Algorithmique - ROB3 - TP1
Algorithmes de tri

Le but de ce TP est d'implémenter plusieurs algorithmes de tri d'un tableau d'entiers, et de comparer expérimentalement ces algorithmes. Chaque algorithme prendra en argument un pointeur vers un tableau *tab* d'entiers et un entier n représentant le nombre d'éléments du tableau. L'algorithme ne retournera rien mais triera le tableau *tab*. Vous donnerez la complexité théorique de chaque algorithme, puis, pour chaque valeur de n, vous exécuterez votre algorithme sur des tableaux contenant n nombres aléatoires et vous tracerez une courbe représentant son temps de d'exécution en fonction de n.

Exercice 1 (Initialisation des tableaux)

Ecrivez une fonction prenant en argument un pointeur vers un tableau *tab* d'entiers, un entier n représentant la taille de *tab*, et deux entiers B_{min} et B_{max}. Cette fonction doit initialiser le tableau *tab* avec des valeurs aléatoires comprises entre B_{min} et B_{max}.

Exercice 2 (Tri par insertion)

Codez l'algorithme de tri par insertion vu en cours. Rappelez sa complexité théorique et tracez sa courbe de temps d'exécution.

Exercice 3 (Tri fusion)

Codez l'algorithme de tri fusion vu en cours. Déterminez sa complexité théorique en utilisant le théorème maître (vous indiquerez la relation de récurrence sur laquelle vous vous fondez), et tracez sa courbe de temps d'exécution.

Exercice 4 (Tri par énumération)

Le principe de cet algorithme repose sur la construction de l'histogramme des données, puis le balayage de celui-ci de façon croissante, afin de reconstruire les données triées. L'algorithme commence par parcourir le tableau de façon à connaître le plus grand B_{max} et le plus petit B_{min} des entiers de *tab*. Il crée ensuite un tableau *histogramme* de taille $B_{max} - B_{min} + 1$ dans lequel $histogramme[i]$ contiendra le nombre de fois que la valeur $B_{min} + i$ apparaît dans *tab*. Si B_{max} et B_{min} sont des constantes, quelle est la complexité de cet algorithme ? Codez cet algorithme et comparez son temps d'éxecution aux autres algorithmes de tri.

Exercice 5 (Tri par base)

Le tri par base est un algorithme de tri qui trie les nombres de *tab* en examinant individuellement les chiffres dont sont composés ces nombres. Il compare étape par étape les chiffres à une même position des nombre à trier, en examinant ces positions de la moins significative (chiffre des unités) à la position la plus significative. Ce tri a été introduit en 1887 par Herman Hollerith et a été utilisé pour trier des cartes perforées.

Une façon simple d'implémenter ce tri est la suivante. Le processus suivant est répété un nombre de fois égal au nombre de chiffres du plus grand nombre du tableau *tab*.

— Les nombres sont placés dans un tableau de 10 cases : chaque case i contient une file contenant tous les nombres dont le chiffre à la position étudiée est égal à i.
Par exemple, si le tableau initial est $[170, 045, 075, 090, 002, 024, 802, 066]$, le tableau à l'issue de la première itération serait :

```
0: 170, 090
1: vide
2: 002, 802
3: vide
4: 024
5: 045, 075
6: 066
7 - 9: vide
```

— Les files sont transformées en un tableau d'entiers, en respectant l'ordre des éléments dans les files. Avec l'exemple précédent, le tableau serait alors $[170, 090, 002, 802, 024, 045, 075, 066]$ à l'issue de la première itération.

Ce processus est répété pour chaque position. Lors de la deuxième itération par exemple, on s'intéresse au chiffre des dizaines, et les files pour l'exemple précédent sont les suivantes :

```
0: 002, 802
1: vide
2: 024
3: vide
4: 045
5: vide
6: 066
7: 170, 075
8: vide
9: 090
```

Le tableau à l'issue de la deuxième itération est le suivant : $[002, 802, 024, 045, 066, 170, 075, 090]$. Enfin, lors de la troisième itération, les files sont :

```
0: 002, 024, 045, 066, 075, 090
1: 170
2 - 7: vide
8: 802
9: vide
```

et le tableau est $[002, 024, 045, 066, 075, 090, 170, 802]$ (il est trié).

Implémentez le tri par base. Quelle est sa complexité ? Comparez expérimentalement cet algorithme aux algorithmes codés auparavant.

Exercice 6 (Horse-racing Duals)

Rendez-vous sur le site www.codingame.com et créez un compte. Ce site permet de s'entrainer à coder en proposant de multiples petits défis et concours. Vous êtes encouragé(e)s à vous exercer dessus afin d'améliorer vos talents de codeurs en langage C. Cependant, il ne permet pas de développer deux compétences importantes pour les TPs d'algorithmique : 1) mener des études expérimentales complètes et précises et 2) gérer de manière rigoureuse des projets informatiques.

Une fois inscrit(e) sur le site, réalisez les opérations suivantes :

- CLiquez sur ENTRAÎNEMENT dans l'onglet "ACTIVITÉ".
- Dans la section "PUZZLE CLASSIQUE - FACILE" cherchez et cliquez sur le puzzle "HORSE RACING DUALS".
- Cliquez sur le bouton "RÉSOUDRE" en haut à droite. Un éditeur de texte s'ouvre.
- Sélectionnez le langage de programmation C.

Pour résoudre le puzzle "HORSE RACING DUALS" vous devez coder votre réponse dans l'éditeur de texte qui vient de s'ouvrir. La description du puzzle est présente à gauche de votre écran. Nous vous proposons de résoudre ce puzzle en stockant dans un tableau la *puissance* de chaque cheval et en triant le tableau obtenu. Comment la réponse au défis peut être obtenue facilement à partir du tableau trié ? Quel tri utilisez-vous et pourquoi ?

Une fois votre réponse codée et le jeu de test passé avec succès, soumettez votre réponse sur le site.

Annexe : mesure d'exécution du temps d'un algorithme et courbes

Pour mesurer le temps mis par le CPU pour effectuer l'appel de la fonction `ma_fonction`, on peut utiliser le code suivant où `temps_cpu` contient le temps CPU utilisé pour l'exécution de `ma_fonction` en secondes.

```
# include < time.h >
...
clock_t temps_initial ; /* Temps initial en micro-secondes */
clock_t temps_final ;    /* Temps final en micro-secondes */
float temps_cpu ; /* Temps total en secondes */
...
temps_initial = clock () ;
ma_fonction() ;
temps_final = clock () ;
temps_cpu = ( temps_final - temps_initial ) * 1e -6;
printf ( "%d %f " , n , temps_cpu ) ;
```

Supposons que l'on veuille visualiser le temps nécessaire à l'exécution d'une fonction en fonction de la taille n de ses données (n peut par exemple être le nombre d'éléments d'un tableau à trier si l'on souhaite tester un algorithme de tri). Pour cela, on répète le code précédent pour chaque fonction étudiée, pour différentes valeurs de n (par exemple des valeurs allant de 100 à 50 000 avec un pas de 5 000). Pour une même valeur de n il est préférable de faire plusieurs mesures et de prendre la moyenne de ces mesures.
On trace ensuite les courbes représentant les séries de nombres obtenues. On peut utiliser pour cela un logiciel extérieur lisant un fichier texte créé par le programme. On utilisera dans ce TP le logiciel gnuplot qui est utilisable en ligne sous linux. Le fichier d'entrée de gnuplot est un fichier texte contenant n lignes, chaque ligne étant composée de la taille d'instance (n) suivie du temps d'exécution de chacune des x fonctions étudiées.
On peut alors lancer gnuplot et taper interactivement les commandes. On peut également utiliser une redirection gnuplot $<$ commande.txt avec un fichier du type

```
plot "01_sortie_vitesse.txt" using 1:2 with lines
replot "01_sortie_vitesse.txt" using 1:3 with lines
replot "01_sortie_vitesse.txt" using 1:4 with lines
set term postscript portrait
set output "01_courbes_vitesse.ps"
set size 0.7, 0.7
replot
```

Les lignes de commande ci-dessus permettent de tracer sur le même graphique les courbes d'éxécution de trois algorithmes résolvant le même problème. Elles créent le fichier postscript (.ps) contenant les trois courbes sur un même graphique. Attention à l'échelle, il est possible que les courbes nécessitent deux dessins pour leurs visualisations. Vérifiez que les courbes de temps obtenues sont cohérentes avec la complexité théorique de vos fonctions.