Cette *vingt-cinquième* colle vous fera travailler sur l'algorithme « Diviser pour Régner » du Tri Fusion, comme vu en cours vendredi 29/04 dernier, mais cette fois sur des tableaux (et pas des listes), et en C et pas en OCaml.

Exercice 0 : rappel de l'idée de algorithme

1. Rappeler au brouillon l'idée de l'algorithme du tri fusion.
   - Spécifier le(s) cas de base,
   - Expliquer comment faire la séparation d'une instance E en un nombre fixé $a \geqslant 1$ d'instances $E_1, \dots, E_a$ de tailles plus petites, et spécifier a ainsi que le facteur $b \geqslant 1$ de division de la taille $|E_i| \leqslant \lceil \frac{|E|}{b} \rceil$,
   - Expliquer comment faire la fusion, en temps linéaire dans la somme des tailles des entrées.

Exercice 1 : considérations théoriques sur l'algorithme

Illustration et exécution sur un exemple

2. Sur l'exemple de tableau suivant : `tab = [8; 1; 6; 3; 4; 7; 2; 5]`, illustrer proprement le fonctionnement du tri fusion, sur un dessin tenant sur plusieurs lignes.

Complexité temporelle

3. Si on note $T(n)$ la complexité temporelle du tri fusion appelé sur un tableau de taille n, donner une relation de récurrence satisfaite par $T(n)$, sous la forme d'une inégalité de récurrence reliant $T(n)$ à des valeurs de T plus petites et un terme en $\mathcal{O}(n^k)$, pour une certaine constante $k \in \mathbb{N}$ à spécifier.
4. Si on applique le « théorème maître » sur cette relation de récurrence, ou si on la résout à la main, quel résultat obtient-on pour une domination asymptotique de $T(n)$ en fonction de n ? On ne demande *pas* de justification.
5. Est-ce que le tri fusion est un tri asymptotiquement plus efficace que les algorithmes de tris suivants : tri à bulle, tri par sélection, tri par insertion ?
6. Citer au moins un autre exemple d'algorithme de tri qui soit asymptotiquement aussi efficace que le tri fusion.

Implémentation en C

On s'intéresse désormais à implémenter en C cet algorithme de tri fusion, sur des tableaux (de valeurs entières), et non plus sur des listes OCaml (simplement chaînées).

Ici, nous utiliserons des *variable-length array*, ou tableaux à tailles paramétriques, pour écrire des signatures de fonctions plus lisibles, par exemple `int* tri_fusion(int n, int tab[n])` pour préciser que `tab` est un tableau (à valeurs entières) de taille égale à `n`.

On compilera avec `COMPILATEUR = gcc` ou `clang`, et la ligne de commande suivante :

```
$ COMPILATEUR -O3 -Wall -Wextra -Werror -fsanitize=undefined
    -pedantic -std=c11 -o Colle_25.exe Colle_25.c && ./Colle_25.exe
```

Séparation en deux sous-tableaux

Pour séparer en deux sous-tableaux un tableau `tab` de taille $n \in \mathbb{N}$, on suppose que $n \geqslant 2$. En effet, si $n \leqslant 1$, il n'y a rien à faire dans le tri fusion et on peut renvoyer `tab` directement.

On propose le choix le plus simple : on découpe au milieu, en `milieu = n/2` (division entière), et le tableau de gauche `tab1` a ses indices qui vont de `0` (inclus) à `milieu` (non inclus), et le tableau de droite `tab2` a ses indices de `milieu` (inclus) à `n` (non inclus).

7. Écrire une fonction `int separe_en_deux(int n)` qui calcule cette valeur de `milieu`.

Fusion des deux sous-tableaux triés

8. Écrire une fonction `int* fusion(int n1, int tab1[n1], int n2, int tab2[n2])`, qui s'occupe de reconstruire un tableau `tab` de taille `n = n1 + n2` trié, à partir des valeurs des deux tableaux `tab1` et `tab2`, déjà triés (par ordre croissant).

On propose l'algorithme suivant :

— si `n1==0`, renvoyer `tab2`,
— si `n2==0`, renvoyer `tab1`,
— sinon, allouer un tableau `tab` de taille `n = n1 + n2`, et le remplir comme cela :
  — on commence à `i=0` (dans `tab1`) et `j=0` (dans `tab2`),
  — tant qu'il reste des valeurs dans `tab1` ET dans `tab2`, on compare la valeur actuelle des deux tableaux, et on affecte dans `tab[i+j]` la plus petite, et on incrémente la position (`i` ou `j`) de la valeur la plus petite,
  — après cette boucle, s'il reste encore des valeurs dans `tab1`, on les ajoute à la suite dans `tab`,
  — idem pour `tab2`.
  — et on termine par renvoyer `tab`.

Tri fusion

Après avoir obtenu l'indice `milieu` où l'on souhaite diviser le tableau `tab` en deux morceaux `tab1` et `tab2`, il faut être capable d'extraire ces sous-tableaux. En Python, c'est très simple avec la notation "slice" : `tab[:milieu]` et `tab[milieu:]`. En C, il va forcément falloir allouer un nouveau tableau (sur le tas, avec donc un `malloc` de la bonne taille), et le remplir avec une boucle.

9. Écrire une fonction `int* sous_tableau(int n, int tab[n], int g, int d)` qui alloue un tableau `sstab` de taille `d - g` et le remplis par les valeurs de `tab` pour `i` allant de `g` (début, `g` inclus) à `d-1` (fin, `d` pas inclus), et le renvoie.
10. Écrire une fonction `int* tri_fusion(int n, int tab[n])`, qui effectue le tri fusion de ce tableau `tab` de taille `n`, en appelant les fonctions `separe_en_deux` (une fois), `sous_tableau` (deux fois), `tri_fusion` récursivement (deux fois) et `fusion` (une fois), sauf pour les cas de bases où `n <= 1`.

Tests

11. Écrire deux fonctions `int* range(int n)` et `int* antirange(int n)` qui allouent et remplissent des tableaux de `n` entiers, entre `0` et `n-1` pour `range` (comme en Python) ou entre `n-1` et `0` pour `antirange`.
12. Écrire une fonction `bool est_croissant(int n, int tab[n])` qui teste si un tableau `tab` est trié par ordre croissant. Votre fonction devra faire au pire `n-1` comparaisons et pas une de plus.
13. Dans votre fonction `int main(void)`, écrire quelques tests avec des petits tableaux créés à la main (ex : `int tab[5] = {3,2,1,5,4};`) et des `assert(est_croissant(n, tri_fusion(n, tab)))` pour vérifier que le tableau obtenu par l'appel à `tri_fusion` est bien trié par ordre croissant.
14. Enfin, écrire quelques tests avec des "grands" tableaux obtenus par `range(n)` ou `antirange(n)`.