Géométrie Algorithmique

TAYLOR MATT / SOCHAJ YOANN

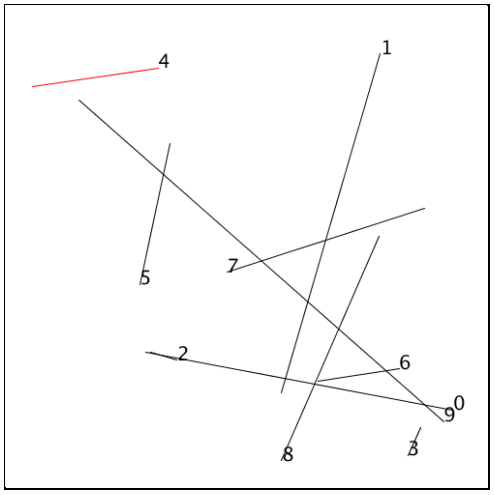
28.03.2021

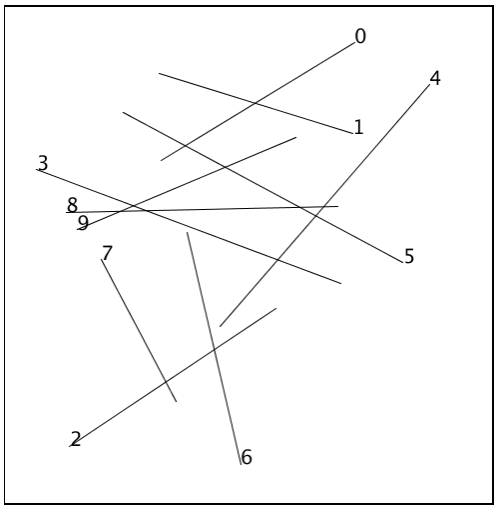
Sommaire:

I. Présentation du projet
II. Les différentes méthodes
   A) Force brute
   B) Diviser pour régner
   C) Ligne de balayage
III. Résultats et Conclusion

I. Présentation du projet

Ce projet de géométrie algorithmique consiste à trouver parmi un nombre de segments **n** générés **aléatoirement**, le **plus grand segment qui n'intersecte aucun autre segment**.





Voici deux exemples de génération de segments:

- le **segment 4** correspondant est affiché en **rouge** *(image de gauche)*.

- si le cas où **tous les segments s'intersectent** se présente alors on affiche un **message** à l'utilisateur dans la **console**: `All the segments intersect` *(image de droite).*

Plusieurs démarches ont été mises en place pour résoudre cet exercice:

Nous avons créé une **classe Segment** qui possède 4 attributs:
- **2 PVector**
- **boolean *hasIntersection***
- **float** pour sa **longueur**

Nous nous sommes également servis de la fonction de **tri intégré à Java** en l'adaptant pour trier nos segments pour avoir des **temps plus rapides**.

Nous avons implémenté le fait de pouvoir choisir la **taille** des segments ce qui a un **impact** important sur la complexité des algorithmes car le nombre d'intersections est beaucoup plus petit.

Ce projet a permis de nous introduire la notion de **complexité** et son rôle **primordial** dans les algorithmes lorsqu'il s'agit de traiter des **données de taille très grandes**.

Nous utilisons plusieurs algorithmes de résolution comme la **force brute**, un algorithme récursif de type **diviser pour régner** et un algorithme itératif de type **ligne de balayage**.

II. Les différentes méthodes

A) Force brute

Nous avons commencé avec l'algorithme de type **force brute**. Celui-ci est la première idée qui nous est venue en tête quand on a essayé de résoudre cet exercice.

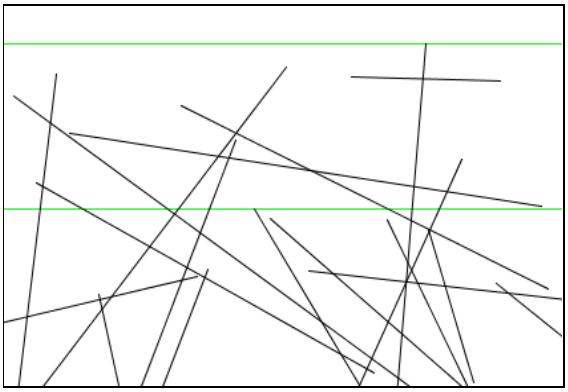
Cette méthode est très **simple à comprendre**, il suffit de faire une **double boucle** for afin de tester pour 1 segment **tous les autres** un à un. Un **boolean** va nous servir à savoir **s'il y a une intersection** ou non à la fin de la deuxième boucle. Si il n'y a pas d'intersection, on regarde si notre segment **MAX** est initialisé. Si il n'est pas initialisé alors on **l'initialise** avec le segment **courant** sinon on regarde si le segment courant est plus grand que **MAX**. S'il est plus grand alors on **change** le max, sinon on retourne juste à la boucle. À la fin des deux boucles nous retournons donc le **plus grand segment sans intersection**

Cette méthode est très **naïve**. Elle est de complexité **O(n^2)**.

B) Diviser pour régner

La méthode **diviser pour régner,** était la méthode **récursive** qu'on a du faire. Cette méthode est **simple** à comprendre, avec les **explications** du professeur et les différentes **interventions** sur le discord. Mais la récursivité est toujours plus "**compliquée**" à mettre en place. Nous allons vous présenter les étapes du programme, mais vous pouvez aussi aller voir sur le code (commenté) car il y aura plus d'informations concrètes.

1. On appelle la fonction **startDivideAndConquer**. Elle **trie** les segments en fonction de la **coordonnée Y,** haut vers le bas. Ensuite on va faire l'appel de la fonction récursive avec un début et une fin (0, nbSegments)



2. Une fois dans la fonction récursive, on crée une liste de **résultats.** Nous avons ensuite la **condition d'arrêt**, ici c'est si on a **strictement moins de 10 segments**, alors on doit faire le **bruteforce** sur ces 10 segments. C'est un bruteforce **modifié** car il doit renvoyer une **liste** de tous les segments sans intersections.

3. Après la **condition d'arrêt**, on doit **lancer** la **récursion** et on va faire cela à l'aide du **milieu**. En effet, on va récupérer le **segment du milieu**. Une fois ceci fait on va faire l'appel récursif du haut et du bas **(0->mid/mid->nbSegment)**. Ça va nous permettre d'avoir les segments du **haut et du bas sans intersections.**

4. Maintenant, il suffit de tester à l'aide d'une **double boucle for**, les segments **candidats** du **haut** avec **tous** les segments du bas (à l'aide des indices 0/end/mid). **Faire pareil** pour les segments **candidats du bas**, mais avec tous les segments du haut (en vert vous pouvez voir une subdivision).

5. Après ce test, on va donc se retrouver avec **TOUS** les segments **sans intersections**, et on envoie cela à la fonction **startDivideAndConquer** (ou on renvoie null si il n'y a pas de segment). On va maintenant regarder la **longueur** des segments, car en effet il faut le **plus grand**. On va faire ceci grâce à **l'attribut length** de Segment. Si il y a qu'un **seul** segment dans la liste, alors on le **renvoie directement** sans regarder la taille.

6. On fait donc une **boucle for** qui va **parcourir** la liste de segments **candidats** pour **récupérer** le plus grand.

7. On **return le plus grand segment** si il y en à un ou **null** si on n'a pas de segment.

C) Ligne de balayage

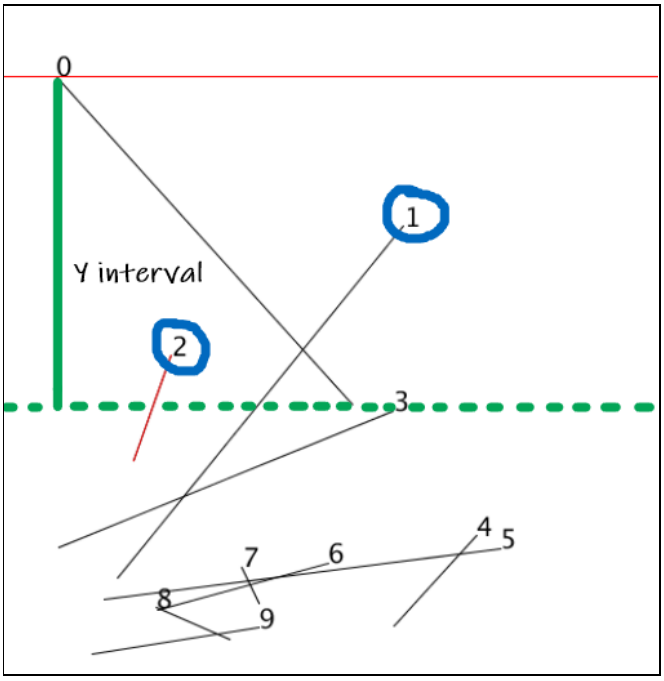
Cette méthode s'appelle **Sweep Line**, le principe est très simple à visualiser mais un peu plus dur à implémenter.

Voici un exemple avec 10 segments:

1. On **trie** nos segments par leurs **coordonnées Y**.

2. On trace une **ligne de balayage** sur le premier point du premier segment de la liste (le plus haut sur la fenêtre), on mesure l'**intervalle Y** de ce segment.

3. On vérifie l'**intersection du segment où se trouve la ligne de balayage et un autre segment seulement s'il est dans cet intervalle** (c'est une perte de temps de vérifier les autres car une intersection ne sera pas possible).
Dans notre cas, il faudrait vérifier l'intersection entre **0 et 1** et **0 et 2**.



Mais une **première optimisation** peut se faire à cette étape.

4. En effet dès que l'on rencontre une intersection on peut **break** et **sortir de la boucle** car ce segment possède une intersection et nous "**intéresse" plus**. On "**marque**" les deux segments en mettant leur **boolean *hasIntersection*** à **vrai**.
On descend la ligne de balayage sur le segment suivant. *(ici, le segment 1 s'intersecte avec 3, on descend encore la ligne de balayage (sur le segment 2),* ***pas besoin de vérifier l'intersection avec le segment 0 car il est déjà "marqué"*** *de même pour le segment 1 et pas d'intersection avec 3 .On a vérifié tous les segments dans l'intervalle et on a aucune intersection).*

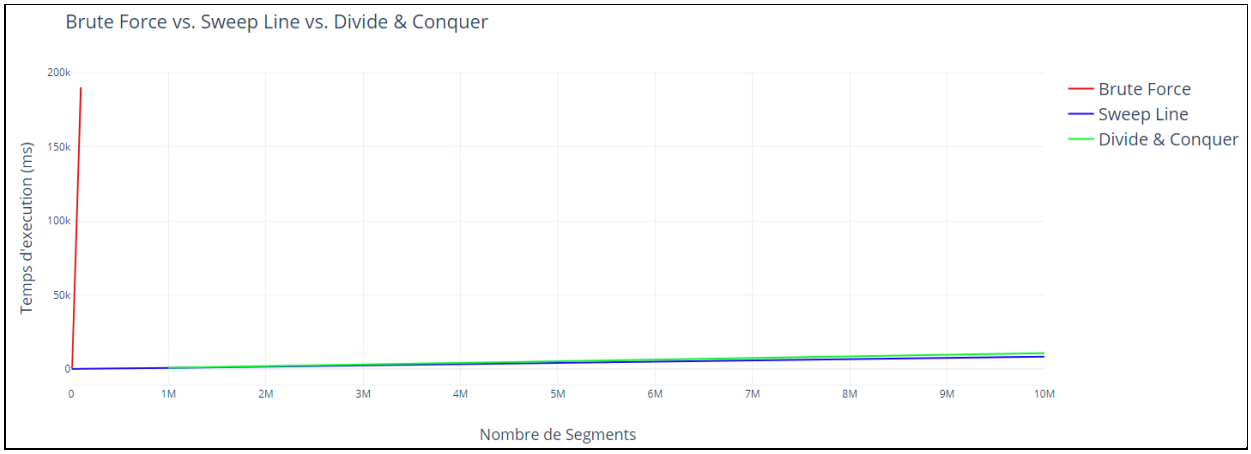
5. Si le segment où se trouve la ligne de balayage ne possède **pas d'intersection, on** le **rajoute** à notre **liste de segments "candidats"** à condition que sa longueur soit supérieure à **maxLength** (qui est initialisé à 0 au début de l'algorithme). Cette valeur est mise à jour et prend la valeur de la longueur du segment que l'on ajoute à notre liste "candidats". Si au moment d'ajouter un segment dans notre liste il y a déjà un segment on l'**enlève pour garder uniquement le plus grand**.

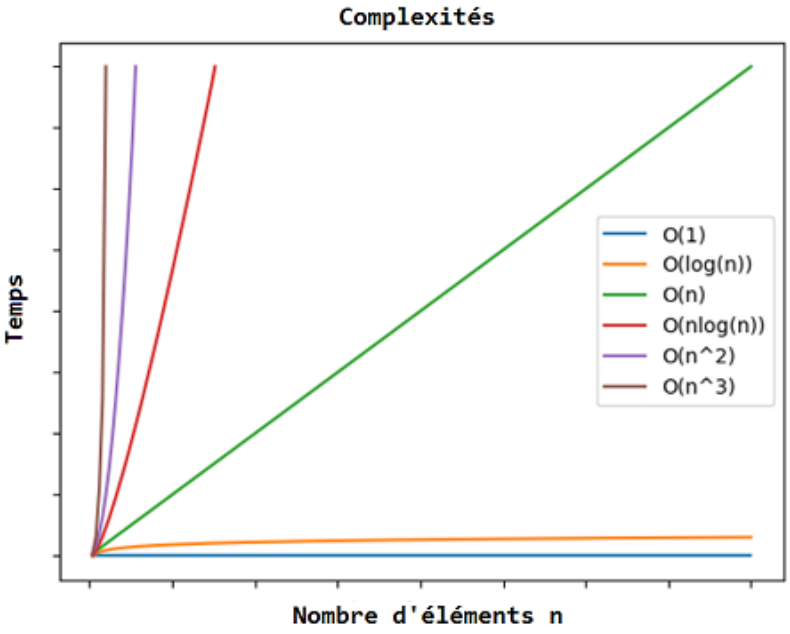
6. L'algorithme s'**arrête** lorsque la **ligne de balayage** se trouve sur le **dernier segment**.

7. Notre **résultat** est le **seul segment** dans notre liste **"candidats"** ou éventuellement **null** si tous les segments s'intersectent.

III. Résultats et Conclusion

_____Voici un graphe qui compare les temps d'exécution de nos trois algorithmes. Evidemment ces calculs ont été faits sur la même machine pour pouvoir les comparer efficacement. On remarque que l'algorithme de type "brute force" a une complexité de **O(n^2)** alors que les deux autres sont beaucoup plus rapides autour de **O(n log(n)),** on ne peut pas faire mieux car il y à une étape de **tri** .





_____Concernant la répartition des tâches nous avons travaillé sur le brute force pour avoir un premier algorithme fonctionnel. Nous avons ensuite travaillé sur les algorithmes l'un après l'autre. A chaque fois qu'on avançait on vérifié avec l'autre personne. On a vu que l'algorithme **Divide & Conquer** était plus lent que **Sweep Line**. Nous avons donc eu besoin de repasser sur le code pour apporter des optimisations (voir la dernière ligne du tableau).

Pour pouvoir se partager le code efficacement nous avons mis en place un espace de dépôt sur **GitHub** ainsi que des envois par **Discord**.

En conclusion nous avons adorés faire ce petit projet d'algorithmique et avons appris beaucoup de choses sur comment rendre nos programmes plus optimisés et connaître la **complexité** de nos algorithmes.

Nous avons regroupé **tous nos résultats** dans un google sheet pour ne pas surcharger le rapport, voici un lien pour les voir si vous êtes intéressé.

Temps moyen d'execution (ms)							
Nombre de segments	10	100	1000	10000	100000	1000000	10000000
Algorithme							
Brute Force	0.8	2	29.2	1779.6	190109		
Sweep Line	1	1.8	4.8	21.8	84	685	8287
Divide And Conquer	0.6	1.4	3	12.4	76.8	816.2	10680
Divide And Conquer Non Optimisé	1.4	2	24	32	497	14824	NA