Timsort

Dans ce problème, nous implantons une version simplifiée du tri utilisé par Python (via la procédure `t.sort()` ou la fonction `sort(t)`). Voici la manière dont il est décrit par Tim Peter, son créateur :

> « This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than lg(N!) comparisons needed, and as few as N-1), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays. »

Texte complet à https://svn.python.org/projects/python/trunk/Objects/listsort.txt.

Comme vous l'avez lu, il est basé sur un tri fusion(« mergesort »). Dans ce devoir, nous allons partir du tri fusion classique et y apporter au fur et à mesure différentes améliorations pour aller en direction du timsort.

1 Fusion en place

On désire écrire une *procédure* `fusionEntre`. Cette procédure prendra en entrée un tableau `t` et trois indices `deb`, `m` et `fin`, tels que les portions de tableau `t[deb:m]` ainsi que `t[m:fin]` soient triées. Son but est de faire en sorte que `t[deb:fin]` devienne triée.

Pour ce, on pourra créer un tableau temporaire `tmp` dans lequel on mettra la fusion de `t[deb:m]` et `t[m:fin]` selon le même algorithme que dans la fonction `fusion` vue en cours. Ensuite, on recopiera `tmp` dans `t[deb:fin]`.

1. Programmer cette fonction.
2. Si on a mis dans `tmp` toute la plage `t[deb:m]`, il est inutile d'y mettre ce qui reste de `t[m:fin]`. Voyez-vous pourquoi ? Si votre fonction n'utilisait pas déjà cette petite amélioration, corrigez-la.

2 Découpage du tableau

Voici le principe de base du timsort : au lieu de découper le tableau initial `t` jusqu'à n'obtenir que des singletons et des tableaux vides, nous le découpons en ses sous-tableaux triés (par ordre croissant). Ensuite, nous fusionnerons les morceaux au moyen de `fusionEntre`.

1. Soit `t=[1,2,0,-1,5,6,3]`. Découper `t` en ses sous-tableaux triés. Combien de fusions faudra-t-il effectuer ensuite ?
2. Quelle est l'amélioration par rapport au tri fusion classique ? Dans quelles circonstances est-ce que cela apportera un gros gain ? Dans quelles circonstances est-ce que cela n'apportera rien ?
3. Écrire une fonction `prochainMorceau` prenant en entrée un tableau `t` et un indice `deb` et renvoyant l'indice i maximal tel que `i<=len(t)` et `t[deb:i]` soit trié.

3 Utilisation d'une pile

Les prochains morceaux à trier seront enregistrés dans une pile, que nous nommerons `àFusionner`. Ce sera une pile de couples d'entiers ; pour tout $(a, b) \in \mathbb{N}^2$, si `àFusionner` contient `(a,b)` cela signifie que `t[a:b]` est une portion triée du tableau, et qu'il faudra la fusionner avec les portions triées voisines au moyen de `fusionEntre`.

En pratique, les piles seront implémentées par des tableaux Python, via les méthodes `pop` et `append`. En particulier le « sommet » de la pile correspond à la « droite » du tableau. Comme le but est ici de travailler l'utilisation des piles, on s'interdira pour `àFusionner` l'usage de `len` et de la syntaxe `àFusionner[i]` sauf indication contraire.

Remarque : Pour insérer un couple (a, b) dans la pile `àFusionner`, la commande est `àFusionner.append((a,b))` : ne pas oublier les deux paires de parenthèses.

Nous ferons en sorte que deux plages à la suite dans la pile correspondent toujours à deux plages contigüs dans le tableau. Ainsi, si `(a, b)` et `(c,d)` sont deux éléments de `àFusionner`, `(a,b)` étant juste au dessus de `(c,d)`, alors `d=a`. De plus, la plage au plus profond de `àFusionner` commencera toujours à 0. On notera (I0) cet invariant de boucle.

Le point clef du timsort est de faire en sorte que les fusions s'effectuent entre des plages de tableau plus ou moins de la même taille (comme c'est le cas pour le tri fusion). En effet, si on fait le calcul, on constate que c'est ce qui permet de réduire au maximum le nombre de comparaisons.

Une manière simple pour y parvenir est de maintenir l'invariant de boucle suivant : En reprenant `(a, b)` et `(c,d)` comme ci-dessus, on impose que `d-c >= 2*(b-a)`. Autrement dit chaque plage enregistrée dans la pile doit être au moins deux fois plus longue que la plage située au-dessus d'elle. Nous noterons (I1) cet invariant de boucle.

L'idée est que si à un moment (I1) n'est plus vérifié, nous fusionnerons les deux plages au sommet de `àFusionner` jusqu'à ce qu'il le redevienne.

1. Pour le tableau `t=[1,2,3,4,0,-1,-2,12]`, la pile `àFusionner` contient à un certain instant `[ (0,4), (4,5), (5,6) ]` (la partie `t[6:]` n'a pas encore été lue et ne figure donc pas dans la pile). L'invariant (I1) est-il vérifié ? Sinon quelle(s) fusion(s) faut-il effectuer pour qu'il le devienne ? Préciser ce que deviendra le tableau une fois effectuée(s) cette ou ces fusion(s).
2. Même question pour `t = [1,2,3,4,-1,1,0,2,1,7,8]` et `àFusionner = [ (0,4), (4,6), (6,8)]`.
3. *Taille maximale de la pile :*
   (a) À un instant donné notons $l =$ `len(àFusionner)`. Notons $(a_0, b_0), ..., (a_{l-1}b_{l-1})$ son contenu, (a_0, b_0) étant au sommet et (a_{l-1}, b_{l-1}) au fond. On suppose que $b_0 - a_0 \geqslant 1$ et que (I1) est satisfait. Quel est pour tout $i \in [\![0, n[\![$ la valeur minimale de $b_i - a_i$?
   (b) On note $n =$ `len(t)`. Grâce à (I0), et au fait que $b_{l-1} \leqslant n$, établir une inégalité liant l et n, et en déduire que $l = O(\log n)$.
4. Écrire une procédure `écraseTout` prenant en entrée le tableau `t` et la pile `àFusionner` et qui fusionne au moyen de `fusionEntre` deux à deux les plages indiquées dans `àFusionner` jusqu'à ce qu'il n'en reste qu'une. Après chaque fusion, on prendra soin de remettre dans la pile le couple correspondant à la nouvelle plage triée qui est apparue.
   On s'autorisera à utiliser `len` pour savoir si la pile contient plus d'un élément.
5. Écrire une procédure `écraseSiBesoin` prenant en entrée la pile `àFusionner` et le tableau `t`, et fusionnant au moyen de `fusionEntre` les plages de `t` correspondant aux deux couples situés au sommet de `àFusionner` jusqu'à ce que (I1) soit vrai.
   On supposera que (I1) est vrai pour la pile privée de son sommet (en pratique, (I1) était vrai avant qu'on empile un nouveau morceau). Ceci signifie qu'il suffit que la plage au sommet de la pile soit au moins deux fois plus courte que la plage d'en dessous pour que (I1) soit vrai.
   Indication : On peut se baser sur une boucle « tant que » ou une fonction récursive.
6. Écrire une procédure `écraseTout` prenant en entrée le tableau `t` et la pile `àFusionner` et qui fusionne au moyen de `fusionEntre` deux à deux les plages indiquées dans `àFusionner` jusqu'à ce qu'il n'en reste qu'une. Après chaque fusion, on prendra soin de remmettre dans la pile le couple correspondant à la nouvelle plage triée qui est apparue.
   On s'autorisera à utiliser `len` pour savoir si la pile contient plus d'un élément.

4 Le programme final

La stratégie finale est simple : on parcourt le tableau de gauche à droite au moyen de `prochainMorceau`. À chaque étape, la plage renvoyée par cette fonction sera empilée dans `àFusionner`, et la fonction `écraseSiBesoin` sera appelée pour maintenir l'invariant (I1).

Une fois parvenu au bout du tableau, on appellera `écraseTout` pour fusionner les derniers morceaux.

1. Appliquer cet algorithme au tableau `t=[1,2,0,-1,5,6,3]`. On indiquera à chaque étape la ou les fusion(s) effectuées et l'état de `àFusionner`.
2. Programmer cet algorithme.
3. Quelle est la complexité dans le meilleur des cas ? Et dans quel cas se produit-elle ? Comparer aux autres tris du programme.

5 Améliorations

Sur ce principe de base, le vrai timsort apporte de nombreuses améliorations. En voici quelques unes, elles sont indépendantes, traitez ce que vous pouvez...

5.1 Fusion optimisée

On améliore ici la fonction `fusionEntre`. Soient `t` un tableau et `deb, m, fin` trois indices, tels que `t[deb:m]` et `t[m:fin]` soient triés. Nous allons déterminer l'indice `a` où `t[m]` devra être inséré. Alors, la plage `t[deb:a]` n'a pas à être bougée. Autrement dit il suffira d'appeler `fusionEntre(t, a, m, fin)` (où `fusionEntre` désigne *l'ancienne* `fusionEntre`, programmée à la partie 1).

1. Écrire une fonction `cherchePlaceGauche` telle que `cherchePlaceGauche(t, deb, m)` renvoie l'indice où `t[m]` devra être inséré (en supposant toujours que `t[deb:m]` est triée).

   Cela signifie qu'elle devra renvoyer $a \in [\![\texttt{deb}, \texttt{m}]\!]$ tel que `t[a-1] <= t[m] < t[a]` (supprimer l'inégalité de gauche si `a==deb`, et l'inégalité de droite si `a==m`).

   On demande une conplexité en $O\big(\log(m - deb)\big)$.
2. Expliquer pourquoi on choisit a tel que `t[a-1] <= t[m] < t[a]` et non `t[a-1] < t[m] <= t[a]` .
3. Expliquer pourquoi il est inutile d'opérer la même optimisation sur le côté droit du tableau, c'est-à-dire de chercher au préalable l'indice $b \in [\![m-1, fin-1]\!]$ où aboutira $t[m-1]$.
4. En déduire une fonction `fusionOpti`. Cette fonction pourra utiliser `fusionEntre`.

5.2 Plages décroissantes

Au lieu de se contenter de repérer les plages croissantes dans `t`, on peut aussi récupérer les plages décroissantes.

1. Écrire une procédure `renverseEntre` prenant un tableau `t` et deux indices `deb` et `fin`, qui inverse l'ordre des éléments de `t[deb:fin]`.
2. Modifier la fonction `prochainMorceau` pour que `prochainMorceau(t, deb)` renvoie l'indice i maximal tel que `t[deb :i]` soit trié, dans l'ordre croissant ou décroissant. Dans le cas décroissant, la plage `t[deb:i]` sera en outre retournée, de sorte qu'elle sera finalement elle aussi triée dans l'ordre croissant.

5.3 Insertion pour peu d'éléments

En comparant les différents tris pour différentes tailles de tableaux, on se rend compte que le tri par insertion est plus efficace pour de petits tableaux. Ainsi, une manière d'optimiser notre tri est de passer au tri par insertion pour de petites plages.

En pratique, nous fixons une variable globale entière `TAILLE_MIN` [1] qui représentera la taille mimimale d'une plage renvoyée par `prochainMorceau`. Si la prochaine plage triée de `t` est de longueur inférieur à `TAILLE_MIN`, nous utiliserons des insertions pour y rajouter le nombre voulu d'éléments.

1. Écrire une procédure `insèreEntre` vérifiant les spécifications :
   - Entrée : un tableau `t` et deux indices `deb` et `i` ;
   - Précondition : `t[deb:i]` est trié ;
   - Effet : `t[i]` est déplacé de sorte que `t[deb:i+1]` devienne trié.

   Dans le vrai Timsort, `TAILLE_MIN` est de l'ordre de 64, mais pour vos tests, prenez plus petit.
2. Modifier la fonction `prochainMorceau` pour faire en sorte que la plage renvoyée soit de longueur au moins `TAILLE_MIN` (sauf naturellement si le bout du tableau est atteint).

1. Traditionnellement, les variables globales sont notées en majuscule.