IN 101 - TD 05
15 octobre 2010

Corrigé

ENSTA ParisTech

Solution 1 : Tri à bulles et variantes

1.a] Voici le code du tri à bulles le plus simple :

```
1  void bubble(int n, int* tab) {
2    int i,j,tmp;
3    for (i=0; i<n-1; i++) {
4      for (j=0; j<n-1-i; j++) {
5        if (tab[j] > tab[j+1]) {
6          tmp = tab[j];  tab[j] = tab[j+1];  tab[j+1] = tmp;
7        }
8      }
9    }
10 }
```

1.b] Pour faire l'initialisation et l'affichage :

```
1  #include <stdio.h>
2  #include <time.h>
3  #include <stdlib.h>
4
5  void print_tab(int n, int* tab) {
6    int i;
7    for (i=0; i<n; i++) {
8      printf("%d ",tab[i]);
9    }
10   printf("\n");
11 }
12
13 void tab_is_sorted(int n, int* tab) {
14   int i;
15   for (i=0; i<n-1; i++) {
16     if (tab[i] > tab[i+1]) {
17       printf("Tableau mal trié !\n");
18       return;
19     }
20   }
21   printf("Tableau trié.\n");
22 }
23
24 int main(int argc, char** argv) {
25   int n,i;
26   if ((argc != 2) || ((n=atoi(argv[1])) == 0)) {
27      printf("Le programme prend en argument un entier non nul.\n");
28      return 2;
```

```
29   }
30   srand(time(NULL));
31   int* tab = (int*) malloc (n*sizeof(int));
32   for (i=0; i<n; i++) {
33     tab[i] = rand() % 256;
34   }
35   print_tab(n,tab);
36   bubble(n,tab);
37   print_tab(n,tab);
38   tab_is_sorted(n,tab);
39   return 0;
40 }
```

1.c] Le tri à bulles avec arrêt anticipé :

```
1  void bubble_stop(int n, int* tab) {
2    int i,j,tmp;
3    int stop;
4    i = 0;
5    do {
6      stop = 1;
7      for (j=0; j<n-1-i; j++) {
8        if (tab[j] > tab[j+1]) {
9          tmp = tab[j];  tab[j] = tab[j+1];  tab[j+1] = tmp;
10         stop = 0;
11       }
12     }
13     i++;
14   } while (!stop);
15 }
```

1.d] Le tri cocktail de base :

```
1  void cocktail(int n, int* tab) {
2    int i,j,tmp;
3    int stop;
4    i = 0;
5    do {
6      stop = 1;
7      for (j=0; j<n-1-i; j++) {
8        if (tab[j] > tab[j+1]) {
9          tmp = tab[j];  tab[j] = tab[j+1];  tab[j+1] = tmp;
10         stop = 0;
11       }
12     }
13     if (stop) {
14       break;
15     }
16     stop = 1;
17     for (j=n-2-i; j>i; j--) {
18       if (tab[j-1] > tab[j]) {
19         tmp = tab[j];  tab[j] = tab[j-1];  tab[j-1] = tmp;
```

```
20          stop = 0;
21        }
22      }
23      i++;
24    } while (!stop);
25 }
```

1.e] Le tri cocktail avec les max :

```
1  void cocktail_minmax(int n, int* tab) {
2    int j,tmp,last;
3    int min = 0;
4    int max = n-1;
5    int stop;
6    do {
7      stop = 1;
8      for (j=min; j<max; j++) {
9        if (tab[j] > tab[j+1]) {
10         tmp = tab[j];  tab[j] = tab[j+1];  tab[j+1] = tmp;
11         stop = 0;
12         last = j;
13       }
14     }
15     max = last;
16     if (stop) {
17       break;
18     }
19     stop = 1;
20     for (j=max; j>min; j--) {
21       if (tab[j-1] > tab[j]) {
22         tmp = tab[j];  tab[j] = tab[j-1];  tab[j-1] = tmp;
23         stop = 0;
24         last = j;
25       }
26     }
27     min = last;
28   } while (!stop);
29 }
```

1.f] Pour compter les comparaisons il suffit d'ajouter dans chaque fonction un compteur `c` initialisé à 0, que l'on incrémente avant chaque test `if` et que l'on retourne à la fin. On ajoute ensuite une boucle `for` dans le `main` pour pouvoir faire des statistiques.

Le nombre moyen (calculé sur un échantillon de 10000 tableaux) de comparaisons pour un tableau aléatoire de taille 100 (avec des éléments entre 0 et 255) est :

`bubble`	4950 comparaisons
`bubble_stop`	4875 comparaisons
`cocktail`	3886 comparaisons
`cocktail_minmax`	3379 comparaisons

Solution 2 : Tri fusion

2.a] Voici le code de la fonction `merge`.

```
1  void merge(int* tab, int p, int q, int r) {
2    int i,j,k;
3    int* tmp = (int*) malloc((r-p) * sizeof(int));
4    for (i=p; i<q; i++) {
5      tmp[i-p] = tab[i];
6    }
7    for (j=q; j<r; j++) {
8      tmp[r-p+q-j-1] = tab[j];
9    }
10   i=0;
11   j=r-p-1;
12   for (k=p; k<r-1; k++) {
13     if (tmp[i]<tmp[j]) {
14       tab[k] = tmp[i];
15       i++;
16     } else {
17       tab[k] = tmp[j];
18       j--;
19     }
20   }
21   // à cette étape, i==j, donc on n'a pas de comparaison à faire
22   tab[r-1] = tmp[i];
23   free(tmp);
24 }
```

Allouer la mémoire à chaque appel de la fonction `merge` est en fait assez coûteux et il serait préférable d'allouer un tableau de la taille de `tab` une fois pour toute au début du tri et d'utiliser ce même tableau pour toutes les opérations de fusion des tableaux.

2.b] Quel que soit l'ordre des éléments du tableau d'entrée le tri fusion fait toujours le même nombre de comparaisons. Pour un tableau de taille 100, il fait toujours exactement 573 comparaisons (ou 672 si votre tri n'économise pas la dernière comparaison).

2.c] Si on coupe le tableau en k à chaque étape, la fonction `merge` doit maintenant prendre en argument k tableaux (soit $k+1$ indices) et devra utilise k compteurs différents. À chaque étape il faut extraire le plus petit de k éléments et incrémenter le compteur de cet élément. Extraire le plus petit élément d'un ensemble de k coûte $\Theta(k)$, et donc, la complexité totale de l'algorithme est $\Theta(kn \times \log_k n) = \Theta(kn \times \frac{\log n}{\log k})$. Le meilleur choix est donc $k = 2$.

Pour améliorer la complexité du tri fusion d'ordre k, on peut aussi trier les k premiers éléments des k tableaux que l'on est en train de fusionner, et à chaque fois extraire le plus petit et insérer un nouvel élément. En utilisant une structure de donnée adéquate (soit un arbre binaire de recherche équilibré, soit un tas, comme nous le verrons au cours 12), cela coûte $\Theta(\log k)$. La complexité du tri fusion est alors $\Theta(n \log n)$ quelle que soit la valeur de k, mais le choix $k = 2$ reste quand même le plus simple !

2.d] Voici le code des fonctions `merge_k` et `merge_sort_k` gérant le compteur `c` du nombre de comparaisons. Il s'agit de la version basique du tri qui cherche à chaque étape le min de k éléments :

```
1  int merge_k(int* tab, int* indices, int k) {
2    int i,j,c,min,argmin;
3    // on alloue un peu trop, mais cela simplifie les calculs d'indices
4    int* tmp = (int*) malloc(indices[k] * sizeof(int));
5    for (i=indices[0]; i<indices[k]; i++) {
6      tmp[i] = tab[i];
7    }
8    int* parcours = (int*) malloc(k*sizeof(int));
9    for (i=0; i<k; i++) {
10     parcours[i] = indices[i];
11   }
12   c = 0;
13   for (j=indices[0]; j<indices[k]; j++) {
14     i=0;
15     while (parcours[i] == indices[i+1]) {
16       c++; // on a fait une comparaison
17       i++;
18     }
19     min = tmp[parcours[i]];
20     argmin = i;
21     i++;
22     while (i<k) {
23       c++; // on va faire une comparaison
24       if ((parcours[i]<indices[i+1]) && (tmp[parcours[i]] < min)) {
25         min = tmp[parcours[i]];
26         argmin = i;
27       }
28       i++;
29     }
30     tab[j] = min;
31     parcours[argmin]++;
32   }
33   free(tmp); free(parcours);
34   return c;
35 }
36 int merge_sort_k(int* tab, int p, int r, int k) {
37   int q,c,i;
38   int* indices;
39   c=0;
40   if ((r-p) > 1) {
41     indices = (int*) malloc((k+1) * sizeof(int));
42     indices[0] = p;
43     for (i=1; i<k+1; i++) {
44       indices[i] = p + ((r-p)*i)/k;
45       c += merge_sort_k(tab, indices[i-1], indices[i], k);
46     }
47     c += merge_k(tab, indices, k);
48     free(indices);
49   }
50   return c;
51 }
```

Dans ces fonctions, le compteur `c` est incrémenté de façon à ce que le nombre de comparaisons soit toujours constant pour k et n donnés. Pour cela, il ne faut pas oublier le `c++` de la ligne 16.

Pour un tableau de 100 éléments, le nombre de comparaisons pour différentes valeurs de k est :

$k = 2$	672 comparaisons
$k = 3$	876 comparaisons
$k = 4$	1116 comparaisons
$k = 5$	1200 comparaisons
$k = 6$	1500 comparaisons
$k = 7$	1800 comparaisons
$k = 8$	1904 comparaisons
$k = 9$	1904 comparaisons
$k = 10$	1800 comparaisons

Notez que pour $k = 2$ cette version fait plus de comparaisons que la version non-généralisée : recopier les deux tableaux dos à dos permet d'économiser une comparaison à chaque fois (on ne fait pas de comparaison quand $i = j$). Au total, cela représente $n - 1$ comparaisons pour un tableau de taille n (ici, 99). Aussi, en fonction des arrondis, le nombre de comparaisons n'augmente pas strictement avec k, contrairement à ce que semble indiquer la complexité asymptotique de $\Theta(kn \times \frac{\log n}{\log k})$.