Start Date : Juin 8th 2021
End Date :

PUSH SWAP

Ce projet demande de trier des données sur une pile, avec un set d'instructions limité, en moins de coups possible. Pour le réussir, il va falloir apprendre à manipuler différents algorithmes de tri et choisir la (ou les ?) solution la plus appropriée pour un classement optimisé des données.

I. Introduction :

Le projet Push_Swap est un **projet d'algorithmique** ou il va falloir **trier de la manière la plus simple et la plus efficace possible**. En somme : nous avons à notre disposition un **ensemble d'entiers**, **deux piles et un ensemble d'instructions** pour manipuler le piles. Avec cela, nous devrons écrire un programme en C au nom de l'intitulé du projet qui **calcule et affiche sur la sortie standard le plus petit programme basé sur un système d'instructions** (cités plus tard) **permettant de trier les entiers passés en paramètres.**

A savoir que les algorithmes de tri, et toute la complexité qui va avec, font partie des grands classiques des entretiens d'embauche. Il est donc important, en cette occasion, de se pencher sérieusement sur la question.

Les objectifs de ce projet sont rigueur, pratique du C et pratique d'algorithmes élémentaires et plus particulièrement la complexité de ces derniers.

Attention, détrompez-vous ! Trier des valeurs c'est simple mais les trier le plus vite possible, c'est moins simple vu que d'une configuration des entiers à trier à l'autre, c'est plus le même algo de tri qui est le plus efficace !

II. Consignes Générales :

Mode de Correction	Ce projet ne sera **corrigé que par des humains** (tout en respectant les contraintes, le noms et l'organisations de nos fichiers dépendent entièrement de nous)
Nom d'Exécutable	**push_swap**
Makefile	Doit compiler le projet tout en contenant les **règles habituelles**: all, make, clean, fclean, re.
Libft	La Libft est **autorisée** mais **ces sources et le Makefile doivent être associées dans un dossier nommé libft qui devra se trouver la racine du dépôt de rendu.**
Fonctions Autorisées	**Write, Read, Malloc, Free, Exit**
Règles Supplémentaires	- En C; - A la norme; - Gérer les erreurs de façon raisonnée; - Aucunes fuites de mémoires doit être présente;

III. Partie Obligatoire :

1. Règles du Jeu :

➢ Le jeu est constitué de **2 piles** nommées **a** et **b**.

➢ Au départ :
- **a contient un nombre arbitraire d'entiers positifs ou négatifs, sans doublons**;
- **b est vide.**

➢ Le but du jeu est de **trier a dans l'ordre croissant**.

➢ On dispose alors des **opérations suivantes** :

sa	*swap a* *//////////*	**Intervertir les 2 premiers éléments au sommet de la pile a**. Ne fait rien s'il y a qu'un ou aucun élément dans a.
sb	*swap b* *//////////*	**Intervertir les 2 premiers éléments au sommet de la pile b.** Ne fait rien s'il y a qu'un ou aucun élément dans b..
ss	*//////////* *//////////*	**sa et sb en même temps**.
pa	*push a* *//////////*	**Prendre le 1er élément au sommet de b et le mettre sur a**. Ne fait rien si b est vide.
pb	*push b* *//////////*	**Prendre le 1er élément au sommet de a et le mettre sur b**. Ne fait rien si a est vide.
ra	*rotate a* *//////////*	**Décaler d'une position vers le haut tous les éléments de la pile a**. Le 1er élément devient le dernier.
rb	*rotate b* *//////////*	**Décaler d'une position vers le haut tous les éléments de la pile b**. Le 1er élément devient le dernier.
rr	*//////////* *//////////*	**ra et rb en même temps**.
rra	*reverse rotate a*	**Décaler d'une position vers le bas tous les éléments de la pile a**. Le dernier élément devient le 1er.
rr b	*reverse rotate b*	**Décaler d'une position vers le bas tous les éléments de la pile b**. Le dernier élément devient le 1er.
rrr	*//////////* *//////////*	**rra et rrb en même temps.**

2. Exemple :

Start Pile	End Pile	Start Pile	End Pile
sa		sb	
1 5 9 0		5 1 9 0		9 0	5 1	9 0	1 5
a	b	a	b	a	b	a	b

Start Pile	End Pile
ss	
9 0	5 1	0 9	1 5
a	b	a	b

Start Pile	End Pile	Start Pile	End Pile
pa		pb	
9 0	5 1	5 9 0	1	9 0	5 1	0	9 5 1
a	b	a	b	a	b	a	b

Start Pile	End Pile	Start Pile	End Pile
ra		rb	
5 9 0 | 1 a | b	9 0 5 | 1 5 a | b	1 5 9 0 | a | b	5 9 0 1 | a | b

Start Pile	End Pile
rr	
5 9 0 | 7 6 1 a | b	9 0 5 | 6 1 7 a | b

Start Pile	End Pile	Start Pile	End Pile
rra		rrb	
1 5 9 0 | a | b	0 1 5 9 | a | b	0 | 9 5 1 a | b	0 | 1 9 5 a | b

Start Pile	End Pile
rrr	
5 9 0 | 4 7 1 a | b	0 5 9 | 1 4 7 a | b

IV. Le Programme Push Swap :

- Le programme réalisé devra **prendre en paramètre la pile a sous la forme d'une liste d'entiers** et ou le **1er paramètre est au sommet de la pile** (donc l'ordre des entiers est très importante). Attention, si aucun paramètre n'est passé, le programme termine immédiatement et n'affiche rien.

- Le programme devra **afficher le programme composé de la plus petite suite d'instructions possible qui permet de trier la pile a**, le plus petit étant au sommet.

- **Les instructions doivent être affichées séparées par un '\n' et rien d'autre**.

- Le but est de **trier les entiers avec le moins d'opérations possible**. En soutenance, le nombre d'instructions que le programme a calculé sera comparé à un nombre d'opérations maximum tolérées.

- En cas d'**erreur**, il faudra **afficher Error suivi d'un '\n' sur la sortie d'erreur**. Par exemple, si certains paramètres ne sont pas des nombres, ou si certains paramètres ne tiennent pas dans un int, ou encore s'il y a des doublons.

- Le programme doit s'**utiliser de la manière suivant :**

Le paramètre **wc -l** (word count -line) **affiche le nombre d'opérations effectuées**: **$>ARG="4 67 3 87 23"; ./push_swap $ARG | wc -l**
Si le **programme checker_OS* affiche KO,** cela signifie que le **programme push_swap calcule un programme qui ne trie pas la liste et qui donc faux** (ce programme est disponible dans les ressources du projet sur l'intranet) : **$>ARG="4 67 3 87 23"; ./push_swap $ARG | ./checker_OS $ARG**

*Checker_OS est un programme qui prend en paramètre la pile a sous la forme d'une liste d'entiers (le premier paramètre est également au sommet). Ensuite, il va attendre de lire des instructions sur l'entrée standard, chaque instruction suivie par un '\n'. Une fois toutes les instructions lues, il va exécuter ces instructions sur la pile d'entiers passée en paramètres. Si la pile a est bien triée et la pile b est vide, alors le programme affiche "OK\n" sur la sortie standard. A l'inverse, il affichera "KO\n". De fois checker_OS va vérifier que le code généré trie bien la pile passée en paramètres. *Réaliser ce programme de vérification fait partie des bonus.*

V. Algorithmes de Tri :

Selon le dictionnaire, "trier" signifie "répartir en plusieurs classes selon certains critères." De fait, le terme de "tri" en algorithmique est la plupart du temps attaché au processus de classement d'un ensemble d'éléments dans une ordre donné. Tout ensemble muni d'un ordre total peut fournir une suite d'éléments à trier.

1. Introduction :

Intuitivement, n'importe quelle personne à qui on donne un ensemble à trier met en place des **stratégies de trie différentes selon le nombre d'éléments de l'ensemble**. On site ainsi les **tri par sélection**, **trie par propagation**, **tri par insertion**, **tri rapide**, **tri par fusion**… Chacunes de ces méthodes ont leurs particularités et leur niveau de performance (qui correspond à la complexité de l'algorithme.

Face à un exemple de taille faible, il peut sembler quelque peu dérisoire de pousser le raffinement jusqu'à trouver un algorithme de tri supplémentaire à appliquer sur un petit nombre d'éléments, car la différence de complexité y sera peu visible. Cependant, il faut garder en mémoire qu'on peut vouloir trier des centaines de milliers d'éléments et que si une différence de stratégie est visible sur des petits nombre tels que 52 ou 200, a fortiori **sur de grands ensembles la différence de complexité sera d'autant plus visible.**

Pour illustrer les méthodes de tri, on suppose qu'on dispose d'un tableau **tab de N entier numérotés de 1 à N** et qu'on cherche à **trier les entiers dans l'ordre croissant** (de gauche à droite).

2. 2 Méthodes de Tri :

Il existe 2 types méthodes de tri **:**

- ➢ Des méthodes qui **trient les éléments deux à deux,** de manière plus ou moins efficace, mais qui nécessitent toujours de **comparer chacun des N éléments avec chacun des N-1 autres éléments.** Parmi ces algorithmes on trouve :
  - Une méthode de tri élémentaire, le **tri par sélection** ;
  - Et sa variante, le **tri par propagation** ou **tri bulle** ;
  - Une méthode qui est par exemple utilisée pour trier les cartes d'un jeu: le **tri par insertion**.
- ➢ Des méthodes qui sont **plus rapides**, car elles **trient des sous-ensembles de ce N éléments puis regroupent les éléments triés**. Elles illustrent le principe "diviser pour régner.". Parmi elles, l'on trouve :
  - Le fameux **trie rapide** ou ***Quicksort*** ;
  - Et enfin, le **tri par fusion**.

Cette liste n'est évidemment pas exhaustive**.** Il existe des **méthodes particulièrement adaptées à certains types de données spécifiques**, comme par exemple le **tri par base** (radix sort).

3. Complexité d'un Algorithme :

a. Mesure de Performance :

Le calcul de la complexité d'un algorithme permet de **mesurer sa performance**. Il existe deux types de complexité :

- La **complexité spatiale** qui permet de quantifier l'utilisation de la mémoire;
- La **complexité temporelle** qui permet de quantifier la vitesse d'exécution.

b. Objectif du la Mesure de Complexité Temporelle :

Pour le moment, nous allons seulement nous concentrer sur la complexité temporelle.

L'objectif d'un calcul de complexité algorithmique temporelle est de pouvoir **comparer l'efficacité d'algorithmes résolvant le même problème.** Dans une situation donnée, cela permet donc d'**établir lequel des algorithmes disponibles est le plus optimal**.

Réaliser un calcul de complexité en temps revient à **compter le nombre d'opérations élémentaires** (affectation, calcul arithmétique ou logique, comparaison…) **effectuées par l'algorithme.** Puisqu'il s'agit seulement de comparer des algorithmes, les règles de ce calcul doivent être indépendantes :

- Du langage de programmation utilisé ;
- Du processeur de l'ordinateur sur lequel sera exécuté le code ;
- De l'éventuel compilateur employé.

c. Mesure de Complexité Temporelle :

Par souci de simplicité, on fera l'hypothèse que **toutes les opérations élémentaires sont à égalité de coût, soit 1 unité de temps**.

La complexité en temps d'un algorithme sera exprimé par une fonction, notée T (pour Time), **qui dépend** :

- **De la taille des données passées en paramètres** : plus ces données seront volumineuses, plus il faudra d'opérations élémentaires pour les traiter (on notera n le nombre de données à traiter);
- **De la donnée en elle-même**, de la façon dont sont réparties les différentes valeurs qui la constituent. Par exemple, si on effectue une recherche séquentielle d'un élément dans une liste non triée, on parcourt un par un les éléments jusqu'à trouver, ou pas, celui recherché. Ce parcours peut s'arrêter dès le début si le premier élément est « le bon ». Mais on peut également être amené à parcourir la liste en entier si l'élément cherché est en dernière position, ou même n'y figure pas.

Délors, en toute rigueur, on peut en effet distinguer **deux formes de complexité en temps** :

- La **complexité dans le meilleur des cas** : c'est la situation la plus favorable (par exemple : recherche d'un élément situé à la première position d'une liste);
- La **complexité dans le pire des cas** : c'est la situation la plus défavorable (par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas). La **complexité dans le pire des cas sera plus souvent calculée que celle dans le meilleur des cas** puisqu'elle est **plus pertinente** (il vaut mieux toujours envisager le pire).

d. Mesure de Complexité Temporelle avec l'Ordre de Grandeur Asymptotique *O* :

Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction T, mais seulement l'**ordre de grandeur asymptotique, noté** O **(« grand O »)** :

Une fonction $T(n)$ est en $O(f(n))$ (« en grand O de f(n)«) si :

$$\exists\, n0 \in \mathbb{N}, \exists c \in \mathbb{R}+, \forall n \in \mathbb{R}+, n \geq n0 \Rightarrow |T(n)| \leq c|f(n)|$$

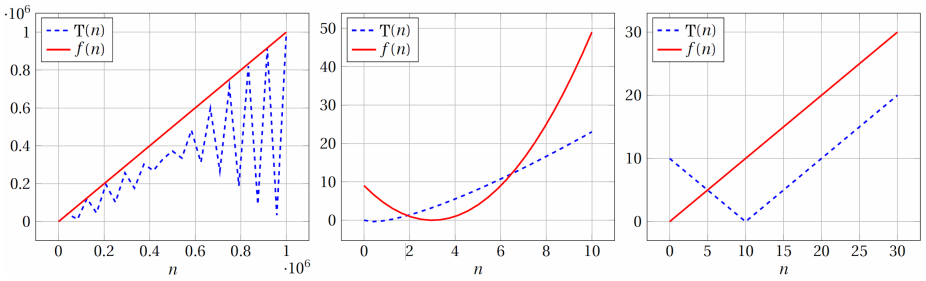
Autrement dit, $T(n)$ **est en** $O(f(n))$ **s'il existe un seuil** $n0$ **à partir duquel la fonction** T **est toujours dominée par la fonction** f, à une constante multiplicative fixée c près.

e. Classe de Complexité :

O	Type de Complexité	Temps pour n = 5	Temps pour n = 50	Temps pour n = 1 000
O(1)	Constante	10 ns	10 ns	10 ns
O(log(n))	Logarithmique	10 ns	20 ns	30 ns
O(n)	Linéaire	50 ns	500 ns	10 µs
O(n×log(n))	Quasi-Linéaire	50 ns	501 ns	10 µs
O(n2)	Quadratique	250 ns	25 µs	10 ms
O(n3)	Cubique	1.25 µs	1.25 ms	10 s
O(2")	Exponentielle	320 ns	130 jours	…
O(n!)	Factorielle	1.2 µs	10^{{48}} ans	…

f. <u>Exemple :</u>

- $T1(n) = 7 = O(1)$;
- $T2(n) = 12n + 5 = O(n)$;
- $T3(n) = 4n2 + 2n + 6 = O(n2)$;
- $T4(n) = 2 + (n - 1) \times 5 = O(n)$.



4. Le Tri par Sélection :

a. Principes :

L'idée est simple : rechercher le plus grand élément (ou le plus petit), le placer en fin de tableau (ou en début), recommencer avec le second plus grand (ou le second plus petit), le placer en avant-dernière position (ou en seconde position) et ainsi de suite jusqu'à avoir parcouru la totalité du tableau.

Dans le cas du push_swap, le tri par sélection consiste en :

- Trouver le plus petit élément et le mettre au début de la liste;
- Trouver le 2ème plus petit élément et le mettre en seconde position;
- Trouver le 3ème plus petit élément et le mettre à la 3ème place;
- Trouver le Nième plus petit élément et le mettre à la Nième place;
- ...

b. Exemple :

0	14	255	-23	67
-23	14	255	0	67
-23	0	255	14	67
23	0	14	255	67
23	0	14	67	255
-23	0	14	67	255

c. Décomposition de l'Algorithme :

- **La fonction max()**, qui prend en **paramètre un tableau et sa taille** et **renvoie l'indice de l'élément le plus grand** : consiste à parcourir l'intégralité du tableau pour à chaque fois comparer l'élément actuel avec le maximum provisoire.
- **La fonction exchange()** qui **échange deux éléments** (dont on connaît les indices) **d'un tableau.** Le principe est le même que la fonction swap(), qui utilise un variable supplémentaire pour stocker temporairement un des contenues à échanger.
- **La fonction sélection()** qui **tant que la taille du tableau est supérieur à 0 :**
    - **Recherche l'indice de l'élément le plus grand** (max());
    - **Echange cet élément avec le dernier élément du tableau** (exchange());
    - **Décrémente la taille.**

d. <u>Calcul de la Complexité :</u>

À la **première itération, on effectue N-1 comparaisons.**
À la **ième itération**, on effectue donc **N-i comparaisons** (puisque à chaque itération on décrémente la taille du tableau).

Le nombre total de comparaisons pour trier un tableau de taille n est donc la **somme de N-i pour i allant de 1 à n-1**. On s'aperçoit donc que la complexité de notre algorithme est **quadratique (en O(n^2))**, ce qui n'est pas très bon (à titre d'exemple, si vous doublez la taille d'un tableau, il vous faudra quatre fois plus de temps pour le trier).

En effet, la simplicité de cet algorithme fait qu'on le qualifie d'**algorithme « naïf »**. Cela ne veut pas pour autant dire qu'il est incorrect, il est juste **trop simpliste pour être réellement efficace**.

e. <u>Conclusion :</u>

En résumé, lorsque on utilise le tri par sélection :

- On effectue environ **\frac{n(n-1)}{2} comparaisons** ;
- On effectue environ **n échange**s ;
- La **complexité moyenne et dans le pire des cas est quadratique.**

5. Tri par Propagation ou Tri à Bulle :

a. Principe :

Le principe du tri à bulles (bubble sort ou sinking sort) est de **comparer deux à deux les éléments e1 et e2 consécutifs d'un tableau et d'effectuer une permutation si e1 > e2**. On continue de trier jusqu'à ce qu'il n'y ait plus de permutation.

b. Complexité :

- ➢ Dans le meilleur des cas, avec des données déjà triées, l'algorithme effectue seulement n - 1 comparaison. Sa **complexité dans le meilleur des cas est donc en Θ(n) : complexité linéaire.**
- ➢ Dans le pire des cas, avec des données triées à l'envers, les parcours successifs du tableau imposent d'effectuer (n2-n)/2 comparaisons et échanges. On a donc une **complexité dans le pire des cas du tri bulle en Θ(n2) : complexité quadratique.**

c. Information Supplémentaire :

Parmi les variantes du tri bulle on peut citer le **tri Shuttle**. Cet algorithme de tri **fonctionne comme le tri bulle mais change de direction à chaque fois qu'il est parvenu à une extrémité.**

On peut également citer le **tri de Oyelam**i ou **le tri à peigne** qui reprend des caractéristiques du tri Shell et du tri à bulles.

d. Exemple :

ROUND 1				
0	14	255	-23	67
0	14	255	-23	67
0	14	255	-23	67
0	14	-23	255	67
0	14	-23	67	255
ROUND 2				
0	14	-23	67	255
0	14	-23	67	255
0	-23	14	67	255

ROUND 3				
0	-23	14	67	255
-23	0	14	67	255

6. Tri par Insertion :

a. Principe :

C'est le tri du joueur de cartes. On fait **comme si les éléments à trier étaient donnés un par un** :

- **Le premier élément constituant, à lui tout seul, une liste triée de longueur 1 ;**
- On range ensuite **le second élément pour constituer une liste triée de longueur 2** ;
- Puis on range **le troisième élément pour avoir une liste triée de longueur 3 ;**
- Et ainsi de suite...

Le principe du tri par insertion est donc d'**insérer à la nième itération le nième élément à la bonne place.**

b. Complexité :

- ➢ Dans le meilleur des cas, avec des données déjà triées, l'algorithme effectue seulement N comparaisons. **Sa complexité dans le meilleur des cas est donc en Θ(n) : complexité linéaire.**
- ➢ Dans le pire des cas, avec des données triées à l'envers, les parcours successifs du tableau imposent d'effectuer (n-1)+(n-2)+(n-3)..+1 comparaisons et échanges, soit (n2-n)/2. On a donc une **complexité dans le pire des cas du tri bulle en Θ(n2) : complexité quadratique.**

c. Information Supplémentaire :

On notera également une propriété importante du tri par insertion : contrairement à celle d'autres méthodes, **son efficacité est meilleure si le tableau initial possède un certain ordre**. L'algorithme **tirera en effet parti de tout ordre partiel présent dans le tableau**. Jointe à la simplicité de l'algorithme, cette propriété le désigne tout naturellement pour "finir le travail" de méthodes plus ambitieuses comme le tri rapide

d. <u>Exemple :</u>

Liste Initiale	0	14	255	-23	67
Nouvelle Liste	0				
Liste Initiale	0	14	255	-23	67
Nouvelle Liste	0	14			
Liste Initiale	0	14	255	-23	67
Nouvelle Liste	0	14	255		
Liste Initiale	0	14	255	-23	67
Nouvelle Liste	-23	0	14	255	
Liste Initiale	0	14	255	-23	67
Nouvelle Liste	-23	0	14	67	255

7. Tri Rapide

a. Principe :

Le **tri rapide** - aussi appelé **"tri de Hoare"** (du nom de son inventeur Tony Hoare) ou **"tri par segmentation"** ou **"tri des bijoutiers"** ou, en anglais **"quicksort"** - est certainement l'**algorithme de tri interne le plus efficace.**

Le principe de ce tri est d'**ordonner le vecteur T.(0)..T.(n) en cherchant dans celui-ci une clé pivot autour de laquelle réorganiser ses éléments**. Il est souhaitable que le pivot soit aussi proche que possible de la clé relative à l'enregistrement central du vecteur, afin qu'il y ait à peu près autant d'éléments le précédent que le suivant, soit environ la moitié des éléments du tableau. Dans la pratique, on prend souvent le dernier élément du tableau.

On **permute ceux-ci de façon à ce que pour un indice j particulier tous les éléments dont la clé est inférieure à pivot se trouvent dans T.(0)...T.(j) et tous ceux dont la clé est supérieure se trouvent dans T.(j+1)...T.(n). On place ensuite le pivot à la position j.**

On applique ensuite le **tri récursivemen**t à, **sur la partie dont les éléments sont inférieurs au pivot et sur la partie dont les éléments sont supérieurs au pivot.**

b. Complexité :

- ➢ Dans le **pire des cas**, c'est à dire si, à chaque niveau de la récursivité le découpage conduit à trier un sous-tableau de 1 élément et un sous-tableau contenant tout le reste, l**a complexité du tri rapide est en Θ(n2) : complexité quadratique.**
- ➢ Par contre, dans le **cas moyen**, cet algorithme a une **complexité en Θ(nlog n) : complexité quasi-linéaire.**

c. Information Supplémentaire :

Dans le tri rapide, **le choix du pivot est essentiel**. Plusieurs options sont possibles :

- Choisir le **premier élément** du tableau ;
- Choisir le **dernier élémen**t du tableau ;
- Choisir un **élément au hasard** ;
- Choisir l'**élément au milieu du tableau** ;
- Trouver le **pivot optimal en recherchant la médiane** (tous les éléments / /2) ;

d. Exemple :

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]	arr[6]	arr[7]	arr[8]	arr[9]
PIVOT = arr[0] = 10									
10	16	8	12	15	6	3	9	5	///
LOW 1 : arr[1] = 16					LOW > PIVOT ? TRUE				
HIGHT 8 : arr[8] = 5					HIGH <= PIVOT ? TRUE				
10	5	8	12	15	6	3	9	16	///
LOW++					HIGH--				
10	5	8	12	15	6	3	9	16	///
LOW 2 : arr[1] = 8					LOW > PIVOT ? FALSE				
HIGHT 7 : arr[7] = 9					HIGH <= PIVOT ? TRUE				
10	5	8	12	15	6	3	9	16	///
LOW 3 : arr[3] = 12					LOW > PIVOT ? TRUE				
HIGHT 7 : arr[7] = 9					HIGH <= PIVOT ? TRUE				
10	5	8	9	15	6	3	12	16	///
LOW++					HIGH--				
10	5	8	9	15	6	3	12	16	///
LOW 4 : arr[4] = 15					LOW > PIVOT ? TRUE				
HIGHT 6 : arr[6] = 3					HIGH <= PIVOT ? TRUE				
10	5	8	9	3	6	15	12	16	///
LOW++					HIGH--				
10	5	8	9	3	6	15	12	16	///
LOW 5 : arr[4] = 6					LOW > PIVOT ? TRUE				
HIGH 5 : arr[6] = 6					HIGH <= PIVOT ? FALSE				
6	5	8	9	3	10	15	12	16	///
PIVOT : arr[5] = 10									
6	5	8	9	3	10	15	12	16	///
arr[0] - arr[4] < PIVOT						arr[6] - arr[9] > PIVOT			

8. Tri par Fusion :

a. Principe :

Il s'agit à nouveau d'un tri suivant le paradigme diviser pour régner. Le principe du tri fusion (ou tri par interclassement) en est le suivant :

- On **divise en deux moitiés la liste à trier** (en prenant par exemple, un élément sur deux pour chacune des listes).
- On **trie chacune d'entre elles.**
- On **fusionne les deux moitiés obtenues pour reconstituer la liste triée.**

b. Complexité :

La complexité dans le **pire des cas** du tri fusion est en **log2(n).**

.

c. Exemple :

Liste Initiale									
10	16	8	12	15	6	3	9	5	0
Liste Divisé 1					Liste Divisé 2				
10	16	8	12	15	6	3	9	5	0
8	16	10	12	15	0	3	9	5	6
8	10	16	12	15	0	3	5	9	6
8	10	12	16	15	0	3	5	6	9
8	10	12	15	16	0	3	5	6	9
Listes Regroupées									
8	10	12	15	16	0	3	5	6	9
0	10	12	15	16	8	3	5	6	9
0	3	12	15	16	8	10	5	6	9
0	3	5	15	16	8	10	12	6	9
0	3	5	6	16	8	10	12	15	9
0	3	5	6	9	8	10	12	15	16

VI. Algorithme PushSwap :

1. Structure et Liste Chaînée :

```
typedef struct    s_node
{
        int             value;
        int             pos;
        int             unité;
        struct s_node   *prev;
        struct s_node   *next;
}               t_node;
```

```
typedef struct    s_list
{
        int             nb_unité;
        int             nb_values;
        int             nb_stackA;
        int             nb_stackB;
        int             *unité3;
        struct s_node   *stackA;
        struct s_node   *stackB;
}               t_list;
```

2. ToolBox :

- ☑ ~~Fonctions de parsing ;~~
- ☑ ~~Fonctions de gestion d'erreur ;~~
- ☐ Fonctions Libft (à mettre à jour une fois l'algorithme réalisées) ;
- ☐ Fonctions primaires d'utilisation des listes chaînées (new list, destroy list, etc..) ;
- ☑ ~~Fonctions pour chaque instructions (pa, pb, sa, sb, ss, ra, rb, rr, rra, rrb, rrr) ;~~
- ☑ ~~Fonctions qui trient les nombres préalablement et en attribuant à la variable pos de chaque node la position finale de sa "value" respective ;~~
- ☑ ~~Fonctions qui calculent, une fois la liste triée, la quantité d'unité de 3 nombres possible selon la taille de liste des nombres donnés ;~~
- ☐ Fonctions qui calculent, une fois la liste trié et les unités distribués, le nombre d'unités de 3 (ensembles de 3 unités);
- ☐ Fonctions qui trient toutes combinaisons de 3 nombre (une unité) sur une même stack (stack A et B) sans utiliser ni les fonctions d'instruction de type rotate ni celle de type reverse) : -23, 0, 14 / -23, 14, 0 / 14, -23, 0 / 14, 0, -23 / 0, 14, -23 / 0, -23, 14) ;
- Fonctions qui trient toutes combinaisons de 3 nombre (une unité) lors d'un push sur l'autre stack A et B (attention, la dernière unité peut varier d'une taille de 3 à 1) ;

3. Base :

- Gestion d'erreur relatif à av et ac ;
- Création de la liste chaînée ;
- Parsing, gestion d'erreur et initialisation de la liste chaînée ;
- Attribuer la position final de la value de chaque node ainsi que son "unité" :

Value	-23	0	67	14	-50	123	110	-35	28	7	-5
Trié	-50	-35	-23	-5	0	7	14	28	67	110	123
Pos	2	4	8	6	0	10	9	1	7	5	3
Unité	0	1	2	2	0	3	3	0	2	1	1

4. Algorithme Brut :

a. First Algorithme :

- Divisé la stack A en deux stack A et B, A ayant tous les nombres de position taille à position taille / 2 et B ayant tous les nombre de position à taille / 2 + 1 à la position 0 (ex pour une liste de 40 : stack A de 40 à 21 (unité de de N2 à G3 A1 à G2) et stack B de 20 à 1 (unité de G2 à N2) -> 40 / 40 move ;
- Isolé tous les éléments qui ne font pas partis de la dernière unité sur la stack B -> 20 / 40 move ;
- Trier l'unité stocké sur la stack A -> 2 / 40 move ;
- Rotate et push si unité suivante jusqu'à ce que les 3 membres de l'unité se trouve sur la stack A -> 9 / 40 move ;
- Trier la nouvelle unité sur A -> 4 / 40 move ;
- Reverse jusqu'à avoir tous les valeurs que l'on a rotate deux étapes avant en haut de stack B -> 6 / 40 move ;
- Répéter les deux dernières étapes jusqu'à obtenir tous les nombres étant originellement sur la stack A -> 8 + 0 + 5 / 40 | 12 + 4 + 9 / 40 | 4 + 1 + 1 / 40 | 6 + 4 + 3 / 40 | 3 + 4 + 0 / 40 -> 60 / 40 move ;
- Répéter pour la stack B -> 19 + 4 + 16 / 40 | 14 + 0 + 11 / 40 | 8 + 0 + 5 / 40 | 9 + 2 + 7 / 40 | 9 + 1 + 6 / 40 | 4 + 1 + 1 / 40 | 3 + 4 + 0 / 40 -> 124 / 40 move;
- Total move : 256 / 40.

b. <u>Second Algorithme :</u>

- Push par unité sur la stack B jusqu'à obtenir la dernière unité seule sur la stack A -> 40 + 22 + 26 + 36 + 30 + 19 + 11 + 11 + 11 + 10 + 10 + 8 + 6 / 40 -> 240 / 40 move;
- Trier l'unité restante sur A et push / trie sur A -> 4 + 3 + 4 + 6 + 5 + 4 + 5 + 3 + 3 + 5 + 4 + 4 + 4 + 2 / 40 -> 54 / 40 move ;
- Total move : 294 / 40.

c. <u>Third Algorithme :</u>

- Push les 3 dernières unités sur la stack B, puis les 3 suivantes, etc… (la première unité doit rester stocké sur la stack A) : 40 + 31 + 18 + 14 + 6 / 40 -> 109 / 40 move ;
- Trier l'unité stocké sur la stack A -> 4 / 40 move ;
- Rotate et push si unité suivante jusqu'à ce que les 3 membres de l'unité se trouve sur la stack A -> 3 / 40 move ;
- Trier la nouvelle unité sur A -> 4 / 40 move ;
- Reverse jusqu'à avoir tous les valeurs que l'on a rotate deux étapes avant en haut de stack B -> 0 / 40 move ;
- Répéter les deux dernières étapes jusqu'à obtenir tous les nombres étant originellement sur la stack A -> 6 + 3 + 3 / 40 | 6 + 3 + 3 / 40 | 3 + 4 + 0 / 40 | 6 + 4 + 3 / 40 | 6 + 0 + 3 / 40 | 3 + 5 + 0 / 40 | 8 + 5 + 5 / 40 | 5 + 4 + 2 / 40 | 3 + 4 + 0 / 40 | 7 + 4 + 4 / 40 | 5 + 1 + 2 / 40 | 2 + 1 + 0 / 40 -> 134 / 40 move;
- Total move : 243 / 40.