ALGORITHMIE
MOBILE EQUILIBRÉ

I. ALGORITHMES

1. PRÉAMBULE

Pour tous les algorithmes, on suppose une structure de données pour les arbres telle que:

```
structure Tree(Node root)
   this.root = root

structure Node(Node left, Node right):
   this.left          = left
   this.right         = right
   this.weight        = left.weight + right.weight
   this.balance       = abs(left.weight - right.weight)
   this.total_balance = left.total_balance + right.total_balance + this.balance

structure Leaf(Integer weight) extends Node:
   this.weight  = weight
   this.left    = this.right = null
   this.balance = this.total_balance = 0
```

Ainsi, tous les algorithmes vont être créés de manière à ne pas devoir recalculer tous les déséquilibres totaux à chaque ajout d'un noeud. (C'est à dire création à partir des feuilles, puis on remonte).

2. ALGORITHME DYNAMIQUE ORDONNÉ

La récurrence de l'algorithme est :

$$\begin{cases} 0 \text{ si } i = j \\ \min(\Delta_{i,k} + \Delta_{k+1,j} + \text{abs}(\partial_{i,k} - \partial_{k+1,j})) \text{ pour tout } k >= i \text{ et } k < j \end{cases}$$

Le but va donc être de remplir un tableau à deux dimensions à l'aide de cette récurrence. On peut donc observer que presque une moitié du tableau ne sera pas utilisée par cette récurrence.

Afin de baisser la complexité de cet algorithme, on suppose un tableau de poids qui évitera de recalculer à chaque fois la formule suivante : $\partial_{i,k} - \partial_{k+1,j}$
Cela sera un tableau à une seule dimension, de taille du nombre de poids + 1, avec 0 pour première valeur.

Par exemple:

```
W = [3, 1, 2]
sums = [0, 3, 4, 6]
```

De plus, cela nous permet de retrouver le poids très simplement à l'aide de la formule suivante :
$\partial_{i,k}$ = sums[k + 1] - sums[i]

Exemple:

```
i = 1, k = 2
sums[k + 1] = 6 et sums[i] = 3
Donc ∂i,k = 6 - 3 = 3
```

De part sa nature et de part ses constructions, cet algorithme a pour complexité une grandeur de $O(n^2.\log n)$.
On doit parcourir l'ensemble du tableau à deux dimensions (donnant le n^2). Et pour chaque case à remplir de ce tableau, on doit itérer de nouveau afin de trouver le minimum (ce qui donne le log n).

Cet algorithme trouve l'arbre optimal à tout les coups.
En effet, lors des calculs de chaque case du tableau, l'algorithme va trouver le plus petit déséquilibre local parmi toutes les possibilités qui s'offrent à lui. La case en haut à gauche (d'indice [0, n]) contient le plus petit déséquilibre total, car tous les poids auront été ajoutés.

Nous pouvons voir chaque case du tableau telle que, [i, j] correspond à un sous arbre contenant tous les poids allant de i à j. Ainsi, chaque case contient le déséquilibre optimal, pour sa liste de poids.

L'algorithme trouve l'arbre optimal tout le temps.

3. ALGORITHME « ORDONNÉ 1 »

Le principe de cet algorithme est très simple : on souhaite mettre le même poids dans le sous arbre gauche et dans le sous arbre droit.
Mais évidemment, il va falloir gérer les déséquilibres, puisque il va être très rare de pouvoir réellement séparer de telle sorte que les deux poids soient égaux.

Pour cela, on va poser W la liste des poids, et S_w sa somme.
On va donc essayer de mettre $S_w/2$ dans le sous arbre gauche et $S_w/2$ dans le sous arbre droit.
Le but de l'algorithme va être d'itérer, en gardant l'ordre, en ajoutant les éléments à gauche, tant que le total du poids entreposé à gauche ne dépasse pas $S_w/2$. (Dans l'algorithme, si on ajoute un élément d'un côté, on soustrait son poids à la limite de poids trouvé).

Pour l'élément qui fait dépasser cette limite, il va falloir vérifier de quel côté il se trouvera être le plus optimisé.
Pour cela, on va regarder quel est le poids le plus petit du sous arbre gauche entre le moment où le dernier poids a été ajouté (que l'on va nommer RSG), et le moment avant l'ajout de ce poids (que l'on va nommer LAST).

Ainsi si abs(RSG) < LAST ou si RSG = 0, alors on ajoute l'élément à gauche
Sinon, on l'ajoute à droite.

On peut considérer RSG et LAST comme étant les déséquilibres entre les deux sous listes en fonction du côté duquel on ajoute l'élément. Et comme l'algorithme cherche à minimiser ces déséquilibres, alors on ajoute du côté où ce déséquilibre se trouve être le plus petit.

Par exemple:

```
W = [7, 4, 10] ; S = 21 ; Ws = 10.5
Wleft = Ws

7       Wleft = Wleft - 7 = 3.5 > 0
7 4     Wleft = Wleft - 4 = -0.5 > 0

On a RSG = -0.5 et LAST = 3.5
|RSG| < LAST, donc on ajoute l'élément à gauche.

On obtient donc l'arbre [[7, 4], 10]
```

On applique ensuite récursivement les règles avec les sous listes gauche et les sous listes droite.

La somme n'étant calculée qu'une seule fois dans l'algorithme implémenté, et n'itérant que sur une partie de chaque sous tableau, la complexité de l'algorithme se trouve être en O(n.log n) en moyenne.

Cet algorithme ne calcule pas forcément des arbres optimaux avec par exemple:

```
W = [20, 3, 7, 19]

L'algorithme optimal (ici le dynamique) renverra : [20, [[3, 7], 19]]
Alors que cet algorithme va renvoyer : [[20, 3], [7, 19]]
```

Après plusieurs observations, l'algorithme se comporte mal lorsque deux petits poids sont entourés par deux poids plus grands.

4. ALGORITHME « ORDONNÉ 2 »

L'idée de cet algorithme était de pouvoir créer l'arbre en partant de la plus petite des feuilles, afin de pouvoir construire l'arbre en remontant (du bottom-up).

Pour cela, l'algorithme va tout d'abord aller chercher l'index du plus petit poids présent dans la liste de poids. Ensuite, il va calculer avec quel noeud il possède le déséquilibre le plus petit, et former un noeud avec.
Cet algorithme résulte d'une observation faite, avec de nombreux tests, montrant que le plus petit élément ne se trouvait jamais (en tout cas, dans les tests effectués) tout seul.

Après avoir retrouvé l'élément le plus petit, ainsi que son partenaire, on garde des pointeurs sur les extrémités gauches et droites de ses deux éléments, afin de pouvoir les sélectionner au fur et à mesure de la construction.

Ainsi, s'il est plus optimal d'ajouter l'élément gauche que le droite, alors on crée un noeud où cet élément se retrouve la feuille à gauche, et l'arbre en construction à droite. Cela est symétrique pour l'élément droite.
Mais si au final, il n'est pas optimal d'ajouter ni à gauche, ni à droite, alors selon les déséquilibres de chacun, on va créer des noeuds contenant des sous arbres avec les éléments de gauche et de droite.
Et si l'un des deux pointeurs se trouve être en dehors des limites, alors on ajoute les éléments de l'autre côté tant que cela ne crée pas de déséquilibre.

Par exemple:

W = [15, 7, 3, 9, 2, 16] ; indexMin = 5 ; node = [9, 2] ; iLeft = 3 ; iRight = 6

balancedLeft = $W_{iLeft} - W_{node}$; balancedRight = $W_{iRight} - W_{node}$
Comme balancedLeft < 0 et balancedLeft < balancedRight alors on applique récursivement tel que:
node = [[rec_left, [9, 2]], rec_right]

Pour rec_right, 16 étant la seul élément, alors on le renvoie en tant que feuille.
Ce qui nous donne donc :
node = [[rec_left, [9, 2]], 16]

Soit W_{rec} = [15, 7, 3] ; indexMinRec = 3; nodeRec = [7, 3]; iLeft = 1; iRight = 4
Comme iRight dépasse le tableau, et que $W_{iLeft} - W_{nodeRec} >= 0$, alors 15 est considéré comme une feuille.

On obtient donc au final l'arbre suivant:
[[[15, [7, 3]], [9, 2]], 16] avec $\Delta = 50$

L'algorithme a pour complexité O(n.log n), car il doit parcourir plusieurs fois le temps : à la recherche de l'index du minimum, et ensuite pour construire l'arbre.

Cet algorithme ne donne pas un arbre optimal, pour l'exemple donné ci dessus. En effet, il ne peut prévoir à l'avance quels vont être les déséquilibres totaux en fonction des éléments gauches et des éléments de droite. Dans l'exemple, il pense que tourner le premier noeud vers la gauche sera plus efficace que vers la droite, alors que ce n'est justement pas le résultat souhaité.

Résultat optimal attendu :

[[15, [7, 3]], [[9, 2], 16]] avec $\Delta = 23$

5. ALGORITHME « DÉSORDONNÉ 1 »

L'idée de cet algorithme est de séparer au maximum les grandes valeurs des petites. En effet, si des grandes valeurs se retrouvent à côté de petites, on se retrouve avec des déséquilibres locaux qui peuvent très vite faire grimper le déséquilibre total.

Pour cela, on doit partir sur une liste triée par ordre décroissant, ainsi qu'une liste d'éléments gauches et une liste d'éléments droits. On suppose que l'ajout d'un élément dans ces listes gardent la propriété que la liste est triée dans l'ordre décroissant.

Tant que la liste de poids courante n'est pas vide, on va ajouter ses éléments à gauche ou à droite, en prenant, chacun leur tour, l'élément le plus grand restant puis l'élément le plus petit restant.
Si le poids de la liste de gauche est plus fort que le poids de la liste de droite, alors on ajoute à droite, et symétriquement si le poids de la liste de droite est plus fort que le poids de la liste gauche.
Par contre, si les deux poids sont égaux, on mettra l'élément du côté possédant la valeur la petit entre les plus grandes valeurs de gauche et de droite.

Par exemple:

$W = [13, 4, 5, 9, 7, 2, 10]$; $W_{trie} = [13, 10, 9, 7, 5, 3, 2]$; $W_{left} = []$; $W_{right} = []$

$W_{left} = [13]$, $W_{right} = []$;; $W_{left} = [13]$, $W_{right} = [2]$;;
$W_{left} = [13]$, $W_{right} = [10, 2]$;; $W_{left} = [13]$, $W_{right} = [10, 3, 2]$;;
$W_{left} = [13, 9]$, $W_{right} = [10, 3, 2]$;; $W_{left} = [13, 9]$, $W_{right} = [10, 5, 3, 2]$;;
$W_{left} = [13, 9]$, $W_{right} = [10, 7, 5, 3, 2]$

Et on applique ensuite récursivement sur les sous listes de gauche et de droite, pour obtenir le résultat suivant :
[[13, 9], [[[10, 5], [7, [4, 2]]]] avec $\Delta = 20$

Cet algorithme a une complexité de l'ordre de O(n.log n).

Cet algorithme ne produit pas forcément l'arbre le plus optimal, avec par exemple:

W = [9, 4, 4, 3, 1, 2] ; Wtrie = [9, 4, 4, 3, 2, 1]

Avec cet algorithme, donnera :
[[9, 3], [4, [3, [2, 1]]]] avec $\Delta = 11$

Avec l'algorithme « Désordonné 2 »:
[9, [[4, 3], [3, [1, 2]]]] avec $\Delta = 7$

6. ALGORITHME « DÉSORDONNÉ 2 »

Après avoir fait de nombreux tests sur de petits arbres, cet algorithme est né d'une observation faite, qui ne fonctionne que sur des très petits arbres en moyenne (voir le benchmark sur cinq valeurs), mais qui dépassant une certaine taille (aux environs de 10), n'est plus du tout efficace.

Le principe est très simple, et repose toujours sur la volonté de séparer une sous liste gauche et une sous liste droite en deux parties les plus égales possible.
Dans un premier temps, on trie la liste dans l'ordre décroissant. On prend la première valeur (étant donc la plus grande de la liste), et on itère en partant de la fin en ajoutant tous les éléments dans la sous liste droite, jusqu'à ce que le poids de cette liste soit supérieur ou égal à la valeur la plus grande.
Ainsi, quand cela se passe, s'il existe une autre plus grande valeur, on crée un noeud avec la première, sinon il devient feuille, et on applique récursivement sur le sous arbre droit.

Par exemple:

$W = [9, 4, 3, 3, 1, 2]$; $W_{trie} = [9, 4, 3, 3, 2, 1]$; $W_{left} = []$; $W_{right} = []$

max = 9
$W_{left} = [9]$ et $W_{right} = [4, 3, 3, 2, 1]$

maxRight = 4
$W_{left} = [4, 3]$ et $W_{right} = [3, 2, 1]$

On obtient donc l'arbre suivant:
[9, [[4, 3], [3, [1, 2]]]] avec $\Delta = 7$

L'algorithme se trouve être en O(n.log n).

Il ne donne pas la solution optimal, avec l'exemple suivant :

```
W = [7, 3, 2, 4, 9]

Son résultat:
   [9, [7, [4, [3, 2]]]] avec Δ = 11
Résultat de l'algorithme « Désordonné 1 »:
   [[9, 4], [7, [3, 2]]] avec Δ = 9
```

II. IMPLÉMENTATIONS

1. ALGORITHMES IMPLÉMENTÉS

i. ALGORITHME DYNAMIQUE ORDONNÉ

L'algorithme dynamique permet, grâce au remplissage d'un tableau, de retrouver à coup sûr le déséquilibre total optimal.
Ainsi, on garde deux tableaux à double dimensions et un tableau à simple dimension, afin de pouvoir optimiser au mieux le temps de travail au détriment de l'espace mémoire. Le complexité de cet algorithme est telle, que cela prendrait beaucoup de temps pour calculer l'arbre optimal pour une liste de poids grande.

Un des tableau à deux dimensions (nommé `*balances*`), contiendra le déséquilibre total pour un sous arbre donné. Le second tableau, nommé `*ways*`, est un tableau contenant l'ensemble des chemins permettant de retrouver l'arbre optimal.
Et enfin le tableau `*sums*`, celui à une seule dimension, n'est présent que pour pré-calculer les sommes. Pour accéder à la somme d'un élément (i, j), on fera appel à la fonction **getRealSum()**, qui recalcule la somme à partir du tableau.

Le premier choix fait était de créer des objets Pair, permettant de stocker le déséquilibre total ainsi que le chemin ensemble, afin de ne devoir garder qu'un seul tableau à double entrée, mais après quelques tests (non donnés…), il y avait une perte d'environs 5 à 6 secondes sur les temps de calculs sur une liste de poids de ~1000 éléments, pour un algorithme tout à fait semblable par rapport à la version fourni.

Étant donné que beaucoup de données sont inutiles (la moitié des tableaux du côté gauche de la diagonale), on limite nos parcours de tableau, afin d'optimiser le nombre de calcul.

On va donc tout d'abord remplir la totalité du tableau `*sums*`, puis calculer toutes les valeurs pour remplir `*balances*` et `*ways*`, en même temps.
Et à partir de `*ways*`, reconstruire l'arbre en partant du bas (afin de ne devoir calculer qu'une seule fois les déséquilibres totaux de chaque sous arbres).

ii. ALGORITHME « ORDONNÉ 1 »

Cet algorithme, se rapprochant de l'idée d'un diviser pour régner, n'utilise qu'un seul tableau linéaire. Une seule fonction sera utilisée pour cet algorithme, prenant les index de début, de fin, et la somme du morceau sélectionné.

Avant de lancer l'algorithme, on calcule une seule fois la somme totale de la liste de poids donnée en argument.

Afin de ne pas devoir recopier la référence du tableau de poids, on garde la référence en tant qu'attribut de l'objet. Cela permet de sauvegarder quelques calculs.

Pour les cas de base, on renverra un noeud ou une feuille.
Pour le reste, on calcule d'abord la moitié de la somme courante, et on fait en sorte de se rapprocher le plus possible de cette moitié de chaque côté.
On va donc itérer jusqu'à atteindre le cas où l'on dépasse cette moitié. On recalcule ensuite la somme de gauche et de droite à partir de la différence de poids obtenue par cette itération, et on applique la fonction récursivement en créant un noeud.
Le faire de cette manière permet une création de l'arbre par le bas, et n'oblige pas à recalculer la totalité des Δ des sous arbres.

iii. Algorithme « Ordonné 2 »

L'idée était de vouloir partir du bas de l'arbre pour remonter petit à petit.
On va avoir l'utilisation de deux fonctions annexes **indexMin()** et **chooseMinUnbalanced()**.
La première fonction, **indexMin()**, prend deux entiers pour correspondre au début et à la fin de la zone à balayer. Elle renvoie l'index du premier plus petit élément trouvé.
Tandis que la fonction **chooseMinUnbalanced()**, prenant trois arguments : le début, la fin et l'index à vérifier, renvoie l'index de l'élément à gauche ou à droite de l'élément pointé par index, provoquant le plus petit déséquilibre possible.

La première étape est de récupérer le tout premier noeud possible, à l'aide des fonctions ci-dessus. Ensuite, on entre dans une boucle permettant de recréer l'arbre en fonction des règles édictées par l'algorithme, et expliquées dans la première partie du problème.

Semblable aux autres algorithmes, il garde un tableau de poids dans l'instance de l'objet, plutôt que de devoir recopier la référence à chaque appel.

iv. Algorithme « Désordonné 1 »

La première étape de l'algorithme est de transférer la liste des poids dans un objet LinkedList, permettant de retirer très simplement des éléments situés au début, et à la fin de la liste. Mais aussi pour insérer des éléments plus facilement.

Le principe va être très simple : on crée une LinkedList pour le sous arbre gauche, et une LinkedList pour le sous arbre droit. On maintient un poids de gauche et un poids de droite, tous les deux initialisés à la moitié de la somme de la liste de poids, et un boolean qui permet de définir où l'on droit prendre le prochain élément (le premier ou le dernier).

A chaque fois que l'on retire un élément, on change la valeur de top.
Ensuite on respecte l'algorithme en changeant les poids de gauche et les poids de droite en fonction des éléments que l'on ajoute.

v. *Algorithme « Désordonné 2 »*

Étant donné que l'algorithme utilise une ArrayList, et qu'il ne possède pas la fonction poll() (permettant de retirer le premier élément d'une file), une fonction de même nom a été rajoutée afin de pouvoir supprimer et renvoyer le premier élément.

L'implémentation est très simple : on maintient trois entiers, le premier correspondant à la valeur la plus grande de la liste, le second qui pourra contenir la seconde plus grande valeur (voir l'algorithme), et la somme, permettant de savoir quand prendre en compte la seconde valeur.

Ensuite, on applique la fonction récursivement avec soit une feuille à gauche (lorsque la somme ne dépasse pas le plus grand élément), soit un noeud comportant les deux plus grandes valeurs.

2. Descriptif des fichiers

Scripts

ALGO	Utilisé pour le lancement du programme. Voir le **Guide d'installation** ci-dessous.
BENCHMARK	Utilisé pour effectuer le benchmark du programme. Celui-ci va créer un nombre de listes de tailles données, et va les lancer sur chacun des algorithmes; il calcule la moyenne des poids de sorties, ainsi que la moyenne du temps passé. Il effectue 100 tests sur des valeurs aléatoires allant d'une taille de 5 à 5000. Une barre de chargement est présente sur chacun des tests. Le benchmark permet d'afficher deux types d'erreurs: les Stack Overflow et les timeout. On considère qu'un algorithme est timed-out à partir du moment où son temps d'exécution dépasse les 15 secondes. Tout le test courant est donc annulé, et placé sous le signe du timeout. Il est possible d'éditer le script afin de réguler le nombre de valeurs, en changeant la variable RAND_LIST, qui correspond à la liste de tout le nombre de poids traités par les algorithmes. **(Attention: il prend beaucoup de temps pour s'exécuter, car certains des algorithmes mettront du temps à renvoyer leurs résultats ; pour le benchmark complet, compter environs 15 minutes.** **Par ailleurs, la commande CTRL-C met du temps à fonctionner (l'arrêt correspondra au timeout du thread en cours)).**

Le package data correspond à la structure de donnée du programme, permettant la création des mobiles équilibrés. Celui-ci calcule au fur et à mesure de la création, en temps constant, l'équilibre local et l'équilibre total d'un noeud.
Le package helpers contient des classes permettant de généraliser des fonctionnalités pour certains algorithmes, mais aussi pour le lancement du programme.
Le package algorithms contient toutes les classes correspondant à l'implémentation de chacun des algorithmes présentés dans ce projet.

FICHIERS SOURCES (JAVA 7)

data.Leaf	Classe correspondant à la structure de donnée d'une feuille. La feuille est considérée comme une extension d'un noeud, ne possédant pas d'enfant et ayant un équilibre (total ou non) de 0.
data.Node	Classe correspondant à la structure de donnée d'un noeud.
data.Tree	Classe correspondant à la structure de donnée d'un arbre. Actuellement la classe ne contient que sa racine, qui n'est autre qu'un noeud.
helpers.CLIParser	CLIParser (alias Command Line Interactive Parser), est une classe statique permettant le parsing des arguments donnés par l'utilisateur, afin mettre à jour les options. Contient une classe de nom Options en son sein, afin de concentrer l'ensemble des options à l'intérieur.
helpers.ExtMath	Contient quelques fonctions utilitaires de mathématique (comme la somme d'un sous-tableau), utilisées par les algorithmes
algorithms.AbstractAlgorithms	Classe abstraite, permettant de généraliser le lancement des algorithmes. C'est elle qui gère les timers, et les outputs en fonction des options données par l'utilisateur.
algorithms.OrderedAlgorithm	Classe correspondant à l'algorithme « Ordonné 1 », étend la classe AbstractAlgorithms.
algorithms.OrderedAlgorithm2	Classe correspondant à l'algorithme « Ordonné 2 », étend la classe AbstractAlgorithms.
algorithms.OrderedDynamicAlgorithm	Classe correspondant à l'algorithme « Dynamique Ordonné », étend la classe AbstractAlgorithms.
algorithms.UnorderedAlgorithm	Classe correspondant à l'algorithme « Désordonné 1 », étend la classe AbstractAlgorithms.
algorithms.UnorderedAlgorithm2	Classe correspondant à l'algorithme « Désordonné 2 », étend la classe AbstractAlgorithms.

3. GUIDE D'INSTALLATION

i. COMPILATION

La compilation se fait tout simplement à l'aide d'un Makefile créé pour l'occasion.
Celui-ci utilisera la commande **ant** par défaut, s'il est installé sur la machine, sinon il utilisera le compilateur **javac**.
Une règle clean a été ajoutée au Makefile afin de supprimer l'intégralité des fichiers *.class* généré lors de la compilation.

```
$ make
(…) compilation output

$ make clean
(…) removal output
```

ii. EXECUTION

Comme indiqué dans le descriptif, un utilitaire a été écrit dans le but de simplifier le lancement du programme java. En effet, sans ce script, il aurait fallu indiquer l'emplacement des *.class*, des libraires (ici, non utilisées) et du fichier correspondant au point d'entrée.

La liste des options disponibles lors de l'exécution du programme :

——noout	N'affiche pas l'arbre de sortie en output. Utilisé notamment lors du benchmark (car totalement inutile), mais aussi lors de l'exécution de tests comportant un nombre important de poids.
—m <int> ——max <int>	Indique le nombre maximum à ne pas dépasser lors de la génération aléatoire de la liste de poids.
—p ——prompt	Lors de la lecture de l'entrée standard, affiche un prompt afin de privilégier la lisibilité.
—r <int> ——random <int>	Indique la volonté de créer une liste de poids aléatoire de taille donnée en argument.
—t ——time	Active l'affichage du temps que l'algorithme a utilisé afin de calculer son résultat.
—od ——ordered-dynamic	Active l'algorithme « Dynamique Ordonné »
—o1 ——ordered-1	Active l'algorithme « Ordonné 1 »
—o2 ——ordered-2	Active l'algorithme « Ordonné 2 »
—u1 ——unordered-1	Active l'algorithme « Désordonné 1 »
—u2 ——unordered-2	Active l'algorithme « Désordonné 2 »

Par défaut, l'algorithme sélectionné est « Dynamique Ordonné ».

Exemple:

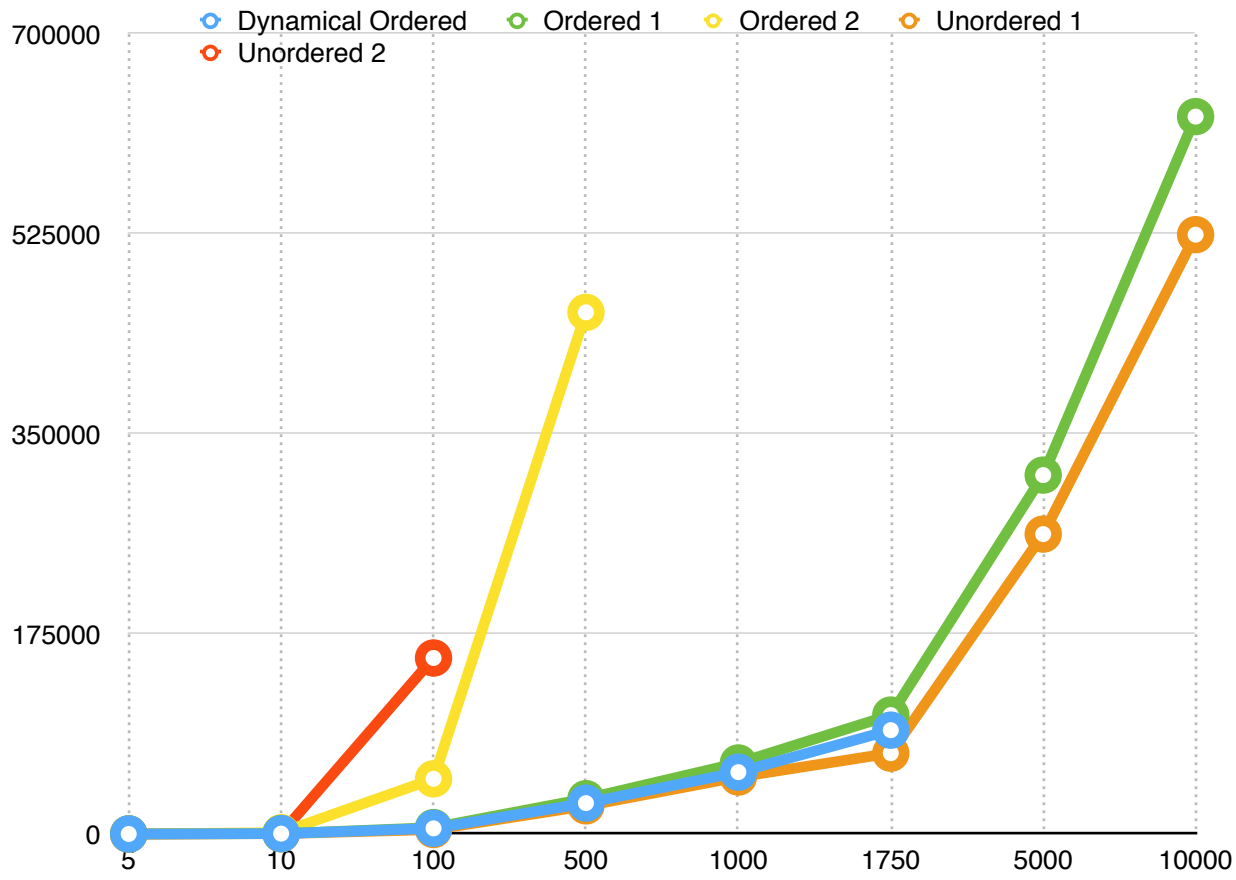
```
$ ./algo ——time —r 100 —m 42
(……) génère l'arbre à l'aide d'une liste de 100 poids compris entre 1 et 42 en utilisant
l'algorithme par défaut, qui est « Dynamique Ordonné »; indique le temps de calcul.

$ ./algo ——time ——prompt —o1
~~~> 7
~~~> 9    <—    Input, saisi par l'utilisateur.
~~~> 3          Un saut de ligne sans aucune saisie
~~~> 10         permet l'arrêt de l'entrée standard, afin
~~~> 2          de procéder à l'algorithme.
~~~>

[[7, 9], [3, [10, 2]]]
20
0.002

(……) Le programme a donc généré un arbre correspondant à la solution et affiche les
informations demandées: le déséquilibre total, le temps, ainsi que l'ensemble des calculs
divisés en trois catégories.
```

III. PERFORMANCES

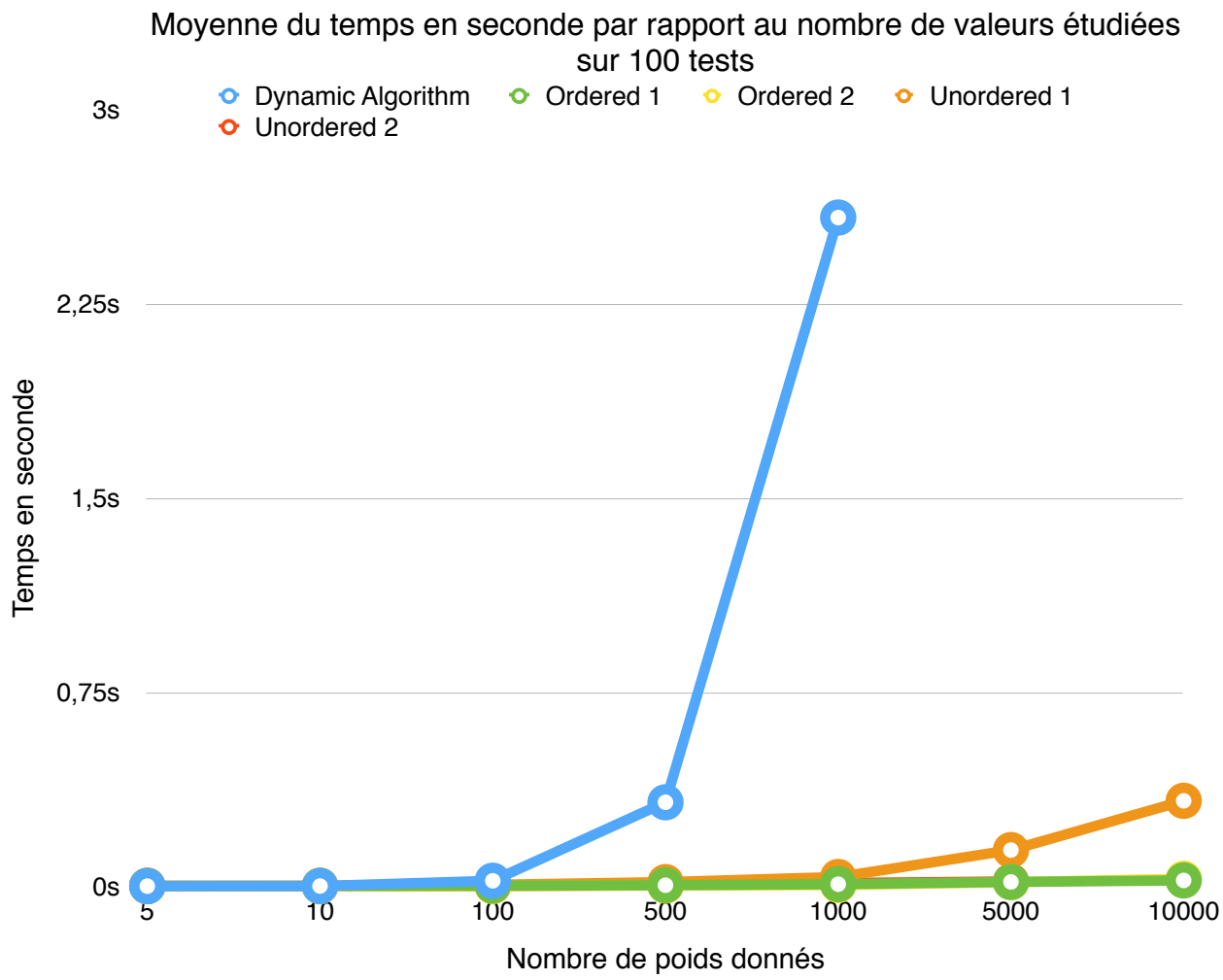
1. GRAPHIQUES

i. EQUILIBRE TOTAL MOYEN / TEMPS

Certains tests des algorithmes « Ordered 2 » et « Unordered 1 » ont été supprimés afin de rendre lisibles les différences sur le graphique (puisque ce sont des algorithmes moins intéressante, voir le benchmark ayant servi pour établir les courbes).



ii. TEMPS/VALEURS



2. BENCHMARK

Voici un bout du benchmark ayant servi à dessiner les courbes ci-dessus. Celui-ci se trouve dans le fichier benchmark2.out; un autre fichier benchmark est disponible (comportant moins de tests que celui-ci)

```
Benchmark for `Dynamical Ordered` with 500 random values, applied 100 times.
  Average weight: 27222
  Average time:   0.32468s
Benchmark for `Ordered 1` with 500 random values, applied 100 times.
  Average weight: 31105
  Average time:   0.00464s
Benchmark for `Ordered 2` with 500 random values, applied 100 times.
  Average weight: 456525
  Average time:   0.00296s
Benchmark for `Unordered 1` with 500 random values, applied 100 times.
  Average weight: 24806
  Average time:   0.01658s
Benchmark for `Unordered 2` with 500 random values, applied 100 times.
  Average weight: 4151250
  Average time:   0.00416s

Benchmark for `Dynamical Ordered` with 1000 random values, applied 100 times.
  Average weight: 54286
  Average time:   2.58156s
Benchmark for `Ordered 1` with 1000 random values, applied 100 times.
  Average weight: 62337
  Average time:   0.0086s
Benchmark for `Ordered 2` with 1000 random values, applied 100 times.
  Average weight: 1284963
  Average time:   0.00508s
Benchmark for `Unordered 1` with 1000 random values, applied 100 times.
  Average weight: 50053
  Average time:   0.03626s
Benchmark for `Unordered 2` with 1000 random values, applied 100 times.
  Average weight: 16797680
  Average time:   0.01288s

Benchmark for `Dynamical Ordered` with 5000 random values, applied 100 times.
  Average weight: N/A (timeout)
  Average time:   N/A (timeout)
Benchmark for `Ordered 1` with 5000 random values, applied 100 times.
  Average weight: 314146
  Average time:   0.01718s
Benchmark for `Ordered 2` with 5000 random values, applied 100 times.
  Average weight: 21865841
  Average time:   0.01636s
Benchmark for `Unordered 1` with 5000 random values, applied 100 times.
  Average weight: 262530
  Average time:   0.14016s
Benchmark for `Unordered 2` with 5000 random values, applied 100 times.
  Average weight: 422048249
  Average time:   0.01906s

Benchmark for `Dynamical Ordered` with 10000 random values, applied 100 times.
  Average weight: N/A (timeout)
  Average time:   N/A (timeout)
Benchmark for `Ordered 1` with 10000 random values, applied 100 times.
  Average weight: 627730
  Average time:   0.02308s
Benchmark for `Ordered 2` with 10000 random values, applied 100 times.
  Average weight: 81377082
  Average time:   0.02958s
Benchmark for `Unordered 1` with 10000 random values, applied 100 times.
  Average weight: 524464
  Average time:   0.33032s
Benchmark for `Unordered 2` with 10000 random values, applied 100 times.
  Average weight: N/A (stack overflow)
  Average time:   N/A (stack overflow)
```

IV. BILAN

1. ALGORITHME ORDONNÉ

Le premier bilan que l'on peut faire, est que l'algorithme « Ordonné 2 » ne fonctionne pas du tout, et produit des arbres possédant des équilibres totaux complètement gigantesques par rapport aux deux autres. Et cela le déclasse directement, même si celui-ci se trouve être plus rapide.

Cependant, les deux premiers algorithmes peuvent convenir tous deux, mais à des besoins différents.
En effet, on s'est assuré que l'algorithme dynamique trouvait à coup sûr les arbres d'équilibre totaux optimaux, mais que son temps d'exécution augmentait par rapport à la taille de la liste.
De manière inverse, l'algorithme « Ordonné 1 » ne trouve pas forcément les arbres optimaux (quoiqu'il s'en rapproche, si l'on regarde bien le benchmark), mais celui se trouve être extrêmement rapide, et peut calculer une taille de poids dépassant des millions (voir le bout de code shell ci-dessous).

```
$ ./algo -r 15000000 -m 100 -o1 --noout --time
453218849
6.304
```

Mais ici, on ne parle que de temps, alors qu'il est aussi très important de prendre en compte la mémoire utilisée (en ignorant la mémoire utilisée par la création de l'arbre). L'algorithme dynamique nécessite tout de même deux tableaux à deux entrée, ainsi qu'un autre à simple entrée (sans compter celui qui contient la liste des poids). On va très vite utiliser énormément de mémoire (quelques tests ont montré une consommation de mémoire jusqu'à **2Go** pour une instance de la machine virtuelle Java, pour une taille de poids de seulement 5000 !).
Tandis que l'algorithme « Ordonné 1 » ne consomme que très peu de mémoire, car en effet, il ne se sert QUE du tableau des poids, et rien d'autre. On n'a donc aucun pic de consommation en mémoire !

En conclusion, pour les algorithmes ordonnés, on peut dire qu'il est préférable de calculer des arbres ayant un déséquilibre total n'étant pas forcément le plus optimal, car il ne demandera pas autant d'espace mémoire, ni de temps.
Mais si les besoins demandent le plus optimal, alors il faut s'assurer que cela ne se fait pas sur une taille de liste des poids trop grande.

2. ALGORITHME DÉSORDONNÉ

D'après les résultats du benchmark, on peut voir que l'algorithme « désordonné » est meilleur sur des petits exemples, mais il se fait très vite dépasser par le premier.

L'algorithme « Désordonné 1 » renverra un arbre, qui n'est pas forcément le plus optimal, mais qui gardera un déséquilibre total moyen (en se basant sur le principe de séparer les plus grands des petits, afin d'éviter des déséquilibres locaux trop grand).

On peut sans conteste dire que le premier est bien plus utile que le deuxième, car il produit un déséquilibre bien plus faible, et surtout, il ne provoque pas de StackOverflow.

3. CONCLUSION

Les algorithmes produisent donc des résultats totalement différents, montrant chacun leurs forces et leurs défauts, et qu'ils peuvent être utilisés selon des besoins totalement différents (une mémoire réduite, un temps réduit, la volonté d'avoir l'arbre le plus optimal, …)