the left or right?" Then, we keep slicing and dicing until we've nabbed our elusive target.

```
function binarySearch(arr: number[], target: number): number {
    // Initialize the left and right pointers
    // left starts at the beginning of the array
    // right starts at the end of the array
    let left = 0;
    let right = arr.length - 1;

    // Keep on truckin' as long as the left pointer is less than or equal to the right pointer
    while (left <= right) {
```

```
        // Calculate the middle index of the current search space
        const mid = Math.floor((left + right) / 2);

        // Check if the middle element is the target
        if (arr[mid] === target) {

            // If it is, return the index of the element
            return mid;
        } else if (arr[mid] < target) {

            // If the middle element is less than the target, swing right
            left = mid + 1;
        } else {

            // If the middle element is greater than the target, veer left
            right = mid - 1;
        }
    }
}
```

And whit this example you will see, clear as the light that guide us in the darkest nights that I'm not lying:

```
// we need to find the index of the target number inside an array of 1024 elements

1024 / 2 = 512 // we halve it and see that the target number is in the right half
512 / 2 = 256 // we halve it again and see that the target number is in the right half
256 / 2 = 128 // we halve it again and see that the target number is in the right half
128 / 2 = 64 // we halve it again and see that the target number is in the right half
64 / 2 = 32 // we halve it again and see that the target number is in the right half
32 / 2 = 16 // we halve it again and see that the target number is in the right half
16 / 2 = 8 // we halve it again and see that the target number is in the right half
8 / 2 = 4 // we halve it again and see that the target number is in the right half
4 / 2 = 2 // we halve it again and see that the target number is in the right half
2 / 2 = 1 // we halve it again and see that the target number is in the right half
1 / 2 = 0.5 // we can't halve it anymore, so we stop

// here comes the magic, if we count the number of steps we have a total of 10 steps to find the target
// do you know what is the logarithm of 1024 in base 2? it's 10 !!
log2(1024) = 10
```