



Les petits plats

## Algorithme de recherche en JavaScript

**Compétences :**

- Analyser un problème informatique.
- Développer un algorithme pour un problème.

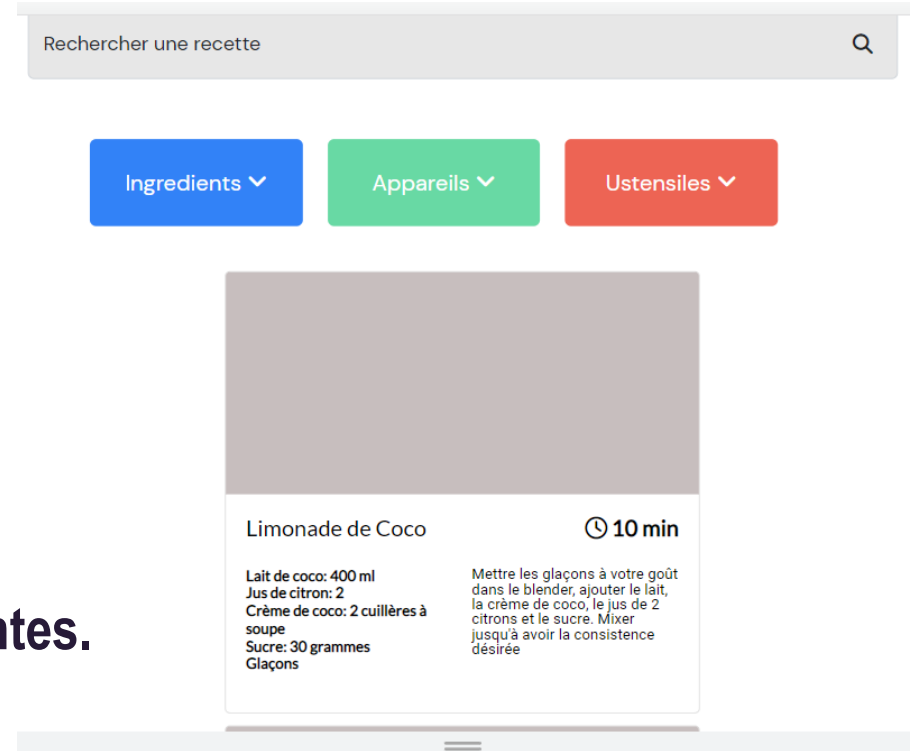
# Sommaire

- Contexte du projet Front-end p3
- Fiche d'investigation de fonctionnalité – Diagramme p4,5,6
- Versions p7
- Structure du projet p 8
- Méthodes Search.js 9
- Méthode de recherche des recettes p 10
- View.js et simulateKeyUp() 11, 12
- Html <a>13
- Html <i>, createTagButton()14
- QuerySelector() 15
- Structure index.js 16
- Code et conclusion 17,
- Test 18
- Comparaison des méthodes de boucles natives et de programmation fonctionnelle 19
- Exemple avec texte et sans tag 20
- Exemple avec texte et tag 21

# Contexte du projet Front-end - Mission de 3 mois

## Création site de recettes de cuisine :

- Fluidité du moteur de recherche.
- Développer d'une manière optimale.
- Création de deux implémentations différentes.
- Comparaison des performances.



[illegible]

<b>Fonctionnalité</b> : Recherche des recettes	<b>Fonctionnalité #99</b>
<b>Problématique</b> : Accéder rapidement à une recette correspondant à un besoin de l'utilisateur dans les recettes déjà reçues	

Dans cette option, nous allons parcourir les tableaux uniquement avec des boucles native comme for et while

- ⊗ Fonction native
- ⊗ Code plus facile à lire
- ⊗ Plus facile pour les débutants

- Moins rapide de 5%
- Plus de code à écrire

Dans cette option, nous allons parcourir les tableaux avec les méthodes de l'objet array

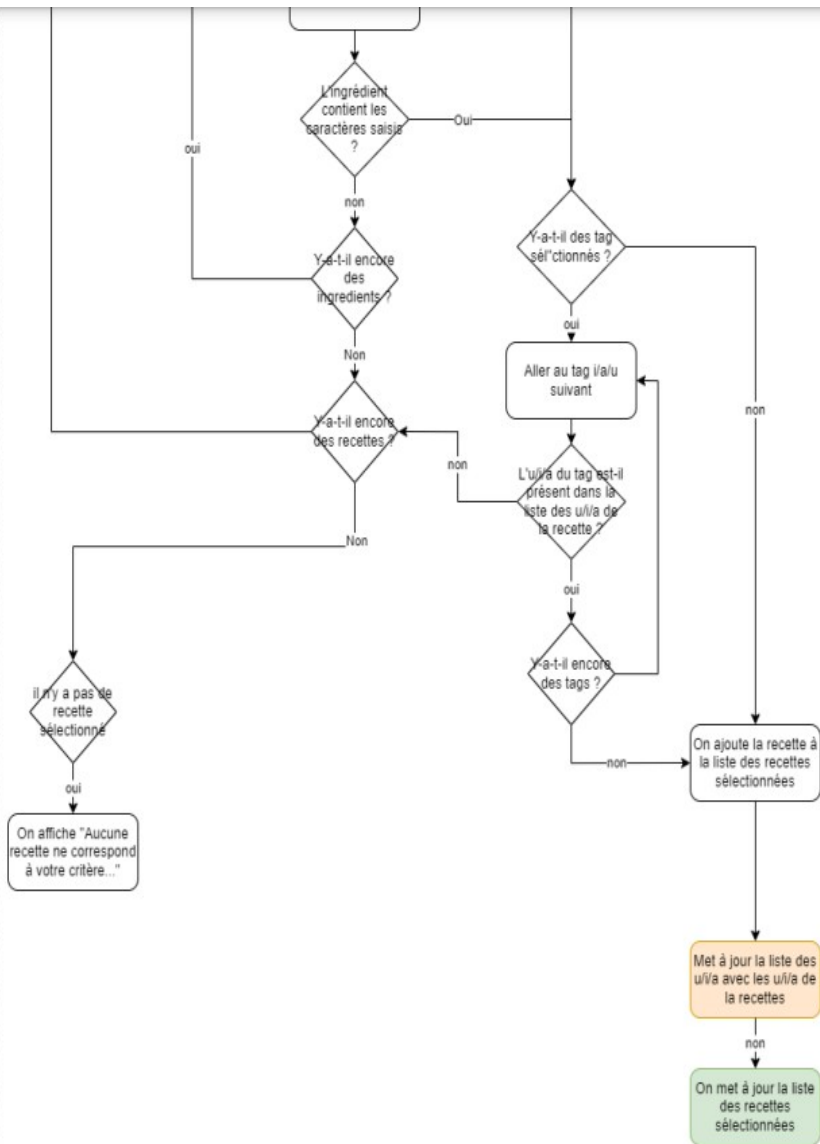
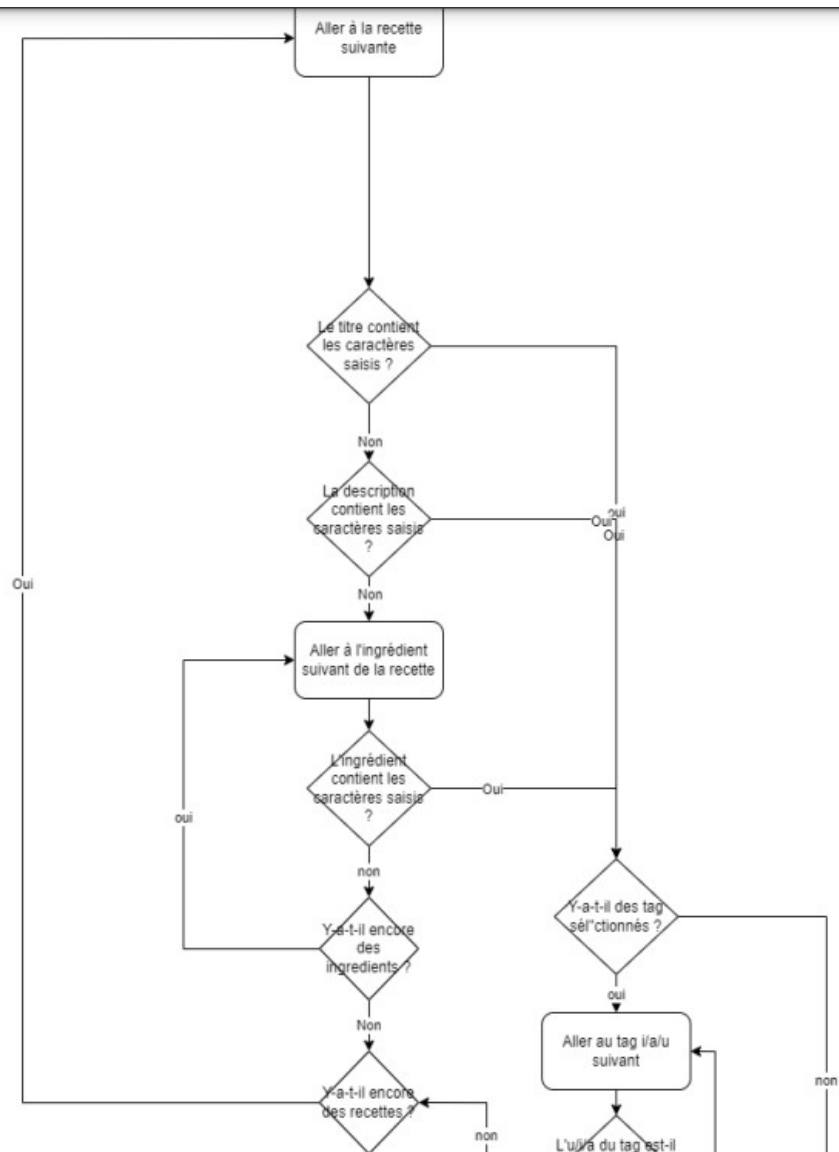
- Meilleures performances
- Moins de code à écrire

- Code plus complexe à lire
- Besoin de bien connaître les nouvelles méthodes `filters`, `forevery` (pouvoir s'arrêter dans le parcours) et `foreach`

On retient l'option 1 car malgré la complexité, les performances sont meilleures <https://isben.ch/ga1n>.

# Accéder à une recette correspondante





**Version en programmation fonctionnelle avec les méthodes de l'objet array (foreach, filter, map, reduce).**

**Version utilisant les boucles natives (while, for...).**

**Document d'investigation de fonctionnalité pour décrire les deux implémentations à comparer.**

# Prendre en main les éléments

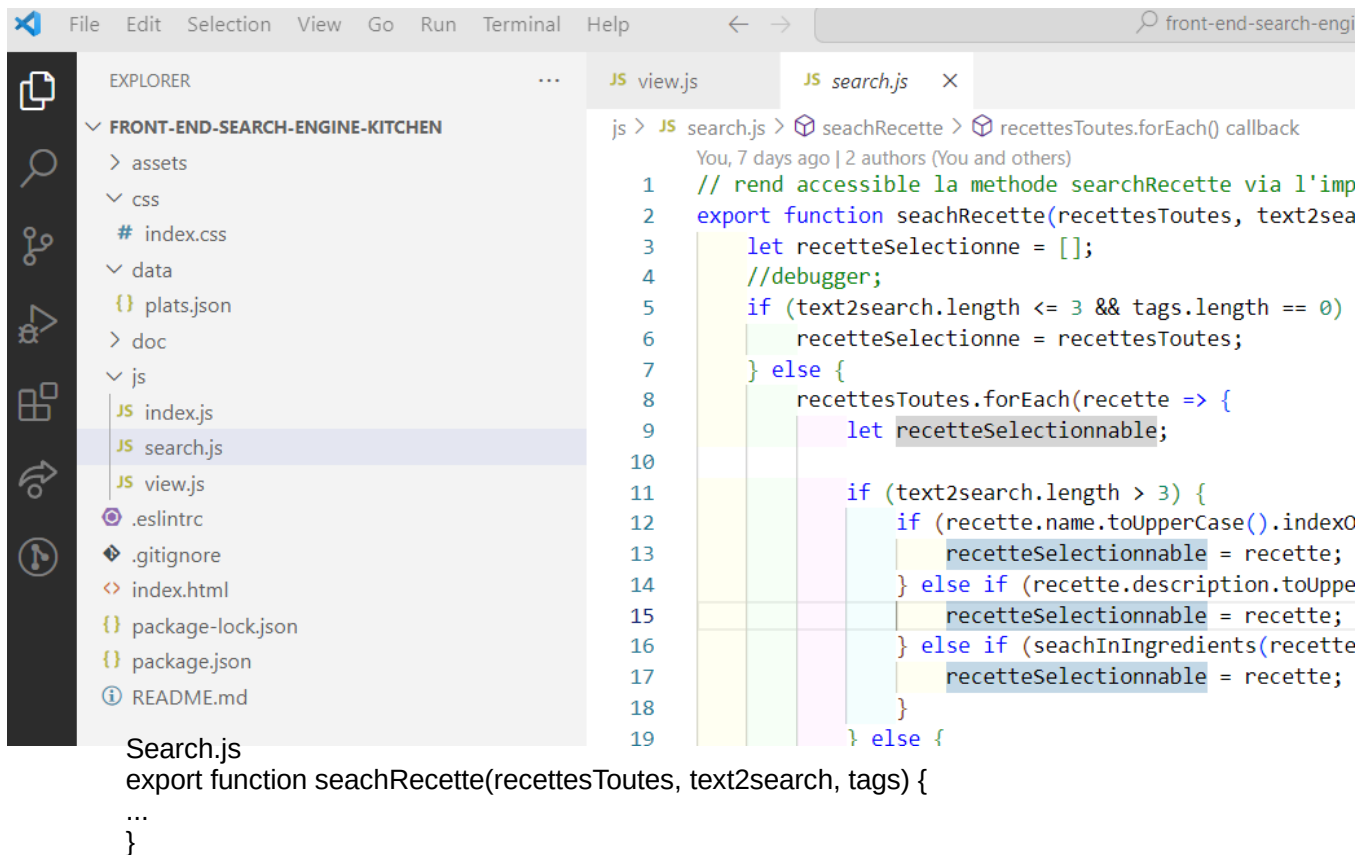
## Script/Index.html

**Index.js** → script principal de la page index.html

## Script

**Search.js** → module d'algo de recherche

**View.js** → module de création d'affichage dynamique, notamment transforme le json en html



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar displays the project structure for 'FRONT-END-SEARCH-ENGINE-KITCHEN'. The files listed are: assets, css (index.css), data (plats.json), doc, js (index.js, search.js, view.js), .eslintrc, .gitignore, index.html, package-lock.json, package.json, and README.md. The 'search.js' file is selected. The main editor area shows the code for 'search.js'. The code defines a function 'seachRecette' (note the typo) that takes 'recettesToutes' and 'text2search' as arguments. It uses 'recettesToutes.forEach()' to iterate over the recipes. Inside the loop, it checks if the search text is empty or if the recipe name, description, or ingredients match the search text. If a match is found, it adds the recipe to 'recetteSelectionnable'. Finally, it logs the selected recipes to the console and returns the array.

```
js > JS search.js > seachRecette > recettesToutes.forEach() callback
You, 7 days ago | 2 authors (You and others)
1 // rend accessible la methode searchRecette via l'imp
2 export function seachRecette(recettesToutes, text2sea
3     let recetteSelectionne = [];
4     //debugger;
5     if (text2search.length <= 3 && tags.length == 0)
6         recetteSelectionne = recettesToutes;
7     } else {
8         recettesToutes.forEach(recette => {
9             let recetteSelectionnable;
10
11             if (text2search.length > 3) {
12                 if (recette.name.toUpperCase().index0
13                     recetteSelectionnable = recette;
14                 } else if (recette.description.toUppe
15                     recetteSelectionnable = recette;
16                 } else if (seachInIngredients(recette
17                     recetteSelectionnable = recette;
18             }
19         } else {
20             console.log("voilà", recetteSelectionne.length);
21             return recetteSelectionne;
22         }
23     }
24 }
```

```
console.log("voilà", recetteSelectionne.length);
return recetteSelectionne;
```



# Méthodes Seach.js

- Foreach : parcourt les éléments d'un tableau = For
- Every : foreach que l'on peut interrompre en faisant un return (on peut ne pas parcourir tous les éléments du tableau) (ex : dans un tableau, on veut trouver le premier élément qui a une certaine valeur)
- Filter : à partir d'un tableau, donne un sous ensemble qui répond à un critère. (ex : tableau de chiffre et de nombre, on ne veut que les nombres)
- Push : ajout un élément dans le tableau

# Méthode pour rechercher des recettes

## export function searchRecette(recettesToutes, text2search, tags) {

La fonction **searchRecette** prend trois paramètres : **recettesToutes** (qui représente toutes les recettes disponibles), **text2search** (le texte de recherche) et **tags** (les tags sélectionnés). Elle renvoie une liste de recettes correspondant aux critères de recherche.

// Vérifier si la longueur du texte de recherche est inférieure à 3 et s'il n'y a pas de tags sélectionnés, si **non**, toutes les recettes sont sélectionnées et assignées à **recetteSelectionne** ; si **oui** méthode parcourt chaque recette dans **recettesToutes** à l'aide d'une boucle **forEach**. Pour chaque recette, une variable **recetteSelectionnable** est initialisée.

La variable **recetteSelectionne** est initialisée comme une liste vide pour stocker les recettes sélectionnées.

// Vérifier si le texte de recherche correspond au nom de la recette, à la description ou aux ingrédients. Si la longueur du texte de recherche est **supérieure** à 2, la méthode **toUpperCase().indexOf(text2search)** est utilisée pour rechercher une correspondance dans chaque cas. Si une correspondance est trouvée, la recette est assignée à **recetteSelectionnable**.

// Vérifier les tags sélectionnés

// Ajouter des conditions pour d'autres **types** de tags si nécessaire

Si la longueur des tags sélectionnés est supérieure à 0 et **recetteSelectionnable** n'est pas nul, la méthode parcourt chaque tag à l'aide de **every** pour effectuer une vérification spécifique au type de tag.

Dans l'exemple donné, seul le type de tag "ustensiles" est pris en compte. Si la recette ne contient pas l'ustensile spécifié dans le tag, **recetteSelectionnable** est assigné à nul et la **boucle** est interrompue.

// Ajouter la recette à la sélection

Enfin, si **recetteSelectionnable** est non nul, la recette est ajoutée à la liste **recetteSelectionne**.

// Méthode pour rechercher dans les ingrédients d'une recette

Une fois toutes les recettes parcourues, la méthode renvoie la liste **recetteSelectionne** qui contient les recettes correspondantes aux critères de recherche.

La **méthode searchInIngredients** est une fonction auxiliaire utilisée pour rechercher une correspondance dans les ingrédients d'une recette. Elle parcourt chaque ingrédient de la recette pour vérifier si le texte de recherche correspond à un ingrédient. Si au moins un ingrédient correspond, la méthode renvoie **true**.

// Rendre la méthode searchRecette accessible via l'import

Enfin, la méthode **searchRecette** est exportée par défaut, ce qui permet de l'importer et de l'utiliser dans d'autres parties de code.

# View.js

Manipulation de tags et à la gestion de l'affichage de cartes de recettes :

1. La fonction ``cleanUpTags`` prend deux paramètres : ``tags`` et ``selectedTags``. Elle filtre les tags en supprimant les doublons, les éléments vides et les tags qui ont été trouvés dans la liste des tags sélectionnés pour la recherche. Elle renvoie les tags filtrés triés en utilisant l'objet ``Intl.Collator`` pour comparer les chaînes de caractères en utilisant les règles de tri spécifiques à la langue française.
2. La fonction ``simuleKeyUp`` simule un événement "keyup" sur l'élément d'ID "search". Cela peut être utilisé pour déclencher une recherche lorsque la touche "Enter" est pressée dans le champ de recherche.
3. La fonction ``simuleKeyUpDropBox`` simule un événement "keyup" sur tous les éléments d'entrée des dropbox. Cela peut être utilisé pour rafraîchir la recherche des dropbox après une nouvelle recherche.
4. La fonction ``displayOptions`` prend deux paramètres : ``type`` et ``tab``. Elle recherche l'élément de la page qui contient les "options" correspondantes au type donné, puis elle vide cet élément. Ensuite, pour chaque élément dans le tableau ``tab``, elle crée une balise ``a`` avec un lien et un gestionnaire d'événement de clic. Cette fonction est utilisée pour afficher les options de tags dans les dropdowns.
5. La fonction ``createTagButton`` crée un bouton de tag. Elle prend deux paramètres : ``type`` et ``tag``. Elle crée un bouton avec la classe correspondant au type donné, et elle ajoute un gestionnaire d'événement de clic à l'icône "croix" qui supprime le tag lorsque l'utilisateur clique dessus. Cette fonction est utilisée pour créer les boutons de tags dans la section des tags.
6. La fonction ``getSelectedTags`` récupère tous les tags créés et retourne leurs types et leurs valeurs dans un tableau d'objets. Elle utilise ``querySelectorAll`` pour récupérer tous les boutons de tags, puis elle itère sur ces boutons pour extraire le texte affiché sur le bouton (en minuscules et sans espaces en trop) et la classe du bouton (en enlevant la chaîne "btn" et en supprimant les espaces en trop). Cette fonction est utilisée pour obtenir les tags sélectionnés lors de la recherche.
7. La fonction ``displayCards`` affiche les cartes de recettes. Elle prend deux paramètres : ``recettes`` (une liste de recettes) et ``tags`` (une liste de tags). Cette fonction commence par vider la section des recettes. Ensuite, elle itère sur chaque recette et crée le HTML pour la carte de recette en utilisant les informations de la recette. Elle ajoute également les ingrédients de chaque recette aux tableaux correspondants (``ingredients``, ``appareils`` et ``ustensiles``). Enfin, elle appelle ``cleanUpTags`` pour supprimer les doublons des tags d'ingrédients, d'appareils et d'ustensiles, puis elle appelle ``displayOptions`` pour afficher les options de tags filtrées dans les dropdowns, et enfin, elle appelle ``simuleKeyUpDropBox`` pour rafraîchir la recherche dans les dropdowns.

Pour filtrer et afficher des recettes en fonction des tags sélectionnés par l'utilisateur, ainsi que pour gérer les interactions avec les dropdowns et les boutons de tags.

# View.js - fonction `simuleKeyUp()`

Gestion des interactions utilisateur et des événements sur les éléments du **DOM** : clics de souris et les frappes de clavier permettent d'ajouter des fonctionnalités interactives et réactives à l'application.

Le `addEventListener()` ajoute des gestionnaires d'événements de clic à certains éléments `<a>` et `<i>`. Ils sont déclenchés lorsque l'utilisateur clique sur ces éléments.

`click` : dans `displayOptions()`, type d'événement pour les éléments `<a>` et `<i>`, pour cacher des éléments, créer des tags, simuler des saisies de recherche.

`keyup` : type d'événement qui se déclenche lorsque l'utilisateur relâche une touche du clavier, touche "Enter".

`simuleKeyUp()` sélectionne l'élément avec l'ID "search" dans le document HTML :

`document.querySelector("#search").`

Puis simule un événement de frappe de touche "Enter" en utilisant `dispatchEvent()` sur l'élément sélectionné pour déclencher `keyup`. Cette fonction peut être utilisée pour déclencher manuellement un événement de recherche ou une autre action qui est normalement déclenchée par une frappe "Enter" dans le champ de recherche

# Création html <a>

En utilisant ces techniques, le code dans le fichier génère du contenu HTML dynamiquement en fonction des données fournies et des interactions de l'utilisateur. Il crée des boutons de tags, des cartes de recettes et des éléments de dropdown, puis les insère dans le document HTML pour les afficher à l'écran. Cela permet de mettre à jour dynamiquement l'interface utilisateur en fonction des actions de l'utilisateur et des données disponibles.

1. Utilisation de `document.createElement` : Cette méthode permet de créer un nouvel élément HTML. Par exemple, dans la fonction `createTagButton`, on utilise `document.createElement("button")` pour créer un élément `<button>`.
2. Attribution d'attributs : Une fois l'élément créé, on peut lui attribuer des attributs en utilisant les méthodes `setAttribute` et les propriétés spécifiques de l'élément. Dans `createTagButton`, on utilise `btn.setAttribute("class", "btn " + type)` pour définir la classe du bouton, et `a.setAttribute("href", "#" + el)` pour définir l'attribut `href` de l'élément `<a>`.
3. Modification du contenu textuel : propriété `textContent`. Dans `displayOptions`, on utilise `a.textContent = el.charAt(0).toUpperCase() + el.slice(1)` pour mettre la première lettre du texte en majuscule.
4. Insertion d'éléments dans le **DOM** : Une fois que l'élément est créé et configuré, on peut l'insérer dans le document en utilisant des méthodes telles que `appendChild`. Dans `displayOptions`, on utilise `divOptions.appendChild(a)` pour insérer chaque élément `<a>` dans la div `divOptions`.

## Création html `<i>` createTagButton()

Balise `<i>` est utilisée pour représenter une **icône** et associée à des **événement clic** pour interagir avec l'utilisateur et de déclencher des actions spécifiques en réponse à ces interactions, utilisée avec un événement, comme `addEventListener` ou `dispatchEvent`.

Dans la fonction `createTagButton()`, le gestionnaire d'événements de clic associé à l'icône `<i>` permet de **supprimer** un élément du **DOM** lorsque l'utilisateur clique sur cette icône.

Dans la fonction `createTagButton()`, lorsque l'utilisateur clique sur l'icône `<i>`, le gestionnaire d'événements de clic est déclenché, ce qui entraîne la suppression de l'élément parent de l'icône `<i>` dans le DOM.

**Méthodes de sélection du DOM pour cibler et manipuler des éléments spécifiques du document HTML en utilisant leurs sélecteurs CSS, leurs ID ou leurs balises. Cela permet de trouver et d'interagir avec les éléments nécessaires pour ajouter des fonctionnalités ou effectuer des modifications sur la page :**

1. `querySelector` est une méthode du **DOM (Document Object Model)** en JavaScript qui permet de sélectionner un élément du document HTML en utilisant un sélecteur **CSS**. Elle renvoie le premier élément correspondant au sélecteur spécifié : `document.querySelector("#search")` est utilisé pour sélectionner l'élément avec l'ID "search" dans le document HTML.
2. `querySelectorAll` ... sous forme de `NodeList` (une collection d'éléments). Cela permet de sélectionner plusieurs éléments en une seule fois : `document.querySelectorAll(".dropdown-content input")` est utilisé pour sélectionner tous les éléments `<input>` situés dans les éléments avec la classe "dropdown-content".
3. `getElementById` permet de sélectionner un élément en utilisant son attribut **id**.
4. `getElementsByTagName` permet de sélectionner tous les éléments d'un document avec un nom de balise spécifique. Elle renvoie une collection d'éléments correspondant à la balise spécifiée : `div.getElementsByTagName("input")` renvoie tous les éléments `<input>` situés à l'intérieur de l'élément **`<div>`**.

# Structure du code – index.js

Index.js c'est une implémentation d'une fonction de recherche et d'affichage de recettes basée sur des critères de recherche et des tags sélectionnés.

Fonctionnement :

La fonction ``searchDisplay()`` gère recherche et affichage des recettes, étapes :

- Récupère la valeur du champ de recherche à partir de l'élément HTML avec l'ID "search".
- Obtient les tags sélectionnés à l'aide de la fonction ``getSelectedTags()`` du module ``View``.
- Appelle la fonction ``seachRecette()`` du module ``Search`` pour rechercher les recettes correspondant aux critères spécifiés (texte de recherche et tags).
- Affiche les recettes sélectionnées à l'aide de la fonction ``displayCards`` du module ``View``.

La fonction ``getData()`` recupe données des recettes à partir d'un fichier JSON. Elle effectue les étapes suivantes :

- Initialise une variable ``recettesToutes`` pour stocker toutes les recettes.
- Effectue une requête HTTP avec ``fetch`` pour récupérer le fichier JSON contenant les recettes.
- Convertit la réponse en JSON avec ``response.json()`` et assigne les données à ``recettesToutes``.
- Ajoute un écouteur d'événements "keyup" sur le champ de recherche pour appeler ``searchDisplay`` à chaque fois que l'utilisateur tape une touche.
- Appelle ``searchDisplay`` une première fois pour afficher toutes les recettes au chargement initial de la page.

4. La fonction ``getData`` est appelée pour lancer le processus de récupération des données et d'affichage des recettes.



# Code - Conclusion

**Utilisation de manière efficace améliore la lisibilité, la maintenance et les performances :**

**Fonctionnalités modernes :**

Fonctions fléchées et les méthodes de tableau, pour réduire la complexité du code en utilisant des expressions plus courtes et plus expressives.

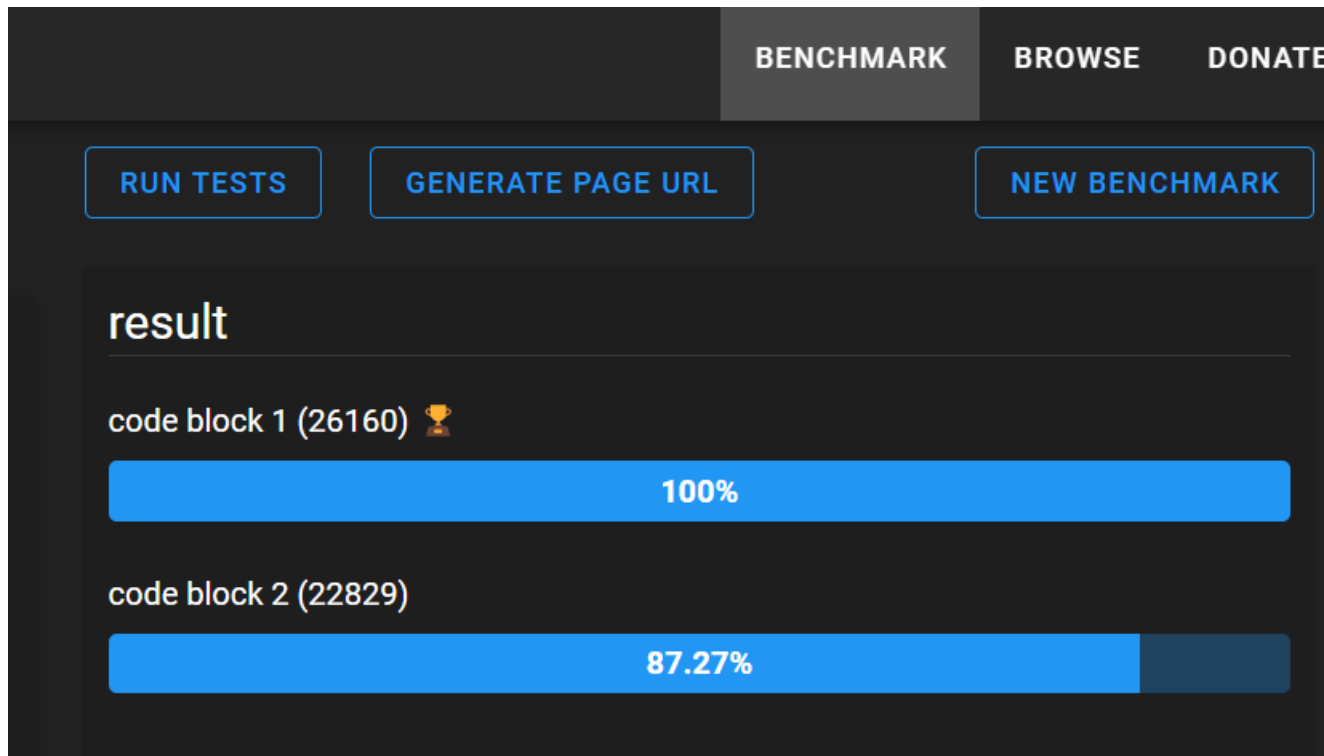
**Conventions du langage** pour accomplir les tâches de manière écrite de manière laconique, clair et précise.

**Avantages :**

1. **Lisibilité** : Facile à lire et à comprendre, réduit la quantité de code redondant ou superflu, pour se concentrer sur la logique métier essentielle.
2. **Maintenance** : Facile à maintenir car il est plus clair et moins sujet aux erreurs. Les modifications et les mises à jour peuvent être effectuées plus rapidement et avec moins de risques de bugs introduits par des changements inutiles.
3. **Performance** : performant car il réduit le temps d'exécution en évitant les boucles inutiles ou les opérations.

# Tests <https://jsben.ch/qa1In>

Retenir l'option 1 – fonctionnelle. Malgré la complexité, les performances sont meilleures.



## Search.js : Option 2 : boucles natives - for

### Option 1 : programmation fonctionnelle - méthodes de l'objet array

Fichier 1 search.js est considéré comme plus concis et expressif car il utilise des méthodes de tableau modernes, des fonctions fléchées et des noms de variables significatifs. Utilise la méthode `every` sur le tableau `tags` pour parcourir les tags et effectuer des vérifications spécifiques en fonction du type de tag.

Fonction `searchInIngredients` :

Utilise une fonction fléchée directement à l'intérieur de `forEach` pour rechercher dans les ingrédients d'une recette.

Définit une fonction séparée `searchInIngredients` pour rechercher dans les ingrédients d'une recette.

Utilisation de **méthodes de tableau** : Le fichier 1 utilise des méthodes telles que `forEach`, `filter` et `every` pour parcourir, filtrer et vérifier les éléments d'un tableau de manière plus déclarative.

Cela rend le code plus facile la lecture et la compréhension de l'intention du code. Moins de code répétitif et non spécifique, inclus dans de nombreux endroits sans y apporter de réelle valeur ou **logique métier**. Moins de code "poids mort" qui n'apporte pas de fonctionnalité spécifique, mais qui est nécessaire pour mettre en place l'infrastructure et les dépendances requises. Il s'agit de code générique qui doit être inclus pour des raisons de structure, de compatibilité ou de conventions.

**Fonctions fléchées** : Le fichier 1 utilise des fonctions fléchées pour définir des fonctions de rappel de bonne manière (réduit mauvais syntaxe).

## "coc" dans le champs de recherche et a sélectionné le tag « Bol »

```
const recettesToutes = [
  {
    name: "Salade de fruits",
    description: "Une délicieuse salade de fruits",
    ingredients: [
      { ingredient: "Pomme" },
      { ingredient: "Banane" },
      { ingredient: "Orange" },
    ],
    ustensils: ["Bol", "Couteau"],
  },
  {
```

Longueur du texte de recherche "coc" est supérieure à 2 caractères, donc nous procédons à la vérification de chaque recette (json), excluant les autres.

Le tag sélectionné est "Bol". Itération sur chaque recette restante, qui est actuellement vide car **aucune recette** n'a passé la vérification précédente.

La méthode se termine et renvoie recetteSelectionne qui est un **tableau vide**.

```
const text2search = "coc";
const tags = [{ type: "ustensiles", name: "Bol" }];

const recetteSelectionne = searchRecette(recettesToutes, text2search, tags);
console.log(recetteSelectionne);
```

# "coc" dans le champs de recherche et a sélectionné le tag « Bol »

Longueur du **texte** de recherche  
**"coc"** est supérieure à 2 caractères  
dans recette, donc nous procédons à  
la vérification de chaque recette  
(json), excluant les autres.

Le **tag** sélectionné est **"Bol"**.  
recette ← recette est sélectionnable ;  
**Type** :  
On parcourt les tag (1='Bol'), ce tag  
c'est 1 '**ustensile**' ; **Type** :  
On vérifie si 'Bol' est dedans (>0) :  
Recette est sélectionnable → recette  
sélectionnées.  
Si faut on enlève la **recette**  
**sélectionnable** (=null) ;  
Pas de recette, pas d'ajout aux  
**recette sélectionnées** ;

