TD 4 de Programmation - Eléments de correction

L2.1 MathInfo

Exercice 3 - Tri par distribution

Soit `Tab` un tableau de n caractères entre `A` et `Z`. Pour trier le tableau selon l'ordre lexicographique, on utilise un tableau auxiliaire `Aux` d'entiers tel que `Aux[i]` indique le nombre de fois que le i-$^{\text{ième}}$ caractère apparaît dans le tableau `Tab`.

1. Pour remplir le tableau `Aux`, on utilise de code ASCII des caractères. Pour cela on a besoin de convertir un caractère en son code ASCII et inversement. On le fait en forçant le type des variables : par exemple dans le code suivant, la variable entière `a` reçoit le code ASCII du caractère `'A'` (c'est-à-dire 65), et la variable de type caractère `c` reçoit le caractère dont le code ASCII est 72 (c'est-à-dire H).

   ```
   int a;
   char c
   a=(int) 'A';
   c=(char) 72;
   ```

   Ceci nous permet de remplir le tableau `Aux`. On commence par le créer en le déclarant comme un tableau d'entiers de taille 26, dont toutes les cases contiennent la valeur 0. Puis on parcourt le tableau `Tab` et pour chaque caractère `Tab[i]` rencontré, on le convertit en position dans l'alphabet (`'A'` est en position 0, `'B'` en position 1, etc ...) grace à l'expression `((int) Tab[i])-((int) 'A')`, et on incrémente la valeur dans le tableau `Aux` qui correspond à ce numéro.

   Pour trier le tableau `Tab`, il suffit de le modifier en respectant les occurences de chaque caractère : les `Aux[0]` premières cases de `Tab` seront des `'A'`, les `Aux[1]` suivantes des `'B'`, etc...

   ```
   void tri_distrib(char Tab[], int taille)
   {
      int i,j;
      int k=0;
      int Aux[26]={0};
      for(i=0;i<taille;i++)
         Aux[((int) Tab[i])-((int) 'A')]++;

      for(i=0;i<26;i++)
         for(j=0;j<Aux[i];j++)
            {
            Tab[k]= (char) (i+((int) 'A'));
            k++;
            }
   }
   ```

2. Si on appelle n la taille du tableau `Tab`, la complexité de cet algorithme est en $\mathcal{O}(2*n+2*26) = \mathcal{O}(n)$ en temps: on parcourt `Tab` une fois pour compter le nombre d'occurences de chaque caractères, et une deuxième fois pour le remplir avec les caractères triés; le tableau auxiliaire `Aux` est parcouru deux fois. En ce qui concerne la complexité en espace, on utilise un tableau auxiliaire d'entiers, de taille fixée (ici 26 pour les 26 lettres de l'alphabet). La complexité en espace est donc en $\mathcal{O}(1)$.

3. Le tri par distribution repose sur le principe suivant : si on veut trier un tableau de taille n qui contient des variables pouvant prendre C valeurs différentes, et que l'on connaît à l'avance quelles sont ces valeurs, alors on peut trier ce tableau en temps $\mathcal{O}(2*n+2*C)$ et en espace $\mathcal{O}(C)$. Pour que ce tri puisse être utilisé, il faut donc que les données à trier prennent des valeurs connues et en petit nombre par rapport à la taille du tableau ($C \in \mathcal{O}(n)$). Par exemple, ce tri peut être utilisé pour trier un tableau d'entiers compris entre 0 et 10, mais on ne peut pas l'utiliser si on ne sait rien a priori sur la taille des entiers. D'autre part, il est inintéressant de trier par distribution des tableaux d'entiers dont on sait que les éléments sont compris entre 0 et 10000 si la taille de ces tableau est 100...

Exercice 5 - Tri de Shell

1. On sait que le tri est insertion est efficace lorsque le tableau de départ est déjà presque trié, mais il est en moyenne peu efficace car il n'échange que des éléments consécutifs dans le tableau. Le tri de Shell est une amélioration du tri par insertion. Le principe est en le suivant: on applique le tri par insertion à des sous-tableaux dont les éléments sont à distance h_n les uns des autres. L'entier h_n est appelé la période et diminue au cours de l'algorithme jusqu'à valoir 1. Lorsque la période 1 est atteinte, on effectue un tri par insertion classique et les traitements précédents font qu'à cet instant, le tableau est presque trié (d'où l'efficacité du tri par insertion).

   Dans cet exercice on utilise la suite de périodes définie par $h_1 = 1$ et $h_n = 2 * h_{n-1}$, mais on peut optimiser le temps d'exécution en utilisant d'autres suites de périodes, comme celle proposée dans l'énoncé.

   On applique le principe de l'algorithme à l'exemple. Le tableau comprenant 9 éléments, la première période qui nous intéresse est $h = 8$. A chaque étape le sous-tableau sur lequel on applique le tri par insertion est grisé.

6	5	4	9	1	7	8	3	2
3	5	4	9	1	7	8	6	2
3	2	4	9	1	7	8	6	5
3	2	4	9	1	7	8	6	5
3	2	4	9	1	7	8	6	5
3	2	4	9	1	7	8	6	5
3	2	4	9	1	7	8	6	5
3	2	4	9	1	7	8	6	5

   On recommence ensuite avec $h = 4$.

3	2	4	9	1	7	8	6	5
3	2	4	8	1	7	9	6	5
3	1	4	8	2	7	9	6	5
3	1	4	8	2	5	9	6	7

   Puis avec $h = 2$.

3	1	4	8	2	5	9	6	7
2	1	3	8	4	5	7	6	9
2	1	3	5	4	6	7	8	9

   Et enfin avec $h = 1$.

2	1	3	5	4	6	7	8	9
1	2	3	4	5	6	7	8	9

2. On commence par chercher quelle sera la première période du tri de Shell. Cette période est stockée dans la variable `h`. Puis on trie par insertion tous les sous-tableaux de période `h`, et on recommence avec $\frac{h}{2}$.

```
void tri_shell(int t[], int taille)
{
     int h, i, j, pivot;

     h=1;

     while(h<taille)
         {
         h=2*h;
         }

     while(h!=0)
         {
         h=h/2;
         for (i=h;i<taille;i++)
             {
             pivot=t[i];
             j=i;

             while((j>(h-1)) && (t[j-h]>pivot))
                 {
                 t[j]=t[j-h];
                 j=j-h;
                 }
             t[j]=pivot;
             }
         }
}
```