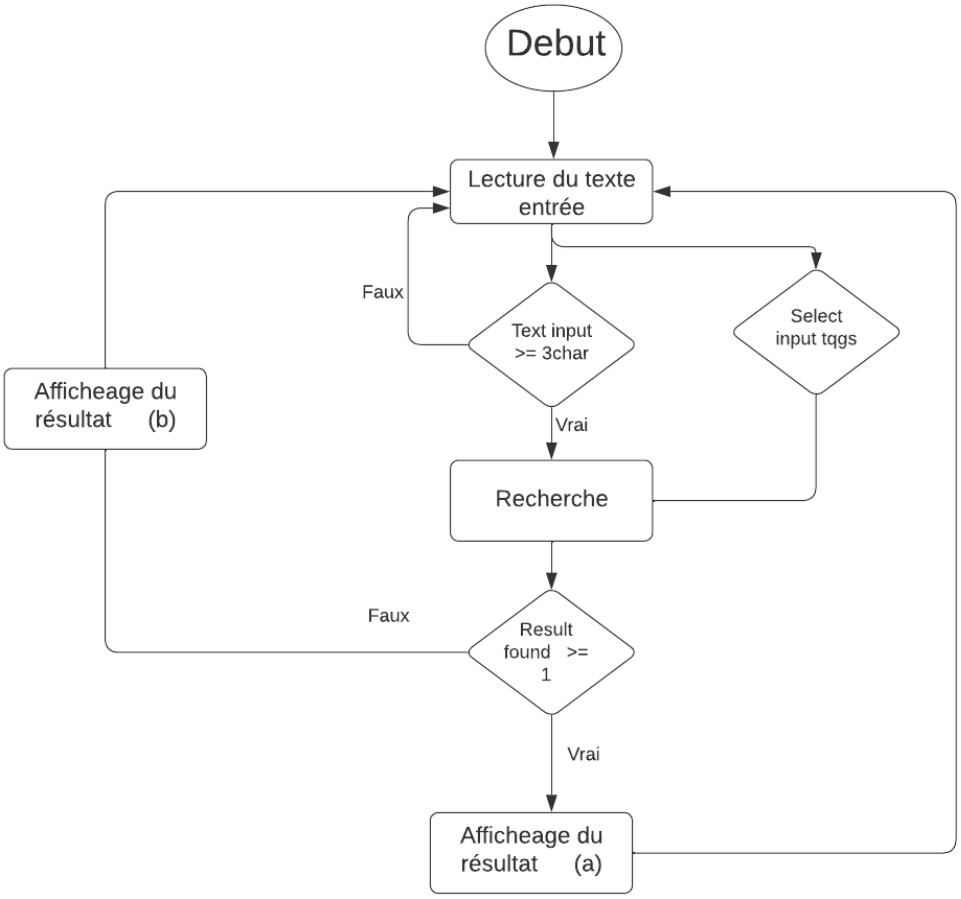
Plan:

I– POINT COMMUN DES DEUX ALGORITHMES

A- Généralité

Ces deux algorithmes de recherche permettent de rechercher des recettes correspondant à un ou plusieurs critères.
ces critères de recherche sont les suivants:

- une saisie dans la barre de recherche déclenche une recherche sur une recette : soit sur son nom, sa description ou ses ingrédient
- une saisie dans les champs de saisie des ingrédients, appareils et ustensils déclenche une recherche respectivement dans les listes ingrédients, appareils, et ustensils
- la sélection d'un ou plusieurs tag(s) parmi les trois catégories (ingrédient, appareil, ustensils) déclenche une recherche dans les clés "ingrédient", "appliances" et "ustensils" de chaque objet recette.
- l'algorithme de recherche par tag est identique pour ces deux approches; **seule la recherche par saisie dans la barre de recherche diffère entre les deux approches.**



B- Fonctionnalité

B-1) Récupération des paramètres des recherches dans l'objet searchParams

```
export const searchParams = {
    textSearch: '',
    ingredients: [],
    appliances: [],
    ustensils: []
};
```

B-2) Fonction search()

elle retourne un ou plusieurs id de recette(s) et peut appeler 4 sous fonctions indépendantes

```
ustensilsSearch()
appliancesSearch()
ingredientsSearch()
keywordSearch()
```

selon le cas de figure

Cas 1: aucun paramètre de recherche:

chargement de la page. lors du chargement de la page, on retourne un tableau de nombre allant de 1 à 50.

Cas 2: 1 mot saisi dans la barre de recherche

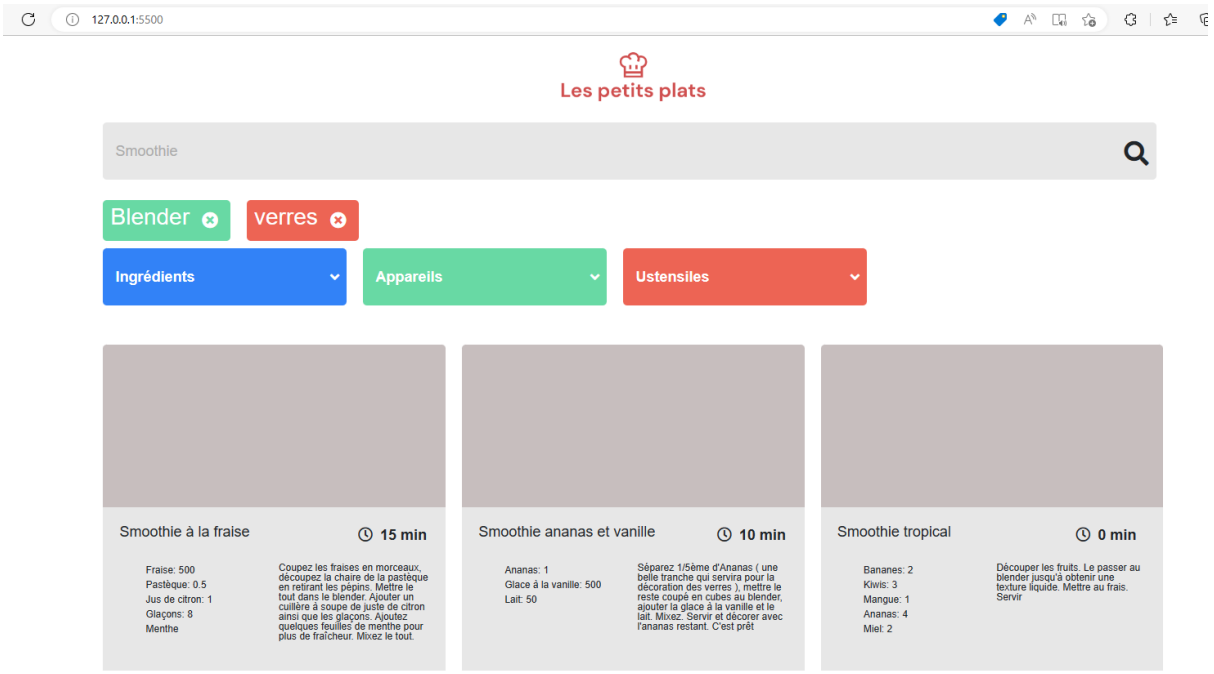
appelle de la fonction ***`keywordSearch()`***

Cas 3: un tag sélectionné dans une liste:

appel de la fonction correspondante: ***`ustensilsSearch()`*** pour les ustensils, ***`appliancesSearch()`*** pour les appareils, ***`ingredientsSearch()`*** pour les ingrédients

exemple:

```
export const searchParams = {
    textSearch: 'Smoothie',
    ingredients: [],
    appliances: [Blender],
    ustensils: [Verres]
};
```



on itère sur les objets activeSearch et searchResults en parallèle dans la fonction ***`search (searchParams)`***:

```
    const activeSearch = {
        ingredients: searchParams.ingredients.length > 0,
        appliance: searchParams.appliences.length > 0,
        ustensils: searchParams.appliences.length > 0,
        text: searchParams.textSearch !== ''
    };
```

```
        const searchResults = {
            ingredients: () => ingredientsSearch(idsFound), //
recherche par un tag ingredient (choisi)
            appliences: () => appliancesSearch(idsFound), // recherche
par un tag appliance (choisi)
            ustensils: () => ustensilsSearch(idsFound), // recherche
par un tag ustensil (choisi)
            text: () => keywordSearch(idsFound) // recherche par une
saisie sur la barre de recherche
        };
```

1. `activeSearch.ingredients` : `false` // car pas de recherche par tag ingredient

2. `activeSearch.appliances` : `true` // car on cherche le tag ‘Blender’ de appliance;

Pour cela, on appelle la fonction associée a la clé appliances comme suit: `searchResults.appliances` : `appliancesSearch(idsFound)`. Cette fonction reçoit donc `idsFound = []` en paramètre car elle a été appelée en premier. comme le tableau est vide, la fonction itérée donc sur le tableau complet des recettes. elle stock les ids des 5 recettes répondant au critère (tag) “Blender” dans `idsFound`

3. `activeSearch.ustensils` : `true` // car on cherche le tag ‘Verres’ de ustensils;

Pour cela, on appelle la fonction associée a la clé ustensils comme suit: `searchResults.ustensils` : `ustensilsSearch(idsFound)`. Cette fonction reçoit donc `idsFound = [1. 17. 18. 19. 49]` (on itère uniquement sur les résultats de la première recherch). on stock donc les ids res 5 recettes répondant au critère (tag) “verres” dans `idsFound`

4. `activeSearch.text` : `true` // car on recherche le texte `Coco` entré dans la barre de recherche. Pour cela, on appelle la fonction associée par sa clé comme suit: `searchResults.text`: `keywordSearch(idsFound)` . cette fonction (keywordSearch(idsFound)) effectuer la recherche sur les élément issus du résultat de la dernière recherche (c’est à dire sur les idsFound = [1, 17, 18, 19, 49]) et renvoi un tableau de 3 ids idsFoun = [17, 18, 49].

Remarque:

l’avantage de cet algorithme est qu’il permet de:

- déclencher la recherche seulement en cas de saisie par l’utilisateur
- restreindre le scope des recherches au résultats des recherches précédentes uniquement

  Ainsi, même si les 4 sous-fonctions de recherche sont indépendantes, dans le cas d’une recherche d’intersection (c’est à dir d’une recette correspondant aux paramètres saisis ou sélectionné), la recherche suivante (n+1) se base sur le résultat de la recherche précédente (n). ce qui évite:
  - que chaque sous fonction itère systématiquement sur les 50 recettes
  - de devoir croiser les résultats à la fin.

selon l’exemple ci-dessus

- recherche 1 (tag ingrédient) : pas de saisie utilisateur, non évalué
- recherche 2 (tag appareil) : première recherche effective, itère sur 50 recette et renvoi 5 recettes
- recherche 3 (tag ustensile) : itère sur 5 recettes et renvoi 5 recettes
- recherche 4 (mot clé) : itère sur 5 recettes et renvoi 3 recettes

par ailleur, comme la fonction de recherche par mot-clé `keywordSearch()` doit vérifier la correspondance entre une chaine de caractère dans chacun des 3 champs Titre, Ingredients, Description, ce qui a priori est plus long que la recherche par “tag”,

on l’appelle en dernier afin de lui donner un minimum de recett dans lesquelles rechercher.

Une optimisation possible de l'algorithme serait de permuter de manière dynamique l’ordre d'appeler des fonctions de recherche afin de déclencher la plus rapide et/ou la plus restrictive en premier.

II- PROGRAMMATION FONCTIONNELLE

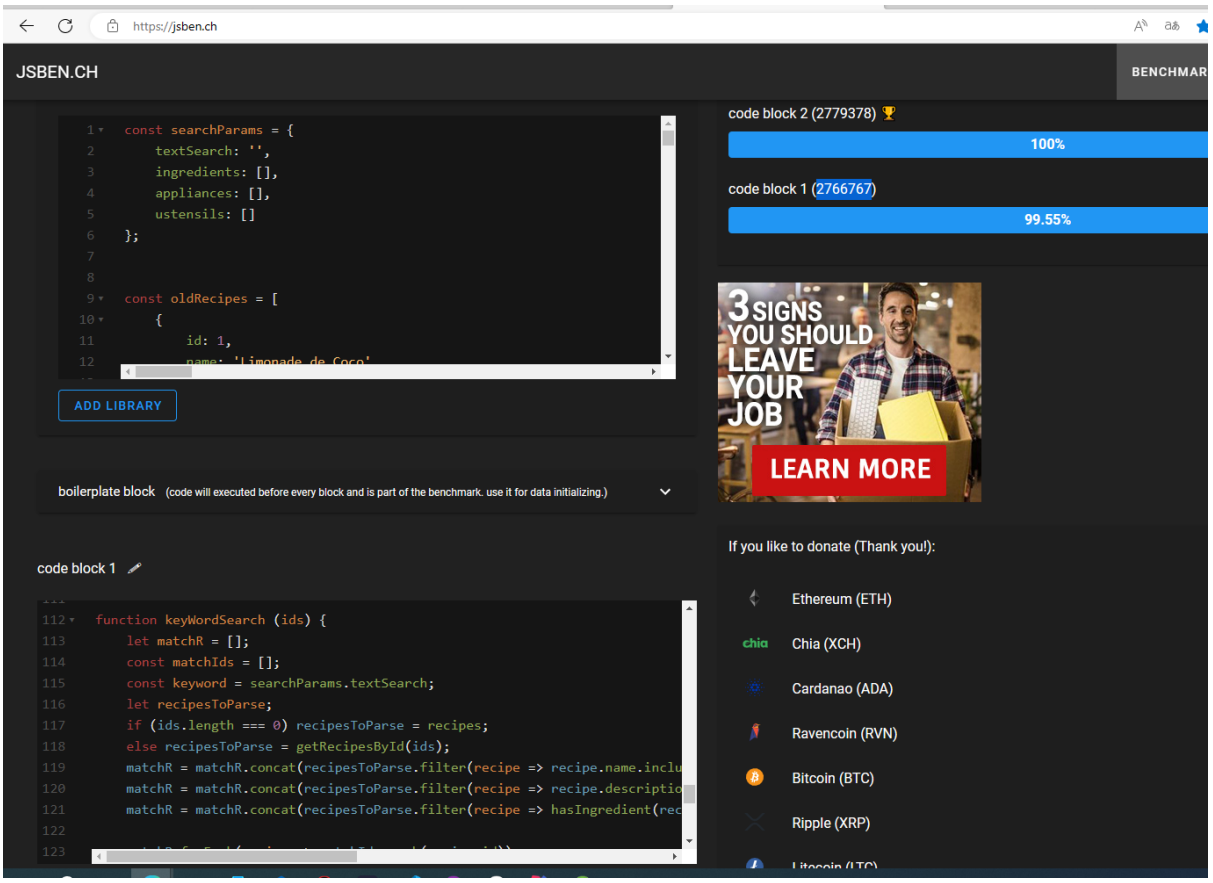
```
function keyWordSearch (ids) {
    let matchR = []
    const matchIds = []
    const keyword = searchParams.textSearch
    let recipesToParse

    if (ids.length === 0) recipesToParse = recipes
    else recipesToParse = getRecipesById(ids)

    matchR = matchR.concat(recipesToParse.filter(recipe =>
recipe.name.includes(keyword)))
    matchR = matchR.concat(recipesToParse.filter(recipe =>
recipe.description.includes(keyword)))
    matchR = matchR.concat(recipesToParse.filter(recipe =>
hasIngredient(recipe, [keyword])))

    matchR.forEach(recipe => matchIds.push(recipe.id))

    return matchIds.filter((value, index, filteredRecipes) =>
filteredRecipes.indexOf(value) === index)
}
```

description:

- pas d’affectation requises (les variables `` `matchR ` `` et `` `matchIds ` `` ont été déclarés juste par souci de lisibilité du code)
- Transparence référentielle : pas d'effet de bord produit par les fonctions : c’est à dire qu' en dehors du résultat attendu, elles ne font plus autre chaise comme par exemple afficher quelque chose à l’écran, écrire sur disque, échanger sur le réseau etc…
- fonction / methode passées en parametres ` concat( filter( include() ) )`

Sans aucun paramètre:
99.55%

plus de 2.7 millions d’opérations effectués

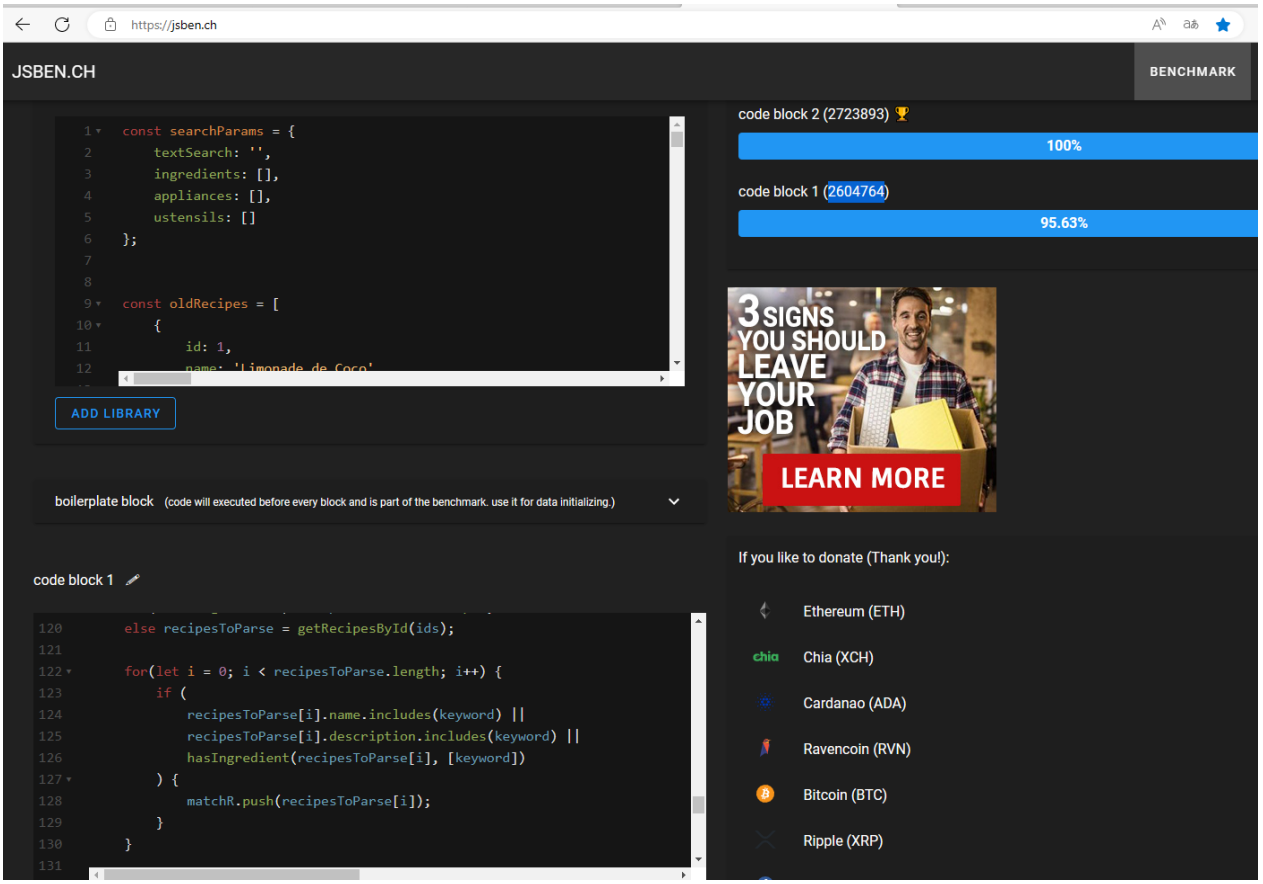


III- PROGRAMMATION IMPÉRATIVE

```
// RECHERCHE IMPERATIVE
function keywordSearch (ids = []) {
    const matchR = [];
    const matchIds = [];
    const result = [];
    const keyword = searchParams.textSearch;
    let recipesToParse;

    if (ids.length === 0) recipesToParse = recipes;
    else recipesToParse = getRecipesById(ids);

    for(let i = 0; i < recipesToParse.length; i++) {
        if (
            recipesToParse[i].name.includes(keyword) ||
            recipesToParse[i].description.includes(keyword) ||
```

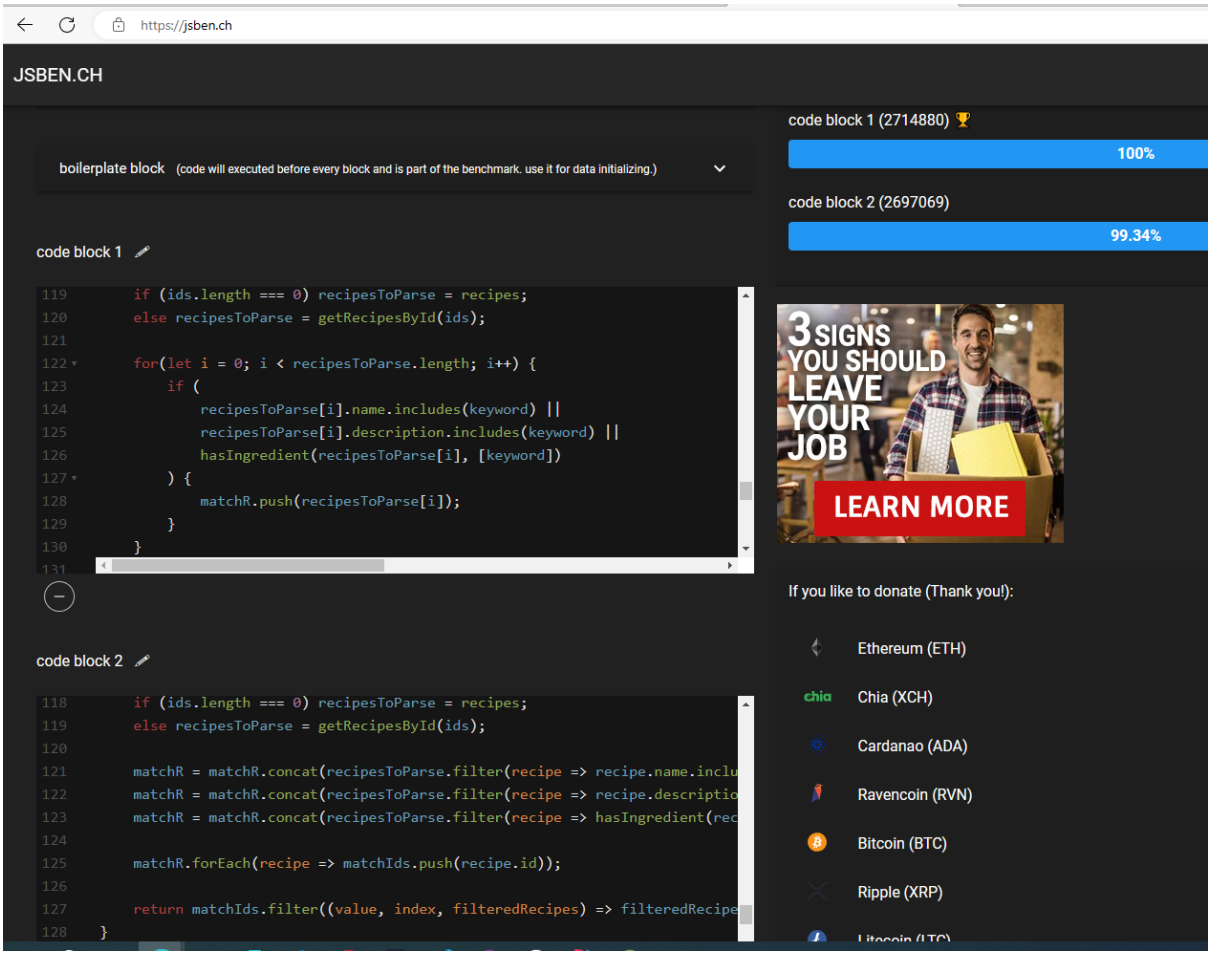
```
            hasIngredient(recipesToParse[i], [keyword])
        ) {
            matchR.push(recipesToParse[i]);
        }
    }

    for (let i = 1; i < matchR.length; i++) {
        matchIds.push(matchR[i].id);
    }

    for (let i = 0; i < matchIds.length; i++) {
        if (matchIds.indexOf(matchIds[i]) === i)
result.push(matchIds[i]);
    }
    return result;
}
```

Sans aucun paramètre:
95.63%
plus de 2.6 millions d'opérations éffectués



SIELINOU NOUBISSIE ERIC ROMUALD – Advance-it groupe

IV- COMPARAISON DES PERFORMANCES



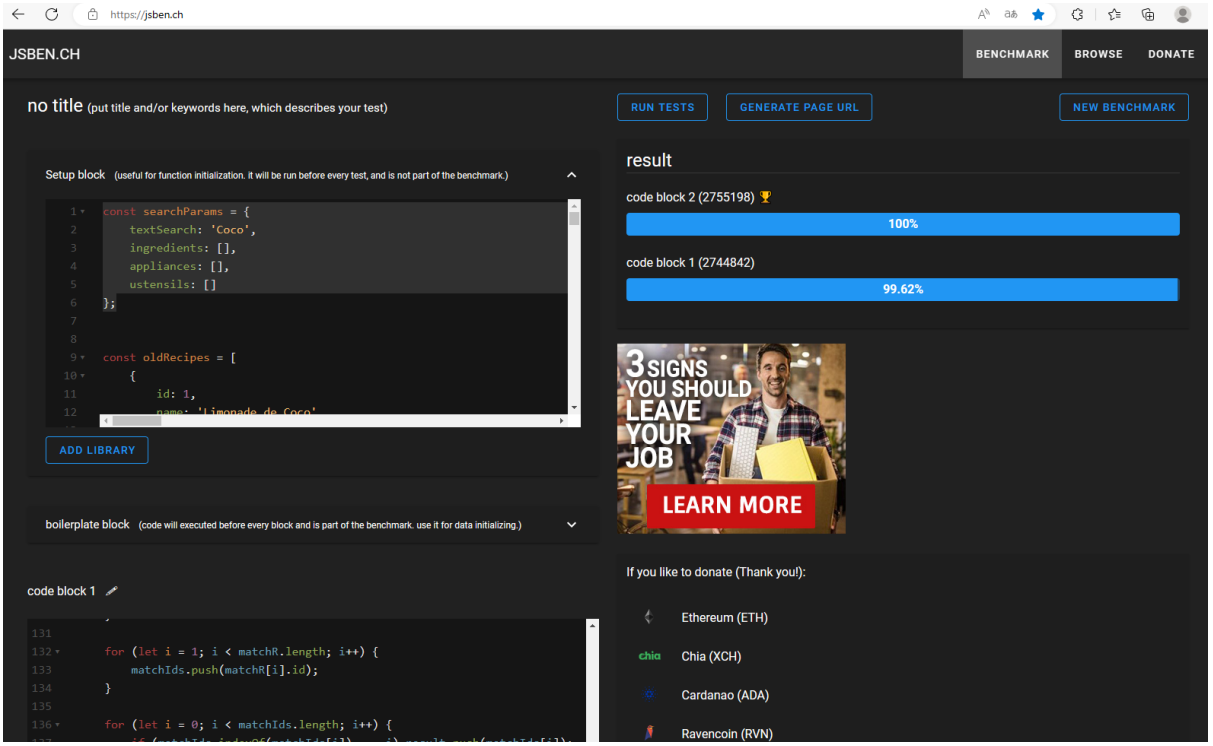
cas 1: pas de saisie, 1 tag sélectionné
Les 2 algorithmes étant identiques pour la recherche par tag, ils ne présentent pas de différence de performance lorsqu'on recherche par tag.

cas 2: 1 mot clé présent dans le titre, les ingrédients
la recherche fonctionnelle est plus rapide

cas 3: 1 mot clé présent dans le titre, les ingrédients, la description
la recherche fonctionnelle est plus rapide
ex: on saisit `Coco` dans la zone de texte et on effectue une recherche avec les deux algorithmes:

```
const searchParams = {
    textSearch: 'Coco',
    ingredients: [],
```

```
    appliances: [],
    ustensils: []
};
```



Remarque:
on remarque que la recherche fonctionnelle qui est dans le block 2 quand elle est terminée (à 100%), la recherche impérative qui est dans le block 1 est non terminée (99.62%)
d'ou une différence de 0.38%

cas 4: 1 mot clé présent dans le titre
la recherche **fonctionnelle est plus rapide**

cas 5: 1 mot clé présent dans les ingrédients
la recherche **fonctionnelle est plus rapide**

cas 6: 1 mot clé présent dans la description
la recherche **fonctionnelle est plus rapide**

cas 7: 1 mot clé présent dans les ingrédients, la description
la recherche **fonctionnelle est plus rapide**

cas 8: 1 mot clé présent dans le titre, les ingrédients
la recherche **fonctionnelle est plus rapide**

V- conclusion

La recherche fonctionnelle est plus rapide que la recherche car elle effectue plus d'opérations. En plus, la recherche fonctionnelle est plus lisible que la recherche impérative.