RECHERCHE RECETTE Fonctionnalité #1

Problématique : Afin de se démarquer des sites concurrents, les résultats d'une recherche doivent être performants en termes de rapidité et de fluidité pour retenir un maximum d'utilisateurs

ENJEU : Implémenter un algorithme de tri performant pour une comparaison lexicographique

OPTION 1 : Algorithme de tri .sort()

La méthode .sort() est disponible par défaut en Javascript.

Elle renvoie le tableau trié et modifie la position des éléments dans le tableau d'origine.

Les éléments sont convertis en chaînes de caractères, puis comparés et triés par ordre lexicographique.

La particularité de cette méthode générique est qu'elle utilise des méthodes de tri différentes selon le navigateur utlisé.

Avantages

- Compare par défaut des chaines de caractère selon les valeurs des unités de code UTF-16
- Aucune fonction à implémenter dans son usage standard
- Tri stable : conserve l'ordre d'apparition des éléments égaux

Inconvénients

- Le tri est effectué sur le tableau courant qui est modifié, aucune copie n'est réalisée
- Nécessité d'implémenter une fonction de comparaison pour trier des nombres *("80" arrive avant "9" selon l'ordre des unités de code UTF-16)*
- Utilise le tri par insertion avec Chrome et le tri par fusion avec Mozilla Firefox et Safari

Performances

Insertion sort

Complexité temporelle *(nb comparaisons nécessaires)*

- Pire cas *(tableau trié par ordre décroissant)* : $O(n^2)$
- Meilleur cas *(tableau déjà trié)* : O(n)

Complexité spatiale *(espace mémoire requis)*

O(1) *(car aucune structure de données supplémentaire)*

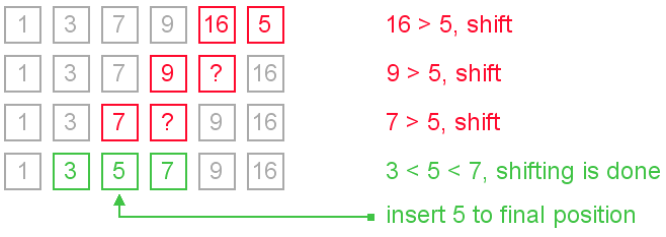


Figure 1. Illustration Insertion Sort

Merge sort

Complexité temporelle *(nb comparaisons nécessaires)*

Pire cas = Meilleur cas : O(n log n)
- division du tableau => O(log n)
- fonction récursive => O(n) pour chaque appel

Complexité spatiale *(espace mémoire requis)*

O(n) car la fonction récursive implique une pile interne pour stocker ses appels et donc un traitement des données de type **LIFO** (Last In First Out)

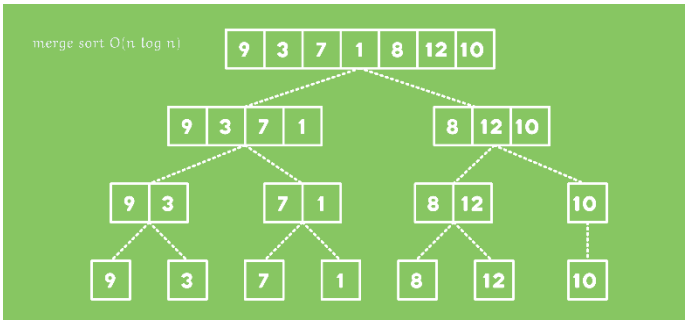


Figure 2. Illustration Merge Sort

OPTION 2 : Algorithme de tri Quick Sort

Le tri rapide (Quick Sort) consiste à diviser le tableau en 2 parties, séparées par l'élément pivot (qui peut être le premier, le dernier ou le médian). L'algorithme utilise deux pointeurs qui analysent séquentiellement le tableau, ce qui est optimal par rapport à la mise en cache.

Option 2.a : Schéma de partition de LOMUTO

Ce schéma de partition prend pour pivot le dernier élément du tableau. L'algorithme maintient l'index i pendant qu'il balaye le tableau en utilisant un autre index j, tel que les éléments avant i (inclus) sont inférieurs ou égaux au pivot, et les éléments i+1 à j-1 (inclus) sont supérieurs au pivot. En parcourant le tableau, si un élément est plus petit, il est échangé avec l'élément courant. Sinon, l'élément actuel est ignoré

Avantages	Inconvénients
⊕ Divise par 2 le nombre d'éléments à trier, ce qui est très efficace sur les grands ensembles de données.	⊖ L'élément pivot est le dernier du tableau (l'élément central comme pivot est la solution la plus optimale)
⊕ Utilise 2 pointeurs qui se déplacent l'un vers l'autre tout en comparant leur valeur respective, permuttée si besoin	⊖ Tri instable : si le tableau contient plusieurs 5, la conservation de leur ordre n'est pas garantie
⊕ Actualise continuellement les paires de valeurs jusqu'au pivot	⊖ Effectue le balayage de l'intégralité des données, quelle que soit leur valeur

Option 2.b : Schéma de partition de HOARE

2 indices qui commencent aux extrémités du tableau en cours de partition, se déplacent l'un vers l'autre, jusqu'à détecter une inversion : une paire d'éléments, l'un supérieur ou égal au pivot, l'autre inférieur ou égal au pivot, qui sont dans le mauvais ordre. Les éléments sont alors permutés. Lorsque les indices se rencontrent, l'algorithme s'arrête et renvoie l'indice final.

Avantages	Inconvénients
⊕ Effectue en moyenne trois fois moins d'échanges que la partition Lomuto	⊖ Le partitionnement de Hoare entraîne également la dégradation de Quicksort en $O(n^2)$ lorsque le tableau d'entrée est déjà trié
⊕ Créé des partitions efficaces même lorsque toutes les valeurs sont égales	⊖ Tri instable sur tableau : si le tableau contient plusieurs 5, la conservation de leur ordre n'est pas garantie
⊕ Sur un tableau déjà trié, aucune permutation n'est opérée	

Performances

Complexité temporelle *(nb comparaisons nécessaires)*

- Pire cas : $O(n^2)$

Si le tableau est trié par ordre décroissant ou que les éléments sont tous identiques, les sous-tableaux sont fortement déséquilibrés

- Meilleur cas : O(n log n)

division du tableau => O(log n)
fonction récursive => O(n) pour chaque appel

Complexité spatiale *(espace mémoire requis)* : O(n)

Car la fonction récursive implique une pile interne pour stocker ses appels et donc un traitement des données de type **LIFO** (Last In First Out)

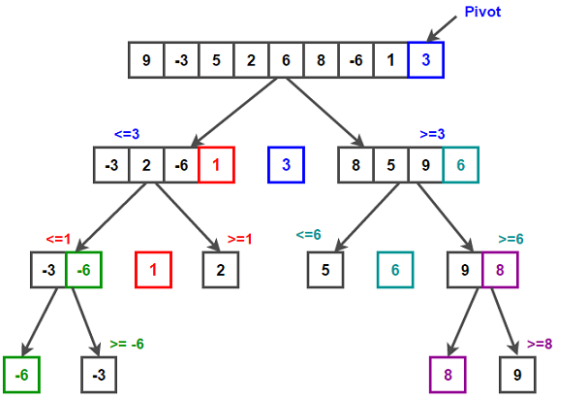


Figure 3. Illustration Quick Sort

RECHERCHE RECETTE **Fonctionnalité #1**

TEST PERFORMANCES

Navigateur utilisé : Chrome

Outil utilisé : Performance Analyser (extension Chrome)

Test 1 : Affichage de la page d'accueil

Test 2 : Affichage des recettes correspondantes d'après la recherche de l'item « chocolat »

Méthodes testées : Tri par défaut de JavaScript => tri par Insertion (insertion sort) / Tri rapide (quick sort) => partition de Hoare

Capture d'écran des tests de performance et tableau d'analyse des résultats : Pages suivantes

Indicateurs retenus :

- Total navigation timming
- DOM content loading
- Slowest call timming
- Average call timming
- Response timming
- Recipes.json timming

Solution retenue

Après tests, comparaison et analyse des résultats, l'algorithme de tri rapide (quick sort) s'avère plus performant que celui par insertion (insertion sort), et ce sur quasi l'intégralité des indicateurs retenus.

Dans le contexte précis du site Les Petits Plats, je retiens donc la solution tri rapide (quick sort)

REMARQUES

La performance totale du tri par insertion se situe autour de 300 ms, ce qui est très bon. Par ailleurs, le différentiel varie de 1 à 64 ms selon les indicateurs et le contexte testé, ce qui n'est pas perceptible à l'échelle humaine et donc peu significatif en termes d'expérience utilisateur. D'autre part, lorsque les collections comptent moins de 50 items, l'algorithme de tri par Insertion est plus performant que ceux qui utilisent la méthode de division, tels que le tri fusion ou le tri rapide.

Dans le cas étudié, la collection est composée de 50 items exactement, certains se déclinant en sous-items. Il serait donc intéressant de tester ces deux algorithmes avec plusieurs tailles de collections (50 ; 500 ; 5 000 ; 50 000) afin de mettre en évidence des écarts de performance significatifs.

A titre d'illustration, voici une comparaison des temps d'exécution pour les principales complexités généralement rencontrées *(sur la base arbitraire d'une milliseconde pour une opération élémentaire)*

$\lg n$	n	$n \lg n$	n^2
3 ms	10 ms	33 ms	100 ms
7 ms	100 ms	664 ms	10 s
10 ms	1 s	10 s	16 mn 40 s
13 ms	10 s	2 mn 13 s	1 j 3 h 46 mn
17 ms	1 mn 40 s	27 mn 41 s	115 j 17 h
20 ms	16 mn 40 s	5 h 32 mn	31 ans 259 j

Temps d'exécution : log n = logarithmique | n = linéaire | n log(n) = linéarithmique | n^2 = quadratique

TEST PERFORMANCES | Accueil

Requests	Domains	Subdomains of TLD	Requests to Host	TLD & Subdomain Requests
15	4	0	10	10

Total	Time to First Byte	DOM Content Loading	DOM Processing	Slowest Call
290ms	13ms	180ms	271ms	137ms

Average Call
57ms

Figure 1.a. Performances Insertion Sort (default)

Requests	Domains	Subdomains of TLD	Requests to Host	TLD & Subdomain Requests
15	4	0	11	11

Total	Time to First Byte	DOM Content Loading	DOM Processing	Slowest Call
255ms	17ms	120ms	229ms	80ms

Average Call
47ms

Figure 1.b. Performances Quick Sort

TEST PERFORMANCES | Accueil

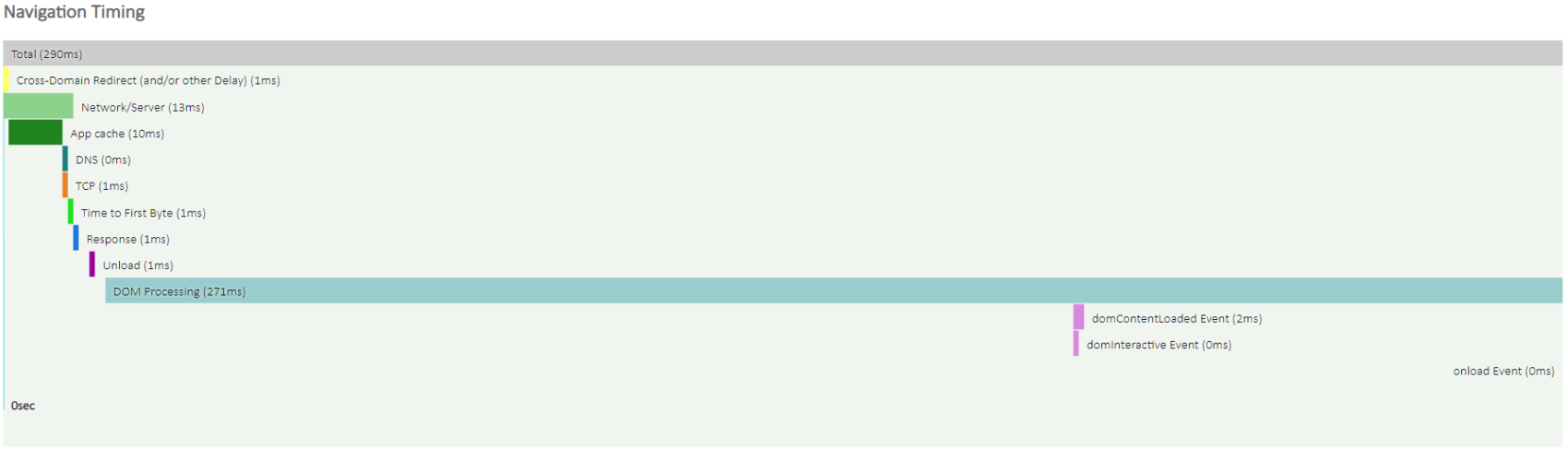


Figure 2.a. Performances Insertion Sort (default)

Figure 2.b. Performances Quick Sort

TEST PERFORMANCES | Accueil

Request FileTypes & Initiators

FileType	Count	Count Internal	Count External	Initiator Type	Count by Initiator Type	Initiator Type Internal	Initiator Type External
css	3	1	2	link	3	1	2
image	3	2	1	img	3	2	1
js	6	6		script	6	6	
font	2		2	css	2		2
other	1	1		fetch	1	1	

Resource Timing

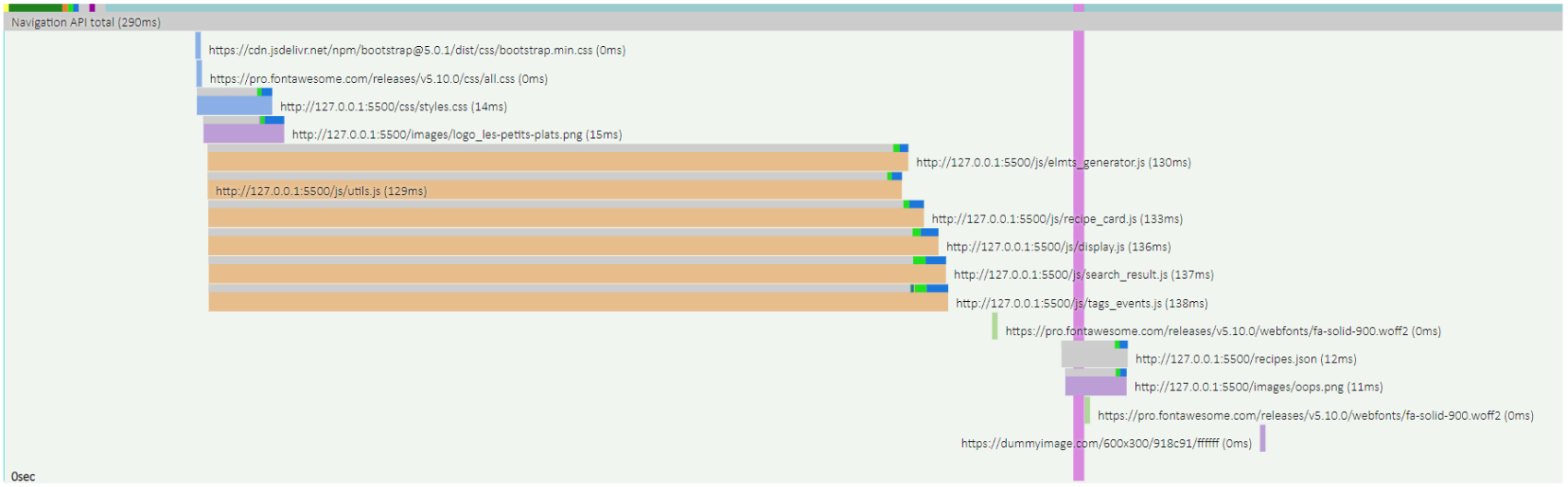


Figure 3.a. Performances Insertion Sort (default)

TEST PERFORMANCES | Accueil

Request FileTypes & Initiators

FileType	Count	Count Internal	Count External	Initiator Type	Count by Initiator Type	Initiator Type Internal	Initiator Type External
css	3	1	2	link	3	1	2
image	3	2	1	img	3	2	1
js	7	7		script	7	7	
font	1		1	css	1		1
other	1	1		fetch	1	1	

Resource Timing

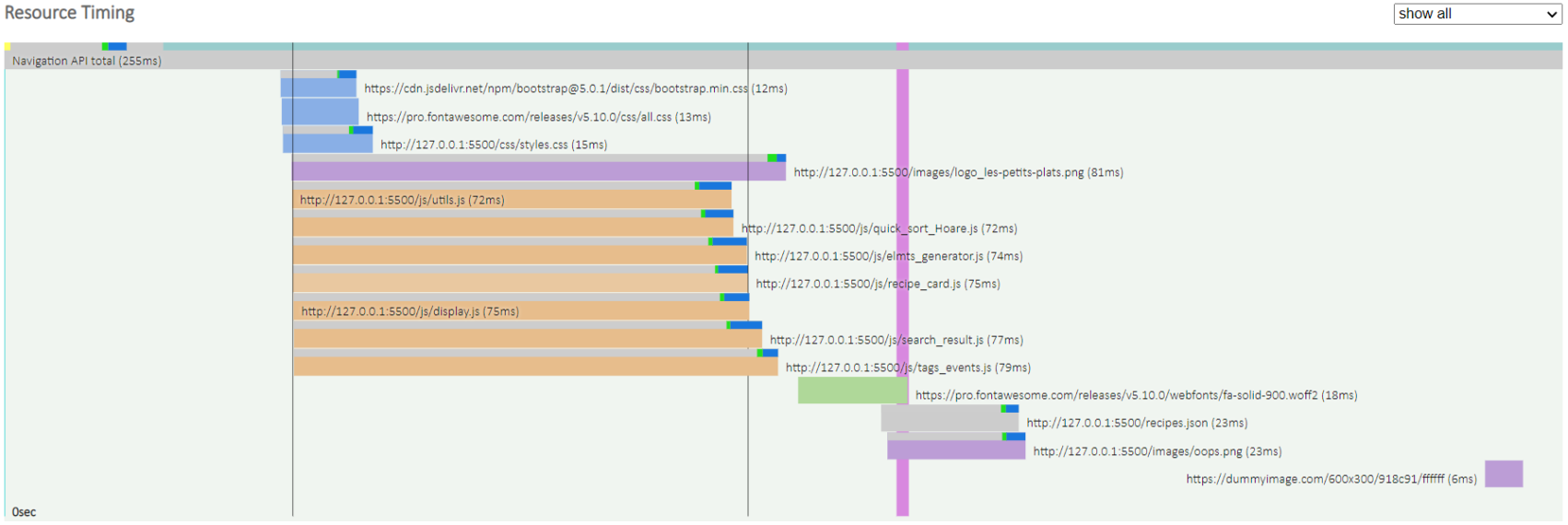


Figure 3.b. Performances Quick Sort

TEST PERFORMANCES | Recherche *item « chocolat »*

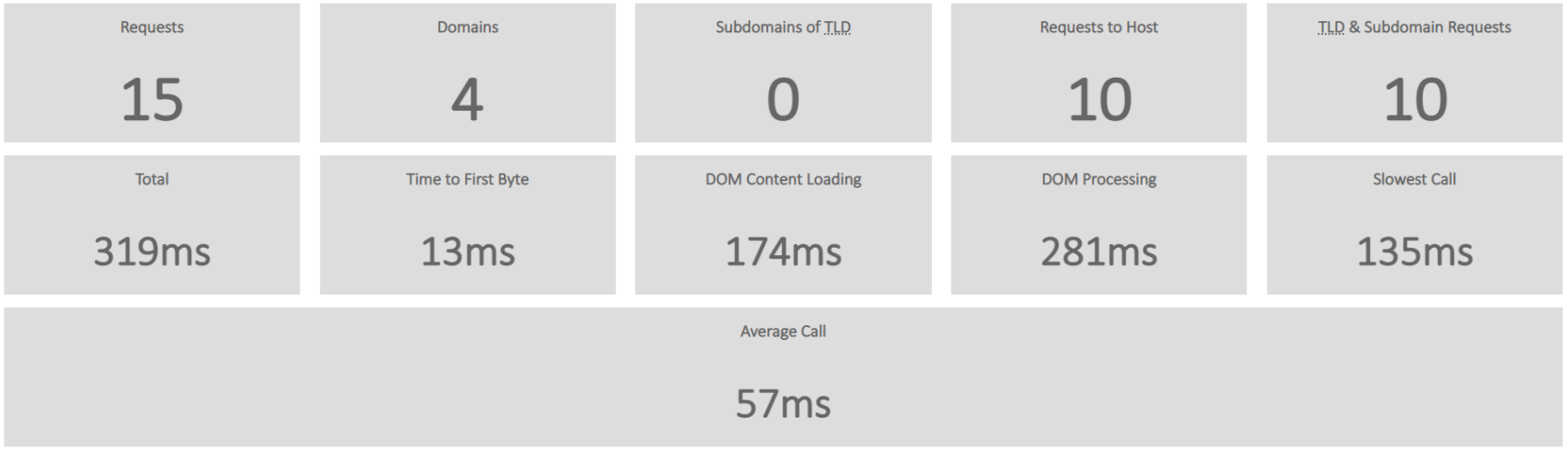


Figure 1.a. Performances Insertion Sort (default)

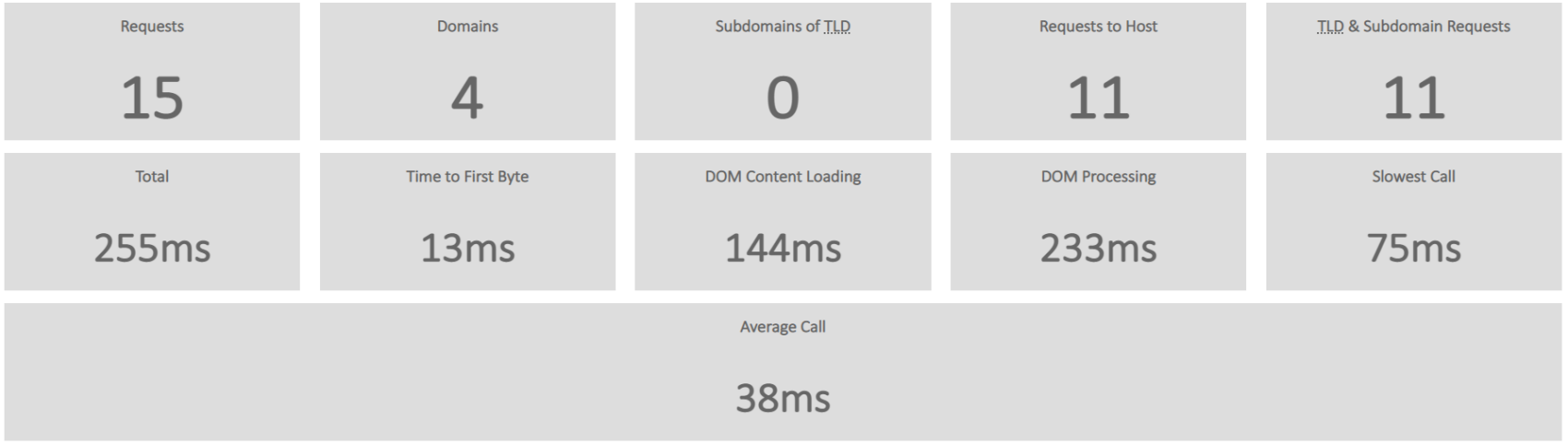


Figure 1.b. Performances Quick Sort

TEST PERFORMANCES | Recherche *item « chocolat »*

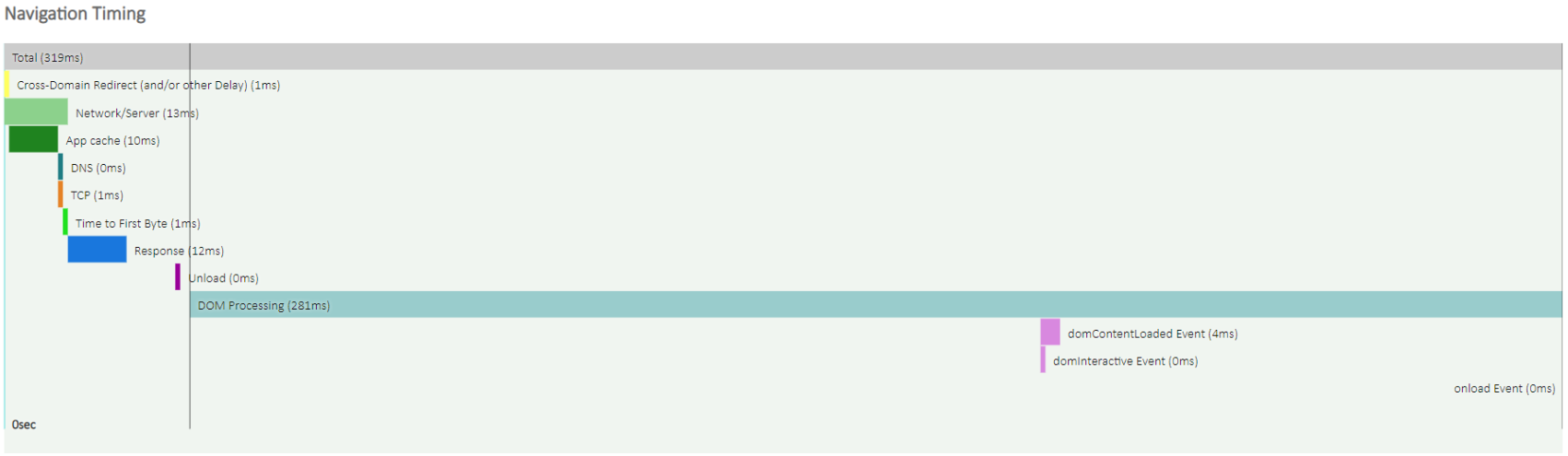


Figure 2.a. Performances Insertion Sort (default)

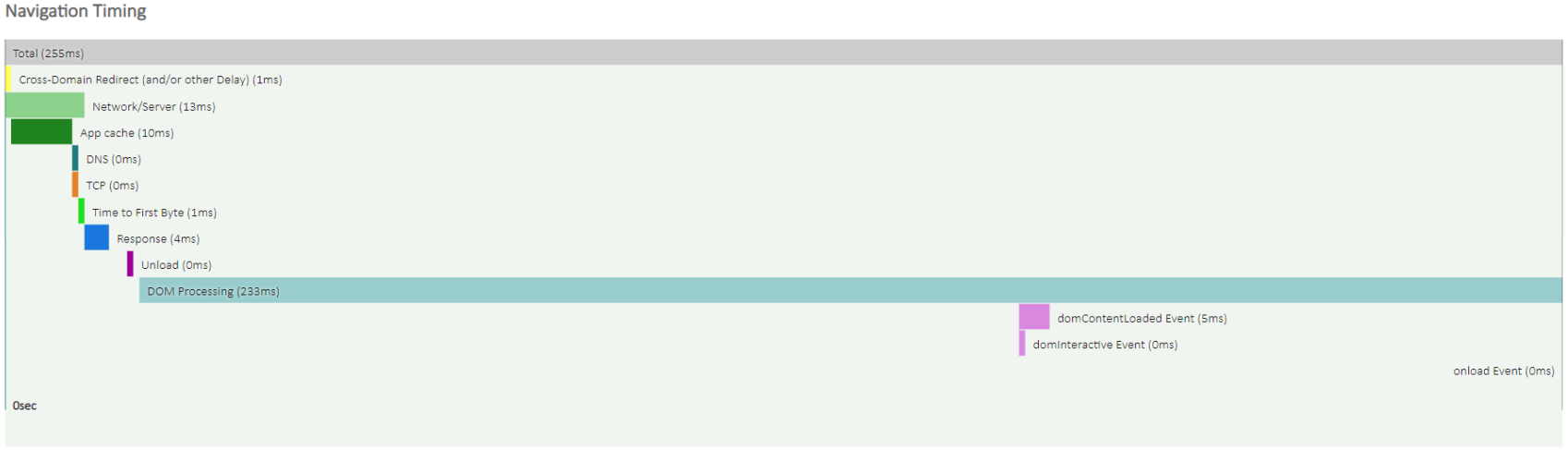


Figure 2.b. Performances Quick Sort

TEST PERFORMANCES | Recherche *item « chocolat »*

Request FileTypes & Initiators

FileType	Count	Count Internal	Count External	Initiator Type	Count by Initiator Type	Initiator Type Internal	Initiator Type External
css	3	1	2	link	3	1	2
js	7	7		script	7	7	
image	4	3	1	img	3	2	1
				other	1	1	
font	1		1	css	1		1
other	1	1		fetch	1	1	

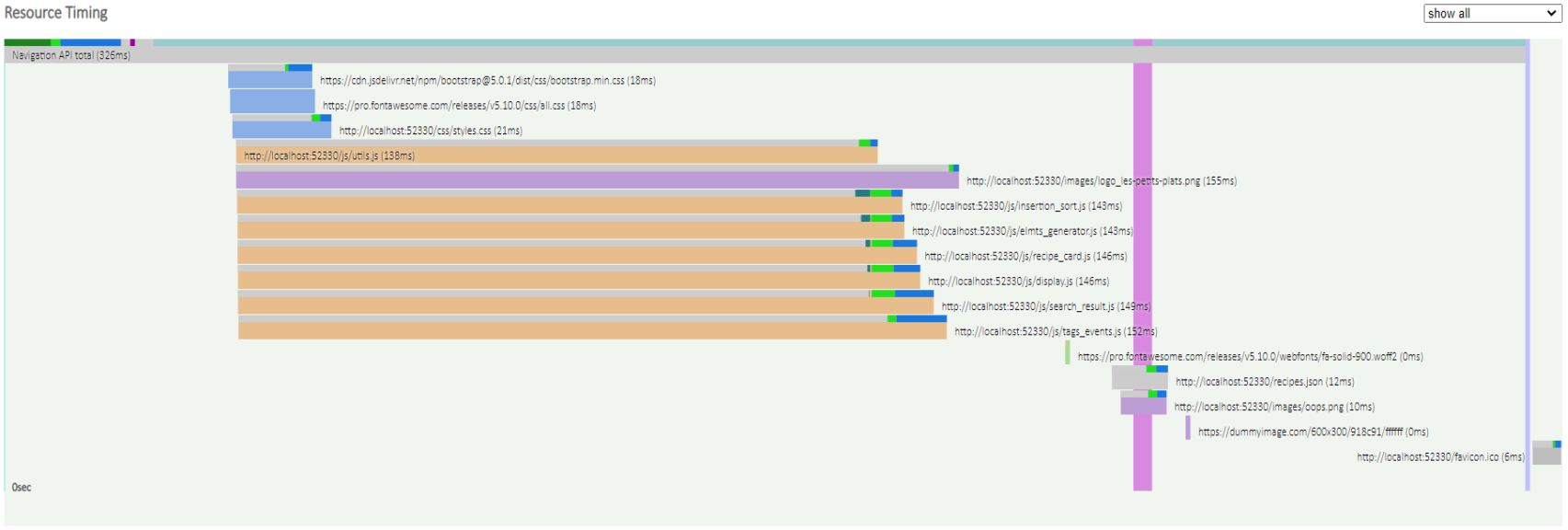


Figure 3.a. Performances Insertion Sort (default)

TEST PERFORMANCES | Recherche *item « chocolat »*

Request FileTypes & Initiators

FileType	Count	Count Internal	Count External	Initiator Type	Count by Initiator Type	Initiator Type Internal	Initiator Type External
css	3	1	2	link	3	1	2
image	3	2	1	img	3	2	1
js	7	7		script	7	7	
font	1		1	css	1		1
other	1	1		fetch	1	1	

Resource Timing

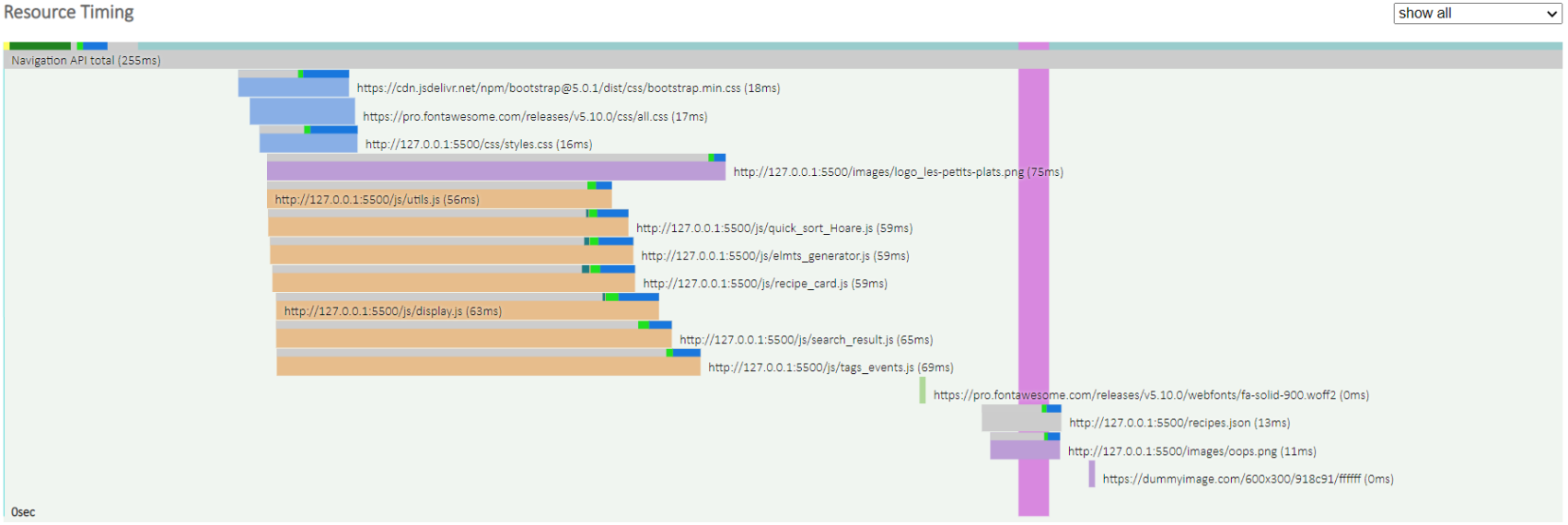
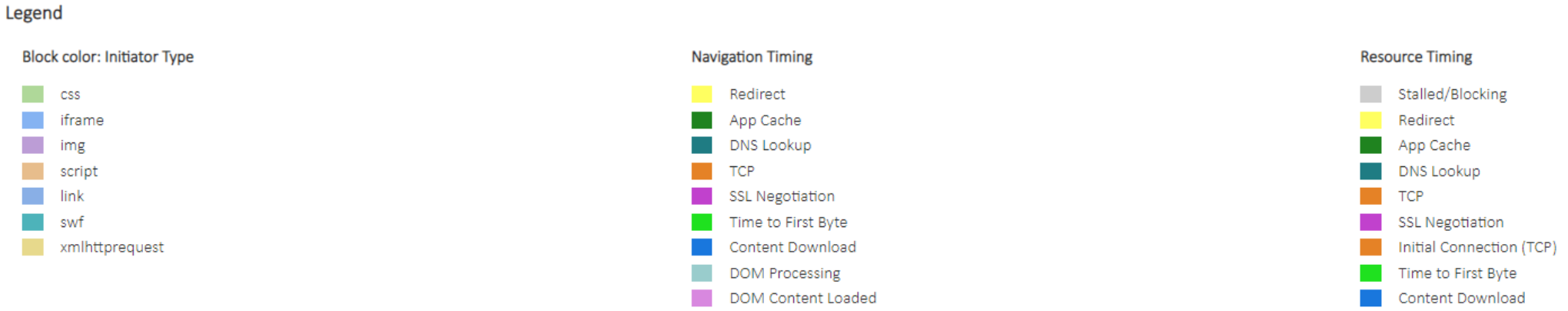


Figure 3.b. Performances Quick Sort

Les petits plats

Fiche d'investigation fonctionnalité #1

TEST PERFORMANCES | Légende



COMPARAISON PERFORMANCES

		Total Nav timming	DOM content loading	Slowest call	Average call	Response timming	Recipes.json
TEST 1 *(ms)*	**Insertion sort**	x	x	x	x	x	x
	Quick sort	x - 35	x - 60	x - 43	x - 10	x + 2	x + 11
TEST 2 *(ms)*	**Insertion sort**	x	x	x	x	x	x
	Quick sort	x - 64	x - 30	x - 60	x - 19	x - 8	x + 1

Annexe

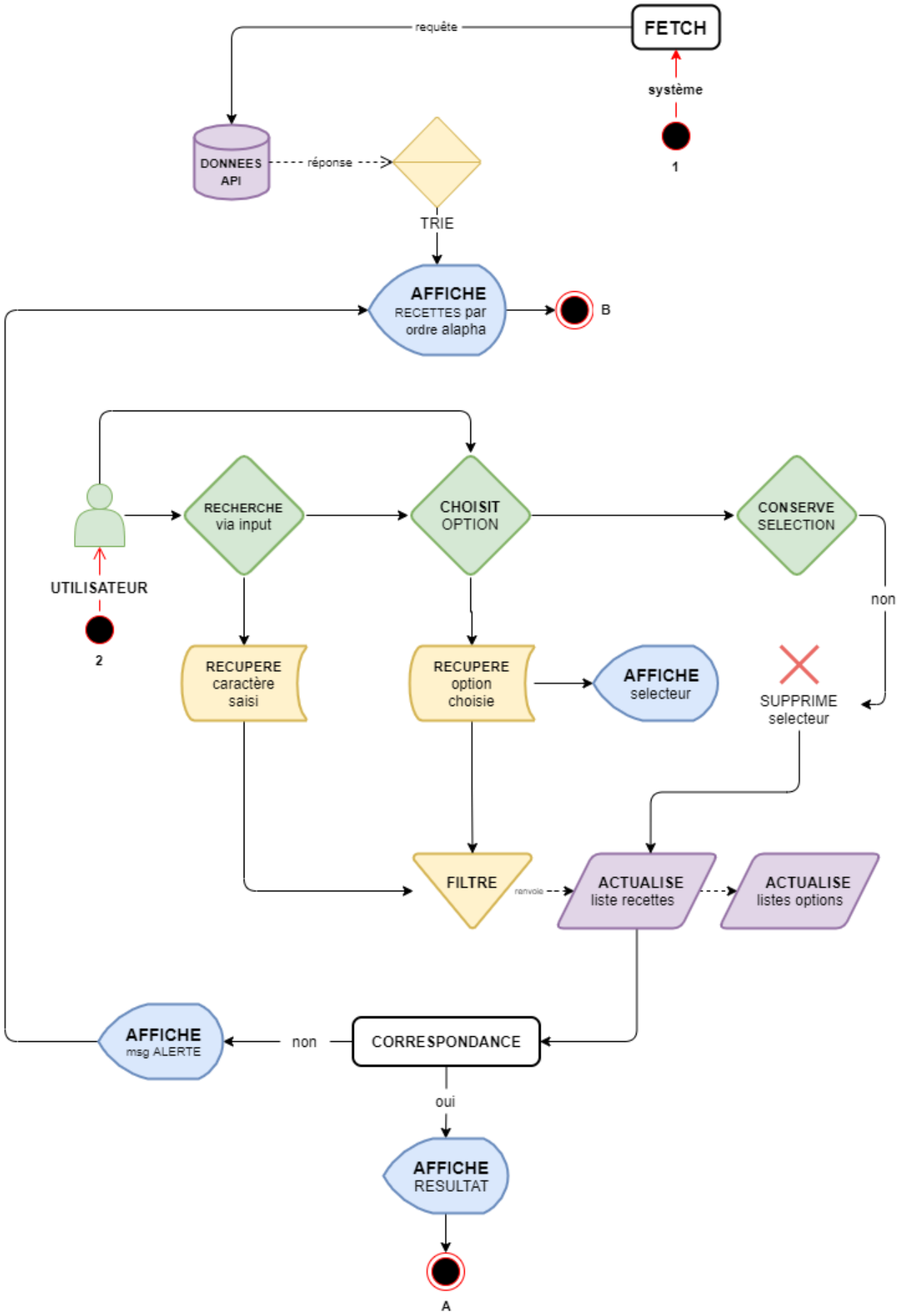


Figure 1 - Diagramme de fonctionnalité

Annexe

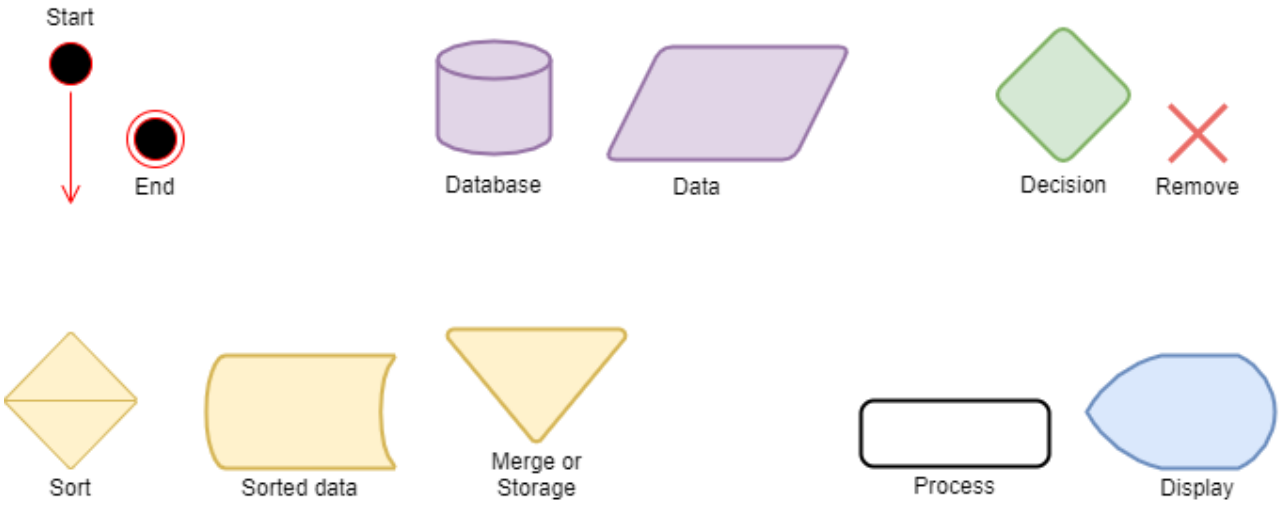


Figure 2 - Légende diagramme de fonctionnalité