# Problème 0 : Optimisation de planning

Ninon Mouhat-Courvoisier
Bastien Brogliato-Martin
Romain Iavarone–Michez
Nathaël Panel-Roques
Titouan Vial

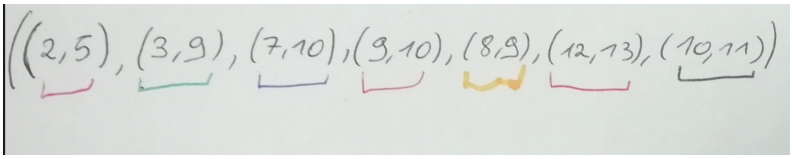
4 avril 2024

Dans le cadre du thème informatique algorithmique et résolution de problèmes, nous avons eu l'occasion de travailler sur un problème d'optimisation de planning. Ce problème consiste à faire succéder un nombre maximal de créneaux horaires sans qu'ils se chevauchent.

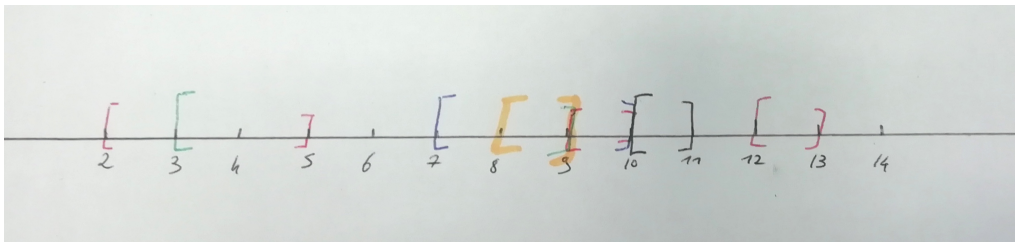
Afin de résoudre ce problème de manière efficace, nous avons chercher à mettre au point un algorithme optimal.

## 1 Réflexions sur le problème

Nous avons eu 3 idées principales lors de nos réflexions. Chaque méthode est détaillée ci-dessous, et illustrée sur l'exemple suivant :

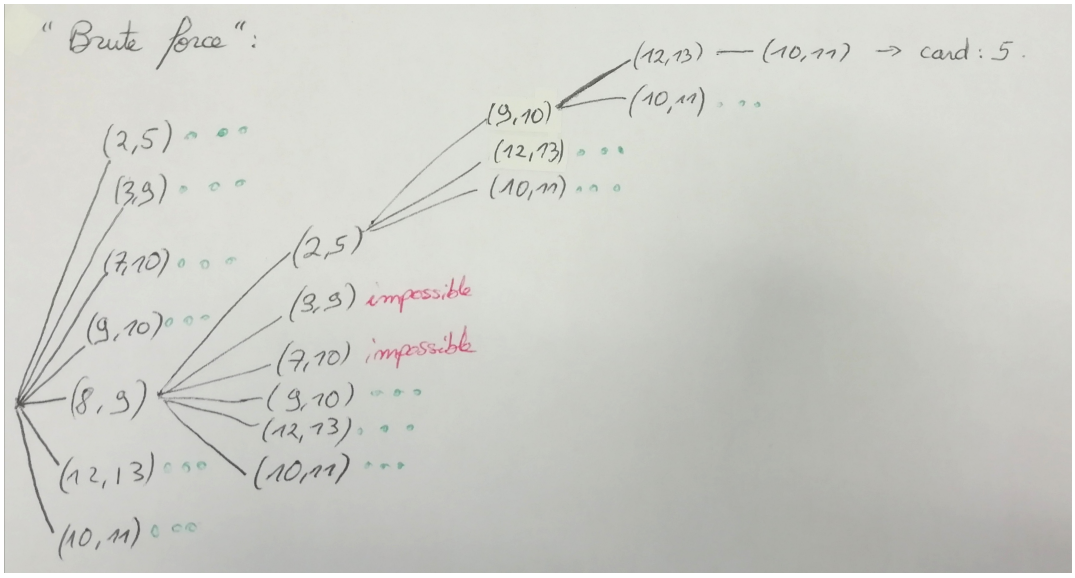



Que l'on représente ainsi comme un ensemble de segments :




Notre première idée était de tester toutes les combinaisons de créneaux possibles et de sélectionner la combinaison de taille maximale (méthode par force brute).

Voici son illustration sur l'exemple :



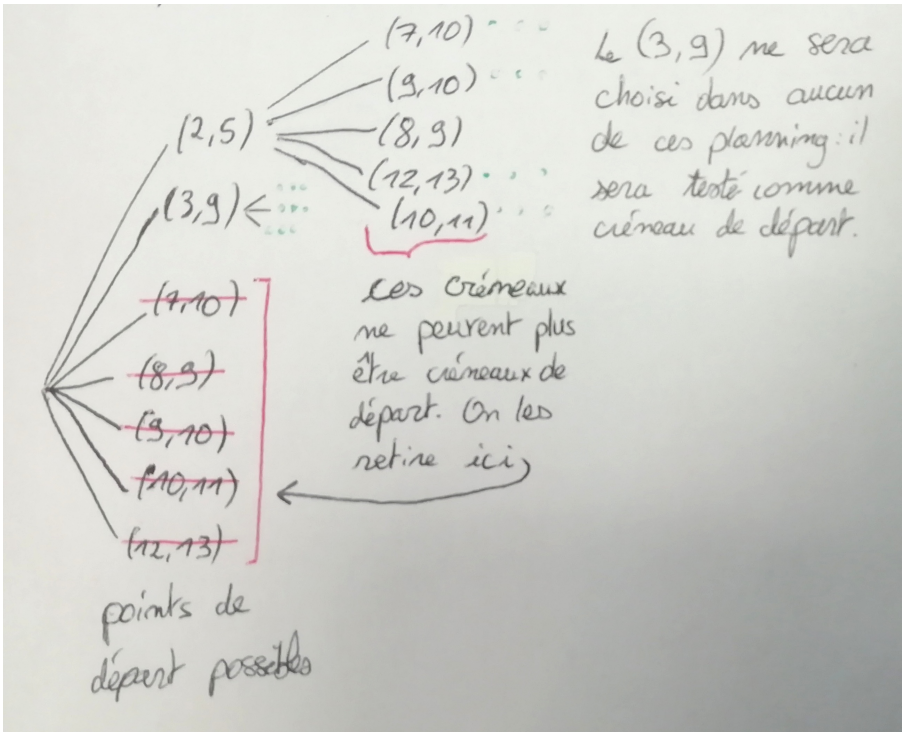

Nous savions que cette idée était fonctionnelle, cependant nous savions également qu'elle était très coûteuse en opérations (complexité en $\mathcal{O}($2.1$)$. Nous avons donc décidé de chercher des algorithmes moins coûteux et de conserver celui-ci afin de vérifier la validité de nos prochains algorithmes.

La deuxième idée que nous avons eu était de de créer un arbre représentant les plannings possibles : chaque branche représente un planning possible, et chaque noeud de la branche à la couche $p$ correspond au $p$-ième créneau du planning en question. Pour que notre algorithme fonctionne nous avons défini la première couche de l'arbre comme ceci : l'ensemble des créneaux triés par ordre croissant selon l'horaire de début puis, en cas d'égalité d'heure de début pour deux créneaux, par ordre croissant selon leur horaire de fin (soit donc un tri lexicographique). Une fois le tri effectué, nous allons créer l'arbre de manière à ce que les seuls planning présents dans cet arbre soit optimaux.

Voici comment l'algorithme créerait l'arbre : pour chaque élément de la première couche pris dans l'ordre dans lesquels on les a triés, on créé le sous arbre ayant comme racine cet élément et on y ajoute des éléments comme ceci : - plaçons-nous à la couche p du sous-arbre actuel, puis dans une des branches de cette couche. On y ajoute l'ensemble des créneaux possibles suite à celui-ci. Ici, tous les créneaux valides sont sous-optimaux en tant que créneau $p-1$ (car tous les plannings qui ont ce créneau à la couche p auront, suite à ce créneau, les mêmes chemins possibles que ceux qui ont le même créneau à la couche $p-1$, mais ils auront donc au final un créneau en moins, ce qui est sous-optimal). Nous supprimons donc ces créneaux des créneaux disponibles comme points de la couche $p-1$.

Une fois tous les éléments traités, on choisit la (ou une des) branches de longueur maximale.

Voici l'illustration de cette méthode :




Lorsque l'on a tenté d'implémenter cette méthode, nous n'avons pas trouvé de manière valide de supprimer les points de départs : soit nous supprimions des solutions possibles donc l'algorithme n'était pas valide (il ne renvoyait pas nécessairement un planning optimal), soit nous ne supprimions pas assez de points de départs et nous revenions à l'algorithme de recherche par force brute légèrement amélioré.

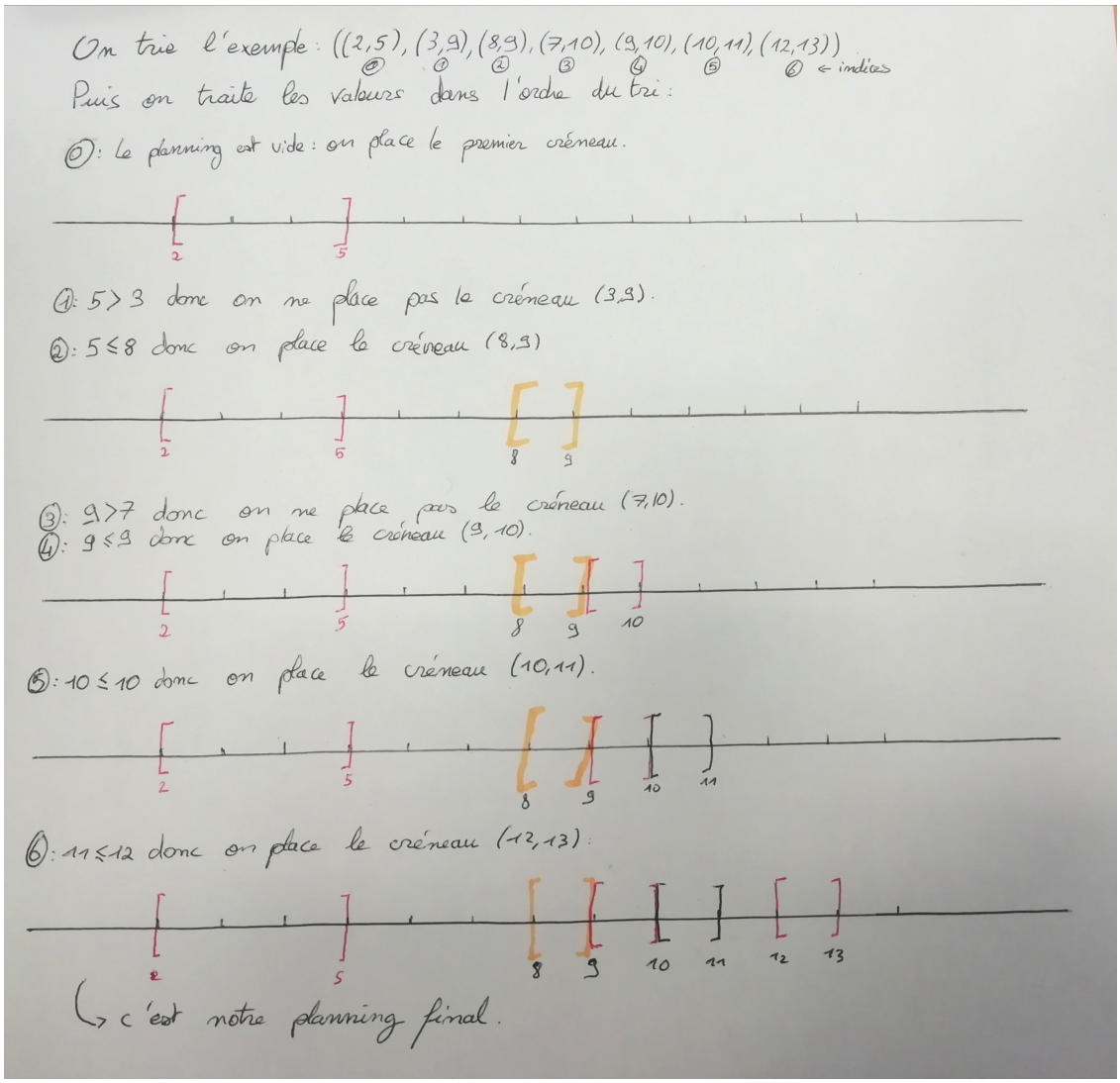
La dernière idée que nous avons est celle que nous avons finalement décidé d'utiliser, et qui est détaillée dans la suite du rapport.

# 2 Algorithme retenu

## 2.1 Principe de l'algorithme

Le principe de l'algorithme retenu est celui d'un algorithme glouton. Le planning optimal est construit créneau par créneau, en lui ajoutant à chaque étape le créneau valide (ie, dont l'horaire de début est supérieure ou égale à l'horaire de fin du dernier créneau du planning) dont l'horaire de fin est la plus faible possible.

Ceci est réalisé à l'aide d'un tri lexicographique croissant des créneaux, en inversant leur horaire de début et de fin pour le tri (ie, les créneaux sont triés d'abord par leur horaire de fin, puis, en cas de conflit, par leur horaire de début). Ceci permet ensuite de construire le planning en un seul parcours de liste, en effectuant uniquement des comparaisons entre l'horaire de fin du dernier créneau du planning et les horaires de début de chacun des créneaux de la liste, les ajoutant au planning dès qu'il est possible de le faire.




Une illustration de la méthode

Nous avons implémenté celle-ci au travers du code python suivant :

```python
def solve_efficient(lst):
    n = len(lst)
    lst = lst[:]
    # tri selon la 2e composante, puis selon la 1re
    lst.sort(key=lambda x: x[::-1])

    solution = []
    # aucun créneau n'a encore été pris, on prend donc une valeur de fin de
    ↪  créneau qui est la "plus petite" possible (dans R) : -infini
    last_end = float('-inf')
    for start, end in lst:
        # si le créneau commence après que le dernier sélectionné ait fini
        if start >= last_end:
            solution.append((start, end))
            last_end = end
    return solution
```

## 2.2 Correction de l'algorithme

**Définitions :**

On définit un créneau comme un couple $(d, f) \in \mathbb{N}^2$ tel que $d < f$. $d$ est appelé « horaire de début » du créneau et $f$ « horaire de fin ». On note $E$ l'ensemble des créneaux dont on dispose dans le contexte du problème.

$D = \{d \mid \exists f \in \mathbb{N}, (d, f) \in E\}$, $F = \{f \mid \exists d \in \mathbb{N}, (d, f) \in E\}$

On définit un planning comme une suite de créneaux $((d_i, f_i))_{1 \leq i \leq k} \in E^k, k \in \mathbb{N}$, telle que $\forall i \in [\![1, k-1]\!], f_i \leq d_{i+1}$

Posons, pour tout $(d, f) \in E, P_{(d,f)} = \{\text{ensemble des plannings débutant par } (d, f)\}$

Posons, pour tout $f \in F$, $E_f = \{(d', f') \in E \mid d' \geq f\}$, l'ensemble des créneaux qu'il est possible de placer après les créneaux se terminant par $f$ dans un planning

De même, posons, pour tout $f \in F$, $D_f = \{d' \mid \exists f' \in \mathbb{N}, (d', f') \in E_f\}$ et $F_f = \{f' \mid \exists d' \in \mathbb{N}, (d', f') \in E_f\}$, les équivalents de $D$ et $F$ dans le sous-problème restreint à $E_f$

On remarque que $\forall (f_1, f_2) \in F^2$, $f_1 \leq f_2 \Rightarrow E_{f_1} \supset E_{f_2}$ et donc $D_{f_1} \supset D_{f_2}$ et $F_{f_1} \supset F_{f_2}$

Posons $N \in \mathbb{N}$ tel que $\forall P$ planning, $P$ est optimal $\Rightarrow \text{card}(P) = N$. L'ensemble des plannings possibles étant un ensemble fini, il admet un élément de plus grand cardinal, ie un planning optimal, ce qui justifie l'existence de $N$, celui-ci étant le cardinal de tous les plannings optimaux.

**Initialisation :**

Posons $(d_1, f_1) \in E$ un créneau tel que $f_1 = \min(F)$.

Pour tout $(d, f) \in E$, pour tout $P \in P_{(d,f)}$, il existe $P_1 \in P_{(d_1,f_1)}$ tel que $\text{card}(P_1) \geq \text{card}(P)$ ($f_1 \leq f$, donc la suite $P_1$ identique à $P$ à l'exception de son premier élément, devenu $(d_1, f_1)$, est un planning valide et tel que $\text{card}(P_1) \geq \text{card}(P)$).

Ainsi, il existe nécessairement un planning optimal débutant par $(d_1, f_1)$, tout planning autre pouvant être modifié en conservant son cardinal pour débuter par $(d_1, f_1)$.

**Hérédité :**

Soit $k \in [\![1, N-1]\!]$, soit $((d_i, f_i))_{1 \leq i \leq k} \in E^k$ un planning valide tel que $\forall i \in [\![1, k-1]\!], f_{i+1} = \min(F_{f_i})$, $f_1 = \min(F)$ et tel qu'il existe un planning optimal dont les $k$ premiers termes sont $((d_i, f_i))_{1 \leq i \leq k}$. Posons $(d_{k+1}, f_{k+1}) \in E_{f_k}$ tels que $f_{k+1} = \min(F_{f_k})$.

Par raisonnement analogue avec celui effectué pour l'initialisation, pour tout planning $P_k$ dont les $k$ premiers termes sont $((d_i, f_i))_{1 \leq i \leq k}$ et dont le $k$+1-ième terme est $(d, f) \in E_{f_k}$, il existe un planning $P_{k+1}$ identique à $P_k$ à l'exception du $k$+1-ième terme qui est remplacé par $(d_{k+1}, f_{k+1})$.

Ainsi, il existe nécessairement un planning optimal dont les $k$+1 premiers termes sont $((d_i, f_i))_{1 \leq i \leq k+1}$. Il est à noter que si $k = N - 1$, ce dernier planning est un planning optimal.

**Conclusion :**

Par récurrence, en construisant un planning en suivant l'algorithme glouton, c'est-à-dire en construisant un planning en prenant à chaque fois le créneau possible finissant le plus tôt, on converge bien vers un planning optimal, chaque ajout de créneau selon cet algorithme n'empêchant jamais d'atteindre un planning optimal, l'algorithme ne pouvant alors s'arrêter que lorsque que le planning créé est optimal. 6.1

## 2.3 Tests pratiques

Pour s'assurer en pratique que notre algorithme fonctionnait bien (pour remarquer une erreur d'implémentation par exemple), nous avons comparé ses résultats avec ceux de la recherche par force brute (dont le résultat est forcément optimal par construction). Pour ce faire, on génère des ensembles de créneau pour faire tourner nos algorithmes dessus. On dispose de trois paramètres pour la création de ces créneaux :

- `n` : le nombre de créneaux disponibles
- `lower_bound` : l'horaire minimal de début d'un créneau
- `higher_bound` : l'horaire maximal de fin d'un créneau

À partir de ces paramètres, on crée une liste de créneaux. On détermine d'abord le début du créneau aléatoirement entre les deux bornes (en retranchant 1 à `higher_bound` pour s'assurer de ne pas avoir un créneau vide où l'horaire de début est égal à celui de fin), puis on détermine son horaire de fin aléatoirement entre son horaire de début (auquel on ajoute 1) et `higher_bound`. On répète l'opération `n` fois pour obtenir une liste de créneaux de taille `n`. Le code utilisé pour générer une liste de créneaux est donc le suivant :

```
lst = []
for _ in range(n):
    start = random.randrange(lower_bound, higher_bound - 1)
    end = random.randrange(start + 1, higher_bound)
    lst.append((start, end))
```

On fait ensuite tourner les deux fonctions (force brute et gloutonne) sur la liste obtenue et on compare la longueur des plannings renvoyés. On ne s'intéresse qu'à la longueur des plannings car c'est celle-ci que nous cherchons à optimiser, indépendamment de leurs contenus. Il est possible que plusieurs solutions optimales (ie, des plannings de même longueur) existent pour un ensemble de créneaux, auquel cas les deux fonctions pourraient renvoyer deux plannings différents. On ne peut donc pas vérifier que les plannings renvoyés sont égaux, mais seulement qu'ils sont de même longueur.

On répète finalement ce procédé un nombre `n_tries` de fois pour bien constater que les fonctions ont bien des résultats similaires sur un grand nombre de cas.

## 2.4 Complexité de l'algorithme

**Complexité théorique**

La complexité théorique de cet algorithme est un $\mathcal{O}(n \log_2(n))$, avec $n$ le nombre de créneaux à optimiser (si la liste d'entrée ne compte aucun doublon, on a $n = \text{card}(E)$).

Le parcours de la liste étant en $\mathcal{O}(n)$ (une comparaison est effectuée pour chaque élément de la liste représentant l'ensemble $E$), sa complexité est alors absorbée par la complexité en $\mathcal{O}(n \log_2(n))$ de la fonction `list.sort` native de Python lors du calcul de la complexité asymptotique totale de l'algorithme.

**Complexité empirique**

Afin de vérifier empiriquement la complexité théorique trouvée ci-dessus, nous avons effectué des mesures du temps d'exécution de notre fonction `solve_efficient` pour différentes valeurs de `n`. Similairement à ce qui est fait dans la partie de tests, on choisit deux bornes (`n_min` et `n_max`) pour les valeurs de `n` à tester. On créé ensuite une liste de créneau de longueur `n_max` (pour une exécution avec $n \in [\![n_{min}, n_{max}]\!]$, on en prendra les $n$ premiers éléments, cela permet d'éviter de répéter la création de la liste à chaque valeur de $n$ ce qui ralentirait sensiblement les tests).

Chaque mesure de durée d'exécution est effectué à l'aide de la fonction `time.perf_counter_ns` dont on prend le résultat avant et après l'appel de la fonction, on obtient alors le temps d'exécution en faisant la différence des deux nombres. On stocke finalement les résultats dans une liste pour pouvoir en faire des graphiques.

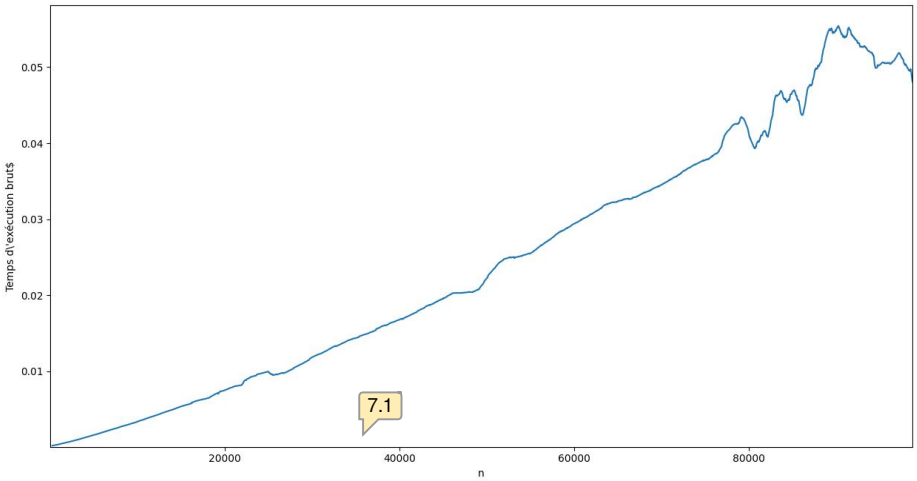



FIGURE 1 – Temps d'exécution de la fonction en fonction de n

Dans la figure 1 on affiche le temps d'exécution de la fonction (en secondes) en fonction de $n$ le nombre de créneaux sur lesquels on exécute la fonction. Afin d'obtenir un résultat lisible on a effectué une moyenne glissante sur les valeurs des temps d'exécution de la fonction car celles-ci sont très bruitées. On constante bien que le temps d'exécution de la fonction semble presque linéaire en fonction de $n$, ce qui est cohérent avec la complexité théorique.

Pour visualiser plus clairement si la complexité théorique se retrouve bien empiriquement, on trace le temps d'exécution de la fonction divisé par $n \log_2(n)$ en fonction de $n$ dans la figure 2. On remarque qu'il y a des légères variations dans la valeur précise du rapport calculé, mais celui-ci reste toujours du même ordre de grandeur et est donc considéré presque constant. Sans pouvoir analyser plus finement ces données, elles semblent bien cohérentes avec une complexité de $\mathcal{O}(n \log_2(n))$.

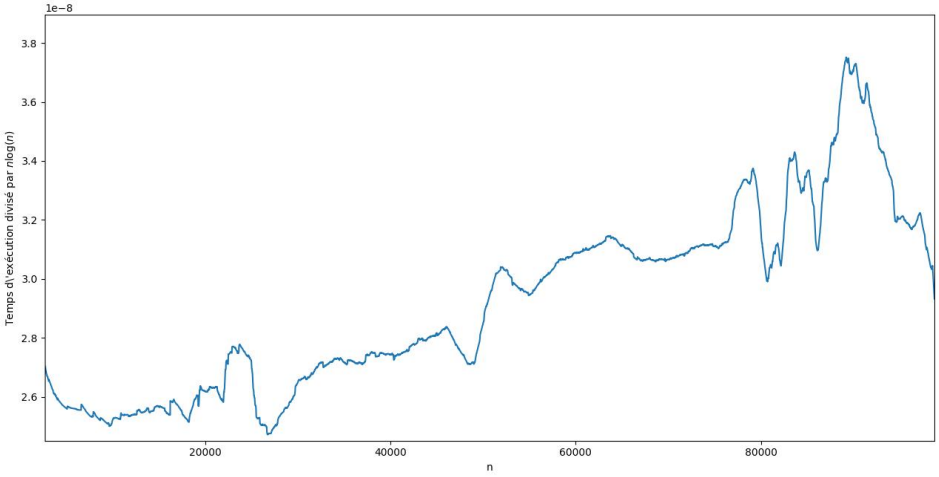



FIGURE 2 – Temps d'exécution de la fonction divisé par $n \log_2 n$

# 3 Conclusion

Pour conclure, nous pensons qu'il était important pour nous de créer plusieurs méthodes pour résoudre ce problème. En effet, cela nous a permis d'extraire les points forts et les points faibles de celles auxquelles nous avions réfléchis. Ainsi, notre algorithme final est un condensé de nos spectres de réflexion individuels, ce qui est un point important pour nous, car ce travail reste d'abord un travail de groupe.

Ainsi une fois notre algorithme établi, nous l'avons implémenté en Python puis nous avons comparé ses résultats avec la méthode de recherche par force brute afin de vérifier sa justesse. Une fois cela fait, nous l'avons démontré mathématiquement.

Chacun a donc contribué à sa manière au travail final, faisant que la solution obtenue et les réflexions ayant menées à son adoption sont indissociables du groupe dans son ensemble, et n'aurait probablement pas pu être obtenue sous cette forme si nous avions travaillé individuellement.

# Index des commentaires