Table des matières

Timsort

Dans ce problème, nous implantons une version simplifiée du tri utilisé par Python (via la procédure `.sort()` ou la fonction `sorted`). 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.
Ce tri a été conçu pour Python en 2012, il a depuis été repris par Java, Android, Octave, V8, Swift et Rust.

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.
Les partie marquées d'un point d'exclamation sont les plus importantes. Les autres apportent des améliorations, dont l'absence n'empêchera pas le fonctionnement de votre programme.

1 Sur listes Caml

1.1 ! Première amélioration : sous-listes déjà triées

Nous appelerons « préfixe » d'une liste `l` tout « début » de `l`. Précisément, si $n = |l|$ et $x_0, ..., x_{n-1}$ sons les éléments de l, alors pour tout $i \in [\![0, n[\![$, $[l_0; ...; l_i]$ est un préfixe de l.

Commencer par récupérer une fonction `fusion` comme utilisée dans le tri fusion. Récupérer également les fonctions élémentaires pour implémenter les files persistantes, ainsi que les tas-min. Dans la suite, nous supposerons disponibles un type `'a file` de files persistantes, et un type `'a arbre` qui pourra être utilisé comme file de priorité.

La première optimisation que nous allons implémenter consiste à découper la liste initiale non pas en singletons et listes vides, mais en sous-listes triées de la liste de départ. Ainsi, si celle-ci contient plusieurs fragments déjà triés, nous nous épargnerons des appels récursifs du tri sur ces sous-listes.

1. Écrire une fonction `prochain_morceau` de type `'a list -> 'a list * 'a list` qui prend une liste `l` et qui renvoie le couple (plus grand préfixe croissant de `l`, suite de `l`).
   Indication : Pour obtenir le morceau directement à l'endroit, il vaut mieux ne pas utiliser d'accumulateur ici.

   solution :

```
let rec prochain_morceau l =

  let rec aux prec = function
    (* Argument supplémentaire : prec le dernier élément pris dans le morceau actuel *)
    (* Renvoie un couple (éléments dans l'ordre et >= prec au début de la liste, suite de
       ↪ la liste)  *)
    |[]  ->[], []
    |t::q when prec <= t ->
```

```
      let morceau, suite = aux t q in t::morceau, suite
   |t::q  ->
      [], t::q
  in

  match l with
  |[]->[],[]
  |t::q-> let m, s = aux t q in t::m, s
;;
```

2. Écrire une fonction `liste_decouped` de type `'a list -> 'a list file` qui prend une liste et qui renvoie la file de ses sous-listes croissantes. Par exemple `liste_decouped [1 ;2 ;0 ;-1 ;3 ;4]` renverra la file contenant `[1;2]`, `[0]`, et `[-1;3;4]`.
solution :

```
''
```

3. Écrire une fonction prenant une file de listes triées et fusionnant deux à deux ces listes jusqu'à ce qu'il n'en reste qu'une. En déduire une fonction de tri.
solution :

```
let tri1 l=

  let rec rassemble f=
  (* f : file d'attente des sous-listes à fusionner *)
    match defiled f with
    |t, q when q=fileVide -> (* une seule liste dans la file : cas d'arrêt*) t
    |t, q  -> let s, q2 = defiled q in
        rassemble (enfiled (fusion s t) q2)
  in

  rassemble (liste_decouped l )
;;
```

4. (bonus) Version fonctionnelle : écrire l'analogue pour les files de `List.fold_left`. Il s'agit donc d'une fonction `fold_file : ('a -> 'b -> 'a)-> 'a -> 'b file -> 'a` . En déduire la fonction de tri en une ligne.
solution : Voici la première fonction :

```
(* Cette première version 'neffectue en fait pas les opération dans 'lordre voulu *)
let rec fold_file op e file =
  (* op : opération à appliquer aux éléments de la file.
      e : élément de départ (souvent le neutre) *)
  if file=fileVide then e
  else
    let x, suite = defiled file in
    op x (fold_file op e suite)
;;

(* Cette version réenfile les résultas intermédiaires afin d'effectuer les opérations
   ↪ dans l'ordre voulu ici.*)
let rec fold_file op e file =
  (* op : opération à appliquer aux éléments de la file.
      e : élément de départ (souvent le neutre) *)
  if file=fileVide then e
  else
    let x, suite = defiled file in
    if suite=fileVide then x
    else let y, ssuite = defiled suite in
      fold_file op e (enfiled (op x y) ssuite)
;;
```

Ensuite pour rassembler une file de listes, on utilise comme opérateur la fonction `fusion` et comme valeur initiale la liste vide.

```
''
let tri_f l=
  fold_file
    fusion
    []
    (liste_decouped l)
;;
```

5. Justifier l'idonéité de la structure de file d'attente ici. On pourra détailler sur l'exemple de `[1 ;2 ; -1 ;0 ;-1 ;4 ;2 ;3]` les différentes fusions effectuées.
   solution : Si tous les morceaux croissants renvoyés par `decoupe_liste` sont de la même taille, la structure de file d'attente permet que les fusions se fassent entre morceaux grosso-modo de la même taille. On fusionnera d'abord deux à deux les listes renvoyées par `decoupe_liste`, puis les listes issues d'une fusion de deux de ces listes initiales, puis les listes obtenues après fusion de quatre listes initiales, etc.

   Ceci minimise la complexité de la fusion de tous les morceaux.
6. Au contraire, imaginer un exemple de liste qui conduirait à des fusions très déséquilibrés. Ce problème sera traité plus loin.
   solution : Soit $n \in \mathbb{N}$. Imaginons que les morceaux renvoyés par `decoupe_liste` soient au nombre de trois, et de taille respective $n, 1, 1$. la première fusion coutera environ n comparaisons et la seconde aussi, donc au total environ $2n$ comparaisons.

   Si on avait d'abord fusionné les deux singletons, la première fusion aurait coûté une comparaison et la seconde environ n, soit un total d'environ n comparaisons.

   Exemple précis : `[0 ;1 ;2 ;3 ;4 ;5 ;6 ;7 ;8 ;9 ;8 ;7]` (ici $n = 10$).

1.2 Deuxième amélioration : sous-listes décroissantes

Modifier la fonction `prochain_morceau` pour qu'elle identifie le prochain morceau croissant ou décroissant. Dans le cas d'un morceau décroissant, il sera retourné (en temps linéaire) avant d'être renvoyé.

Indication : On propose une fonction auxiliaire prenant comme arguments supplémentaires :

- `prec` : dernier élément pris
- `croissant` booléen indiquant si on cherche une sous-liste croissante (si faux c'est qu'on cherche une sous-liste décroissante).

Ou alors carrément écrire deux fonctions séparées : l'une pour récupérer un préfixe croissant, l'autre pour un préfixe décroissant. Si on souhaite optimiser au maximum : la fonction pour récupérer un préfixe décroissant devrait utiliser un accumulateur, de sorte que le résultat sera immédiatement retourné.

solution :

```
''
let rec aLenvers l=
  let rec aux accu = function
    |[] -> accu
    |t::q -> aux (t::accu) q
  in
  aux [] l;;

aLenvers [1;2;3];;


let prochain_morceau l =
  (* Renvoie le couple (préfixe monotone de l retourné s'il était décroissant, suite de l) *)

  let rec aux prec croissant = function
  (* arg supplémentaire : prec élément précédent
                          croissant : booléen qui indique si on extrait une sous-suite
                              ↪ croissante*)
    |[] -> [], []
    |t::q when (t>=prec && croissant) || (t<= prec && not croissant) -> (* On continue *)
       let morceau, suite = aux t croissant q in
       t::morceau, suite
    |t::q  -> (* fini *)
       [], t::q
```

```
  in

  match l with
  |[] -> [], []
  |[t] -> [t], []
  |s::t::q when s<= t -> (* On cherche un morceau croissant *)
     let morceau, suite = aux t true q in (s::t::morceau, suite)

  |s::t::q -> (* On cherche un morceau décroissant. Le retourner avant de le mettre dans la
      ↪ file !  *)
     let morceau, suite = aux t false q in (aLenvers (s::t::morceau), suite)
;;
```

Ensuite on garde `decoupe_liste` et `tri` exactement comme avant.

1.3 Troisième amélioration : insertion

1. Le tri par insertion est meilleur que le tri fusion pour des petites listes. Fixer une variable globale `taille_min` (le Timsort choisit une valeur entre 32 et 64, mais pour vos tests vous pouvez prendre plus petit), et lors de la découpe de la liste initiale, faire en sorte que chaque sous-liste soit au moins de longueur `taille_min`, en rajoutant au moyen de la fonction `insertion` des éléments si besoin.

   Indication :

   On peut programmer une fonction `insertion` prenant un argument booléen supplémentaire `croissant` qui indique si on veut insérer dans une liste croissante ou décroissante. Ou plus conceptuel : prendre en argument directement la relation d'ordre à utiliser.

   Cette fois, utiliser un accumulateur dans la fonction auxiliaire, sur lequel on pourra utiliser `insertion`. Du coup, c'est dans le cas où on cherche une suite croissante qu'on va devoir retourner le résultat.

   solution :

```
;;
let taille_min = 6;;


let rec insertion x croissant = function
  (* croissant est un booléen qui indique si on insère dans une suite croissante. Dans le
      ↪ cas contraire, on insère dans une suite décroissante.*)
  |[] -> [x]
  |t::q when (x<= t && croissant) || (x>=t && not croissant) -> x::t::q
  |t::q -> t:: (insertion x croissant q)
;;




let prochain_morceau l =
  (* Renvoie le couple (préfixe monotone de l retourné s'il était décroissant, suite de
      ↪ l) *)

  let rec aux prec croissant taille accu = function
    (* arg supplémentaire : prec : élément précédent
                            croissant : booléen qui indique si on extrait une sous-suite
                                ↪ croissante
                            taille : nb d'éléments déjà pris
     Le morceau renvoyé est dans l'ordre not croissant *)
    |[] -> (accu  , [])

    |t::q when (t>=prec && croissant) || (t<= prec && not croissant) -> (* On continue *)
        aux t croissant (taille +1) (t::accu) q

    |t::q  -> (* fin du morceau monotone *)
       if taille < taille_min then
         (* On rajoute un élément par insertion *)
```

```
   (* Attention : il faut garder prec dans l'appel récursif, car t va être mis au milieu
      ↪ de la liste *)
   aux prec croissant (taille+1) (insertion t (not croissant) accu) q
       else
   (accu, t::q)
  in


  match l with
  |[] -> [], []
  |[t] -> [t], []
  |s::t::q when s<= t -> (* On cherche un morceau croissant *)
     let morceau, suite = aux t true 2  [ t; s] q in (aLenvers morceau, suite)

  |s::t::q -> (* On cherche un morceau décroissant *)
     let morceau, suite = aux t false 2 [s;t] q in ( morceau, suite)
;;

prochain_morceau [0;1;3;2;1;4;6;9;0;1];;
```

2. *(facultatif :)* Choisir `taille_min` en fonction de la longueur de la liste initiale pour faire en sorte que dans le cas où tous les morceaux découpés seraient de longueur `taille_min`, le nombre de morceaux soit une puissance de 2 ou juste inférieur à une puissance de 2. Cela permettra d'optimiser le nombre de fusions nécessaires par la suite. [1] Timsort prend les 6 bits les plus significatifs de la taille du tableau, plus 1 si au moins un des bits restant est non nul. Ainsi, si la taille du tableau est $\overline{a_{62}a_{61}...a_1a_0}^{(2)}$, notons i le plus grand indice tel que $a_i \neq 0$, alors on prend `taille_min` $= \overline{a_ia_{i-1}...a_{i-5}}^{(2)} + 1$, sans le -1 si $\overline{a_{i-6}...a_0}^{(2)} = 0$.
solution :

```
''
let calcule_taille_min n =
  (* Entrée : la longueur de la liste à trier
     Sortie : la taille optimale des morceaux. *)


  let rec bit_de_plus_grand_poids k =
    (* indice du bit de plus haut poids de k. (Bit des unités d'indice 0) *)
    if k=0 then -1
    else 1+ bit_de_plus_grand_poids (k/2)
  in

  let i = bit_de_plus_grand_poids n
  in n lsr (i-6)
     +
       (if n mod (1 lsl (i-7)) = 0 then 0 else 1)
;;
```

1.4 ! Tas de listes

1. Modifier encore une fois la fonction `decoupe_liste` pour que les sous-listes soient mises dans un tas-min. Afin qu'elles soient triées selon leur longueur, y mettre des couples de la forme *(longueur de la sous-liste, la sous-liste)*.
solution :

```
''
(* On a besoin de la taille des morceaux, modifions très légèrement prochain_morceau pour
   ↪ qu'il la renvoie *)
let prochain_morceau4 l =
  (* Renvoie le triplet (préfixe monotone de l retourné s'il était décroissant, suite de
     ↪ l, taille du morceau) *)

  let rec aux prec croissant taille accu = function
```

1. Dans le calcul de la complexité du tri fusion classique, on trouve environ $n \log n$ comparaisons lorsque n est une puissance de 2, et $2n \log n$ lorsque n est une puissance de 2 plus 1.

```
223     (* arg supplémentaire : prec : élément précédent
224                              croissant : booléen qui indique si on extrait une sous-suite
                                     ↪ croissante
225                              taille : nb d'éléments déjà pris
226      Le morceau renvoyé est dans l'ordre not croissant *)
227     |[] -> (accu  , [], taille)
228
229     |t::q when (t>=prec && croissant) || (t<= prec && not croissant) -> (* On continue *)
230         aux t croissant (taille +1) (t::accu) q
231
232     |t::q  -> (* fin du morceau monotone *)
233        if taille < taille_min then
234          (* On rajoute un élément par insertion *)
235    (* Attention : il faut garder prec dans l'appel récursif, car t va être mis au milieu
          ↪ de la liste *)
236    aux prec croissant (taille+1) (insertion t (not croissant) accu) q
237        else
238    (accu, t::q, taille)
239   in
240
241
242   match l with
243   |[] -> [], [], 0
244   |[t] -> [t], [], 1
245   |s::t::q when s<= t -> (* On cherche un morceau croissant *)
246      let morceau, suite, taille = aux t true 2  [ t; s] q in (aLenvers morceau, suite,
           ↪ taille)
247
248   |s::t::q -> (* On cherche un morceau décroissant *)
249      let morceau, suite, taille = aux t false 2 [s;t] q in ( morceau, suite, taille)
250 ;;
251
252
253
254 (* J'entasse des couples (-taille, morceau trié), afin que les petits morceaux se
       ↪ retrouvent en haut du tas.
255    C'est juste une astuce pour pouvoir utiliser des tas-max là où un tas-min serait plus
          ↪ pertinent. *)
256
257 let rec decoupe_liste2 = function
258     |[] -> Vide
259     |l -> let morceau, suite, taille = prochain_morceau4 l in
260        entasse (-taille, morceau) (decoupe_liste2 suite)
261 ;;
```

2. En déduire une nouvelle version de votre fonction de tri ; cette fois vous prendrez soin que chaque fusion concerne toujours les deux plus petites sous-listes disponibles.
solution :

```
1   ''
265 let tri4 l=
266
267   let rec rassemble f=
268   (* f : file de priorité des sous-listes à fusionner *)
269     match extraitMax f with
270     |(taille, l), Vide -> (* une seule liste dans la file : cas d'arrêt*) l
271     |(taille1, l1), q  -> let (taille2, l2), q2 = extraitMax q in
272         rassemble (entasse (taille1+taille2, fusion l1 l2) q2)
273   in
274
275   rassemble (decoupe_liste2 l )
276 ;;
```

3. Quel inconvénient voyez-vous à cette nouvelle approche ?
   solution : Les opérations élémentaires sur les tas sont en $O(\log n)$ alors que celles sur les files sont en $O(1)$.

2 Sur tableaux Python

La version proposée dans ce paragraphe est plus proche du véritable timsort. Comme dans le tri rapide vu en cours, nous allons utiliser des fonctions auxiliaires chargée de trier une portion donnée du tableau de départ.

2.1 Fusion

1. ! Écrire une fonction `fusionEntre` prenant en entrée un tableau `t` et trois indices `deb`, `m` et `fin` tels que `t[deb:m]` et `t[m:fin]` soient triés et qui fait en sorte que pour que `t[deb:fin]` devienne trié.
   On propose tout simplement de créer un nouveau tableau `res` dans lequel mettre la fusion de `t[deb:m]` et de `t[m:fin]` tout comme vu en cours ; ce tableau sera ensuite recopié dans `t[deb:fin]`.
   solution :

```
def fusionEntre(t, deb, m, fin):
    """ Fusionne en place t[deb:m] et t[m:fin], qui doivent être triés au préalable."""
    temp=[]# Contiendra la fusion de t[deb:m] et t[m:fin]
    i, j = deb, m

    while i < m and j < fin :
        # Imvariant de boucle : temp contient la fusion de t[deb:i] et t[m:j]
        if t[i] < t[j]:
            temp.append(t[i])
            i+=1
        else:
            temp.append(t[j])
            j+=1

    temp.extend(t[i:m])
    # Les éléments de t[j:fin] sont déjà dans leur place finale : inutile de les mettre dans temp pour les remettre ensuite dans t

    # On recopie temp dans t:
    for k in range(len(temp)):
        t[deb+k]=temp[k]
```

2. **Amélioration 4 :** Écrire une fonction `fusionOpti` prenant les mêmes entrées et produisant le même effet que `fusionEntre`, mais `FusionOpti` commencera par chercher dichotomie l'indice `a` dans $[\![deb, m]\!]$ où devra aboutir `t[m]`, ainsi que l'indice `b` dans $[\![m-1, fin[\![$ où devra aboutir `t[m-1]`. Ainsi les deux plages `t[deb:a]` et `t[b:fin]` n'ont pas à être manipulées : il suffira alors d'appeler `fusionEntre(t,a,m,b+1)` pour conclure.
   Indication : Juste pour insister : on garde donc la procédure `fusionEntre` : la procédure `fusionOpti` va juste appeler celle-ci sur une plage plus petite.

   solution :

```
# ATTENTION : fin est inclu, c'est perturbant en Python !
def cherchePlaceGauche(t, deb, fin, x):
    """ Renvoie un indice i où insérer x dans t[deb:fin] afin de conserver le caractère trié de cette tranche de tableau.
    Précisément, on aura : t[i-1] <= x < t[i]
                        et    deb <= i <= fin
    Rema : dans le cas où plusieurs places sont possibles, je renvoie la plus à droite, vu la place de l'inégalité large ci-dessus.
    """
    if deb==fin:
        return deb
    else:
        m=(deb+fin)//2 # Au pire, m==deb
        if x<t[m]:
            return cherchePlaceGauche(t, deb, m, x)
```

```
        else:#x >= t[m]
            return cherchePlaceGauche(t, m+1, fin, x)

# version impérative. Ecrire l'invariant de boucle!
def cherchePlaceGauche(t, deb, fin, x):
    """ Renvoie i tel que :  t[i-1] <= x < t[i]
                        et      deb <= i <= fin
    """
    d, f = deb, fin
    while d < f:
        # Invariant : t[d-1] <= x < t[f]
        m=(d+f)//2
        if x<t[m]:
            f=m
        else: # t[m] <= x
            d=m+1
    # sortie de boucle : d==f, et vu l'invariant ce nombre est le bon résultat.
    return d



def cherchePlaceDroite(t, deb, fin, x):
    """
    Renvoie €i[deb,fin] tel que t[i] < x <= t[i+1]
    """
    if deb==fin:
        return deb
    else:
        m=(deb+fin+1)//2 # Au pire, m==fin
        if x<=t[m]:
            return cherchePlaceDroite(t, deb, m-1, x)
        else:#x > t[m]
            return cherchePlaceDroite(t, m, fin, x)


def fusionEntreOpti(t, deb, m, fin, debug=False):
    assert deb <= m <= fin
    a= cherchePlaceGauche(t, deb, m, t[m])
    b= cherchePlaceDroite(t, m-1, fin-1, t[m-1])
    if debug:
        print(t[m], "sera mis case ", a)
        print("et", t[m-1], "sera mis case ", b)
    fusionEntre(t, a
                , m
                , b+1 # Attention ici
                )
```

3. **Amélioration 5 :** Dans `fusionEntre(t, deb, m, fin)`, on peut économiser de la mémoire en évitant de faire la fusion dans un tableau séparé. Il suffit de sauvegarder la plus petite des deux plages à fusionner (`t[deb:m]` ou `t[m:fin]`). On pourra alors mettre le résultat de la fusion directement dans `t[deb:fin]`.

   Indication : On peut commencer par sauvegarder systématiquement `t[deb:m]`. Si on souhaite sauvegarder `t[m:fin]` lorsque cette plage est plus petite, afin d'économiser plus de mémoire, il faudra remplir `t[deb:fin]` en partant de la fin, donc procéder à la fusion en partant des plus grands éléments.

   solution : Voici la version simple :

```
def fusionEntre(t, deb, m, fin):
    morceau1 = t[deb:m] # NB : avec des tableaux numpy ceci ne marcherait pas car ne
        ↪ créeait pas une copie
    n1 = m-deb # nb d'éléments dans le premier morceau
    i, j = 0, m # i indique le prochain morceau à prendre dans morceau1, j le prochain
        ↪ élément à prendre dans t
```

```
    k=deb # emplacement où écrire le prochain élément dans t
    while i < n1 and j < fin :
        if morceau1[i] < t[j]:
            t[k] = morceau1[i]
            i+=1
        else:
            t[k] = t[j]
            j+=1
        k+=1

    # On met les éléments restant dans morceau1
    while i < n1:
        t[k] = morceau1[i]
        i+=1
        k+=1
```

Remarque : On constate que la recherche du b de la question précédente devient inutile avec cette amélioration...

2.2 ! Pile de plages à fusionner

Contrairement à la version Caml où nous avions dans un premier temps découpé la liste initiale en sous-listes triées, et dans un second temps fusionné tout le monde, dans la version Python nous effectuons les découpes et les fusions au fur et à mesure.

Nous maintiendrons cette fois une pile `aFusionner` (en pratique un objet de type `List` sur lequel nous utiliserons les méthodes `pop` et `append`). Cette pile contiendra des couples d'entiers : un couple `(deb, fin)` présent dans la pile signifie que `t[deb:fin]` est trié, et qu'il faudra donc fusionner cette plage avec une plage voisine. Au passage, la longueur de cette plage est donc `fin-deb`. Nous ferons en sorte que deux plages consécutives dans la pile soient toujours voisines dans le tableau.

Nous parcourrons le tableau `t` une seule fois. Chaque plage déjà triée que nous rencontrerons sera placée dans la pile. Et après chaque insertion, nous fusionnerons les deux plages au sommet de la pile tant que l'invariant de boucle suivant n'est pas vérifié :

Pour tout `(d, f)` et `(d', f')` deux éléments consécutifs dans `aFusionner`, `(d, f)` étant au dessus de `(d', f')`, on doit avoir `f'-d' >= 2*(f-d)`.

En particulier, les plages les plus profondes dans la pile sont les plus grandes.

Remarque : Ceci est une simplification. Le vrai invariant de boucle du Timsort est pour tout (d, f), (d', f'), (d'', f'') éléments consécutifs de la pile, $\begin{cases} f-d > f'-d' \ + \ f''-d'' \\ f'-d' > f''-d'' \end{cases}$.

N.B. Pour implémenter des piles mutables en Python, il suffit d'utiliser le type `List`, et ses méthodes `pop` et `append`.

1. Prendre un exemple de tableau et montrer l'évolution de la pile `aFusionner` au cours de l'exécution de l'algorithme.
2. Soit k la taille de la pile à un moment donné. Notons $\big((d_{k-1}, f_{k-1}), \dots, (d_0, f_0)\big)$ le contenu de la pile à ce moment, (d_0, f_0) étant le sommet de la pile.
   (a) Si l'invariant de boucle est bien vérifié, que vaut au minimum, pour $i \in [\![0, k[\![$, $f_i - d_i$?
   solution : Par récurrence immmédiate, pour tout $i \in [\![0, n[\![$, $f_i - d_i \geqslant 2^i(f_0 - d_0)$.
   (b) En déduire le nombre minimum d'éléments de `t` qui ont été lus.
   solution : Le nombre d'éléments du tableau que ont été lu est égal à la somme des longueurs des intervalles dans la pile, c'est-à-dire $\sum_{i=0}^{k-1}(f_i - d_i)$. Par la question précédente, ceci est au moins $\sum_{i=0}^{k-1} 2^i(f_0 - d_0)$. En outre, $f_0 - d_0 \geqslant 1$ (et même supérieur à `taille_min` si cette amélioration est prise en compte), donc finalement, le nombre d'éléments lus est $\geqslant 2^k - 1$.
3. Notons n la longueur de `t`. Quelle sera la taille maximale de la pile lors de l'exécution de l'algorithme ?
   solution : À chaque instant, le nombre d'éléments lus est $\leqslant n$. Donc d'après la question précédente, en notant encore k le nombre d'éléments de la pile, on a $2^k - 1 \leqslant n$, d'où $k \leqslant \log_2(n+1)$. La taille maximale de la pile est donc logarithmique en la taille du tableau.

4. Écrire une fonction d'entête `prochainePlage(t,deb)` prenant en entrée le tableau `t` et un indice `deb` et renvoyant l'indice `fin` maximal tel que `t[deb:fin]` est trié dans l'ordre croissant.
   solution :

```
€
def prochainePlage(t,deb):
    """ Renvoie i maximal tel que t[deb:i] est trié dans l'ordre croissant."""

    res=deb+1
    n=len(t)
    while res < n and t[res-1]<=t[res]:
        res+=1
        # Ici, t[deb:res] est trié
    return res
```

5. Écrire une procédure `ecrase` qui prendra en entrée `t` et la pile `aFusionner` et qui fusionnera les deux plages de `t` décrites par les deux éléments du sommet de la pile jusqu'à ce que l'invariant de boucle soit vérifié.
   Indication : On s'autorisera à utiliser **`len`** pour savoir si la pile contient un seul élément. Il peut être plus pratique ici d'utiliser la récursivité.

   solution :

```
€
```

6. En déduire une procédure de tri.
   solution : Je commence par une fonction pour fusionner tout ce qu'il reste dans la pile. Elle sera appelée à la fin de la procédure de tri.

```
€
def fusionne_tout(t, àFusionner):
    while len(àFusionner) >1:
        (d0, f0) = àFusionner.pop()
        (d1, f1) = àFusionner.pop()
        fusionEntreOpti(t, d1, f1, f0)
        àFusionner.append( (d1, f0) )
```

Et voici alors la première version du Timsort version Python :

```
€
def timsort0(t, debug=False):
    n=len(t)
    i=0
    àFusionner=[]

    while i <n:
        if debug : print(àFusionner, t)
        fin = prochainePlage(t,i)
        àFusionner.append( (i,fin) )
        écrase(t, àFusionner)
        i=fin

    fusionne_tout(t, àFusionner)
    #Maintenant, àFusionner contient un seul élément : (0,n). Donc t[0:n], càd t, est
        ↪ trié.
```

2.3 Améliorations précédentes

Modifier la fonction `prochainePlage` pour implémenter les améliorations de la version sur les listes. Si vous détectez une plage décroissante, vous la remettrez à l'endroit avant de renvoyer l'indice `fin`. Si la plage détectée est trop petite, vous effectuerez des insertions en place pour l'agrandir.

Il sera plus lisible d'écrire des fonctions à part pour l'insertion et pour renverser une tranche de tableau.

2.4 Bonus : mode galop

Pour obtenir le vrai Timsort, il faut implémenter une dernière amélioration. Lors de la fusion de deux plages, disons $p1$ et $p2$, si un grand nombre d'éléments consécutifs sélectionnés viennent de la même plage, on suppose que le tableau à trier est ainsi fait qu'un grand nombre des éléments suivant à sélectionner viendront encore de la même plage.

On fixe une valeur `min_galop` (environ 7 initialement). Si au moins `min_galop` éléments ont été sélectionnes à la suite par exemple dans $p1$, on lance le « mode galop », qui va rechercher directement le nombre k des prochains éléments à prendre dans $p1$. Pour trouver k on procède par une recherche « exponentielle », ou recherche « galopante ». Il s'agit dans un premier temps de trouver i tel que $k \in [\![2^i, 2^{i+1}[\![$ (c'est là la phase « exponentielle »), et dans un deuxième temps de lancer une recherche dichotomique classique dans $[\![2^i, 2^{i+1}[\![$.