DM Tri par insertion - CS353

1 Introduction

L'objectif de ce document est de faire un compte rendu du travail réalisé lors de la mise en oeuvre de l'algorithme de tri par insertion dans un langage de haut niveau. Pour cela, j'ai choisi d'utiliser le langage Python car celui-ci offre plusieurs avantages par rapport à un langage plus bas niveau comme le C et qui sont les suivants:

- La syntaxe est plus souple et moins sensible que des langages comme le C ou Java.
- On dispose de nombreuses "bibliothèques" et outils afin de réaliser des opérations complexes comme des représentations graphiques, etc.
- Il nécessite moins de lignes de code car il est plus abstrait et de plus haut niveau que le C par exemple.
- Enfin, c'est un langage de plus en plus utilisé que j'apprécie et que j'utilise souvent.

Ce document a été fièrement rédigé en LaTeX.
Je tiens à dire que l'ensemble de ce travail a uniquement était réalisé par moi-même sans plagiat en utilisant le plus possibles de outils libres et éthiques.
Les sources et les compilations liées à ce travail sont sous licence CC-BY et disponibles ici: `https://github.com/matodonte/`

2 Démarches

Comme conseillé dans la partie cours, j'ai réalisé trois cas d'études:

1. Cas d'un tableau déjà trié (cas simple).
2. Cas d'un tableau avec des éléments répartis suivant une probabilité gaussienne (cas moyen).
3. Cas d'un tableau trié dans l'ordre décroissant (cas complexe).

De ces cas d'études, on obtiendra des représentations graphiques des résultats que je commenterais afin d'en donner le sens.

3 Le code source

Mon code source Python est segmenté en quatre partie.
On a tout d'abord l'importation des "bibliothèques" classiques utilisées.

```
import random as ra
import matplotlib.pyplot as plt
import time
```

Puis, j'ai implémenté en Python l'algorithme de tri par insertion:

```
def tri_par_insertion(tableau):
    # variable pour avoir le temps d'execution de la fonction
    temps_init = time.time()
    # variable pour compter le nombre de comparaisons
    compteur=0
    for parcours in range (1,len(tableau)):
        # variable "pivot" qui va stocker une variable du tableau qui va
        # bouger pour trier le tableau
        pivot = tableau[parcours]
        indice = parcours-1
        compteur+=1
        while indice >= 0 and tableau[indice] > pivot:
            compteur+=1
            tableau[indice+1] = tableau[indice]
            indice -= 1
        tableau[indice+1] = pivot
    # on soustrait le temps cumule avec le temps de depart pour avoir
    # le temps d'execution de la fonction
    temps = time.time() - temps_init
    return tableau,compteur,temps
```

Cette fonction renvoie un tuple contenant trois éléments (3-uplet) qui sont respectivement:

1. Le tableau trié.
2. Un entier contenant le nombre de comparaisons effectuées.
3. Le temps d'exécution en seconde mis par la fonction.

Ensuite, j'ai créé les fonctions permettant de générer les tableaux pour réaliser mes études de cas:

```
def genere_tableau_aleatoire_trie(taille):
  return [i for i in range(0,taille)]
```

Cette fonction génère (renvoie) un tableau de taille "taille" dont les éléments sont déjà triés. Ce tableau représentera notre cas dit "facile".

```
def genere_tableau_aleatoire(distance_moyenne, taille):
  return [int(ra.gauss(0,distance_moyenne)) for i in range(0,taille)]
```

Cette fonction génère (renvoie) un tableau de taille "taille" dont les éléments suivent une distribution de probabilité gaussienne ayant une espérance de 0 et un écart-type de "distance_moyenne". Ce tableau représentera notre cas dit "moyen".

```
def genere_tableau_aleatoire_pas_trie(taille):
  return [i for i in range(taille,0,-1)]
```

Cette fonction génère (renvoie) un tableau de taille "taille" dont les éléments sont déjà triés mais dans l'ordre décroissant. Ce tableau représentera notre cas dit "complexe".

Finalement je crée une dernière fonction qui permettra de faire un diagnostic de tous ces tests et de tirer des résultats intéressants:
(L'affichage du code source n'est pas très bien ici, il sera plus lisible dans le fichier d'origine)

Cette fonction va simplement faire une boucle $taille\_fin - taille\_debut$ fois avec des tailles de tableaux différents et va exécuter la fonction de tri avec ces différents tableaux. On récupérera dans des listes le nombre de comparaisons ainsi que le temps d'exécution pour chaque tableau. Ensuite, nous afficherons tout d'abord le nombre de comparaisons pour les trois cas en fonction de la taille des tableaux puis nous afficherons le temps d'exécution pour les trois cas en fonction de la taille des tableaux.
On commentera ensuite ces résultats et les comparerons avec la théorie.

```
def affiche_courbe(d_moy,taille_debut,taille_fin):
  print("########################")
  print("DIAGNOSTIC EN COURS ...")
  print("########################")
  # caclul le temps pour executer la fonction finale
  temps_init = time.time()
  compteur=0
  # les listes contenant les valeurs des comparaisons effectuees dans les
  # Voir plus loin
  liste_c_trie = []
  liste_c_pas_trie = []
  liste_c_gauss = []

  # listes contenant les temps d'execution pour des tableaux et des taill
  liste_t_trie = []
  liste_t_pas_trie = []
  liste_t_gauss = []

  for taille in range(taille_debut,taille_fin):
      tab_trie,c_trie,t_trie=tri_par_insertion(genere_tableau_aleatoire_t
      tab_pas_trie,c_pas_trie,t_pas_trie = tri_par_insertion(genere_table
      tab_gauss,c_gauss,t_gauss = tri_par_insertion(genere_tableau_aleato
      #print(taille , c_trie)
      liste_c_trie.append(c_trie)
      liste_c_pas_trie.append(c_pas_trie)
      liste_c_gauss.append(c_gauss)
      liste_t_trie.append(t_trie)
      liste_t_pas_trie.append(t_pas_trie)
      liste_t_gauss.append(t_gauss)
      compteur+=1
      print(compteur/(taille_fin-taille_debut)*100, "%")

  fig_trie, ax_trie = plt.subplots()
  ax_trie.plot([i for i in range(taille_debut,taille_fin)], liste_c_trie)
  plt.title("Nombre de comparaisons pour un tableau deja trie")
  plt.xlabel("Taille tableau")
  plt.ylabel("Nombre de comparaisons")
  fig_pas_trie, ax_pas_trie = plt.subplots()
  ax_pas_trie.plot([i for i in range(taille_debut,taille_fin)], liste_c_p
  plt.title("Nombre de comparaisons pour un tableau trie dans l'ordre dec
  plt.xlabel("Taille tableau")
  plt.ylabel("Nombre de comparaisons")
```

```
    fig_gauss, ax_gauss = plt.subplots()
    ax_gauss.plot([i for i in range(taille_debut,taille_fin)], liste_c_gaus
    plt.title("Nombre de comparaisons pour un tableau genere \n aleatoireme
    plt.xlabel("Taille tableau")
    plt.ylabel("Nombre de comparaisons")

    fig_temps_trie, ax_temps_trie = plt.subplots()
    ax_temps_trie.plot([i for i in range(taille_debut,taille_fin)], liste_t
    plt.title("Evolution du temps d'execution avec des tailles \nde tableau
    plt.xlabel("Taille tableau")
    plt.ylabel("Temps d'execution (en secondes)")
    fig_temps_pas_trie, ax_temps_pas_trie = plt.subplots()
    ax_temps_pas_trie.plot([i for i in range(taille_debut,taille_fin)], lis
    plt.title("Evolution du temps d'execution avec des tailles de tableaux
    plt.xlabel("Taille tableau")
    plt.ylabel("Temps d'execution (en secondes)")
    fig_temps_gauss, ax_temps_gauss = plt.subplots()
    ax_temps_gauss.plot([i for i in range(taille_debut,taille_fin)], liste_
    plt.title("Evolution du temps d'execution avec des tailles de tableaux\
    plt.xlabel("Taille tableau")
    plt.ylabel("Temps d'execution (en secondes)")

    temps_finale = time.time() - temps_init

    print("DIAGNOSTIC FINI")
    print("######################## \n")
    print("Le diagnostic s'est execute en {} secondes ! ".format(temps_fina
```

Cette fonction affiche 6 courbes que nous allons expliciter ci-dessous.

4 Les résultats et conclusions

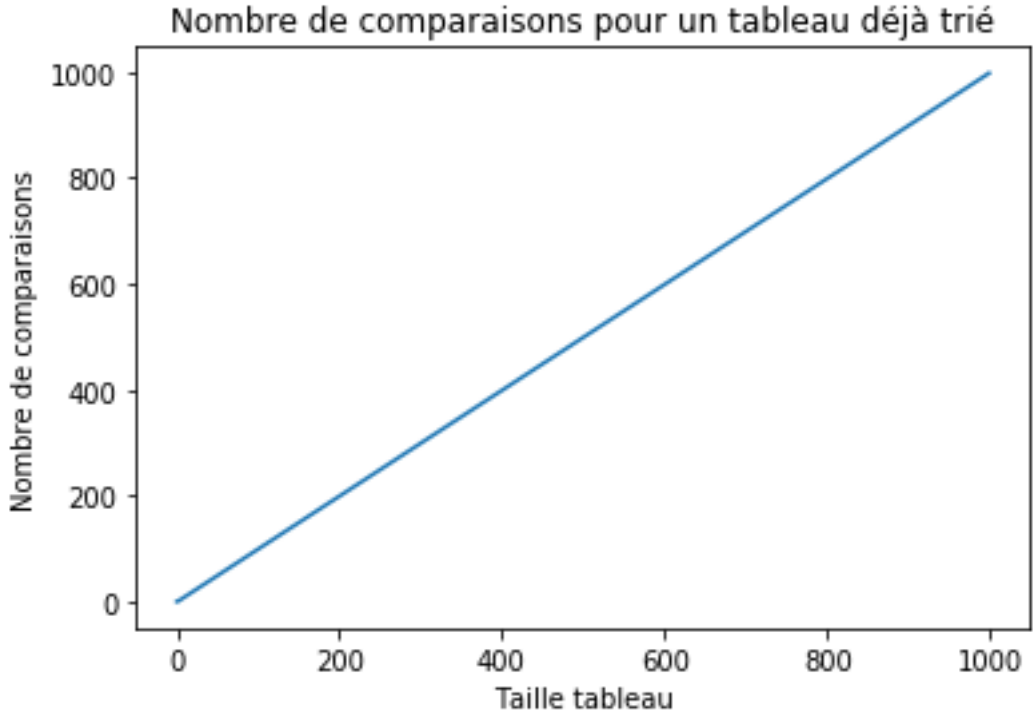
Nous lançons donc dans un shell le programme "affiche_courbe()" comme ci-dessous:

```
affiche_courbe(20,0,1000)
```

Nous prenons une distance moyenne entre chaque éléments de 20 (pour le tableau "moyen") et une taille maximale de 1000 éléments. Cela va donc générer 1000 tableaux (échantillons).

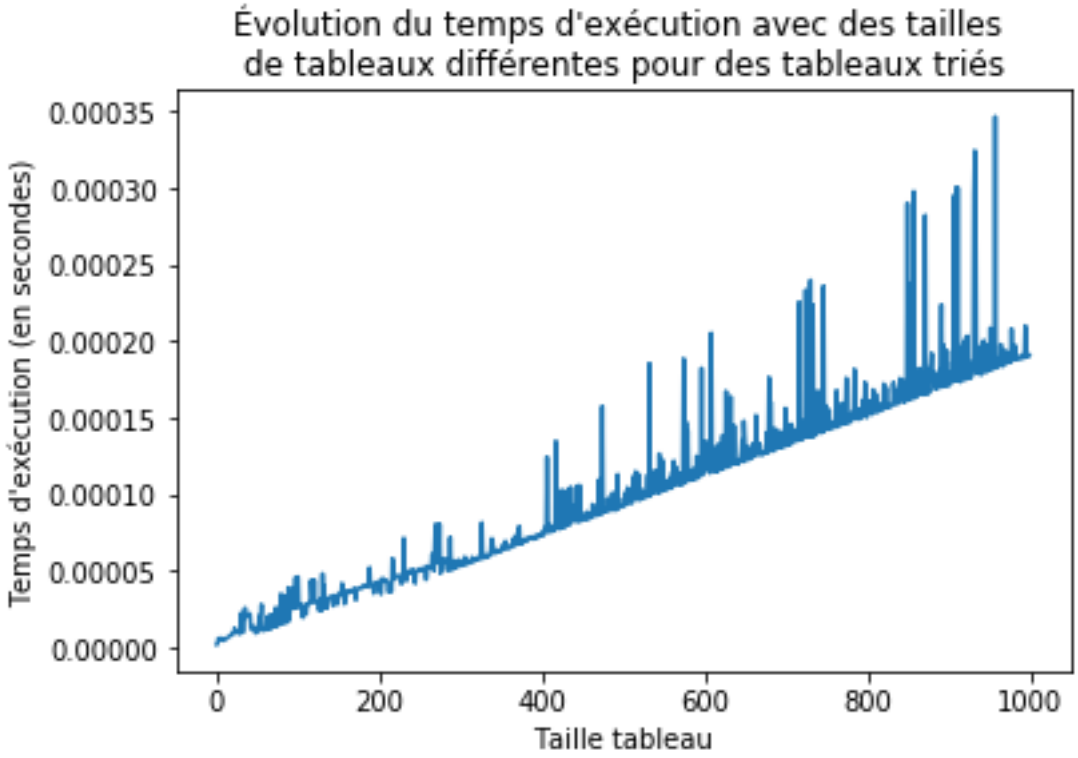
4.1 Cas tableau trié (cas "facile")

On obtient ainsi la courbe suivante montrant le nombre de comparaison en fonction de la taille du tableau:



Pas de surprise, comme vu en théorie, le nombre de comparaison est linéaire et est en $\mathcal{O}(n)$. La pratique confirme bien la théorie, ouf ...

Ensuite, nous affichons les temps d'exécution pour ces mêmes tableaux triés:



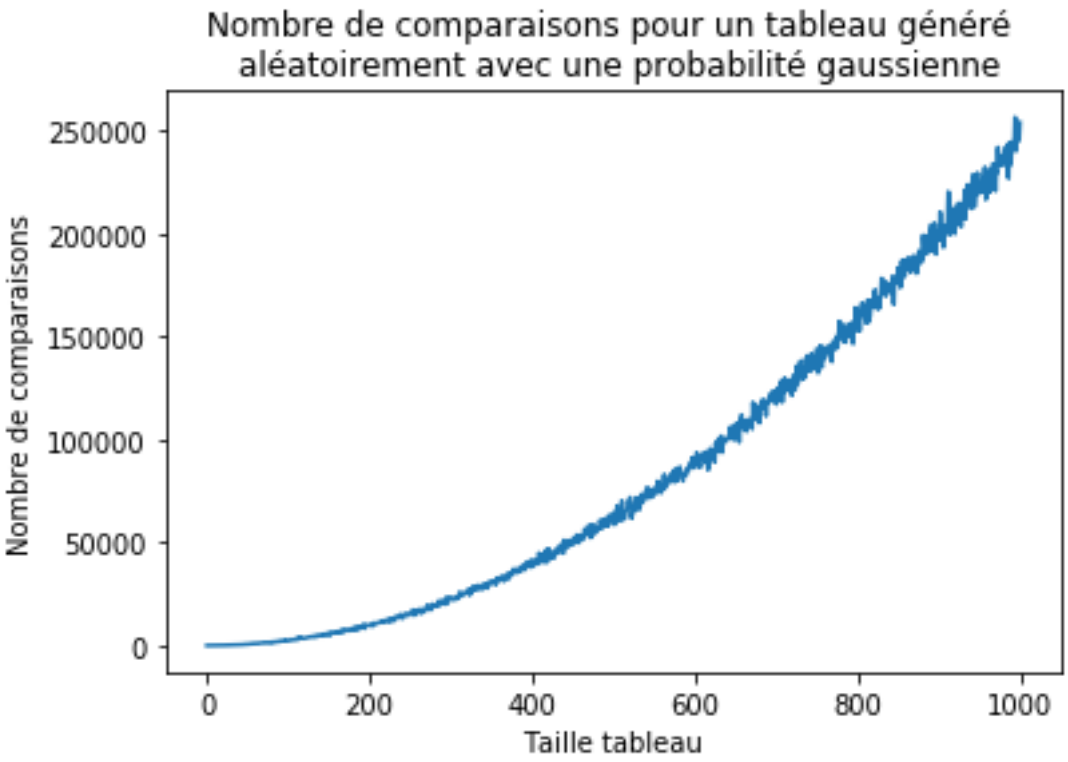
On obtient ici quelque chose d'intéressant: on peut remarquer que la courbe semble linéaire mais est complètement irrégulière. Il semblerait que pour certain tableau, le temps d'exéctution (réel) mis par la fonction soit des fois bien plus long que "la normale". Cette différence (qui peut s'avérer "énorme" dans certain cas) est surement du au fait que mon processeur avait d'autres

chats à fouéter à ce moment là. Cette théorie s'est confirmée lorsque j'ai relancé plusieurs fois ce petit programme. J'obtenais des résultats bien différents et les "pics" étaient déplacés.
Cependant, les ordres de grandeurs sont très petits (de l'ordre de la microseconde) et le processeur (i5 ici) semble se débrouiller.

En conclusion de cette partie on peut donc dire que les résultats obtenus sont en cohérence avec la théorie et que les temps d'exécution (réels) dépendent fortement des programmes en cours lancés sur la machine.

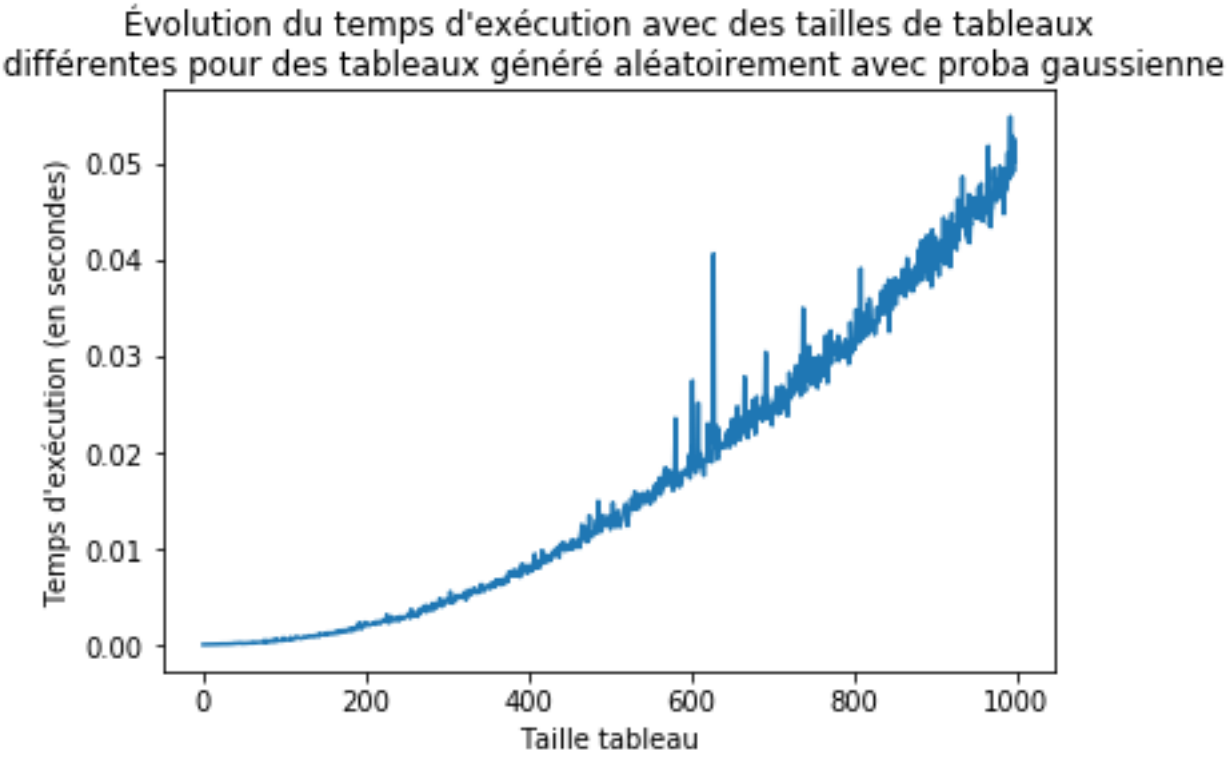
4.2 Cas du tableau "gaussien" (cas "moyen")

Nous allons réaliser la même démarche qu'avec la partie précédente donc pas beosin de se répeter.
On a ci-dessous, un graphique montrant le nombre de comparaisons pour des tableaux de différentes tailles générés avec une distribution gaussienne. La distance moyenne entre les éléménts est de 20.



On peut effectuer plusieurs commentaires sur ce graphique. Premièrement, on peut "voir" que la courbe semble approximer une fonction $f : x \mapsto ax^2$ où $a \in \mathbb{R}_+^*$. On peut aussi remarquer que la courbe "s'élargit" quand la taille des tableaux augmente. Cela est du au fait que les valeurs sont générés aléatoirement et donc que des fois les tableaux sont "un peu" trié.
Quoi qu'il en soit; on avait déterminé dans la théorie (en cours), qu'un tableau d'éléments distribués aléatoirement uniformément était en $\mathcal{O}(n^2)$ et on peut donc le confirmer ici. Il faut beaucoup plus d'opérations qu'avec un tableau déjà trié.

En ce qui concerne les temps d'exécution, nous obtenons le résultat suivant:

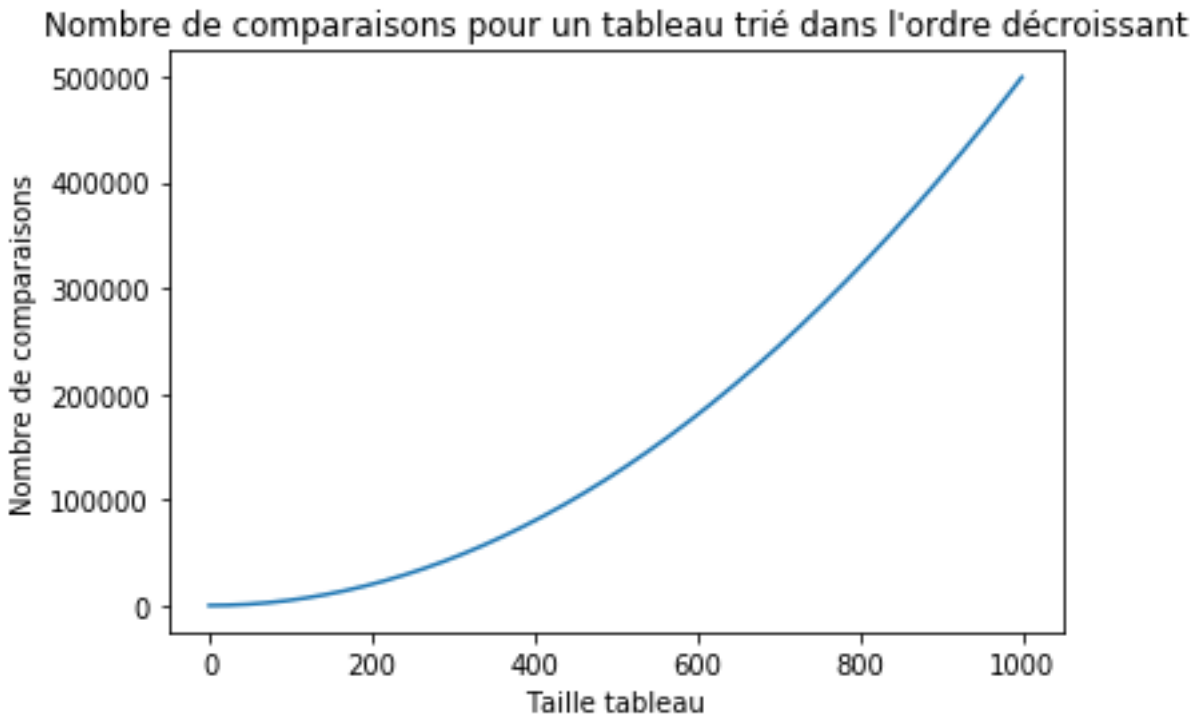


Le temps d'exécution semble lui aussi être en $\mathcal{O}(n^2)$ mais plus choquant encore est la valeur de ces temps d'exécution ! Contrairement au cas précédent, la courbe semble être "presque" parfaite. On peut remarquer quelques cas particulier dans les alentours des tailles 600 où les temps d'exécution ne suivent pas la courbe. Cela, comme le cas précédent est du au processus lancés sur ma machine. on peut voir que pour un tableau de 1000 éléments on arrive à un temps d'exécution de 0.05 secondes ce qui est énorme.

On peut conclure sur cette partie que le nombre de comparaisons et le temps d'exécution (réel sur cette machine) sont en $\mathcal{O}(n^2)$ comme nous avions prédit en théorie.

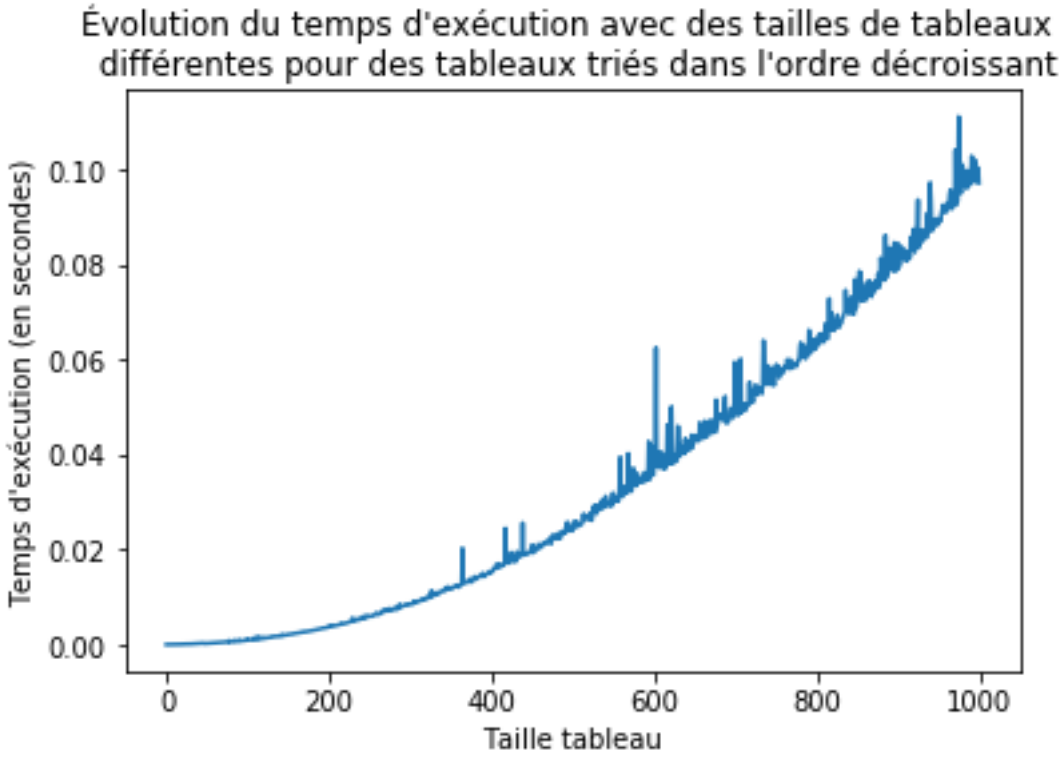
4.3 Cas du tableau trié dans l'ordre décroissant

Pour finir, nous étudions le cas d'un tableau trié dans l'ordre décroissant. Le nombre de comparaisons est donné ci-dessous:



Comme dans le cas précédent, on peut voir que la courbe est en $\mathcal{O}(n^2)$. Cela est sans surprise pour nous car nous l'avions prouvé dans la partie théorique. Cependant, contrairement au cas précédent, le coefficent b de la fonction $g : x \mapsto bx^2$ semble plus élevé. On peut même conjecturer que $b = 2 \times a$.
Quoi qu'il en soit, le nombre de comparaisons est bien plus grand que le premier et deuxième cas.

Intéressons nous au temps d'exécution maintenant:



À première vue, nous pouvons remarquer que la courbe est aussi en $\mathcal{O}(n^2)$. La répartition des temps semble lisse mais des cas particuliers apparaissent. Sur ce dernier cas, on peut remarquer que les temps d'exécution sont (encore) plus grands que les cas précédants.
On arrive pour une taille de tableau de 1000 éléments à un temps réel de 0.1 secondes.
Coïncidence ou pas, mais le coefficient de la courbe pour ce cas semble aussi être 2 fois plus grand que le cas précédant.
En réalité, nous avions démontré qu'il ne s'agissait pas que d'une coïcidence est que le nombre de comparaisons était à peu près deux fois plus grand pour ce cas que le cas précédant... Encore une fois la pratique confirme la théorie... ouf.

En conclusion de cette partie, on peut dire que le nombre de comparaisons et le temps réel d'exécution sont en $\mathcal{O}(n^2)$ et que la théorie est bien confirmée par la pratique.

5 Conclusion générale

En conclusion, on peut voir que le tri par insertion aura (sauf sans un seul cas particulier) un nombre de comparaisons et un temps d'exécution en $\mathcal{O}(n^2)$.
Le choix d'une distribution de probabilité gaussienne pour le cas dit "moyen" permettait de représenter des cas "réels" car la plupart des représentations de données suivent cette densité de probabilité.
Bien évidement, le temps d'exécuion sur ma machine dépend de pleins d'autres paramètres que je n'ai pas bien détaillés cependant, ils peuvent être négligés dans cette étude.
Étant donné que tous les tests se sont fait sur des tailles d'échantillons finis, nous ne pouvons pas voir de résultats asymptotiques, cependant, avec un peu d'imagination on se rend compte de la "réalité".
Certaines parties du code source n'ont pas été beaucoup détaillés et commentés mais je compte sur la bonne compréhension du lecteur.