Le tri à bulles et quelques variations

Voici un corrigé (assez long) de ce petit devoir, j'y reprends tous les algorithmes demandés, avec un peu plus de variantes, on y trouve quelques tests, permettant d'évaluer les nombres moyens de comparaisons et d'échanges (quand cela a un sens). J'ai finalement mis un programme Python implémentant l'un des algorithmes (le plus élaboré) avec lequel j'ai fait quelques tests (mais Python n'est vraiment pas mon langage de prédilection pour tester quelqu'algorithme que ce soit, et les tests les plus exhaustifs, je les ai fait en programmant dans un autre langage). On trouvera donc dans la suite :

— de cette page-ci à la page 4 des remarques d'ordre général sur le travail que j'avais donné à faire, le travail que vous m'avez rendu, et des particularités du problème qui n'étaient pas toujours très explicites ;

— de la page 5 à la page 14, vous trouverez la correction des exercices d'un point de vue algorithmique exclusivement (avec un petit bonus [ou malus, don't know which]) ;

— de la page 15 à la page 19 on trouve un petit programme Python implémentant le *shaker-sort* et permettant de faire des tests, de manière pas trop chronophages (pour moi, pas pour l'ordinateur).

— donc, pour les gens pressés, les seules pages qui concernent très précisément le travail algorithmique qui était demandé sont les pages 5–6 et 11–12 ; tout le reste, en dehors de la programmation, ce sont des choses qui tournent autour du même thème mais qui n'étaient pas explicitement demandées pour l'évaluation.

Il y a forcément des erreurs dans tout ce que j'ai écrit ; si vous en trouvez, je vous remercie de me les signaler. Si certaines choses ne sont pas claires pour vous dans ce que j'ai écrit là, posez-moi des questions et j'essaierai de vous expliquer.

Quelques remarques d'ordre général sur vos travaux

1. Dans les expérimentations de l'algorithme de base du tri par bulle, ce n'est pas la peine de compter, par programme, le nombre de comparaisons puisque ce nombre est $n(n-1)/2$ dans tous les cas (ou parfois $(n-1)^2$ quand les choses ne sont pas très bien faites, mais en tous cas, il est déterminé exactement avant exécution et ne dépend que de la taille du tableau, pas de son contenu précis). Il n'y a qu'après les optimisations successives permettant d'arrêter le tri lorsque l'on a constaté que le tableau était trié qu'il faut les compter, puisqu'alors leur nombre est variable pour une même taille de tableau.

2. Comme cela transparaît dans ce que je viens de dire, certains, parmi vous, ont implémenté *bubble-sort* par deux boucles imbriquées comme ceci :

for i **in** $1, \ldots, n-1$ **loop**
  for j **in** $1, \ldots, n-1$ **loop**
    .....

Or, sans chercher l'optimisation ultime, il n'est quand même pas compliqué de constater que la plage que doit examiner la boucle interne est d'une amplitude qui, dans le pire des cas, diminue d'une unité à chaque itération de la boucle principale, l'invariant assurant qu'une partie du tableau est à sa place définitive (aux indices hauts).

3. Beaucoup, parmi vous, se sont « amusés » à comparer les différents algorithmes entre eux. Si vous voulez vraiment comparer ces algorithmes, alors cela peut être fait sur les mêmes jeux de données, c'est-à-dire, en tirant un tableau au hasard et en le passant à la moulinette de chaque algorithme (mais cela nécessite de recopier ce tableau pour le sauvegarder avant de le trier afin d'opérer exactement sur le même tableau quel que soit l'algorithme). Évidemment, ceci n'a aucune

importance (en théorie au moins) si l'on fait un grand nombre d'expérimentations pour calculer des moyennes.

4. Si la plupart d'entre vous ont prouvé la correction du principe de l'algorithme du tri à bulles (plus ou moins facilement et plus ou moins simplement), aucun d'entre vous n'a prouvé que les optimisations demandées dans l'exercice 4 étaient justifiées ; or, il est aussi important de prouver que des optimisations sont correctes que de prouver qu'un algorithme de base est correct.

5. Vous avez pratiquement tous testé vos programmes sur des échantillons de tableaux de grande taille mais aucun parmi vous (sauf si quelque chose m'a échappé [auquel cas, je vous prie de m'excuser] malgré mes vérifications) ne semble avoir utilisé une fonction de vérification qu'un tableau est bien trié. Or, en particulier après les optimisations de l'exercice 4, il n'est pas inutile de vérifier cela sur les tableaux de grande taille tirés au hasard puis triés (la gestion des indices est assez délicate et il faut être sûr de son coup). J'ai même vu des programmes qui affichaient des tableaux de taille 20 000, faut avoir le moral et la patience !

6. Sur votre programmation Python, je n'ai pas grand chose à dire (je ne suis pas un programmeur Python, comme je l'ai déjà dit… et le dirai probablement encore plusieurs fois). Cependant, même non spécialiste de Python, j'ai bien vu que vous aimiez bien, pour la plupart, traduire l'échange de variables $x \xleftrightarrow{\text{swap}} y$, par l'unique instruction « `x, y = y, x` », ce qui semble effectivement très rigolo. Sachez tout de même traduire l'échange des valeurs de deux variables par un algorithme transposable dans n'importe quel langage : $aux \longleftarrow x$; $x \longleftarrow y$; $y \longleftarrow aux$. De plus, il n'est pas inutile de savoir[1] comment Python traduit votre instruction « `x, y = y, x` » et comment il traduit la séquence classique.

7. Pour terminer sur une note gaie, j'ai été impressionné par la qualité de la présentation de la plupart de vos tests statistiques, avec des courbes, des diagrammes en bâtons, etc.

Remarques supplémentaires

1. Je ne présenterai pas exhaustivement les programmes qui m'ont permis de faire des tests, mais voici quelques résultats sur des tableaux de taille croissante, pour trois algorithmes : *bubble-sort* de base, sans aucune optimisation (repéré par B1 dans le tableau), une seconde version de *bubble-sort* (notée B3, qui n'était pas demandée dans mes exercices) et qui fait le type d'optimisation qui était demandé dans l'exercice 3 mais pour *bubble-sort*, et enfin, la version de *shaker-sort* qui était demandée dans l'exercice 3 (notée S3).

n	Échanges	Tests-B1	Tests-B3	Tests-S3	B3/B1	S3/B1
1000	253210	499500	495619	339648	0.992	0.680
2000	989186	1999000	1996157	1330679	0.999	0.666
4000	4019203	7998000	7991991	5354019	0.999	0.669
8000	16149960	31996000	31980560	21500480	1.000	0.672
16000	64453809	127992000	127930544	85893289	1.000	0.671
32000	257439014	511984000	511903305	343672122	1.000	0.671
64000	1023259219	2047968000	2047664141	1362774348	1.000	0.665
128000	4102542054	8191936000	8191452346	5469582038	1.000	0.668
256000	16373588598	32767872000	32766616978	21835145435	1.000	0.666
512000	65 489 305 419	131 071 744 000	131 071 000 448	87 331 709 424	0.999	0.666
1024000	261 986 507 809	524 287 488 000	524 273 301 952	349 355 756 470	0.999	0.666

On peut faire quelques observations, à partir des chiffres que l'on voit là, également à propos des chiffres que l'on ne voit pas dans ce tableau (si on ne les y voit pas, c'est parce qu'ils sont inutiles, mais il n'est pas anodin de souligner pourquoi).

1. Après tout, vous êtes un peu des informaticiens.

2. Comme cela est dit par ailleurs, le nombre d'échanges effectués par les différents algorithmes sur un même tableau est constant : quelle que soit l'optimisation (ou la non-optimisation) mise en place, cela ne change en rien le nombre d'échanges (qui est égal au nombre d'inversions de la permutation sous-jacente [confer exercice 2]), ce n'est donc pas la peine de répéter ce nombre pour chaque algorithme.

3. On vérifie donc bien expérimentalement ce qui est démontré dans l'exercice 2, à savoir que le nombre moyen d'échanges est proche de $n(n-1)/4$ (la moitié du nombre de tests de l'algorithme B_1, que l'on trouve dans la colonne « Tests-B1 »).

4. Pour ce qui est du nombre de tests, on constate, expérimentalement, que tous les algorithmes, à l'exception du dernier, font à peu près le même nombre de tests, ce qui montre que, tant que l'on ne fait pas des allers-retours dans le tableau, on ne change pas fondamentalement le nombre de tests, autrement dit, les optimisations ne servent pas à grand chose ; mais pour ce qui concerne la version programmée de *shaker-sort*, dans l'exercice 3, il semble bien que le nombre de tests soit proche des 2/3 du nombre de tests effectués par l'algorithme de base (je n'en ai pas de démonstration, j'ai constaté cela en rédigeant cette note), c'est assez surprenant et, pour l'instant, ce n'est qu'une hypothèse.

5. Pour information (et peut-être comparaison), mon test des trois algorithmes pour une taille de tableau de 1 024 000 a demandé sur ma machine un poil plus que 3h30 (en fait, le calcul de la dernière et de la pénultième lignes de mon tableau a demandé 4h30).

6. Enfin, il ne faut pas oublier que bubble-sort et toutes ses variantes forment la famille des tris les plus inefficaces dans le cas général ; en revanche, *bubble-sort* et, mieux, *shaker-sort*, dans leurs versions optimisées sont extrêmement efficaces lorsqu'on les applique à un tableau presque trié (ou très peu dérangé) ; cela revient à dire que ces deux algorithmes sont des tris très efficaces lorsqu'il n'y a rien à faire… il ne faut pas rire, c'est une situation qui peut se présenter ; dans le cas, par exemple, où il n'y a qu'une poignée de valeurs qui ne sont pas à leur place, et qu'on le sait à l'avance, alors *shaker-sort* sera probablement le plus efficace des tris (largement devant les bolides que sont *quicksort*, *merge-sort.* ou *heapsort*).

Évaluation

1. En ce qui concerne l'évaluation, j'ai validé le travail à partir du moment où l'invariant de *bubble-sort* (dans l'exercice 1) était à peu près correctement exprimé, que sa preuve d'invariance. était au moins esquissée, et que la compexité était évaluée correctement (nombre exact de comparaisons, majorant du numbre d'échanges). J'ai fait l'hypothèse qu'avec l'expérience, tout ceci deviendrait un peu plus naturel et plus fluide (car, cela peut être très lourd… les preuves d'algorithmes ne sont pas réputées pour leur légèreté).

2. Les deux optimisations du tri (qu'il soit *bubble* ou *shaker*) ont parfois été totalement comprises, parfois seule l'optimisation consistant à arrêter l'algorithme dès qu'il n'y a plus d'échange était envisagée (et dans ce cas, il aurait été bien d'utiliser un vrai booléen au lieu d'utiliser un entier). Mais comme je l'ai dit plus haut, il aurait été bien, dans tous les cas, de donner des éléments de preuve pour montrer que ces optimisations sont correctes. Je m'explique : quand on optimise au mieux (selon ce qui est fait ici dans les exercices 6 et 3), il ne faut pas se tromper dans les sauts d'indices que l'on effectue ; il faut avoir prouvé que si le dernier échange que l'on fait en montant est $t_i \overset{\text{swap}}{\longleftrightarrow} t_{i+1}$ alors la plage d'indices allant de $i+1$ à n est triée et en place (mais pas la plage allant de i à n), et que lorsque le dernier échange en descendant est $t_{j-1} \overset{\text{swap}}{\longleftrightarrow} t_j$, alors la plage d'indices allant de 1 à $j-1$ est triée et en place (mais pas la plage allant de 1 à j). Bref, une optimisation, ça se prouve comme le reste (et même encore plus que le reste).

Remarques terminales : numérotation des exercices

1. Après avoir lu vos devoirs, je me suis aperçu que j'aurais dû être plus précis sur certaines choses, en particulier pour vous faire mieux sentir les optimisations que j'attendais dans l'exercice 4) ; j'ai donc modifié la rédaction de l'ensemble des exercices ce qui fait que, maintenant, dans ce corrigé, vous avez, pour le même prix 6 exercices au lieu de 3.

2. Vous verrez, à la suite, que les exercices sont présentés dans un ordre qui n'est pas celui de leur numéro. Malgré le changement de présentation des algorithmes demandés, j'ai essayé de conserver la numérotation originale du devoir. Ainsi, les exercices 1, 2 et 3 demandés dans le devoir sont toujours les exercices 1, 2 et 3 qui apparaissent ici, mais ils ont été désordonnés[2]. Ils figurent à des places qui me semblent plus conformes à une progression (pédagogique, ou dans l'ordre de difficulté croissante), ainsi, la suite des exercices que vous pouvez lire dans la suite est : 1, 4, 5, 6, 3, 2.

2. C'est un peu normal pour un devoir sur les tris.

Exercice 1. Le tri à bulles (bubble-sort)

Voilà le principe du tri à bulles pour un tableau T indexé par $\{1, \ldots, n\}$. On parcourt le tableau T en examinant les paires d'éléments consécutifs : si les deux éléments d'une paire sont correctement ordonnés l'un par rapport à l'autre, on ne fait rien, sinon on les échange de manière à les réordonner.

a. Si l'on parcourt le tableau à trier des indices faibles vers les indices forts en appliquant le procédé précédent, quelle propriété possède la valeur située à l'indice n dans le tableau T après ce premier passage ? L'algorithme du tri par la méthode de la bulle consiste à répéter ce processus de parcours et d'échanges tant que le tableau n'est pas totalement trié. Établir et prouver l'algorithme du tri à bulles.

b. Pour cet algorithme, quel est le nombre de comparaisons d'éléments de tableau ? Quel est le nombre maximum d'échanges d'éléments de tableau ? Le nombre minimum ? Exhiber dans chacune de ces deux dernières situations des configurations du tableau les réalisant.

Solution de l'exercice 1.

a. Il n'est pas difficile de prouver qu'à la fin du premier parcours, la valeur qui se trouve alors à l'indice n est la plus grande valeur se trouvant dans le tableau, il ne reste alors qu'à itérer ce processus, sur la partie de tableau en excluant le dernier indice, etc. Plus généralement, l'algorithme suivant, agissant sur un tableau indexé par $\{p, \ldots, q\}$:

for j **in** $p, \ldots, q-1$ **loop**
  T_j est la plus grande valeur dans la tranche $p, \ldots, j$
  if $T_j > T_{j+1}$ **then**
    $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j+1}$;
  end if ;
  T_j est la plus grande valeur dans la tranche $p, \ldots, j+1$
end loop ;

a la propriété qu'en fin d'algorithme, les valeurs du tableau ont pu être permutées ensemble et, à l'indice q, on trouve la valeur la plus grande du tableau. L'assertion indiquée en début et en fin de boucle est clairement valide et prouve qu'en fin de boucle ($j = q - 1$), alors T_q est la plus grande valeur du tableau.

Pour le tri à bulles, cet algorithme est une routine auxiliaire, et l'invariant de l'algorithme, que l'on va préciser dans un instant, raconte que les valeurs dans les indices hauts du tableau sont les plus grandes valeurs du tableau et qu'elles sont rangées en ordre croissant (on voit une représentation graphique de l'invariant sur la figure 1). Voici l'algorithme :

Algorithme *bubble-sort*, version de base, B_1

for i **in reverse** $2, \ldots, n$ **loop**
  T est trié entre $i+1$ et n et $T(1, \ldots, i) \leq T(i+1, \ldots, n)$
  for j **in** $1, \ldots, i-1$ **loop**
    if $T_{j+1} < T_j$ **then** $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j+1}$; **end if** ;
  end loop ;
  T est trié entre i et n et $T(1, \ldots, i-1) \leq T(i, \ldots, n)$
end loop ;

L'invariant de cet algorithme est précisément l'assertion $\mathcal{P}_i$:

— les valeurs du tableau T sont celles qui étaient initialement dans le tableau, éventuellement permutées (on ne perd pas d'information, et on n'en invente pas !) ;
— les valeurs situées aux indices i à n sont les plus grandes valeurs contenues dans le tableau ;
— de surcroît, ces valeurs sont rangées en ordre croissant (et à leur place définitive).

L'invariant de cet algorithme est exactement le même que dans le cas du tri par sélection, c'est seulement le procédé de conservation de cet invariant qui différencie l'algorithme du tri à bulles de l'algorithme du tri par sélection.

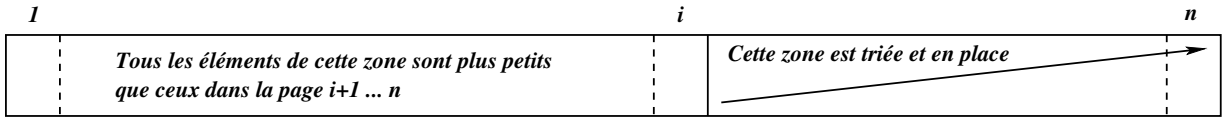

Figure 1. Invariant du tri par la méthode de la bulle

Dans l'algorithme, l'assertion $\mathcal{P}_{i+1}$ est valide en début d'itération de la boucle principale ($\mathcal{P}_i$ étant valide en fin d'itération). Lorsque l'algorithme se termine, l'assertion $\mathcal{P}_2$ est vérifiée ; elle dit que le tableau est trié entre les indices 2 et n, mais elle dit aussi que les valeurs à ces indices sont au moins égales à la valeur à l'indice 1 ; par conséquent, le tableau entier est trié.

b. Le nombre de tests effectués par l'algorithme est évidemment de $n(n-1)/2$, aucun test n'est évité. Le nombre d'échanges, dans le meilleur des cas est 0 (tableau déjà trié), et de $n(n-1)/2$ dans le pire des cas (tableau inversé).

Exercice 4. *Shaker-sort*, **une petite variation de** *bubble-sort*

L'algorithme *shaker-sort* est simplement une petite adaptation de *bubble-sort* dans laquelle on fait successivement des passes ascendantes et des passes descendantes dans le tableau : une passe ascendante fait « monter » les valeurs les plus grandes aux indices forts du tableau et une passe descendante fait « descendre » les valeurs les plus petites aux indices faibles du tableau. Donner cet algorithme. Quelle est sa complexité en nombre de comparaisons et en nombre d'échanges ?

Solution de l'exercice 4.

La seule « difficulté » de cette variation « rigolote » est la gestion des indices pour les boucles, le plus simple est d'utiliser deux variables, l (pour *left*) qui indique l'indice le plus faible et r (pour *right*) qui indique l'indice le plus fort de la plage d'indices que l'on considère ; chacune de ces deux variables est incrémentée ou décrémentée à chaque itération et on s'arrête lorsque ces deux indices se croisent (notion qu'il faut préciser, voir plus bas).

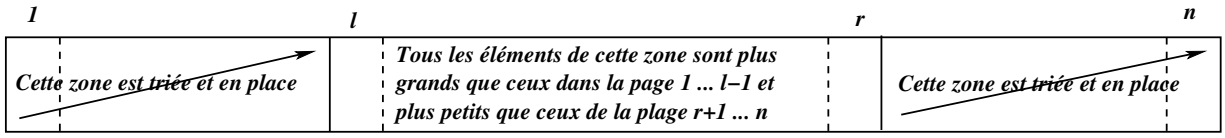

Figure 2. Invariant du tri par l'algorithme *shaker-sort*

Voici tout d'abord l'invariant, sur la figure 2, et puis, l'algorithme, ci-dessous :

Algorithme *shaker-sort*, version de base, S_1

$l \longleftarrow 1$; $r \longleftarrow n$;
loop
  $T(1,\ldots,l-1) \leq T(l,\ldots,r) \leq T(r+1,\ldots,n)$, les plages extrêmes étant triées
  exit when $r \leq l$
  $T(1,\ldots,l-1) \leq T(l,\ldots,r) \leq T(r+1,\ldots,n)$, les plages extrêmes étant triées et $l+1 \leq r$
  for j **in** $l,\ldots,r-1$ **loop**
    if $T_{j+1} < T_j$ **then** $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j+1}$; **end if** ;
  end loop ;
  $r \longleftarrow r-1$;
  $T(1,\ldots,l-1) \leq T(l,\ldots,r) \leq T(r+1,\ldots,n)$, les plages extrêmes étant triées
  exit when $r \leq l$;
  $T(1,\ldots,l-1) \leq T(l,\ldots,r) \leq T(r+1,\ldots,n)$, les plages extrêmes étant triées et $l \leq r-1$
  for j **in reverse** $l+1,\ldots,r$ **loop**
    if $T_j < T_{j-1}$ **then** $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j-1}$; **end if** ;
  end loop ;
  $l \longleftarrow l+1$;
end loop ;

Le nombre de tests exécutés par cet algorithme est exactement le même que celui opéré par *bubble-sort*, à savoir $n(n-1)/2$. Le nombre d'échanges des valeurs dans le tableau est également le même, au minimum, au maximum, et en moyenne, les raisonnements combinatoires (confer exercice 2) ne changent pas sous prétexte que l'on alterne les passes ascendantes et les passes descendantes.

Exercice 5. Tri par la méthode de la bulle : suppression d'itérations inutiles

Combien d'itérations de *bubble-sort* sont **nécessaires** pour trier le tableau suivant (et pour décréter qu'il est trié) ?

1																													**32**
1	2	3	4	5	6	7	25	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	26	27	28	29	30	31	32

Comment peut-on prendre en compte ce phénomène et optimiser un tantinet l'algorithme de *bubble-sort*. Cela change-t-il la complexité de l'algorithme et, si oui, en quoi ?

Solution de l'exercice 5.

Partant du tableau organisé comme suit :

1																													**32**
1	2	3	4	5	6	7	25	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	26	27	28	29	30	31	32

une unique itération (montante) amène à sa place la seule valeur qui n'y est pas (le nombre 25 à l'indice 8), et le tableau est alors trié ; mais il faut une seconde itération de *bubble-sort* pour pouvoir affirmer que le tableau est trié. L'événement déclencheur de cette optimisation est le fait que l'on effectue une passe complète (une itération montante) sans aucune permutation de valeurs ; cela correspond d'ailleurs exactement à l'exécution de l'algorithme standard qui permet de déterminer si un tableau est trié ou pas (une passe de bubble-sort sur toute la partie de tableau concerné sans aucun échange). Voilà ce que donne cet algorithme, où *Sorted* est une variable locale booléenne :

Algorithme *bubble-sort*, version légèrement améliorée, B_2

for i **in reverse** $2, \ldots, n$ **loop**
  T est trié entre $i+1$ et n et $T(1, \ldots, i) \leq T(i+1, \ldots, n)$
  $Sorted \longleftarrow True$;
  for j **in** $1, \ldots, i-1$ **loop**
    if $T_{j+1} < T_j$ **then** $Sorted \longleftarrow False$; $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j+1}$; **end if** ;
  end loop ;
  Si *Sorted* vaut vrai, alors le tableau est également trié entre 1 et i
  exit when *Sorted* ;
  T est trié entre i et n et $T(1, \ldots, i-1) \leq T(i, \ldots, n)$
end loop ;

Pour ce qui est de la complexité, son ordre de grandeur, que ce soit du point de vue des comparaisons ou des échanges, reste le même, $\mathcal{O}(n^2)$, la différence est simplement que le nombre de tests qui, auparavant était toujours de l'ordre de $n^2/2$ peut maintenant chuter à n (si le tableau est trié dès le début, ou pas loin de l'être).

Exercice 6. Tri par la méthode de la bulle : optimisation ultime

Considérer le tableau suivant, et lui appliquer une première itération (montante) de *bubble-sort*.

1																														**32**
12	1	6	2	3	4	5	7	20	8	9	10	11	13	27	14	15	16	17	18	19	21	22	23	24	25	26	28	29	30	31	32

À partir de quel moment, lors de cette première itération, n'y a-t-il plus d'échanges de valeurs dans le tableau ? Que peut-on en déduire ? Appliquer alors une seconde itération (toujours montante) de *bubble-sort* et faire les mêmes observations qu'après la première itération. En déduire un optimisation de *bubble-sort* qui réduit autant qu'il est possible, à chaque itération, l'intervalle d'indices dans lequel on examine les valeurs dans le tableau.

Solution de l'exercice 6.

On part donc du tableau suivant :

1																														**32**
12	1	6	2	3	4	5	7	20	8	9	10	11	13	27	14	15	16	17	18	19	21	22	23	24	25	26	28	29	30	31	32

et on applique une passe (ascendante) du tri par bulles. Cette passe commence par transporter la valeur 12, jusqu'à l'indice 9 où l'on rencontre la valeur 20 qui prend la suite, puis à l'indice 17 on trouve la valeur 27 qui va être transportée jusqu'à l'indice 27 (étonnant, non ?). Le dernier échange concernait les indices 26 et 27, et après cela, il n'y a plus eu d'échanges. Par conséquent, à partir de l'indice 27, les valeurs du tableau sont triées et à leur place définitive. Plus besoin d'examiner les valeurs entre les indices 27 et 32. On schématise cela dans la représentation de l'état du tableau après cette première passe

1																								**26**						
1	6	2	3	4	5	7	12	8	9	10	11	13	20	14	15	16	17	18	19	21	22	23	24	25	26	27	28	29	30	31	32

À la suite de cela, on reprend une passe (ascendante de bubble-sort) entre les indices 1 et 26 ; cette passe va transporter la valeur 6 à sa place, puis la valeur 12 à sa place, et enfin la valeur 20 à sa place.

1																	**19**													
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Le dernier échange ayant eu lieu aux indices 19 et 20, on en déduit qu'il est inutile d'examiner les valeurs situées aux indices 20 à 32, elles sont à leur place définitive. On fait alors encore une passe ascendante, entre les indices 1 et 19, et on constate qu'aucun échange n'est effectué, ce qui montre que le tableau est maintenant trié. Ce tri a demandé 74 comparaisons (au lieu de 496 avec l'algorithme de base, ou 90 avec la version légèrement améliorée, version B_2) et 38 échanges (chiffre constant quel que soit la méthode de la bulle utilisée). Voici maintenant l'algorithme implémentant cette méthode :

Algorithme *bubble-sort*, version optimisée, B_3

$r \longleftarrow n$;
while $r > 1$ **loop**
  T est trié entre $r+1$ et n et $T(1,\ldots,r) \leq T(r+1,\ldots,n)$
  $Last \longleftarrow 1$;
  for j **in** $1,\ldots,r-1$ **loop**
    if $T_{j+1} < T_j$ **then** $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j+1}$; $Last \longleftarrow j$; **end if** ;
  end loop ;
  $r \longleftarrow last$;
end loop ;

L'invariant est exprimé en début de boucle, il est pratiquement identique à celui de la version de base de *bubble-sort* (au nom de l'indice-clef près). La complexité en nombre d'échanges est identique à celle des autres versions de *bubble-sort*, mais la complexité en comparaisons est imprévisible dans le cas général mais reste dans $\mathcal{O}(n^2)$, avec des chutes drastiques dans certaines configurations comme celle que l'on a montrée en exemple plus haut.

Exercice 3. Optimisation du shaker-sort

En triant effectivement quelques tableaux avec la méthode de la bulle ou *shaker-sort*, on constate assez rapidement que, bien souvent, le tableau est trié alors que l'algorithme poursuit son exécution jusqu'à la fin prévue[3]. Dans le cas de *shaker-sort*, on peut examiner l'exemple suivant de tableau :

1																												**32**
1	2	3	20	4	5	7	8	9	27	10	11	13	14	15	16	6	17	18	19	21	22	23	24	25	26	12	28	29	30	31	32

Examiner soigneusement l'exécution de cet algorithme et déterminer la plage exacte des indices qui doivent être examinés à chaque itération. Il est bien clair que les plages d'indices qui doivent être examinées successivement pour la bonne exécution de *shaker-sort* ne diminuent pas régulièrement mais, au contraire, par bonds.

L'optimisation ultime de *shaker-sort* consiste donc à réduire la plage d'indices d'examen du tableau en tenant compte de ces observations.

Solution de l'exercice 3.

Voici donc comment l'on pourrait trier le tableau donné en exemple, en appliquant des passes alternées de la méthode de la bulle, une fois dans un sens, l'autre dans l'autre sens ; nous partons de ce tableau :

1																												**32**
1	2	3	20	4	5	7	8	9	27	10	11	13	14	15	16	6	17	18	19	21	22	23	24	25	26	12	28	29	30	31	32

On fait tout d'abord une passe dans le sens des indices croissants, de l'indice 1 à l'indice 32, et on se retrouve dans la situation suivante :

1																							**26**						
1	2	3	4	5	7	8	9	20	10	11	13	14	15	16	6	17	18	19	21	22	23	24	25	26	12	27	28	29	30	31	32

Dans cette situation, on remarque qu'après l'indice 26 (dernier indice où l'on a fait un échange, qui est $t_{26} \overset{\text{swap}}{\longleftrightarrow} t_{27}$), le tableau est rangé définitivement ; par conséquent ce n'est plus la peine d'examiner la plage d'indices allant de 27 à 32. On peut maintenant faire une passe descendante sur le tableau, de l'indice 26 à l'indice 1 et on obtient alors la configuration suivante :

						7																	**26**						
1	2	3	4	5	6	7	8	9	20	10	11	13	14	15	16	12	17	18	19	21	22	23	24	25	26	27	28	29	30	31	32

On fait maintenant une observation analogue à la précédente : à partir de l'indice 6, dernier indice qui a été l'occasion de faire un échange (qui est précisément $t_6 \overset{\text{swap}}{\longleftrightarrow} t_7$), jusqu'à l'indice 1, le début du tableau est trié et à sa place définitive, on peut donc exclure des traitements suivants, la plage d'indices allant de 1 à 6. On fait alors une autre passe ascendante sur le tableau, dans la plage d'indices comprise entre 7 et 26, qui nous conduit à la configuration suivante :

						7												**19**											
1	2	3	4	5	6	7	8	9	10	11	13	14	15	16	12	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Et dans cette configuration, une observation analogue à celle que nous avons faite lors de la première passe ascendante nous conduit à exclure de la suite la plage d'indices allant de 20 à 26. On fait alors de nouveau une passe descendante, qui nous donne :

											13							**19**											
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Et le tableau est trié… mais il faut une dernière passe pour s'en rendre compte !

Le bilan est que nous avons effectué $31 + 25 + 19 + 12 + 6 = 93$ tests, au lieu de $32 \times 31/2 = 496$ qui auraient été effectués par la version initiale de *shaker-sort* (ou celle de *bubble-sort*, puisqu'elles ont strictement la même complexité.

3. Heureusement ! si un algorithme réfléchissait, ce ne serait plus un algorithme !

Pour concrétiser l'algorithme, nous allons considérer deux indices, l (pour *left*) et r (pour *right*), qui délimitent la plage d'indices à l'intérieur de laquelle on doit encore trier le tableau (aux indices qui ne sont pas dans cette plage, les valeurs du tableau sont à leur place définitive). Et on va alterner les passes ascendantes et les passes descendantes en gérant soigneusement ces deux indices afin de réduire autant que faire se peut les itérations suivantes de l'algorithme. Donc, l'invariant que l'on va maintenir à des points clefs de l'algorithme sera :

> $\mathcal{P}$: la plage d'indices $\{1, \ldots, l-1\}$ est triée et les valeurs qu'elle contient sont $\leq$ à toutes les autres valeurs contenues dans le tableau ; la plage d'indices $\{r+1, \ldots, n\}$ est triée et les valeurs qu'elle contient sont $\geq$ à toutes les autres valeurs contenues dans le tableau (et, bien sûr, on ajoute à cela que les valeurs contenues dans le tableau sont exactement les mêmes que celles initialement contenues dans le tableau, elles ont seulement pu changer de place).

Il est clair que lorsque $l = r$, le tableau est trié, puisque l'on a, $t_{1,\ldots,l-1} \leq t_{l=r} \leq t_{r+1,\ldots,n}$; cette égalité sera donc notre condition d'arrêt

Algorithme *shaker-sort*, version optimisée, S_3

$l \longleftarrow 1$; $r \longleftarrow n$;
loop
  L'invariant $\mathcal{P}$ est vrai
  exit when $r \leq l$;
  L'invariant $\mathcal{P}$ est vrai et $r \geq l+1$
  $last \longleftarrow l$;
  for j **in** $l, \ldots, r-1$ **loop**
    if $T_{j+1} < T_j$ **then** $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j+1}$; $last \longleftarrow j$; **end if** ;
  end loop ;
  $r \longleftarrow last$;
  L'invariant $\mathcal{P}$ est vrai
  exit when $r \leq l$;
  L'invariant $\mathcal{P}$ est vrai et $r \geq l+1$
  $first \longleftarrow r$;
  for j **in reverse** $l+1, \ldots, r$ **loop**
    if $T_j < T_{j-1}$ **then** $T_j \overset{\text{swap}}{\longleftrightarrow} T_{j-1}$; $first \longleftarrow j$; **end if** ;
  end loop ;
  $l \longleftarrow first$;
end loop ;

Pour compléter la preuve donnée précédemment, il faut remarquer que lors d'une passe ascendante, l'indice r diminue strictement mais reste $\geq l$ et, lors d'une passe descendante, l'indice l augmente strictement, mais reste $\leq r$; la plage d'indices examinés (entre l et r) a donc une amplitude qui diminue strictement ce qui assure que $r - l$ diminue strictement, et atteint donc une valeur ≤ 0, mais les remarques sur les valeurs transformées de r et l indique que l'on atteint 0 exactement, par conséquent, le test $r \leq l$ est équivalent opérationnellement à $l = r$ (l'inégalité n'étant qu'une sécurité en cas de petite erreur de programmation).

Le nombre d'échanges effectués par l'algorithme est majoré par $n(n-1)/2$, et cette borne est atteinte pour un tableau rangé en ordre décroissant. On peut noter (et c'est vrai pour toutes les variation de *bubble-sort* que le nombre d'échanges effectué par les algorithmes bubble-sort ou shaker-sort, pour un même tableau, sont en nombre identique, c'est la méthode qui veut ça, ce n'est donc pas la peine de faire une étude spécifique à chaque variante de bubble-sort pour déterminer le nombre moyen d'échanges, les seuls chiffres qui changent sont les nombres de comparaisons effectuées).

Exercice 2. Complexité du tri à bulles

Le but de cet exercice est d'établir, par trois méthodes de calcul différentes — le nombre moyen d'échanges effectués par l'algorithme du tri à bulles décrit dans l'exercice 1. On se base pour ce calcul de complexité sur l'algorithme opérant par parcours du tableau dans un seul sens (soit des indices faibles vers les indices forts, soit l'inverse).

a. Comme très souvent lorsqu'il s'agit de tri de tableau, on prend comme modèle du dérangement du tableau une permutation de $\{1, \ldots, n\}$ (ce qui n'est évidemment pas un modèle totalement réaliste, mais, là au moins, on peut faire quelques calculs). En considérant, précisément, que le tableau contient des nombres entiers tous distincts compris entre 1 et n, et en identifiant le tableau avec une permutation σ de $[1, n]$, comment peut-on décrire le nombre d'échanges d'éléments de tableau en termes de permutation ? Quel est le nombre maximum d'inversions d'une permutation de $[1, n]$?

Dans la suite de l'exercice, on considère que toutes les permutations — représentant les dérangements possibles du tableau — sont équiprobables.

b. Si σ est la permutation définie par le tableau T (i.e. $\sigma(i) = T_i$), on désigne par $\tilde{\sigma}$ la permutation définie par le tableau miroir de T. Quel rapport existe entre le nombre d'inversions de σ et celui de $\tilde{\sigma}$? Soient E_1 et E_2 les ensembles de permutations suivants :

$$E_1 = \{\sigma \mid \sigma(1) > \sigma(n)\} \qquad E_2 = \{\sigma \mid \sigma(1) < \sigma(n)\}$$

Quel est le rapport entre E_1 et E_2 ? En déduire le nombre moyen d'inversions dans une permutation. Conclure en ce qui concerne la complexité moyenne d'échanges du tri à bulles.

c. On note X_j la variable aléatoire, définie sur l'ensemble des permutations, dont la valeur est le nombre d'inversions de j. Quelle est l'espérance mathématique de cette variable aléatoire ? En déduire le nombre moyen d'échanges effectués par l'algorithme du tri à bulles.

d. Si l'on connaît la somme des nombres d'inversions de toutes les permutations de $[1, n-1]$, est-il possible d'en déduire la somme des nombres d'inversions de toutes les permutations de $[1, n]$? Résoudre la récurrence qui apparaît dans cette expression, et en déduire le nombre moyen d'échanges de l'algorithme.

Solution de l'exercice 2.

a. Lors du déroulement de l'algorithme, il y a échange de deux éléments consécutifs chaque fois que ces éléments ne sont pas dans le bon ordre relatif. Le nombre d'échanges est donc égal au nombre de couples (i, j) tels que $j < i$ et $\sigma(j) > \sigma(i)$. C'est donc *le* nombre d'inversions de la permutation sous-jacente au dérangement du tableau.

Le nombre maximal d'inversions d'une permutation de $[1, n]$ est $n(n-1)/2$.

b. Considérons l'ensemble $E = \{(i, j) \mid i < j\}$, alors, on a la réunion disjointe suivante :

$$E = \{(i, j) \mid i < j \wedge \sigma(i) > \sigma(j)\} \cup \{(i, j) \mid i < j \wedge \sigma(i) < \sigma(j)\}$$

Or le cardinal du premier ensemble de cette réunion est le nombre d'inversions de la permutation σ, et le cardinal du second est le nombre d'inversions de la permutation $\tilde{\sigma}$.

Par conséquent, à toute permutation σ correspond, de manière biunivoque, une autre permutation $\tilde{\sigma}$, vérifiant la propriété : le nombre d'inversions de σ et le nombre d'inversions de $\tilde{\sigma}$ sont complémentaires à $n(n-1)/2$.

Il est par conséquent clair que les deux ensembles E_1 et E_2 sont liés par la relation : $E_2 = \{\tilde{\sigma} \mid \sigma \in E_1\}$. Les ensembles E_1 et E_2 ont donc même cardinal. De surcroît, ils sont complémentaires l'un de l'autre.

Si la probabilité d'apparition d'une permutation est $1/n!$ (cas équiprobable), on peut alors écrire — en désignant par ι_σ le nombre d'inversions d'une permutation σ :

$$\begin{aligned}\sum_{\sigma\in\Sigma_n} \iota_\sigma \times \mathrm{Prob}(\sigma) &= \sum_{\sigma\in E_1} \frac{\iota_\sigma}{n!} + \sum_{\sigma\in E_2} \frac{\iota_\sigma}{n!} = \sum_{\sigma\in E_1} \frac{\iota_\sigma}{n!} + \sum_{\sigma\in E_1} \frac{\iota_{\tilde{\sigma}}}{n!} \\ &= \sum_{\sigma\in E_1} \frac{\iota_\sigma + \iota_{\tilde{\sigma}}}{n!} = \sum_{\sigma\in E_1} \frac{1}{n!}\frac{n(n-1)}{2} = \frac{n(n-1)}{4}\end{aligned}$$

c. $X_j(\sigma)$ est donc le nombre d'inversions de j dans σ. Son espérance mathématique est donc

$$E(X_j) = \sum_{k=0}^{n-j} k\,\mathrm{Prob}(X_j = k) = \sum_{k=0}^{n-j} k\frac{\#\{\sigma \mid X_j(\sigma) = k\}}{n!}$$

Il ne reste donc plus qu'à compter les permutations pour lesquelles le nombre d'inversions de j est k. Il faut donc dénombrer les tableaux σ pour lesquels $i < j$ et $\sigma(i) > \sigma(j)$. Il y en a clairement (petit dessin) $n!/(n-j+1)$. Par conséquent,

$$E(X_j) = \sum_{k=0}^{j-1} \frac{k}{n-j+1} = \frac{(n-j)(n-j+1)}{2(n-j+1)} = \frac{n-j}{2}$$

Le nombre d'inversions d'une permutation est une variable aléatoire égale à $\sum X_j$, par conséquent, le nombre moyen d'inversions d'une permutation est égal à $\sum E(X_j) = n(n-1)/4$.

d. Soit I_n la somme des nombres d'inversions de toutes les permutations de $[1, n]$. On partitionne l'ensemble des permutations de $[1, n]$ en fonction de la position de n (i.e. l'image réciproque de n). Si $\sigma^{-1}(n) = i$ alors le nombre d'inversions de σ est égal à $n - i$ (nombre d'inversions de n) plus le nombre d'inversions de la permutation de $[1, n-1]$ sous-jacente. Sachant qu'il y a $(n-1)!$ permutations pour lesquelles l'image réciproque de n est i :

$$I_n = \big((n-1)(n-1)! + I_{n-1}\big) + \big((n-2)(n-1)! + I_{n-1}\big) + \cdots + \big(1.(n-1)! + I_{n-1}\big) + \big(0.(n-1)! + I_{n-1}\big)$$

La relation de récurrence que l'on obtient finalement est donc :

$$I_n = \frac{n-1}{2}n! + nI_{n-1},$$

récurrence qui se résout en $I_n = n(n-1)n!/4$, et nous fournit encore le résultat (on vient de calculer le nombre moyen d'inversions d'une permutation quelconque).

Annexe A : Shaker-sort, programmation et tests en Python

Voici quelques éléments de la programmation en Python de *shaker-sort*, dans la version finale (version 3), avec ce qu'il faut pour tester le programme et effectuer quelques mesures ; attention : ce que j'utilise ici ne sont pas de vrais tableaux, donc on ne peut pas mesurer n'importe quoi avec ces « listes » de Python en espérant que ce soit représentatif de ce qui se passerait avec de vrais tableaux ; ici, je pense qu'il n'y a pas de problème. Ce préliminaire étant fait, dans toute la suite, j'appellerai **tableau** la structure construite en Python avec la syntaxe [· · · · · ·].

1 Les fonctions annexes pour les tests

Parmi les outils utiles au test des tris, on trouve des fonctions de génération aléatoire de tableaux (pour tester sur de très grands tableaux) et la fonction de vérification qu'un tableau est trié (sinon, comment tester et vérifier, en vraie grandeur, des algorithmes de tri ?).

A.1.a ⟨ **Fichier "bubble_tools.py"** ⟩ ≡

```
import random
⟪ Les fonctions de construction aléatoire de tableaux ⟫ A.1.b
⟪ La fonction de vérification qu'un tableau est trié ⟫ A.1.d
```

La première fonction d'initialisation aléatoire d'un tableau possède deux arguments et renvoie le tableau construit et initialisé. Les deux arguments de cette fonction sont les suivants tout d'abord, la taille de la structure qui va être créée, et ensuite la borne maximum (le maximum étant exclu) des entiers naturels qui seront tirés au hasard pour l'initialisation.

A.1.b ⟨ Les fonctions de construction aléatoire de tableaux ⟩ ≡

```
def random_list2(n, N):
    return [random.randint(0, N-1) for i in range(n)]
```

• fragment défini en A.1.b, A.1.c, et utilisé en A.1.a

La fonction `random.randint(a, b)` renvoie un entier tiré aléatoirement dans $\{a, a+1, \ldots, b-1, b\}$ de manière équiprobable. Il est à noter que cette façon de procéder ne permet pas d'assurer que les valeurs que l'on va trier soient uniques, au contraire, il y aura très probablement des répétitions, mais c'est la vie d'un algorithme de tri ! Cela étant, si l'on veut des valeurs uniques, on peut utiliser la fonction `random.sample(population, count)` qui fait un tirage sans remise de `count` individus dans la `population` fournie, de manière équiprobable, et construit une liste avec ce tirage (évidemment, si le nombre demandé est plus grand que la taille de la population, c'est une erreur et une exception est levée).

Je fournis également une seconde version de la fonction de génération aléatoire qui n'a qu'un seul argument : la taille de l'échantillon voulu, la taille maximum des valeurs tirées aléatoirement étant 100 fois la taille de l'échantillon, en espérant que cela ne produira pas une quantité déraisonnable de répétitions dans l'échantillon.

A.1.c ⟨ Les fonctions de construction aléatoire de tableaux ⟩ ***A.1.b*** +≡

```
def random_list1(n):
    return [random.randint(0, 100*n - 1) for i in range(n)]
```

• fragment défini en A.1.b, A.1.c, et utilisé en A.1.a

Vérifier qu'un tableau est trié est très simple, comme on l'a compris, c'est en fait l'étape de base de *bubble-sort* : faire une passe, dans un sens ou dans l'autre, dans le tableau et vérifier que les valeurs adjacentes sont correctement placées l'une par rapport à l'autre ; c'est tout !

A.1.d ⟨ La fonction de vérification qu'un tableau est trié ⟩ ≡

```
def is_sorted(t):
    for i in range(0, len(t)-1):
        if t[i+1] < t[i]:
            return False
    return True
```

• fragment utilisé en A.1.a

2 Les fonctions auxiliaires de *bubble-sort* et de *shaker-sort*

Ces fonctions sont celles qui effectuent une passe dans un sens ou dans l'autre dans une tranche du tableau, afin de transporter la valeurs maximum en haut de la tranche (ou la valeur minimum en bas de la tranche).

A.2.a ⟨ **Fichier "bubble_passes.py"** ⟩ ≡

```
⟪ Fonctions effectuant une passe ascendante dans le tableau ⟫ A.2.b
⟪ Fonctions effectuant une passe descendante dans le tableau ⟫ A.2.d
```

La première de ces fonctions est la fonction effectuant une passe ascendante dans le tableau, entre les indices l (à gauche, aux indices faibles) et r (à droite, aux indices forts). Rien de particulier à dire, si ce n'est qu'il faut éviter de manipuler des tranches de tableaux, du genre `t[range(a,b)]` car cela consiste à une création de tableau pour la tranche considérée, avec recopie dans un sens et dans l'autre (si je ne me trompe pas) et donc peut conduire à des (mauvaises) surprises lors de tests de complexité ; par conséquent, les indices l et r sont fournis comme arguments à la fonction, la fonction fournit l'indice où a eu lieu le dernier échange opéré (selon les préconisations de l'étude théorique) :

A.2.b ⟨ Fonctions effectuant une passe ascendante dans le tableau ⟩ ≡

```
def bubble_ascending(t, l, r):
    last = l
    for i in range(l, r, +1):
        if t[i+1] < t[i]:
            last = i ; t[i+1], t[i] = t[i], t[i+1]
    return last
```

• fragment défini en A.2.b, A.2.c, et utilisé en A.2.a

Une seconde version de cette fonction compte en plus le nombre d'échanges effectués, afin de faire des mesures de complexité :

A.2.c ⟨ Fonctions effectuant une passe ascendante dans le tableau ⟩ ***A.2.b*** +≡

```
def trace_bubble_ascending(t, l, r):
    last = l ; ex = 0
    for i in range(l, r, +1):
        if t[i+1] < t[i]:
            last = i ; t[i+1], t[i] = t[i], t[i+1] ; ex = ex + 1
    return (last, ex)
```

• fragment défini en A.2.b, A.2.c, et utilisé en A.2.a

On notera la manière dont on peut transmettre le résultat de cette fonction — la paire : (dernier indice d'échange, nombre d'échanges) — en construisant un tuple (j'avoue que je n'ai pas réussi rapidement à savoir si l'on pouvait avoir des arguments passés par référence s'ils correspondent à des types scalaires). On peut noter également qu'il est inutile de passer dans le résultat de cette fonction le nombre de comparaisons effectuées, ce nombre est prédéterminé, c'est $r - l$ (puisqu'il y a $r - l + 1$ valeurs dans la plage considérée).

La construction de la fonction qui effectue une passe descendante est tout à fait semblable, c'est seulement le parcours du tableau qui est fait dans l'autre sens, grâce à l'argument `step = -1` de la fonction `range` :

A.2.d ⟨ Fonctions effectuant une passe descendante dans le tableau ⟩ ≡

```
def bubble_descending(t, l, r):
    last = r
    for i in range(r, l, -1):
        if t[i] < t[i-1]:
            last = i ; t[i-1], t[i] = t[i], t[i-1]
    return last
```

• fragment défini en A.2.d, A.2.e, et utilisé en A.2.a

On peut remarquer que (dans le cas de la passe montante)le résultat de `range(a, b, +1)` est la séquence croissante $a, a+1, \ldots, b-1$ (ou bien la séquence vide si $b - 1 < a$) et que (dans le cas de la passe descendante), le résultat de `range(b, a, -1)` est la séquence $b, b-1, \ldots, a+1$ (qui est vide si $a + 1 > b$.

Voici maintenant la version de la passe descendante avec comptage du nombre d'échanges opérés :

A.2.e ⟨ Fonctions effectuant une passe descendante dans le tableau ⟩ ***A.2.d*** +≡

```
def trace_bubble_descending(t, l, r):
    last = r ; ex = 0
    for i in range(r, l, -1):
        if t[i] < t[i-1]:
            last = i ; t[i-1], t[i] = t[i], t[i-1] ; ex = ex + 1
    return (last, ex)
```

• fragment défini en A.2.d, A.2.e, et utilisé en A.2.a

3 La fonction de tri par l'algorithme *shaker-sort*

Ceci n'est qu'une implémentation directe de l'algorithme étudié dans l'exercice précédent, en deux versions : l'une qui fait le tri, tout simplement, et la seconde qui évalue en sus les nombres d'échanges et de comparaisons. Dans ces deux fonctions, je colle à l'étude théorique en utilisant une boucle virtuellement infinie (`while True`) et des sorties de boucles aux deux endroits stratégiques (`if r <= l: break`)

A.3.a ⟨ Les algorithmes de tri *shaker-sort* ⟩ ≡

```
def shaker_sort3(t):
    n = len(t) ; l = 0 ; r = n-1
    while True:
        if r <= l:
            break
        r = bubble_ascending(t, l, r)
```

```
        if r <= l:
            break
        l = bubble_descending(t, l, r)
def trace_shaker_sort3(t):
    n = len(t) ; l = 0 ; r = n-1 ; cmp = 0 ; exch = 0
    while True:
        if r <= l:
            break
        (r, nbexch) = trace_bubble_ascending(t, l, r)
        exch = exch + nbexch ; cmp = cmp + (r-l)
        if r <= l:
            break
        (l, nbexchx) = trace_bubble_descending(t, l, r)
        exch = exch + nbexch ; cmp = cmp + (r-l)
    return (cmp, exch)
```

• fragment utilisé en A.3.b

Toutes les collections des fonctions précédentes peuvent être considérées comme définissant deux modules : `bubble_tools` et `bubble_passes` ; je définis maintenant un autre module, `shaker_sort` qui contient les deux fonctions principales de ce petit développement : les fonction de tri par l'algorithme *shaker-sort*.

A.3.b ⟨ **Fichier "shaker_sort.py"** ⟩ ≡

```
from bubble_passes import bubble_ascending, bubble_descending, \
     trace_bubble_ascending, trace_bubble_descending
⟨⟨ Les algorithmes de tri shaker-sort ⟩⟩ A.3.a
```

4 Script permettant de faire des tests en batch

Pour faire les tests, il ne reste plus qu'à combiner certaines des fonctions précédentes, dans un script que l'on pourra exécuter en batch ; ce que je veux faire est exécuter une commande comme celle qui suit (je travaille sous Unix) :

« `python3 test-shakersort.py 1000 2000 4000 8000 16000 32000 64000` »

et, en attendant qu'elle se termine, aller regarder pousser mes salades.

Voilà le script que je vais utiliser pour cela : il importe des modules précédents exclusivement les fonctions nécessaires, puis passe en revue les arguments de la ligne de commande pour déterminer les tailles des tableaux à construire et trier de tels tableaux produits aléatoirement :

A.4.a ⟨ **Fichier "test-shaker-sort.py"** ⟩ ≡

```
from bubble_tools import random_list1, is_sorted
from shaker_sort import trace_shaker_sort3
⟨⟨ La gestion des arguments de la ligne de commande et les tests ⟩⟩ A.4.b
```

Dans ce qui suit, `sys.argv` donne un tableau (une liste !) des arguments de la ligne de commande, le premier est le nom du script, ici `testshakersort.py`, et les suivants sont les différentes chaînes de caractères représentant les tailles que je mets sur la ligne de commande. Voici le code utilisé dans le script (le programme principal, en quelque sorte) :

A.4.b ⟨ La gestion des arguments de la ligne de commande et les tests ⟩ ≡

```
import sys
```

```
sizes = [int(arg) for arg in sys.argv[1:]]
print("Size / Number of comparisons / Number of exchanges")
for n in sizes:
    t = random_list1(n)
    (cmp, exch) = trace_shaker_sort3(t)
    if not is_sorted(t):
        raise RuntimeError("Array not sorted: bug in shakersort")
    print(n, cmp, exch)
```

• fragment utilisé en A.4.a

Le tableau `arguments` est le tableau des arguments de la ligne de commande, le premier argument étant le nom du fichier de script (qui ne m'intéresse pas) ; le tableau `sizes` donne la suite des tailles des tableaux que je veux tester (un test par taille, mais c'est facile à modifier et facile de calculer une moyenne si l'on veut) ; on fait ce à quoi on s'attend : construction aléatoire d'un tableau de la bonne taille, tri de ce tableau, vérification que le résultat du tri un bien un tableau trié (sinon une exception est levée), puis affichage des données numériques (taille, nombre de comparaisons, nombre d'échanges).

Pour info, le test mentionné plus haut, pour les tailles de tableau doublant de 1000 à 64 000 a demandé 5.30 minutes sur ma machine, le calcul pour 128 000 a demandé environ 18 minutes, et pour 256 000 environ 73 minutes. Les résultats obtenus pour $n = 512\,000$ sont environ 5h20 de calcul, un nombre de comparaisons de 87 285 519 859 et un nombre d'échanges de 65 422 873 170 (ce qui est cohérent avec ce qui figure dans le tableau de la page 3 effectué avec d'autres tirages et des programmes dans un autre langage).