

Mes conseils si je devais recommencer minishell



Mostafa Omrane · [Follow](#)

6 min read · May 3, 2024



5



1



Je note pleins de conseils si je devais recommencer ce projet 🙌

```
bash --posix

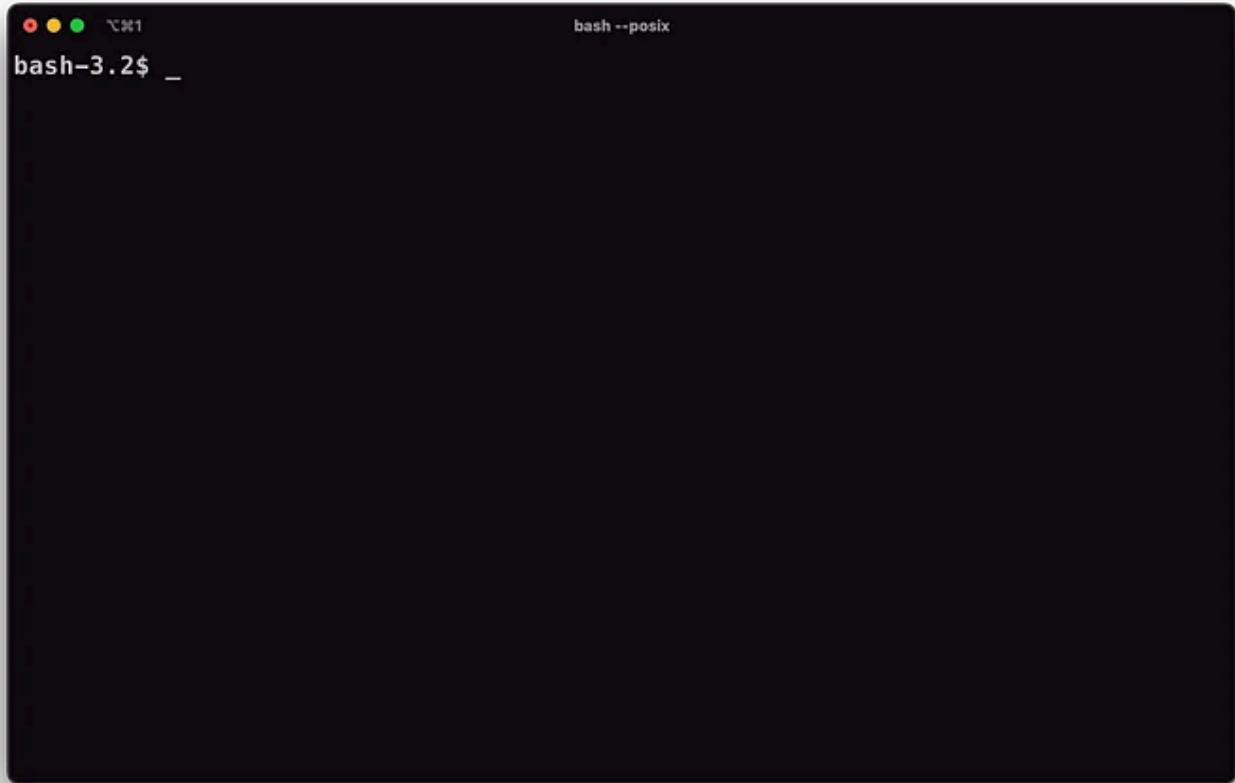
[minispell] > ls
Makefile      bin          libft        minishell    src
README.md     inc          micro        readline.supp todo.md
[minispell] > ls | wc -l
10
[minispell] > echo "Hi !" "I'm using" my minishell
Hi ! I'm using my minishell
[minispell] > echo "Hi !" "I'm using" my minishell > outfile
[minispell] > << EOF
> hi !
> I'm in an heredoc
> EOF
[minispell] > cat outfile
Hi ! I'm using my minishell
[minispell] > cat outfile | grep "xxx"
[minispell] > echo $?
1
[minispell] > lssss
minishell: lssss: command not found
[minispell] > echo $?
127
[minispell] > > outfile
[minispell] > cat outfile
[minispell] > _
```

Voici quelques exemples d'utilisation de notre minishell !

Contexte

Je suis étudiant à 42 Paris et nous avons un projet nommé minishell. À première vue, ce projet est très simple à comprendre : on le lance et il faut pouvoir faire comme si on utilisait bash.

Si j'ouvre un terminal sur mon ordinateur, j'ai un truc qui peut ressembler à ceci :

Capture d'écran de l'application iTerm2

Bash est un programme qui va prendre une ligne de commande et l'interpréter comme il se doit. Dans l'exemple ci-dessous, bash va simplement afficher l'endroit où on se situe dans le système de fichier de l'ordinateur :



J'ai simplement écrit pwd puis j'ai appuyé sur la touche ENTER



• Introduction

J'ai envie d'écrire l'article que j'aurais aimé lire avant de commencer minishell donc c'est partie pour décrire les meilleures étapes à suivre selon moi (à remettre en question si vous souhaitez).



• Utiliser GitHub

Je conseille fortement d'utiliser Git et GitHub pour ce projet. Crée un nouveau repository et ajoute ton ou ta coéquipier·ère dessus. Il y a des configurations à faire (ssh notamment) mais l'objectif, c'est que vous puissiez tous les deux git clone le projet, faire des modifications (ajouter/supprimer fichiers, écrire dedans), et push vos commits.

Une fois que tous est bien mis en place, let's go ici :

<https://learngitbranching.js.org/>

C'est le meilleur site qu'on ait trouvé pour comprendre comment fonctionnent les branches. Je te laisse passer quelques niveaux (pas obligé de tous faire) et surtout essaye de manipuler avec ton ou ta coéquipier·ère pour vraiment assimiler les différentes notions.



• Notre méthode de travail

Au début du projet, on est tombé sur [cette vidéo](#) et on en a essentiellement retenu qu'il ne fallait pas juste séparer le projet en deux gros morceaux puis se retrouver pour fusionner le tout. Je n'ai pas tout regardé (2:28:06) mais on s'est dit qu'expliquer son code à l'autre tout du long du projet allait être super important. Mais je pense qu'il y a autant de façon de travailler qu'il y a de groupe de travail, donc je vais juste décrire la méthode que je trouve intéressante avec mon expérience.

Plus tu comprends les étapes que bash suit pour interpréter une ligne de commande et plus ça sera simple de reproduire son comportement. On a surtout fonctionné avec des mini objectifs. Au début, on a commencé la partie “tokenisation” ensemble sur le même ordi. Une fois qu’on a vu qu’on pouvait séparer le travail en deux (créer la liste de token et identifier le type de chaque token) on a créé chacun sa branche pour travailler chacun de notre côté.

Quand on a fini chacun ce qu’on voulait faire, on s’explique rapidement notre code chacun notre tour puis on essaye de fusionner notre travail sur la branche principale (main) en utilisant la commande git merge (cf. le lien super pratique plus haut).

Après, on se définit un nouvel objectif et on refait pareil : on commence ensemble devant le même ordi (peer programming) puis on se sépare des tâches qu’on pourrait faire parallèlement.

• **readline**

Crée ton repository puis commence à coder un petit programme qui utilise la fonction readline et toutes les autres fonctions qui gravitent autour (rl_clear_history, rl_on_new_line, rl_replace_line, rl_redisplay, add_history).

Plus tu comprends comment elles fonctionnent en les testant et en lisant le man, et mieux, tu les utiliseras. Il ne suffit pas de juste faire fonctionner readline, mais de vraiment comprendre son fonctionnement (ce qu’elle retourne, ce qu’elle affiche, etc).



• **Comprendre la ligne de commande**

Maintenant que readline n'a plus de secret pour toi, tu vas récupérer sa valeur de retour (une chaîne de caractère) et tu dois l'interpréter.

Ce que j'entends par interpréter, c'est d'être capable de la comprendre pour pouvoir l'exécuter. Voici une énorme liste de commande qu'il faudra être capable d'interpréter : [800 tests pour minishell](#)

Trouvé ici : https://github.com/vietdu91/42_minishell

☹️ • Notre plus grande erreur

Avec ce projet, on n'a pas pris au sérieux cet encadré :



You should limit yourself to the subject description. Anything that is not asked is not required.

If you have any doubt about a requirement, take **bash** as a reference.

C'est un truc trop important parce qu'on peut facilement tomber dans un puits sans fond. Pour comprendre, voici l'exemple qui me vient en tête :

```
bash --posix
bash-3.2$ export_
```

En appuyant sur la touche ENTER, bash m'affiche les variables d'environnement (je te laisse essayer). Je me suis dit qu'il fallait faire en sorte qu'avec mon programme, je puisse avoir le même comportement, mais pas besoin parce que dans le *man export* il y a un passage qui dit :

When no arguments are given, the results are unspecified.

• **bash et le standard POSIX**

Pour tous les tests que tu feras, je te conseille de lancer bash comme ceci :

```
bash --posix
```

Dans le sujet du projet, il y a un lien vers la documentation à laquelle nous devons nous référer : [GNU Bash manual](#)

Open in app ↗

Sign up

Sign in

Medium



Search



Write



autre.

• **Le discord**

Dans le salon minishell du Discord de l'école, n'hésite pas à utiliser l'outil de recherche si tu as une question ou des doutes. Il y a sûrement une personne qui a déjà posé ta question et au pire, tu pourras la poser aux gens.

• **Les deux grandes phases**

Dans ce projet, on peut identifier deux grandes phases à ce projet. Il y a la première partie où ton programme va essayer d'analyser la ligne de commande fournit par readline puis il y a une phase dans laquelle tu dois exécuter cette ligne de commande.

🤔 • Analyser

Pour analyser la ligne de commande, voici un schéma :

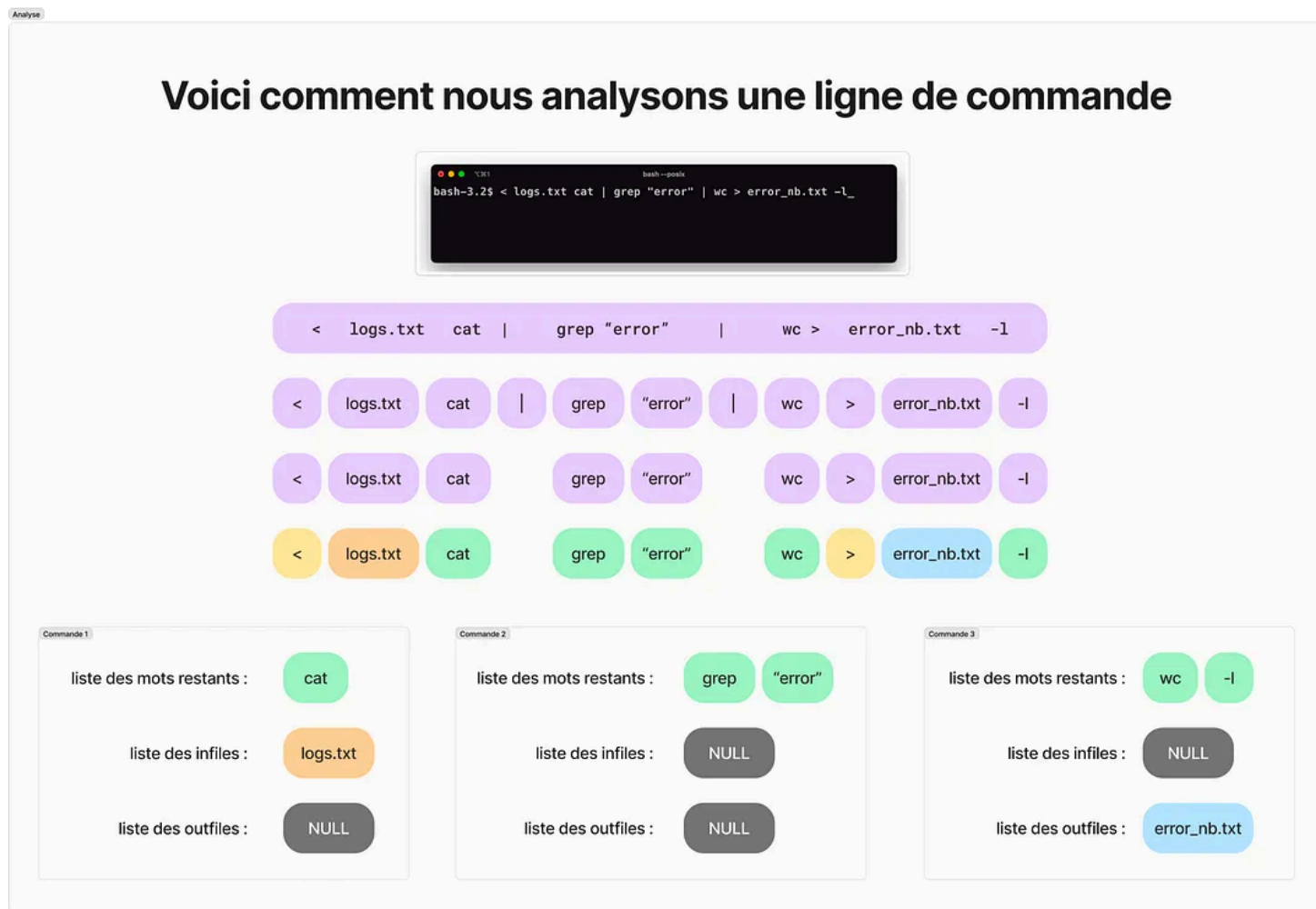


Schéma Figma Jam pour représenter nos différentes étapes permettant de comprendre une ligne de commande et pour ensuite lancer la partie exécution.

🤖 • Exécuter

Avant d'exécuter, il faut comprendre certaines notions.

Les file descriptors (descripteur de fichier ou fd)

Le plus simple pour comprendre ce qu'est un fd, c'est de comprendre ce petit exemple :

Let's understand, what's happen in this program

```

1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <string.h>
4
5 int main(void)
6 {
7     int fd;
8     char *msg;
9     char buf[1];
10
11     fd = open("file.txt", O_WRONLY | O_APPEND);
12     msg = "Some text from fd.c program appended in the file.txt\n";
13     write(fd, msg, strlen(msg));
14     close(fd);
15     fd = open("file.txt", O_RDONLY);
16     write(1, "Content of file.txt:\n", 21);
17     while (read(fd, buf, 1))
18         write(1, buf, 1);
19     close(fd);
20     return (0);
21 }

```

Dans ce programme, on ouvre un fichier appelé "file.txt" avec les modes suivants : `O_WRONLY | O_APPEND` afin de préciser qu'on pourra seulement écrire dans ce fichier sans écraser son contenu.

La fonction `open` a retourné un int appelé **file descriptor** que l'on stock dans une variable afin de le réutiliser plus bas. On utilise la fonction **write** pour écrire dans le fichier `file.txt` en spécifiant son file descriptor. On peut désormais fermer le fichier avec la fonction **close(fd)**.

On va ouvrir à nouveau le fichier mais avec un mode différent : `O_RDONLY` Puis en utilisant la fonction **read**, on peut lire dans le fichier `file.txt` en spécifiant son fd.

On peut remarquer l'utilisation de la fonction **write** avec comme premier argument le chiffre 1 alors que précédemment, nous avons utilisé la variable `fd`.

Le chiffre 1 fait référence à l'utilisation du fd STDIN (qui a la valeur 1) pour écrire dans la sortie standard. Dans de nombreux programmes C, l'affichage à l'écran est réalisé en écrivant dans la sortie standard (STDOUT), qui est associée au fd 1.

Cela ne veut pas forcément dire qu'il existe un fichier associé au fd 1 mais ça sert d'abstraction aux différents programmes. Les programmes s'attendent à recevoir des données en provenance de l'extérieur via le fd 1 (STDIN).

C'est une convention qu'on n'est pas obligé de respecter bien sûr.

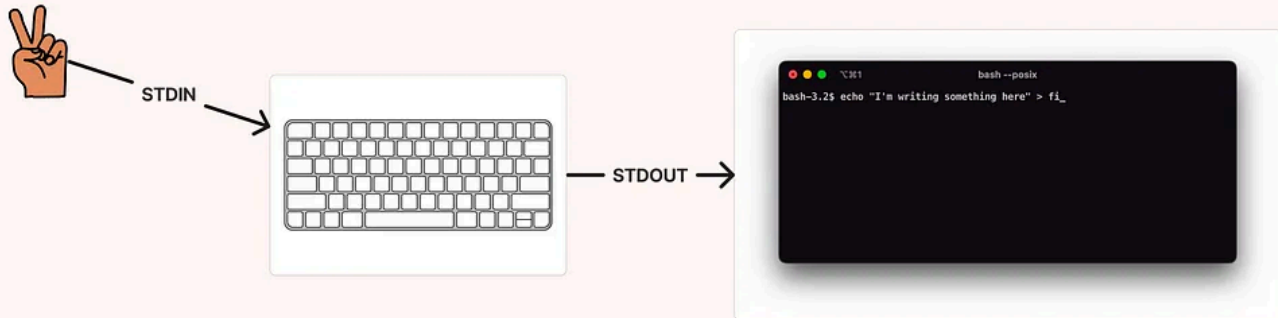


Code commenté pour comprendre comment utiliser les fd (réalisé sur Figma Jam)

STDIN, STDOUT et STDERR

J'ai essayé de schématiser ce que j'ai compris de ces fd standards.

Section 1



1. Mes doigts sont le STDIN pour mon clavier.
2. Le clavier dirige le résultat de l'appuie sur chaque touche vers le terminal via son STDOUT.
3. Le terminal reçoit sur son STDIN le STDOUT du clavier.
4. Puis le terminal pourra rediriger le résultat de la ligne de commande je suis en train d'écrire vers son STDOUT.
5. Et ainsi de suite...

Voici ce que j'ai compris avec STDIN STDOUT (et par extension STDERR).

pipe, dup2 et fork

Les fonctions `pipe()` et `dup2()` sont deux fonctions, utiles pour la communication entre processus. Dans minishell, on va devoir créer des processus enfants avec la fonction `fork()`, pour recréer la logique des pipes dans une ligne de commande.

Voici comment j'ai compris ce que fait concrètement un `fork()` :

Comprendre le fork()

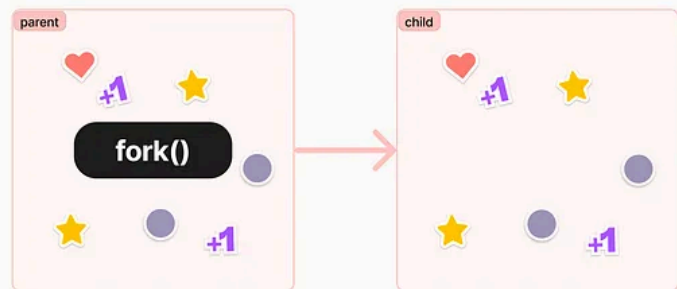
T1 : un process random né et se développe avec pleins de nouvelles données



T2 : le process lance la fonction fork()

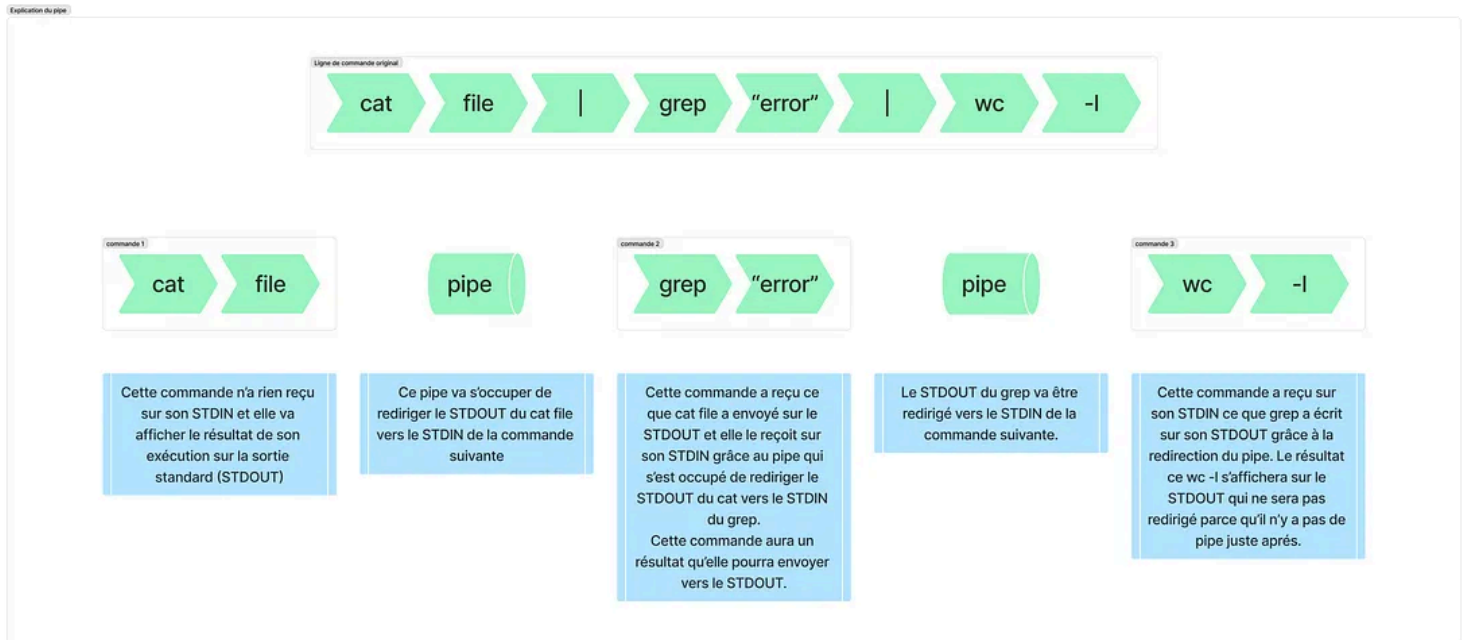


T3 : le process initial devient parent parce qu'un nouveau process est né



Un nouveau process est né du parent. Celui-ci hérite de tout ce que possède son parent. La valeur de retour du fork est un int supérieur à 0 pour le process parent et égale à 0 pour le process enfant. C'est à partir de là que chaque process fait sa vie.

Maintenant, il faut faire en sorte que l'enfant et le parent puissent communiquer entre eux. C'est là qu'on va expliquer la notion de pipe :



J'explique l'enchainement qu'il y a pendant l'exécution d'une commande avec deux pipe.

Merci d'avoir pris le temps de me lire et toujours plus d'infos dans mon bento : <https://bento.me/mostafa>

[42 Network](#)[Minishell](#)[Bash](#)[Shell](#)[Tuto](#)



Written by Mostafa Omrane

1 Follower

Follow

Je suis actuellement étudiant à 42 Paris et j'ai hâte de trouver un CDI 🌸

Recommended from Medium

ALEXANDER NGUYEN
Software Development Engineer

Mar. 2020 – May 2021

- Developed Amazon checkout and payment services to handle traffic of 10 Million daily global transactions
- Integrated Iframes for credit cards and bank accounts to secure 80% of all consumer traffic and prevent CSRF, cross-site scripting, and cookie-jacking
- Led Your Transactions implementation for JavaScript front-end framework to showcase consumer transactions and reduce call center costs by \$25 Million
- Recovered Saudi Arabia checkout failure impacting 4000+ customers due to incorrect GET form redirection

Projects

NinjaPrep.io (React)

- Platform to offer coding problem practice with built in code editor and written + video solutions in React
- Utilized Nginx to reverse proxy IP address on Digital Ocean hosts
- Developed using Styled-Components for 95% CSS styling to ensure proper CSS scoping
- Implemented Docker with Seccomp to safely run user submitted code with <2.2s runtime

HeatMap (JavaScript)

- Visualized Google Takeout location data of location history using Google Maps API and Google Maps heatmap code with React
- Included local file system storage to reliably handle 5mb of location history data
- Implemented Express to include routing between pages and jQuery to parse Google Map and implement heatmap overlay



Alexander Nguyen in Level Up Coding

The resume that got a software engineer a \$300,000 job at Google.

1-page. Well-formatted.



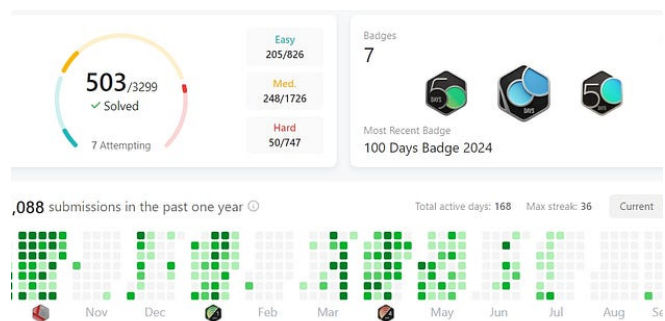
Jun 1



23K



465



Surabhi Gupta in Code Like A Girl

Why 500 LeetCode Problems Changed My Life

How I Prepared for DSA and Secured a Role at Microsoft



Sep 26



1.94K



47




Lists



General Coding Knowledge

20 stories · 1634 saves

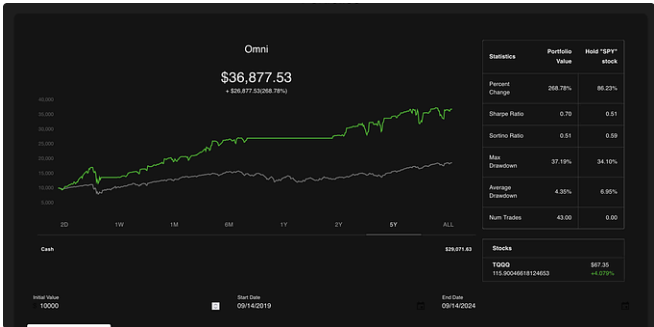


 Dylan Cooper in Stackademic

Google Disturbs the Python Community?! Core Developers...

O1 Chaos in the Python Community After Core Developer Suspended

★ Oct 1 🖱️ 708 💬 22 

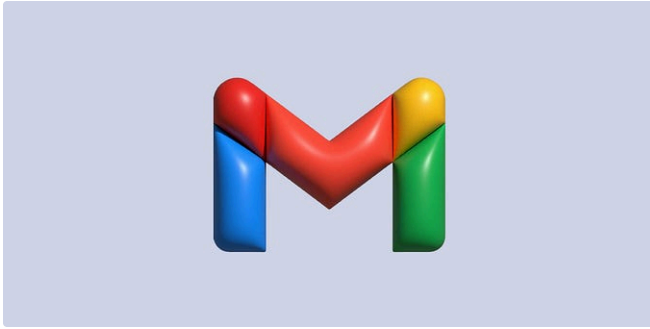


 Austin Starks in DataDrivenInvestor

I used OpenAI's o1 model to develop a trading strategy. It is...

It literally took one try. I was shocked.

★ Sep 15 🖱️ 3.4K 💬 95 



Anshul Kummar in Bouncin' and Behavin' Blogs

Goodbye Gmail: The Hard Truth About Why It's Time for a Change

The end of an era.



Sep 18



7.1K



173



Joe Procopio in Entrepreneurship Handbook

An Honest Recruiter Told Me Why Most Job Seekers Don't Get Hired

If you want a great job, you need some tough love. Here it is.



Sep 23



5.8K



99



See more recommendations