**Fiche d'investigation fonctionnalité**

**Fonctionnalité :** Recherche principale - Site web "Les Petits Plats"

**Problématique**

Afin d'offrir une expérience utilisateur optimale, nous cherchons à implémenter une fonctionnalité de recherche efficace et performante pour le site "Les Petits Plats".

**Option 1 : Recherche par** **boucles natives JavaScript**

Dans cette option, nous utilisons des boucles JavaScript (for, while) pour parcourir le tableau de recettes et effectuer la recherche.

**Avantages**

- ⊕ Contrôle précis sur l'itération des données

- ⊕ Possibilité d'arrêter la boucle dès qu'une correspondance est trouvée (potentiellement plus rapide pour de petits ensembles de données)

- ⊕ Facilité de compréhension pour les développeurs débutants

I**nconvénients**

- ⊖ Peut être moins performant pour de grandes quantités de données

- ⊖ Code potentiellement plus verbeux et moins lisible

- ⊖ Risque plus élevé d'erreurs de logique dans les boucles imbriquées

**Algorithme**

```javascript

function searchRecipes(query, recipes) {

const results = [];

for (let i = 0; i < recipes.length; i++) {

const recipe = recipes[i];

if (

recipe.name.toLowerCase().includes(query.toLowerCase()) ||

recipe.description.toLowerCase().includes(query.toLowerCase()) ||

recipe.ingredients.some(ing => ing.ingredient.toLowerCase().includes(query.toLowerCase()))

) {

results.push(recipe);

}

}

return results;

}

```

**Complexité algorithmique (Big O)**

- Temps : O(n \* m), où n est le nombre de recettes et m est le nombre moyen d'ingrédients par recette.

- Espace : O(k), où k est le nombre de recettes correspondant aux critères de recherche.

Explication : Dans le pire des cas, nous devons parcourir toutes les recettes (n) et pour chaque recette, nous pourrions avoir à parcourir tous ses ingrédients (m). La méthode `includes()` a une complexité de O(p) où p est la longueur de la chaîne, mais nous pouvons la considérer comme constante pour cette analyse.

**Option 2 : Recherche utilisant les méthodes d'array (filter)**

Cette option utilise les méthodes d'array modernes de JavaScript, notamment `filter`, pour effectuer la recherche.

**Avantages**

- ⊕ Code plus concis et lisible

- ⊕ Approche fonctionnelle qui réduit les effets de bord

- ⊕ Potentiellement plus rapide pour de grandes quantités de données (optimisations internes du navigateur)

- ⊕ Facilité d'ajout de critères de recherche supplémentaires

**Inconvénients**

- ⊖ Peut être moins intuitif pour les développeurs débutants

- ⊖ Parcourt toujours l'ensemble des données (ne peut pas s'arrêter prématurément)

- ⊖ Potentiellement moins performant pour de très petits ensembles de données

**Algorithme**

```javascript

function searchRecipes(query, recipes) {

return recipes.filter(recipe =>

recipe.name.toLowerCase().includes(query.toLowerCase()) ||

recipe.description.toLowerCase().includes(query.toLowerCase()) ||

recipe.ingredients.some(ing => ing.ingredient.toLowerCase().includes(query.toLowerCase()))

);

}

```

**Complexité algorithmique (Big O)**

- Temps : O(n \* m), où n est le nombre de recettes et m est le nombre moyen d'ingrédients par recette.

- Espace : O(k), où k est le nombre de recettes correspondant aux critères de recherche.

Explication : Bien que nous utilisions des méthodes d'array plus modernes, la complexité temporelle reste la même que pour l'approche avec des boucles natives. Nous devons toujours parcourir toutes les recettes (n) et pour chaque recette, nous examinons potentiellement tous ses ingrédients (m). La méthode `filter()` crée un nouveau tableau, ce qui explique la complexité spatiale.

**Solution retenue**

Nous avons retenu l'approche utilisant les méthodes d'array (Option 2) pour plusieurs raisons :

1. Le code est plus lisible et maintenable, ce qui facilitera les futures évolutions de la fonctionnalité.

2. Cette approche est généralement plus performante pour de grandes quantités de données, ce qui est crucial pour un site de recettes qui pourrait potentiellement contenir des milliers d'entrées.

3. L'utilisation de méthodes d'array comme `filter` et `some` rend le code plus déclaratif, permettant de comprendre rapidement l'intention de la recherche.

4. Cette méthode s'adapte plus facilement à l'ajout de nouveaux critères de recherche ou à la modification des critères existants.

Bien que la complexité algorithmique soit identique pour les deux approches, les optimisations internes des navigateurs pour ces méthodes d'array peuvent offrir de meilleures performances dans la pratique, en particulier pour de grands ensembles de données.

**Considérations supplémentaires sur la complexité**

Bien que les deux approches aient la même complexité théorique, il est important de noter que :

1. Les implémentations des méthodes d'array comme `filter` et `some` sont souvent optimisées par les moteurs JavaScript des navigateurs, ce qui peut conduire à de meilleures performances dans la pratique.

2. Pour de très petits ensembles de données, la méthode utilisant des boucles natives pourrait être légèrement plus rapide en raison de la possibilité d'arrêter prématurément la recherche.

3. Pour améliorer davantage les performances, surtout pour de grands ensembles de données, on pourrait envisager des structures de données plus avancées (comme des index inversés) ou des algorithmes de recherche plus sophistiqués, qui pourraient potentiellement réduire la complexité à O(log n) ou mieux.

**Annexes**

Ces organigrammes illustrent visuellement le processus de chaque méthode de recherche. Voici une brève explication des différences clés :

1. Méthode par boucles natives :

   - Utilise une boucle explicite pour parcourir les recettes.

   - Crée un nouveau tableau de résultats et y ajoute les recettes correspondantes.

   - Permet potentiellement d'arrêter la recherche prématurément si tous les résultats souhaités sont trouvés.

2. Méthode par méthodes d'array :

   - Utilise la méthode `filter` pour créer un nouveau tableau de résultats.

   - Applique la méthode `some` sur le tableau d'ingrédients.

   - Le processus est plus déclaratif et les boucles sont implicites.

Ces organigrammes mettent en évidence que, bien que le flux général soit similaire, la méthode utilisant les fonctions d'array encapsule certaines étapes (comme la création du tableau de résultats) dans les méthodes elles-mêmes, ce qui rend le code plus concis et potentiellement plus facile à lire et à maintenir.

**Annexe 1 : Organigramme de la méthode de recherche par boucles natives**

graph TD

    A[Début] --> B[Initialiser tableau de résultats vide]

    B --> C[Parcourir chaque recette]

    C --> D{Nom de la recette contient la requête?}

    D -->|Oui| E[Ajouter la recette aux résultats]

    D -->|Non| F{Description contient la requête?}

    F -->|Oui| E

    F -->|Non| G[Parcourir les ingrédients de la recette]

    G --> H{Ingrédient contient la requête?}

    H -->|Oui| E

    H -->|Non| I{Tous les ingrédients vérifiés?}

    I -->|Non| G

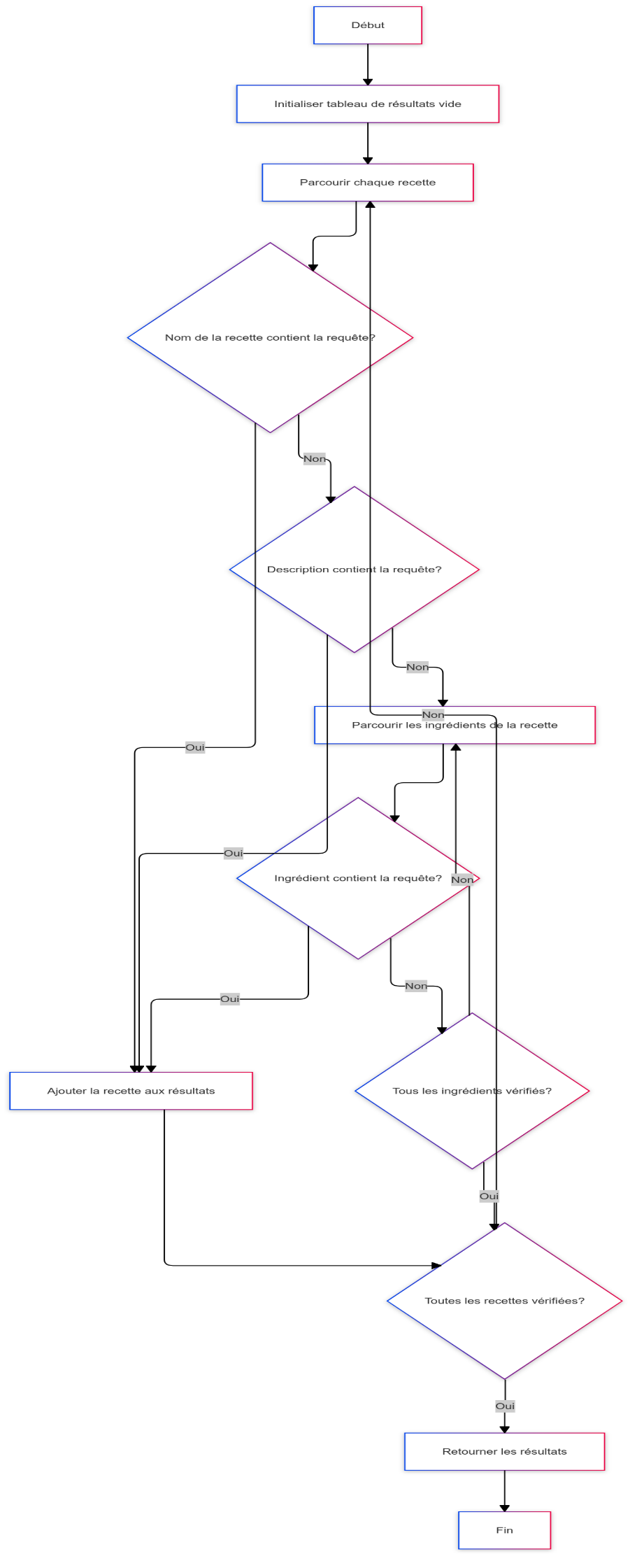
    I -->|Oui| J{Toutes les recettes vérifiées?}

    E --> J

    J -->|Non| C

    J -->|Oui| K[Retourner les résultats]

    K --> L[Fin]`



**Annexe 2 : Organigramme de la méthode de recherche par méthodes d'array**

graph TD

    A[Début] --> B[Appliquer filter sur le tableau de recettes]

    B --> C{Pour chaque recette}

    C --> D{Nom de la recette contient la requête?}

    D -->|Oui| E[Inclure dans les résultats]

    D -->|Non| F{Description contient la requête?}

    F -->|Oui| E

    F -->|Non| G[Appliquer some sur les ingrédients]

    G --> H{Un ingrédient contient la requête?}

    H -->|Oui| E

    H -->|Non| I[Exclure des résultats]

    I --> J{Toutes les recettes vérifiées?}

    E --> J

    J -->|Non| C

    J -->|Oui| K[Retourner le tableau filtré]

    K --> L[Fin]

```

