CS221 - TP1

Vincent MOUCADEAU - Rémi MAZZONE — 2A

23/11/2022

Table des matières

1 Introduction **2**

2 Préparation **2**
2.1 Pseudo code du tri "Bubble sort" 2
2.2 Makefile 2

3 Tri bulle **3**

4 Tri par insertion **4**

5 Tri fusion **5**

6 Conclusion **7**

1 Introduction

Dans ce TP, maintenant que nous avons pris en main les outils de bases, nous allons passer à l'étude d'un système un peu plus complexe, qui traduit un vrai problème. En effet, nous allons étudier le mouvement d'un pendule (sans frottements) avec deux approches différentes : nous utiliserons Simulink dans la première partie et uniquement Matlab dans la deuxième.

2 Préparation

2.1 Pseudo code du tri "Bubble sort"

Listing 1: Pseudo code du tri "Bubble sort" optimisé

```
input: int *tab, int n
output: int nb_swaps
nb_swaps = 0
pour i = 0 a n-1 faire
    bool swapped = false
    pour j = 0 a n-i-1 faire
        si tab[j] > tab[j+1] alors
            echanger tab[j] et tab[j+1]
            nb_swaps++
            swapped = true
        fin si
    fin pour
    si swapped == false alors
        retourner nb_swaps
    fin si
fin pour
```

2.2 Makefile

Listing 2: Makefile du projet

```
main:main.o fonctions.o
    gcc -o $@ $^

main.o: main.c fonctions.h
    gcc -c $<

fonctions.o: fonctions.c fonctions.h
    gcc -c $<

clean:
    rm -rf *.o
    rm -rf main
```

3 Tri bulle

1. Avec le makefile écrit précédemment, le programme se compile et s'exécute correctement. Le programme affiche bien le contenu du tableau `tab1`.

2. Implémentation de la fonction `swap` en C. On utilise une variable temporaire pour échanger les valeurs des deux variables.

Listing 3: Implémentation de la fonction swap

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

3. Implémentation du tri bulle en C. On utilise la fonction `swap` pour échanger les valeurs des deux variables. On utilise une variable `swapped` pour savoir si un échange a eu lieu. Si aucun échange n'a eu lieu, on peut arrêter le tri.

Listing 4: Implémentation optimisée du tri bulle

```c
int bubbleSort(int *tab, int n) {
    int nb_swaps = 0;
    int i, j;
    bool swapped;
    for (i = 0; i < n-1; i++) {
        trie = true;
        for (j = 0; j < n-i-1; j++) {
            if (tab[j] > tab[j+1]) {
                swap(&tab[j], &tab[j+1]);
                nb_swaps++;
                trie = false;
            }
        }
        if (trie) {
            break;
        }
    }
    return nb_swaps;
}
```

4. On vérifie que le tri fonctionne correctement à l'aide de la fonction `compare` qui compare élément par élément le tableau trié avec le tableau de référence. On affiche le nombre d'échanges effectués.

5. Résultats du tri bulle pour les tableaux donnés :

   - `tab1` : Comparaison OK, 682 échanges

- `tab2` : Comparaison OK, 1216 échanges
- `tab3` : Comparaison OK, 63 échanges
- `ref` : Comparaison OK, 0 échanges

6. Dans le pire des cas (tableau trié dans l'ordre décroissant) pour chaque élément du tableau, il y a $n-1$ comparaison. Donc le nombre d'opérations est de $n(n-1)$. Dans le meilleur des cas (tableau trié dans l'ordre croissant), il n'y a pas de comparaison. Donc le nombre d'opérations est de n. En résumé :

   - Pire des cas : $O(n^2)$
   - Moyenne : $O(n^2)$
   - Meilleur des cas : $O(n)$

   On peut dire que l'algorithme est adaptatif car on interrompt le tri dès qu'il n'y a plus d'échanges. Il s'adapte bien au tableau d'entrée, cela permet d'éviter des comparaisons inutiles et donc d'améliorer les performances dans certains cas.

7. Complexité spatiale : 2 variables temporaires, variable `nb_swaps` et variables de boucle. Donc $O(1)$, il ne nécessite pas de tableau temporaire. Il est stable car il ne modifie pas l'ordre des éléments égaux.

4 Tri par insertion

1. Implémentation du tri par insertion en C. On utilise une variable `nb_swaps` pour compter le nombre d'échanges effectués.

Listing 5: Implémentation du tri par insertion

```c
int tri_insertion(int *a, int n){
    int swap_number = 0;
    int i, j;
    for (i = 1 ; i < n ; i++){
        for (j = i ; j > 0 ; j--){
            if (a[j] < a[j-1]){
                swap(&a[j], &a[j-1]);
                swap_number++;
            }
        }
    }
    return swap_number;
}
```

2. Résultats du tri insertion pour les tableaux donnés :

   - `tab1` : Comparaison OK, 682 échanges
   - `tab2` : Comparaison OK, 1216 échanges

- `tab3` : Comparaison OK, 63 échanges
- `ref` : Comparaison OK, 0 échanges

3. Dans le pire des cas (tableau trié dans l'ordre décroissant) l'algorithme effectue de l'ordre de $\frac{n^2}{2}$ opérations. Dans le meilleur des cas (tableau trié dans l'ordre croissant), il y a $n-1$ comparaisons. Donc le nombre d'opérations est de au plus n.
   On remarque que la complexité de l'algorithme du tri par insertion est linéaire quand le tableau est presque trié. Il est même plus efficace que le tri fusion ou le tri rapide dans ce cas. En résumé :
   - Pire des cas : $O(n^2)$
   - Moyenne : $O(n^2)$
   - Meilleur des cas : $O(n)$
4. Complexité spatiale : 2 variables temporaires, variable `nb_swaps` et variables de boucle. Donc $O(1)$. L'algorithme ne nécessite pas de tableau temporaire. Il est stable car il ne modifie pas l'ordre des éléments égaux.

5 Tri fusion

1. Si on déclare le tableau `tmp` en écrivant `int *tmp = tab;` on ne crée pas un nouveau tableau mais on crée un pointeur qui pointe vers le même tableau que `tab`. Si on modifie `tmp` on modifie aussi `tab`. Donc on ne peut pas utiliser `tmp` pour trier le tableau.
   Si on déclare le tableau `tmp` en écrivant `int tmp[n];` on obtient une erreur de segmentation. En effet, la valeur de n doit être connue à la compilation
2. Pour résoudre ce problème, on peut utiliser la fonction `malloc` qui permet de créer un nouveau tableau. On peut aussi utiliser la fonction `memcpy` qui permet de copier un tableau dans un autre. Si on déclare le tableau `tmp` en écrivant `int *tmp = malloc(n * sizeof(int));` on crée un nouveau tableau de taille n et `tmp` pointera vers le premier élément de ce nouveau tableau. On peut donc utiliser `tmp` pour trier le tableau.
3. Fonction `merge` qui fusionne deux tableaux triés en un seul tableau trié. On utilise une variable `nb_swaps` pour compter le nombre d'échanges effectués.

Listing 6: Implémentation de la fonction `merge`

```
1 void merge (int *tab, int *tmp, int left, int mid, int right,
     int *cnt) {
2     int init_mid = mid;
3     bool end_left = false, end_mid = false;
4     for(int i = left; i < right; i++) {
5         if(!end_left && !end_mid) {
6             if(tab[left] < tab[mid]) {
7                 tmp[i] = tab[left];
8                 if(left < init_mid - 1) {
```

```
                    left++;
                }
                else {
                    end_left = true;
                }
            }
            else {
                tmp[i] = tab[mid];
                if(mid < right - 1) {
                    mid++;
                }
                else {
                    end_mid = true;
                }
            }
        }
        else if(end_left) {
            tmp[i] = tab[mid];
            if(mid < right - 1) {
                mid++;
            }
            else {
                end_mid = true;
            }
        }
        else if(end_mid) {
            tmp[i] = tab[left];
            if(left < init_mid - 1) {
                left++;
            }
            else {
                end_left = true;
            }
        }
    }
}
```

4. Fonction `tri_merge` qui utilise lance le tri fusion avec `merge_aux` pour trier le tableau. On utilise une variable `nb_swaps` pour compter le nombre d'échanges effectués.

Listing 7: Implémentation de la fonction `tri_merge`

```
int tri_merge(int *a, int n) {
    int swap_number = 0;
    int *tmp = malloc(n * sizeof(int));

    merge_aux(a, tmp, 0, n, &swap_number);

    for(int i = 0; i < n; i++) {
```

```
        a[i] = tmp[i];
    }
    free(tmp);

    return swap_number;
}
```

Listing 8: Implémentation de la fonction `merge_aux`

```
void merge_aux(int *tab, int *tmp, int left, int right, int *
   cnt) {
    int delta = right - left;
    if(delta > 1) {
        int mid = (left + right) / 2;
        tri_fusion(tab, tmp, left, mid, cnt);
        tri_fusion(tab, tmp, mid+1, right, cnt);
        merge(tab, tmp, left, mid, right, cnt);
    }
}
```

5. Résultats du tri fusion pour les tableaux donnés :
   - `tab1` : Comparaison OK, 682 échanges
   - `tab2` : Comparaison OK, 1216 échanges
   - `tab3` : Comparaison OK, 63 échanges
   - `ref` : Comparaison OK, 0 échanges
6. Complexité temporelle : $O(n \log(n))$
7. La complexité spatiale de cet algorithme est de $O(n)$. En effet, on utilise un tableau de taille n pour stocker les valeurs triées. On utilise aussi une variable `nb_swaps` et des variables de boucle. On remarque que la complexité spatiale est supérieure à celle de l'algorithme du tri par insertion (ou du tri bulle) qui est de $O(1)$. L'algorithme est stable car on ne modifie pas l'ordre des éléments qui sont égaux dans la fonction `merge`.

6 Conclusion

Lors de ce TP, nous avons pû découvrir trois algorithmes de tri différents. D'une part nous avons les algorithme du tri bulle et du tri par insertion, qui sont assez simples à comprendre mais pas très efficaces. D'autre part, nous avons pu voir que le tri fusion est plus efficace que les deux autres algorithmes car il a une complexité temporelle de $O(n \log(n))$ alors que les deux autres algorithmes ont une complexité temporelle de $O(n^2)$. Néanmoins, dans certains cas, le tri par insertion peut être plus efficace que le tri fusion (notamment pour des tableaux presques triés). Il faut donc choisir l'algorithme de tri en fonction du tableau à trier.