Contrôle continu de travaux pratiques n° 1

Travail par binômes – 3 TP encadrés – Remise : lundi 7 novembre 2016

Le but de ce projet est d'étudier différents algorithmes répondant au problème d'identification de l'élément le plus fréquent dans un tableau. On s'intéressera à des tableaux de caractères, et on lèvera les éventuels ex-æquo en choisissant toujours le caractère le plus petit.

Ce travail devra être réalisé dans un unique fichier source CPP, proprement commenté et bien testé. Les analyses théoriques, résultats pratiques et comparaisons demandés seront reportés dans un rapport PDF d'une dizaine de pages.

Outillage

On va considérer des tableaux de caractères de longueur $Taille$ entièrement remplis, définis ainsi en C++ :

```
const size_t Taille = ...;
typedef char Tabchar[Taille];
```

Votre premier travail consiste à écrire un ensemble de sous-algorithmes outils :

1. une procédure `affiche(const Tabchar & t)` qui affiche les éléments du tableau t en une seule ligne ;
2. une procédure `aleatoire(Tabchar & t, char inf, char sup)` qui remplit le tableau t avec des caractères aléatoires entre inf et sup en utilisant la fonction `rand` (cf. TP2).
3. une fonction `size_t occurrences(Tabchar & t, char c)` qui retourne le nombre d'occurrences du caractère c dans le tableau t.

Méthode par comptage

Une première approche simple du problème consiste à compter le nombre d'occurrences de chaque élément du tableau. L'algorithme qui en découle est le suivant :

```
fonction frequenceComptage(ⓓ Tabchar tab) : Caractère
variables
    Entiers occ, max, i
    Caractère res
début
1   max ← 0
2   res ← tab[1]
3   pour i de 1 à Taille faire
4       occ ← occurrences(tab,tab[i])
5       si (occ>max ou (occ=max et res>tab[i])) alors
6           max ← occ
7           res ← tab[i]
        finsi
    finp
8   retourner res
fin
```

L'ordre de complexité temporelle de cette méthode est $\Theta(Taille^2)$. Elle n'emploie pas de mémoire auxiliaire.

Méthode par tri

Une autre stratégie consiste à réaliser d'abord le tri du tableau d'entrée, ce qui a pour effet de regrouper toutes les occurrences d'un même caractère et permet alors leur comptage de façon efficace. Cela nécessite cependant de passer le tableau par modification puisqu'il faut le trier. L'algorithme résultant est le suivant :

```
fonction frequenceTri(ⓜ Tabchar tab) : Caractère
variables
    Entiers occ, max, i
    Caractères c, res
début
01 tri(tab)
02 max ← 0
03 res ← tab[1]
04 occ ← 0
05 c ← tab[1]
06 pour i de 1 à Taille faire
07     si (t[i] = c) alors
08         occ ← occ+1
       sinon
09         si (occ>max) alors
10             max ← occ
11             res ← c
           finsi
12         c ← tab[i]
13         occ ← 1
       finsi
   finp
14 si (occ > max) alors
15     res ← c
   finsi
16 retourner res
fin
```

Il utilise la procédure `tri` qui réalise le tri du tableau `tab`. Le choix de la procédure à utiliser est laissé libre. Il pourrait être intéressant de comparer deux algorithmes différents aux propriétés variées, e.g., tri par insertion ou permutation d'une part, et tri par fusion ou pivotage d'autre part.

L'ordre de complexité temporelle de cette méthode est au mieux $\Omega(Taille)$ et au pire $\mathcal{O}(Taille^2)$: cela dépend du tri utilisé, aussi faudra-t'il préciser ces bornes en fonctions de la (ou des) procédure(s) choisie(s). Elle emploie potentiellement une mémoire auxiliaire si le tri utilisé n'est pas en place.

Méthode par énumération

Sous l'hypothèse qu'on connaît l'ensemble des caractères pouvant être présents dans le tableau, et que ce nombre est petit devant $Taille$, une troisième stratégie très efficace est applicable. On l'utilisera ici pour des tableaux remplis uniquement de lettres minuscules. Elle consiste à comptabiliser, pour chaque caractère possible, son nombre d'occurrences puis à retourner celui qui est le plus fréquent. L'algorithme correspondant est le suivant :

```
fonction frequenceRapide(ⓓ Tabchar tab) : Caractère
variables
    Tableau de 26 Entiers freq
    Entiers max, i, j
    Caractères res
début
01 pour i de 1 à 26 faire
02     freq[i] ← 0
   finp
03 pour i de 1 à Taille faire
04     j ← ord(tab[i]) − ord('a') + 1
05     freq[j] ← freq[j] + 1
   finp
```

```
06 max ← 0
07 pour i de 1 à 26 faire
08     si (freq[i]>max) alors
09         max ← freq[i]
10         res ← car(ord('a')+i-1)
       finsi
   finp
11 retourner res
fin
```

L'ordre de complexité temporelle de cette stratégie est $\Theta(Taille)$. Elle emploie en revanche une mémoire auxiliaire proportionnelle au nombre de caractères différents possibles, ici 26.

Travail demandé

Pour chacune des méthodes proposées :

1. Implémentez-la en C++.
2. Donnez sa spécification complète en commentaires `Doxygen` dans votre programme (cf. TP1).
3. Définissez un jeu de tests représentatif pour des tableaux de $Taille = 3$; indiquez-le en commentaires Doxygen et implémentez en C++ une procédure de tests unitaires qui exécute ce jeu de tests (cf. TP1).
4. Démontrez son ordre de complexité temporelle et identifiez les formes des données associées aux meilleur et pire cas (⇒ rapport).
5. Démontrez son ordre de complexité spatiale et identifiez les formes des données associées aux meilleur et pire cas (⇒ rapport).
6. Après avoir équipé son implémentation d'un chronomètre, étudiez sa complexité temporelle pratique pour des tableaux de $Taille = 10, 100, 1\,000, 10\,000, \ldots$ (cf. TP2). Reportez les mesures en meilleur cas, pire cas, et en moyenne (statistique) dans des tableaux et en tirer des graphiques (⇒ rapport).
7. Comparez ses résultats pratiques à sa complexité temporelle théorique (⇒ rapport).

L'analyse de la méthode par tri pour différentes procédures de tri donnera lieu à des points de bonus.