UPJV UFR des SCIENCES

Licence STS 3ème Année
Informatique
Aide à la Détection d'Erreurs

Corrigé de la feuille de TD n°3

Écrire une fonction qui, étant donnée une liste d'entiers, renvoie une liste constituée des mêmes éléments mais triés en ordre croissant. Sa complexité en nombre de comparaisons devra être en $\theta(N \times \log_2(N))$, où N est la longueur de la liste.
Vous devrez procéder suivant les 6 étapes vues en cours (cf. recherche dichotomique), y compris la vérification de la complexité.

ATTENTION : pour obtenir une solution raisonnable il est nécessaire de décomposer le problème en sous-problèmes. Il y aura donc plusieurs algorithmes à fournir (chacun devant suivre les 6 étapes vues en cours).

On donne ci-dessous les notations algébriques afin de pouvoir décrire les idées et leur justification, puis les fonctions primitives de gestion des listes pour l'écriture des algorithmes.

Notations algébriques :

Soient L, L' deux listes d'entiers et e_i des entiers.
On note $L = \langle e_1, e_2, \dots, e_N \rangle$ la liste constituée des N éléments $e_1, e_2, \dots, e_N$ dans cet ordre (attention deux éléments distincts peuvent avoir même valeur).
Si L est vide, on note $L = \langle\ \rangle$.
Si L est non vide (L contient N éléments avec $N > 0$), on note aussi $L = (e_1, L')$, où $L' = \langle e_2, \dots, e_N \rangle$.
Si L contient N éléments ($N > 0$) et si $i \in \{1, \dots, N\}$, on note aussi $\mathrm{Elt}(L, i)$ le $i^{ème}$ élément de L (i.e. e_i)

Primitives sur les listes :

Soit ListeEntier le type liste d'entiers.

- Fonction avec résultat Booléen **TestListeVide(ListeEntier L)** qui retourne vrai ssi L est une liste d'entiers vide.
- Fonction avec résultat ListeEntier **ListeVide()** qui retourne une liste d'entiers vide.
- Fonction avec résultat ListeEntier **Cons(entier X, ListeEntier L)** qui retourne une liste d'entiers ayant comme premier entier X suivi des entiers de L.
- Fonction avec résultat Entier **Premier(ListeEntier L)** qui retourne le premier entier de L, ATTENTION : précondition : L doit être non vide.
- Fonction avec résultat ListeEntier **Suite(ListeEntier L)** qui retourne la liste constituée des entiers de L sans son premier entier, ATTENTION : précondition : L doit être non vide.

Toutes ces primitives ont une complexité en temps constante.

N.B. Les primitives appellent « naturellement » une approche récursive pour l'écriture des algorithmes. Une approche itérative compliquerait cette écriture.

I. Analyse générale du tri

1) Compréhension du problème

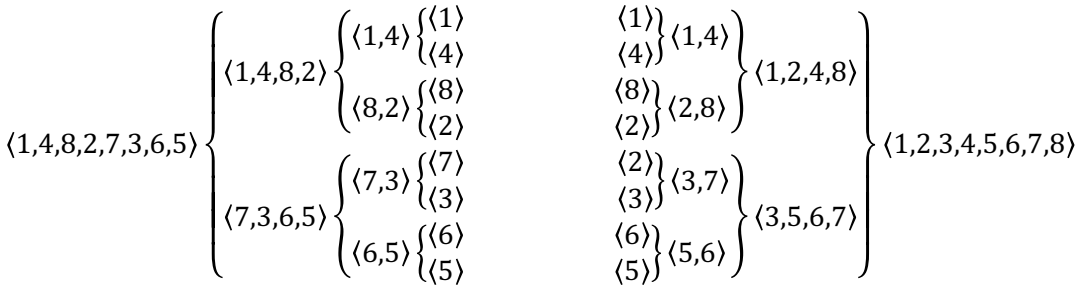
Si $L = \langle\ \rangle$ alors le résultat est la liste L (ou la liste vide), de même, si $L = \langle e \rangle$, le résultat est la liste L.
Si L contient plus d'un élément alors la construction naïve de la liste résultat L' consistant en la comparaison de chaque élément de L aux éléments de L' pour placer cet élément à la bonne place dans L' mène à une complexité en nombre de comparaisons en $\theta(N^2)$. Cette approche n'est donc pas valide, voir l'exemple ci-dessous :

> Exemple 1 : $L = \langle 1,2,3,4,5,6,7 \rangle$ alors $L' = \langle 1 \rangle$ puis $L' = \langle 1,2 \rangle$, ce qui nécessite 1 comparaison, puis $L' = \langle 1,2,3 \rangle$, avec 2 comparaisons… et enfin $L' = \langle 1,2,3,4,5,6,7 \rangle$, avec 6 comparaisons, soit au total 21 comparaisons $\left(\frac{7\times6}{2}\right)$. Si $L = \langle 1,2,3,4,5,6,7,8 \rangle$, il faut 28 comparaisons $\left(\frac{8\times7}{2}\right)$. On a bien, dans le pire des cas, $\theta(N^2)$ comparaisons, où N est le nombre d'éléments de L.

Pour faire apparaître le $\log_2(N)$, il est classique de diviser la taille du problème en 2, voir l'exemple ci-dessous :

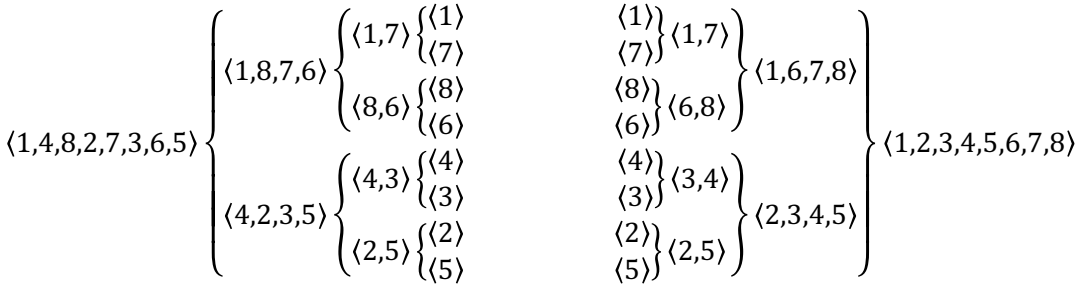
Exemple 2 : $L = \langle 1,4,8,2,7,3,6,5\rangle$, on sépare L en deux listes : $L_1 = \langle 1,4,8,2\rangle$ et $L_2 = \langle 7,3,6,5\rangle$, puis on réitère le procédé : $L_{1,1} = \langle 1,4\rangle$ et $L_{1,2} = \langle 8,2\rangle$, $L_{2,1} = \langle 7,3\rangle$ et $L_{2,2} = \langle 6,5\rangle$ et à nouveau pour obtenir des listes à 1 seul élément : $L_{1,1,1} = \langle 1\rangle$ et $L_{1,1,2} = \langle 4\rangle$, $L_{1,2,1} = \langle 8\rangle$ et $L_{1,2,2} = \langle 2\rangle$, $L_{2,1,1} = \langle 7\rangle$ et $L_{2,1,2} = \langle 3\rangle$, $L_{2,2,1} = \langle 6\rangle$ et $L_{2,2,2} = \langle 5\rangle$. Toutes ces listes à 1 éléments sont triées, on les fusionne 2 à 2 en triant les éléments, ce qui donne : $L'_{1,1} = \langle 1,4\rangle$ pour la fusion de $L_{1,1,1}$ et $L_{1,1,2}$, $L'_{1,2} = \langle 2,8\rangle$ pour la fusion de $L_{1,2,1}$ et $L_{1,2,2}$, $L'_{2,1} = \langle 3,7\rangle$ pour la fusion de $L_{2,1,1}$ et $L_{2,1,2}$, $L'_{2,2} = \langle 5,6\rangle$ pour la fusion de $L_{2,2,1}$ et $L_{2,2,2}$, soit 4×1 comparaisons. On fusionne maintenant ces listes à 2 éléments, ce qui donne : $L'_1 = \langle 1,2,4,8\rangle$ pour la fusion de $L'_{1,1}$ et $L'_{1,2}$, $L'_2 = \langle 3,5,6,7\rangle$ pour la fusion de $L'_{2,1}$ et $L'_{2,2}$, soit au plus 2×3 comparaisons. Puis la dernière fusion qui donne $L' = \langle 1,2,3,4,5,6,7,8\rangle$ pour la fusion de L_1 et L_2, soit au plus 1×7 comparaisons. Soit au total au plus 17 comparaisons, plus précisément, ce nombre de comparaisons est compris entre 4×3 et 7×3 où 3 est le nombre d'« étages » de comparaisons, 4 est le nombre de comparaisons maximal pour l'« étage » le moins coûteux (fusion des listes à 1 élément) et 7 est celui de l'« étage » le plus coûteux (pire des cas pour la fusion des deux listes restantes pour obtention de la liste résultat). On peut réécrire cet encadrement comme suit : $\frac{8}{2}\times\log_2(8)$ et $(8-1)\times\log_2(8)$, si on double le nombre d'éléments de L, on ajoute un « étage » de fusion et chaque étage comporte de 8 à 15 comparaisons, soit au total entre $\frac{16}{2}\times\log_2(16)$ et $(16-1)\times\log_2(16)$. Le nombre de comparaisons est donc bien en $\theta(N\times\log_2(N))$.

On peut représenter cet exemple sous la forme suivante où le schéma de gauche illustre la séparation de L en listes plus petites et le schéma de droite la construction de L' :



Remarque : Comme les primitives ne comportent pas de calcul de la longueur d'une liste, on peut séparer une liste en deux listes ayant pratiquement la même longueur en séparant les éléments de rang impair et les éléments de rang pair. L'exemple ci-dessus devient alors :

Exemple 3 :



2) Spécification

- Pré-conditions : $L \in \mathbb{Z}^N$ avec $N \in \mathbb{N}$
- On pose $\mathrm{N}(e, A)$ le nombre d'occurrences de la valeur de e dans la liste A.

  Post-conditions : $\big((\text{longueur}(\text{TriListe}(L)) = \text{longueur}(L))\big) \wedge$
  $(\forall e, \mathrm{N}(e, \text{TriListe}(L)) = \mathrm{N}(e, L)) \wedge$
  ($\text{TriListe}(L)$ est triée en ordre croissant)

- Remarque : la première post-condition est déductible de la seconde.

3) Idée et justification

Idée :

- Quand peut-on faire simplement ?
  Si $L = \langle\,\rangle$ ou $L = \langle e \rangle$, on renvoie la liste L.
- Et dans les autres cas ?
  Soient les fonctions ImpairListe(L) et PairListe(L) qui partitionnent L en deux listes de longueurs quasiment égales. Le résultat pour le tri de L est alors la fusion (en ordre croissant) de la liste triée à partir de Impair(L) avec la liste triée à partir de Pair(L) où la fusion (en ordre croissant) consiste à construire la liste triée en ordre croissant constituée exactement des éléments de deux listes triées en ordre croissant.

Justification :

- Si les fonctions ImpairListe() et PairListe() et la fusion renvoient le bon résultat on obtient bien la liste initiale classée en ordre croissant.
- La complexité en nombre de comparaisons dépend de la linéarité de la fusion, puisque le nombre d'appels en profondeur est de l'ordre du log (division « quasiment » par 2 de la longueur de la liste à chaque appel récursif). Donc si fusion a une complexité linéaire, on a bien de l'ordre de $\theta(N \times \log_2(N))$ comparaisons où N est la longueur de la liste à trier.
- On peut aussi remarquer que les fonctions ImpairListe() et PairListe() peuvent facilement s'implanter de telle façon que leur complexité globale soit linéaire par rapport à la longueur de le la liste paramètre. Ainsi si la comparaison de deux entiers se fait en temps constant, la complexité globale obtenue pour ce tri est aussi en $\theta(N \times \log_2(N))$.

4) Algorithme

Fonction avec résultat ListeEntier **TriListe(ListeEntier L)**

// Retourne la liste d'entiers constituée des éléments de L triés en ordre croissant
// Pré-conditions : L $\in \mathbb{Z}^N$
/* Post-conditions : (On pose N(e,A) le nombre d'ocurrences de la valeur de e dans la liste A)
((longueur(TriListe(L)) = longueur(L)) ET
(∀ e, N(e, TriListe(L)) = N(e, L)) ET
(TriListe(L) est triée en ordre croissant) */
// Complexité : θ(N x log(N)) si le temps de comparaison est constant.

```
Début
    Si (TestListeVide(L)) Alors
        Renvoyer(L)
    Sinon
        Si (TestListeVide(Suite(L))) Alors
            Renvoyer(L)
        Sinon
            Renvoyer(Fusion(TriListe(ImpairListe(L)),TriListe(PairListe(L))))
        FinSi
    FinSi
Fin
```

5) Démonstrations

Voir V

6) Complexité en temps

Voir V

II. Fusion

1) Compréhension du problème

Voir exemples vus en TD.

2) Spécification

- Pré-conditions : $A \in \mathbb{Z}^p$, $B \in \mathbb{Z}^q$ avec $p \in \mathbb{N}$ et $q \in \mathbb{N}$ et A et B sont triées en ordre croissant.
- Post-conditions : $\big((\text{longueur}(\text{Fusion}(A,B)) = \text{longueur}(A) + \text{longueur}(B))\big) \wedge$
  $(\forall e, \text{N}(e, \text{Fusion}(A,B)) = \text{N}(e,A) + \text{N}(e,B)) \wedge$
  $(\text{Fusion}(A,B)$ est triée en ordre croissant$)$
- Remarque : la première post-condition est déductible de la seconde.

3) Idée et justification

Idée :

- Quand peut-on faire simplement ?
  Si $A = \langle\,\rangle$ ou $B = \langle\,\rangle$, on renvoie la liste B si $A = \langle\,\rangle$, A sinon.
- Et dans les autres cas ?
  Comme ni A ni B ne sont vides, on peut écrire $A = (a, A')$ et $B = (b, B')$.
  On distingue deux cas :
  - $a \leq b$: soit $Res = \text{Fusion}(A', B)$, alors tous les éléments de Res sont supérieurs ou égaux à a, donc le résultat est la liste (a, Res).
  - $a > b$: soit $Res = \text{Fusion}(A, B')$, alors tous les éléments de Res sont supérieurs ou égaux à b, donc le résultat est la liste (b, Res).

Justification :

Dans les deux cas l'élément sélectionné x ($x = a$ ou $x = b$) est bien le plus petit élément des 2 listes et donc (x, Res) contient bien exactement les éléments de A et B triés en ordre croissant.

4) Algorithme

```
Fonction avec résultat ListeEntier Fusion(ListeEntier A, ListeEntier B)
/* Retourne la liste d'entiers constituée de la fusion de A et B, deux listes d'entiers triées dans l'ordre croissant */
// Pré-conditions : A ∈ ℤ^p, B ∈ ℤ^q avec p ∈ ℕ et q ∈ ℕ et A et B sont triées en ordre croissant
/* Post-conditions : On pose N(e,A) le nombre d'ocurrences de la valeur de e dans la liste A
                     (longueur(Fusion(A,B)) = longueur(A) + longueur(B)) ET
                     (∀ e (N(e, Fusion(A,B)) = N(e,A)+N(e,B)) ET
                     (Fusion(A,B) est triée en ordre croissant)*/

Début
    Si (TestListeVide(A) OU TestListeVide(B)) Alors
        Si (TestListeVide(A)) Alors
            Renvoyer(B)
        Sinon
            Renvoyer(A)
        FinSi
    Sinon
        Si (Premier(A) ≤ Premier(B)) Alors
        // P1 : ∀x, ((x élément de A) → (x ≥ Premier(A))) ∧ ((x élément de B) → (x ≥ Premier(A)))
            Renvoyer(Cons(Premier(A), Fusion(Suite(A),B)))
        Sinon
        // P2 : ∀x, ((x élément de A) → (x ≥ Premier(B))) ∧ ((x élément de B) → (x ≥ Premier(B)))
            Renvoyer(Cons(Premier(B), Fusion(A, Suite(B))))
        FinSi
    FinSi

Fin
```

5) Démonstrations

a) Démonstration de l'arrêt

Arrêt des instructions élémentaires.

La fonction TestListeVide() utilisée dans Fusion a sa précondition vérifiée à chaque appel puisque le paramètre est bien un objet du type ListeEntier. Le 1er test assure que l'utilisation de Suite() et Premier() se fait sur une liste non vide, ces deux fonctions s'arrêtent donc. Comme Premier() renvoie un entier et que le second paramètre est du type ListeEntier (résultat de Fusion()), chaque appel de Cons() s'arrête.

D'autre part, à chaque appel récursif les préconditions de Fusion sont vérifiées : en effet si L est une liste non vide triée en ordre croissant alors Suite(L) est aussi une liste (éventuellement vide) triée en ordre croissant. Il reste alors à montrer que le nombre d'appels récursifs est fini.

Nombre fini d'appels récursifs.

Remarque 1 : dans les cas triviaux où $\text{longueur}(A) = 0$ ou $\text{longueur}(B) = 0$, c'est-à-dire dans le cas où l'une des deux listes est vide, il n'y a pas d'appel récursif (exécution du premier « alors »).

On pose le prédicat d'arrêt suivant :

$\forall n \in \mathbb{N}, PA(n)$: pour toutes listes A et B triées dans l'ordre croissant,
$(\text{longueur}(A) + \text{longueur}(B) = n) \rightarrow$ (le nombre d'appels récursifs de $\text{Fusion}(A, B)$ est fini)

Initialisation :
On sait que $PA(0)$ est vraie (cf. Remarque 1).

Induction :
Il faut montrer : $\forall n \in \mathbb{N}$, $PA(n) \vDash PA(n+1)$ (hérédité simple).

On rappelle la définition de $PA(n+1)$:

$PA(n+1)$: pour toutes listes A et B triées dans l'ordre croissant,
$(\text{longueur}(A) + \text{longueur}(B) = n+1) \rightarrow$ (le nombre d'appels récursifs de $\text{Fusion}(A, B)$ est fini).

Soient A et B deux listes triées telles que $\text{longueur}(A) + \text{longueur}(B) = n + 1$.

Nous avons 3 cas à étudier :

1. $\text{Longueur}(A) = 0$ ou $\text{longueur}(B) = 0$, alors d'après la remarque 1, $PA(n+1)$ est vraie.
2. $\text{Longueur}(A) > 0$, $\text{longueur}(B) > 0$ et $\text{Premier}(A) \leq \text{Premier}(B)$. Dans ce cas on appelle $\text{Fusion}(\text{Suite}(A), B)$ or $\text{longueur}(\text{Suite}(A)) + \text{longueur}(B) = n$. Par hypothèse d'hérédité $PA(n)$, $\text{Fusion}(\text{Suite}(A), B)$ a un nombre fini d'appels récursifs, donc $\text{Fusion}(A, B)$ a aussi un nombre fini d'appels récursifs.
3. $\text{Longueur}(A) > 0$, $\text{longueur}(B) > 0$ et $\text{Premier}(A) > \text{Premier}(B)$. Ce cas est similaire au cas précédent.

Dans tous les cas on a bien $PA(n) \vDash PA(n+1)$.

Conclusion :
$PA(0)$ est vraie et $\forall n \in \mathbb{N}$, l'hérédité simple est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PA(n)$ est vraie.

Le nombre d'appels récursifs est donc fini et, puisque les autres instructions s'arrêtent, la fonction s'arrête.

b) Démonstration du bon résultat

i) Si $\text{longueur}(A) + \text{longueur}(B) = 0$, c'est-à-dire dans le cas où les deux listes sont vides, le résultat renvoyé est le bon : B, c'est-à-dire une liste vide.
ii) Si $\text{longueur}(A) + \text{longueur}(B) > 0$, mais dans le cas où A (respectivement B) est vide, c'est-à-dire $\text{Longueur}(A) = 0$ (resp. $\text{Longueur}(B) = 0$), le résultat renvoyé est le bon : B (resp. A).

On pose PBR la Propriété de Bon Résultat de Fusion :

$\forall n \in \mathbb{N}, PBR(n)$: pour toutes listes A et B triées dans l'ordre croissant,
$(\text{longueur}(A) + \text{longueur}(B) = n) \rightarrow (\text{Fusion}(A, B)$ renvoie la liste constituée des éléments de A et B triés dans l'ordre croissant).

Initialisation :
On sait que $PBR(0))$ est vraie : cf. i).
On sait que $PBR(1)$ est vraie, car dans ce cas une des 2 listes est vide : cf. ii).

Induction :
Il faut montrer : $\forall n \in \mathbb{N}^*, PBR(n) \vDash PBR(n+1)$ (hérédité simple).

On rappelle la définition de $PBR(n+1)$:

$PBR(n+1)$: pour toutes listes A et B triées dans l'ordre croissant,
$(\text{longueur}(A) + \text{longueur}(B) = n + 1) \rightarrow$ (Fusion(A, B) renvoie la liste constituée des éléments de A et B triés dans l'ordre croissant).

Soient A et B deux listes triées telles que $\text{longueur}(A) + \text{longueur}(B) = n + 1$.

Nous avons 3 cas à étudier :

1. $\text{Longueur}(A) = 0$ ou $\text{longueur}(B) = 0$, alors selon ii) $PBR(n+1)$ est vraie.

2. $\text{Longueur}(A) > 0$ et $\text{longueur}(B) > 0$ et $\text{Premier}(A) \leq \text{Premier}(B)$. Dans ce cas on appelle Fusion(Suite(A),B) or $\text{longueur}(\text{Suite}(A)) + \text{longueur}(B) = n$. Donc par hypothèse de récurrence, Fusion(Suite(A),B) renvoie la liste constituée exactement de tous les éléments[1] de $\text{Suite}(A)$ et B triés dans l'ordre croissant. Comme seul $\text{Premier}(A)$ est ajouté à Fusion(Suite(A),B), le résultat est de longueur $n+1$ et est constitué d'exactement tous les éléments de A et B. De plus comme A et B sont triées en ordre croissant, nous savons que
   - pour tout entier y dans $\text{Suite}(A)$, $\text{Premier}(A) \leq y$,
   - $\text{Premier}(A) \leq \text{Premier}(B)$) (hypothèse sur le test),
   - pour tout entier y dans $\text{Suite}(B)$, $\text{Premier}(A) \leq y$.

   donc pour tout entier y dans A, $\text{Premier}(A) \leq y$. On en déduit que pour tout entier y dans Fusion(Suite(A),B), $\text{Premier}(A) \leq y$. Donc Cons(Premier(A), Fusion(Suite(A),B)) donne bien la liste constituée exactement de tous les éléments de A et B triés dans l'ordre croissant. Donc $PBR(n+1)$ est vraie.

3. $\text{Longueur}(A) > 0$ et $\text{longueur}(B) > 0$ et $\text{Premier}(A) > \text{Premier}(B)$. Ce cas est similaire au cas précédent.

Dans tous les cas $PBR(n+1)$ est vraie.

Conclusion :
$PBR(0)$ et $PBR(1)$ sont vraies et $\forall n \in \mathbb{N}^*$, l'hérédité simple est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}^*, PBR(n)$ est vraie.

6) Complexité en temps

[1] Je me permets ici d'avoir recours à cette formulation moins formelle puisque la rédaction formelle de la postcondition en donne sans ambiguïté la signification précise.

Remarque : Si l'on suppose que les comparaisons se font en temps constant alors toutes les instructions et les fonctions appelées, à part Fusion(), sont en temps constant. Il suffit donc de calculer le nombre d'appels récursifs pour déterminer la complexité de Fusion(). Si les comparaisons ne se font pas en temps constant, la complexité calculée ici donne le nombre de comparaisons effectuées et il suffit alors de multiplier cette complexité par celle d'une comparaison.

On pose le prédicat de calcul du nombre d'appels récursifs suivant :

$\forall n \in \mathbb{N}^*, PNA(n)$: pour toutes listes A et B triées dans l'ordre croissant,
$(\text{longueur}(A) + \text{longueur}(B) = n) \rightarrow$ (le nombre d'appels récursifs de Fusion est au pire $= n - 1$)

Initialisation :
Si $n = 1$, On sait que $PA(1)$ est vraie car l'une des deux listes et de longueur nulle et il n'y a pas d'appel récursif (cf. remarque 1).

Induction :
Il faut montrer : $\forall n \in \mathbb{N}^*$, $PA(n) \vDash PA(n+1)$ (hérédité simple).

On rappelle la définition de $PA(n+1)$:

$PA(n+1)$: pour toutes listes A et B triées dans l'ordre croissant,
$(\text{longueur}(A) + \text{longueur}(B) = n + 1) \rightarrow$ (le nombre d'appels récursifs de Fusion est au pire $= n$).

Soient A et B deux listes triées telles que $\text{longueur}(A) + \text{longueur}(B) = n + 1$.

Nous avons 3 cas à étudier :

1. $\text{Longueur}(A) = 0$ ou $\text{longueur}(B) = 0$, alors d'après la remarque 1, il n'y a pas d'appels récursifs et $PA(n+1)$ est vraie.
2. $\text{Longueur}(A) > 0$, $\text{longueur}(B) > 0$ et $\text{Premier}(A) \leq \text{Premier}(B)$. Dans ce cas on appelle $\text{Fusion}(\text{Suite}(A), B)$ or $\text{longueur}(\text{Suite}(A)) + \text{longueur}(B) = n$. Par hypothèse d'hérédité $PA(n)$, le nombre d'appels récursifs est au pire $n - 1$, donc le nombre d'appels pour $\text{Fusion}(A, B)$ est au pire n.
3. $\text{Longueur}(A) > 0$, $\text{longueur}(B) > 0$ et $\text{Premier}(A) > \text{Premier}(B)$. Ce cas est similaire au cas précédent.

Dans tous les cas on a bien $PA(n) \vDash PA(n+1)$.

Conclusion :
$PA(0)$ est vraie et $\forall n \in \mathbb{N}$, l'hérédité simple est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PA(n)$ est vraie.

Le nombre d'appels récursifs pour deux listes dont la somme des longueurs est n est donc, au pire, $n - 1$ (ce cas est atteint lorsque les deux listes sont non vides jusqu'au dernier appel récursif où l'une des listes est de longueur 1 et l'autre est vide) et, puisque les autres instructions s'exécutent en temps constant, le temps d'exécution de la fonction est donc en $\theta(n)$. On peut remarquer que le nombre de comparaisons est égal au nombre d'appels récursifs et est donc égal lui aussi au pire à $n - 1$.

III. Impair

1) Compréhension du problème

Voir exemples vus en TD.

2) Spécification

- Pré-conditions : $L \in \mathbb{Z}^p$ avec $p \in \mathbb{N}$.
- Post-conditions : $\big((\text{longueur}(\text{ImpairListe}(L)) = (\text{longueur}(L) + 1) \text{ div } 2)\big) \wedge$

$$\left(\forall i, \left((1 \leq i) \wedge (i \leq \text{longueur}(\text{ImpairListe}(L)))\right) \rightarrow (\text{Elt}(\text{ImpairListe}(L), i) = \text{Elt}(L, 2i-1))\right).$$

- Remarque : la première post-condition est déductible de la seconde.

3) Idée et justification

Idée :

- Quand peut-on faire simplement ?
  - $L = \langle\ \rangle$, on renvoie la liste vide.
  - $L = \langle a \rangle$, on renvoie L.
- Et dans les autres cas ?
  On observe que :
  - L contient au moins 2 éléments, i.e. $L = (e_1, (e_2, L'))$,
  - la parité du rang est respectée entre L et L'.

  Supposons que nous sachions calculer $Res' = \text{ImpairListe}(L')$, alors le résultat que nous souhaitons construire serait la liste $Res = (e_1, Res')$ et les éléments de cette liste sont dans le même ordre que dans L (i.e. : le $i^{ème}$ élément de la liste résultat Res est le $(2i-1)^{ème}$ élément de la liste L).

4) Algorithme

Fonction avec résultat ListeEntier **ImpairListe(**ListeEntier L**)**
// Retourne la liste d'entiers constituée des éléments de rang impair de L (en conservant l'ordre)
// Pré-conditions : L ∈ $\mathbb{Z}^p$ avec p ∈ $\mathbb{N}$
/* Post-conditions : (longueur(ImpairListe(L)) = (longueur(L) + 1) div 2) ET
((∀ i (1 ≤ i ≤ longueur(ImpairListe(L))) → (Elt(ImpairListe(L), i) = Elt(L, 2i – 1)) */

```
Début
    Si (TestListeVide(L)) Alors
        Renvoyer(ListeVide())
    Sinon
        Si (TestListeVide(Suite(L))) Alors
            Renvoyer(L)
        Sinon
            Renvoyer(Cons(Premier(L), ImpairListe(Suite(Suite(L)))))
        FinSi
    FinSi
Fin
```

5) Démonstrations

a) Démonstration de l'arrêt

Arrêt des instructions élémentaires.

La fonction TestListeVide() utilisée dans ImpairListe() a sa précondition vérifiée à chaque appel puisque le paramètre est bien un objet du type ListeEntier. Les deux tests assurent que l'utilisation de Suite(Suite()) et Premier() se fait sur une liste non vide, ces deux fonctions s'arrêtent donc. Comme Premier() renvoie un entier et que le second paramètre est du type ListeEntier (résultat de ImpairListe()), chaque appel de Cons() s'arrête.
D'autre part, à chaque appel récursif les préconditions de ImpairListe sont vérifiées : en effet si L est une liste avec au moins deux éléments alors Suite(Suite(L)) est aussi une liste (éventuellement vide). Il reste alors à montrer que le nombre d'appels récursifs est fini.

Nombre fini d'appels récursifs.

Remarque 2 : dans les cas triviaux où Longueur(L) = 0 ou Longueur(L) = 1, il n'y a pas d'appel récursif (exécution du premier ou du second « alors »).

On pose le prédicat d'arrêt suivant :

$\forall n \in \mathbb{N}, PA(n)$: pour toute liste L,
$(\text{longueur}(L) = n) \rightarrow$ (le nombre d'appels récursifs de $\text{ImpairListe}(L)$ est fini)

Initialisation :
On sait que $PA(0)$ et $PA(1)$ sont vraies (cf. Remarque 2).

Induction :
Il faut montrer : $\forall n \in \mathbb{N}^*, \bigwedge_{i=0}^{n} PA(i) \vDash PA(n+1)$ (hérédité forte).

On rappelle la définition de $PA(n+1)$:
$PA(n+1)$: pour toute liste L,
$(\text{longueur}(L) = n+1) \rightarrow$ (le nombre d'appels récursifs de $\text{ImpairListe}(L)$ est fini).

Soit L une liste telle que $\text{longueur}(L) = n+1$. Comme $n+1 \geq 2$, on exécute le second « sinon » et on appelle ImpairListe( ) avec comme paramètre $\text{Suite}(\text{Suite}(L))$ qui est une liste de longueur $n-1 \geq 0$. On peut donc appliquer l'hypothèse d'hérédité et le nombre d'appels récursifs pour $\text{Suite}(\text{Suite}(L))$ est fini, il l'est donc pour L.
On a bien $\bigwedge_{i=0}^{n} PA(i) \vDash PA(n+1)$.

Conclusion :
$PA(0)$ et $PA(1)$ est vraie et $\forall n \in \mathbb{N}^*$, l'hérédité forte est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PA(n)$ est vraie.

Le nombre d'appels récursifs est donc fini et, puisque les autres instructions s'arrêtent, la fonction s'arrête.

b) Démonstration du bon résultat
i) Si $\text{longueur}(A) + \text{longueur}(B) = 0$, c'est-à-dire dans le cas où les deux listes sont vides, le résultat renvoyé est le bon : B, c'est-à-dire une liste vide.
ii) Si $\text{longueur}(A) + \text{longueur}(B) > 0$, dans le cas où A (respectivement B) est vide, c'est-à-dire $\text{Longueur}(A) = 0$ (resp. $\text{Longueur}(B) = 0$), le résultat renvoyé est le bon : B (resp. A).

On pose PBR la Propriété de Bon Résultat de ImpairListe :
$\forall n \in \mathbb{N}, PBR(n)$: pour toute liste L,
$(\text{longueur}(L) = n) \rightarrow (\text{ImpairListe}(L)$ renvoie la liste constituée des éléments de rang impair de L dans le même ordre).

Initialisation :
$PBR(0)$ est vraie, car ImpairListe($\langle\ \rangle$) retourne la liste vide qui est le bon résultat.
$PBR(1)$ est vraie, car ImpairListe($\langle e \rangle$) retourne la liste $\langle e \rangle$ qui est le bon résultat.

Induction :
Il faut montrer : $\forall n \in \mathbb{N}^*, \bigwedge_{i=0}^{n} PA(i) \vDash PA(n+1)$ (hérédité forte).

On rappelle la définition de $PA(n+1)$:
$PA(n+1)$: pour toute liste L,
$(\text{longueur}(L) = n+1) \rightarrow (\text{ImpairListe}(L)$ renvoie la liste constituée des éléments de rang impair de L dans le même ordre).

Soit L une liste telle que $\text{longueur}(L) = n+1 \geq 2$. Soit $L' = \text{Suite}(\text{Suite}(L))$. Comme $\text{longueur}(L') = n-1$ et $n-1 \geq 0$, $\text{ImpairListe}(L')$ renvoie, par hypothèse d'hérédité, une liste Res' telle que :

$$(\text{longueur}(Res') = (\text{longueur}(L') + 1) \text{ div } 2) \wedge$$
$$\Big(\forall i, \big((1 \leq i) \wedge (i \leq \text{longueur}(Res'))\big) \rightarrow (\text{Elt}(Res', i) = \text{Elt}(L', 2i-1))\Big)$$

Par construction on peut remarquer que le rang des éléments de L' est décalé de 2 par rapport à celui des éléments de L, i.e. $\forall i, \big((1 \leq i) \wedge (i \leq \text{longueur}(Res'))\big) \rightarrow \text{Elt}(L', i) = \text{Elt}(L, i+2)$. Par construction de Res ($Res = \text{ImpairListe}(L)$) et par hypothèse d'hérédité nous avons donc :

$$\text{longueur}(Res) = 1 + \text{longueur}(Res') = 1 + (\text{longueur}(L') + 1) \text{ div } 2$$

donc :

$$\text{longueur}(Res) = (\text{longueur}(L') + 2 + 1) \text{ div } 2 = (\text{longueur}(L) + 1) \text{ div } 2$$

D'autre part, $\text{Elt}(Res, 1) = \text{Elt}(L, 1)$ et pour tout $i \geq 2$, $\text{Elt}(Res, i) = \text{Elt}(Res', i-1)$, or par hypothèse d'hérédité, nous avons $\text{Elt}(Res', i-1) = \text{Elt}(L', 2i-3)$ et par décalage des éléments de L par rapport à ceux de L', nous avons $\text{Elt}(L', 2i-3) = \text{Elt}(L, 2i-1)$ et donc pour tout $i \geq 2$, $\text{Elt}(Res, i) = \text{Elt}(L, 2i-1)$.

On a donc :

$$\forall i, \big((1 \leq i) \wedge (i \leq \text{longueur}(Res))\big) \rightarrow (\text{Elt}(Res, i) = \text{Elt}(L, 2i-1))$$

Donc $\text{ImpairListe}(L)$ est bien la liste constituée des éléments de rang impair de L dans le même ordre et l'hérédité est démontrée.

Conclusion :

$PBR(0)$ et $PBR(1)$ sont vraies et $\forall n \in \mathbb{N}^*$, l'hérédité forte est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PBR(n)$ est vraie.

6) Complexité en temps

On note qu'à part les appels récursifs, toutes les instructions s'exécutent en temps constant. Pour simplifier les calculs, on pose c le coût de l'exécution pour arriver à un retour de la liste vide ou de L ($L = \langle e \rangle$) et C le coût pout arriver à un appel récursif.

On pose $\text{CIL}(n)$ la complexité de $\text{ImpairListe}(L)$ où $n = \text{longueur}(L)$.

Nous avons donc $\text{CIL}(0) = c$, $\text{CIL}(1) = c$ et pour $n \geq 2$, $\text{CIL}(n) = C + \text{CIL}(n-2)$.

- $\text{CIL}(2) = C + \text{CIL}(0) = C + c$
- $\text{CIL}(3) = C + \text{CIL}(1) = C + c$
- $\text{CIL}(4) = C + \text{CIL}(2) = 2C + c$
- $\text{CIL}(5) = C + \text{CIL}(3) = 2C + c$
- $\text{CIL}(6) = C + \text{CIL}(4) = 3C + c$
- $\text{CIL}(7) = C + \text{CIL}(5) = 3C + c$
- ...

On peut en déduire que la formule close est de la forme $\text{CIL}(n) = (n \text{ div } 2) \times C + c$ pour tout $n \in \mathbb{N}$.

Démontrons par récurrence qu'il s'agit effectivement de la bonne formule :

On pose PC la Propriété sur le Coût de ImpairListe :

$\forall n \in \mathbb{N}, PC(n)$: pour toute liste L, $(\text{longueur}(L) = n) \rightarrow (\text{CIL}(n) = (n \text{ div } 2) \times C + c)$.

Initialisation :

$PC(0)$ et $PC(1)$ sont vraies, car $\text{CIL}(0) = c$ et $\text{CIL}(1) = c$.

Induction :

Il faut montrer : $\forall n \in \mathbb{N}^*$, $\bigwedge_{i=0}^{n} PC(i) \vDash PC(n+1)$ (hérédité forte).

On rappelle la définition de $PC(n+1)$:

$PC(n+1)$: pour toute liste L, $(\text{longueur}(L) = n+1) \rightarrow \big(\text{CIL}(n+1) = \big((n+1) \text{ div } 2\big) \times C + c\big)$.

Soit L une liste d'entiers telle que $\text{longueur}(L) = n + 1 \geq 2$. On exécute donc le second « sinon » et $\text{CIL}(n+1) = C + \text{CIL}(n-1)$. Comme $n - 1 \geq 0$, on peut appliquer l'hypothèse d'hérédité, ce qui donne :

$$\text{CIL}(n+1) = C + \text{CIL}(n-1) = C + \big((n-1) \text{ div } 2\big) \times C + c = \big((n-1) \text{ div } 2 + 1\big) \times C + c$$

donc

$$\text{CIL}(n+1) = \big((n+1) \text{ div } 2\big) \times C + c.$$

et $PC(n+1)$ est vérifiée.

Conclusion :

$PC(0)$ et $PC(1)$ sont vraies et $\forall n \in \mathbb{N}^*$, l'hérédité forte est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PC(n)$ est vraie.

On a donc l'encadrement suivant : $a \times n + c - C \leq \text{CIL}(n) \leq a \times n + c$, où a, c et C sont trois constantes ($a = {}^{C}/_{2}$ non nul) donc $\text{CIL}(n) \in \theta(n)$ où n est la longueur de la liste.

IV. Pair

1) Compréhension du problème

Voir exemples vus en TD.

2) Spécification

- Pré-conditions : $L \in \mathbb{Z}^p$ avec $p \in \mathbb{N}$.
- Post-conditions : $\big((\text{longueur}(\text{PairListe}(L)) = \text{longueur}(L) \text{ div } 2)\big) \wedge$
  $\Big(\forall i, \big((1 \leq i) \wedge (i \leq \text{longueur}(\text{PairListe}(L)))\big) \rightarrow (\text{Elt}(\text{PairListe}(L), i) = \text{Elt}(L, 2i))\Big)$.
- Remarque : la première post-condition est déductible de la seconde.

3) Idée et justification

Idée :

- Quand peut-on faire simplement ?
  $L = \langle\ \rangle$, on renvoie la liste vide.
- Et dans les autres cas ?
  On observe que :
  - L contient au moins 1 élément, i.e. $L = (e_1, L')$,
  - la parité du rang est inversée entre L et L'.

  Posons $Res = \text{ImpairListe}(L')$, alors le résultat que nous souhaitons construire serait la liste Res elle-même et les éléments de cette liste sont dans le même ordre que dans L (i.e. : le $i^{ème}$ élément de la liste résultat Res est le $2i^{ème}$ élément de la liste L car, d'après III, le $i^{ème}$ élément de Res est le $(2i-1)^{ème}$ élément de la liste L'.)

4) Algorithme

Fonction avec résultat ListeEntier **PairListe(**ListeEntier L**)**
// Retourne la liste d'entiers constituée des éléments de rang pair de L (en conservant l'ordre)
// Pré-conditions : L ∈ $\mathbb{Z}^p$ avec p ∈ $\mathbb{N}$
/* Post-conditions : (Longueur(PairListe(L)) = (Longueur(instance)) div 2) ET
((∀ i, (1 ≤ i ≤ Longueur(PairListe(L))) → (Elt(PairListe(L), i) = Elt(L, 2i) */

```
Début
    Si (TestListeVide(L)) Alors
        Renvoyer(ListeVide())
    Sinon
        Renvoyer(ImpairListe(Suite(L)))
    Finsi
Fin
```

5) Démonstrations

a) Démonstration de l'arrêt

Les instructions et les fonctions utilisées dans PairListe ont leurs préconditions vérifiées à chaque appel (le test assure que l'utilisation de Suite se fait sur une liste contenant au moins 1 élément) , toutes les instructions s'arrêtent donc toutes. Donc PairListe s'arrête.

b) Démonstration du bon résultat

Dans le cas trivial où $\text{longueur}(L) = 0$, le résultat renvoyé est le bon : c'est la liste vide. Si $\text{longueur}(L) > 1$, alors $\text{PairListe}(L)$ renvoie la liste des éléments de rangs impairs de $\text{Suite}(L)$ dans le même ordre, ce qui est bien la liste des éléments de rang pair de L dans le même ordre.
La démonstration plus formelle étant assez facile, elle est laissée en exercice…

6) Complexité en temps

La complexité de PairListe est identique à celle de ImpairListe à une constante additive près (l'appel de ImpairListe), donc le coût de PairListe est en $\theta(n)$ où n est la longueur de la liste. Si l'on note respectivement $AI(n)$ et $AP(n)$ le nombre d'appels à ImpairListe de ImpairListe et de PairListe, on a : $AP(n) = 1 + AI(n-1)$.

Ce qui donne :

- n pair : $AP(n) = AI(n)$
- n impair : $AP(n) = 1 + AI(n)$

V. Compléments sur le tri

1) Compréhension du problème

Voir I 1.

2) Spécification

Voir I 2.

3) Idée et justification

Voir I 3.

4) Algorithme

Voir I 4.

5) Démonstrations

a) Démonstration de l'arrêt

Arrêt des instructions élémentaires.

La fonction TestListeVide() a sa précondition vérifiée à chaque appel puisque le paramètre est du type ListeEntier. Le premier test assure que l'utilisation de Suite() se fait sur une liste non vide dans le « sinon », cette fonction s'arrête donc. Les préconditions de ImpairListe(), et PairListe() sont aussi vérifiées (le paramètre est bien du type ListeEntier) et les préconditions de Fusion() le sont aussi (ses deux paramètres sont bien du type ListeEntier : résultat de TriListe()).

D'autre part, à chaque appel récursif les préconditions de TriListe() sont vérifiées. Il reste alors à montrer que le nombre d'appels récursifs est fini.

Nombre fini d'appels récursifs.

Remarque 4 : dans les cas triviaux où $\text{longueur}(L) = 0$ ou $\text{longueur}(L) = 1$, il n'y a pas d'appel récursif (exécution du premier ou du second « alors »).

On pose le prédicat d'arrêt suivant :

$\forall n \in \mathbb{N}, PA(n)$: pour toute liste L,
$(\text{longueur}(L) = n) \rightarrow$ (le nombre d'appels récursifs de $\text{TriListe}(L)$ est fini)

Initialisation :
On sait que $PA(0)$ et $PA(1)$ sont vraies (cf. Remarque 4).

Induction :
Il faut montrer : $\forall n \in \mathbb{N}^*, \bigwedge_{i=0}^{n} PA(i) \vDash PA(n+1)$ (hérédité forte).

On rappelle la définition de $PA(n+1)$:

$PA(n+1)$: pour toute liste L,
$(\text{longueur}(L) = n+1) \rightarrow$ (le nombre d'appels récursifs de $\text{TriListe}(L)$ est fini).

Soit L une liste telle que $\text{longueur}(L) = n+1$. Comme $n+1 \geq 2$, on exécute le second « sinon » et on appelle $\text{TriListe}(L)$ avec comme paramètre $\text{ImpairListe}(L)$ et $\text{PairListe}(L)$ qui sont des listes de longueur respective n_i et n_p avec $\text{n}_i + \text{n}_p = n+1$, $1 \leq \text{n}_i < n$ et $1 \leq \text{n}_p < n$. On peut donc appliquer l'hypothèse d'hérédité et le nombre d'appels récursifs pour $\text{ImpairListe}(L)$ et $\text{PairListe}(L)$ est fini, il l'est donc pour L.

On a bien $\bigwedge_{i=0}^{n} PA(i) \vDash PA(n+1)$.

Conclusion :
$PA(0)$ et $PA(1)$ est vraie et $\forall n \in \mathbb{N}^*$, l'hérédité forte est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PA(n)$ est vraie.

Le nombre d'appels récursifs est donc fini et, puisque les autres instructions s'arrêtent, la fonction s'arrête.

b) Démonstration du bon résultat
On pose PBR la Propriété de Bon Résultat de TriListe :
$\forall n \in \mathbb{N}, PBR(n)$: pour toute liste L,
$(\text{longueur}(L) = n) \rightarrow (\text{TriListe}(L)$ renvoie la liste constituée exactement des éléments de L triés en ordre croissant).

Initialisation :
$PBR(0)$ est vraie, car $\text{TriListe}(\langle\ \rangle)$ retourne $\langle\ \rangle$ qui est le bon résultat.
$PBR(1)$ est vraie, car $\text{TriListe}(\langle e \rangle)$ retourne la liste $\langle e \rangle$ qui est le bon résultat.

Induction :
Il faut montrer : $\forall n \in \mathbb{N}^*, \bigwedge_{i=0}^{n} PBR(i) \vDash PBR(n+1)$ (hérédité forte).

On rappelle la définition de $PBR(n+1)$:
$PBR(n+1)$: pour toute liste L,
$(\text{longueur}(L) = n+1) \rightarrow (\text{TriListe}(L)$ renvoie la liste constituée exactement des éléments de L triés en ordre croissant).

Soit L une liste telle que $\text{longueur}(L) = n+1$. Comme $n+1 \geq 2$, on exécute le second « sinon ». $\text{ImpairListe}(L)$ et $\text{PairListe}(L)$ sont des fonctions qui s'arrêtent et renvoient respectivement des listes L' et L'' de longueur respective n_i et n_p avec $\text{n}_i + \text{n}_p = n+1$, $1 \leq \text{n}_i < n$ et $1 \leq \text{n}_p < n$ et telles que L' et L'' contiennent à elles deux exactement les éléments de L. Par hypothèse de récurrence, chacun des appels $\text{TriListe}(L')$ et $\text{TriListe}(L'')$ renvoient deux listes triées R' et R'' constituées respectivement des éléments de L' et L''. Comme $\text{Fusion}(R', R'')$ renvoie une liste triée constituée des éléments des deux listes triées R' et R'', nous pouvons en conclure que $\text{TriListe}(L)$ renvoie une liste triée de tous les éléments de L. Donc $PBR(n+1)$ est vraie.

Conclusion :
$PBR(0)$ et $PBR(1)$ sont vraies et $\forall n \in \mathbb{N}^*$, l'hérédité forte est vérifiée. On en déduit donc que $\forall n \in \mathbb{N}$, $PBR(n)$ est vraie.

6) Complexité en temps

Remarque : Le calcul du coût n'est pas aussi simple que pour les autres fonctions, car même certaines instructions qui ne sont pas des appels récursifs à TriListe() ne s'exécutent pas en temps constant. Là encore on suppose que les comparaisons se font en temps constant. On va donc simplement calculer le nombre d'appels sur toutes les fonctions utilisées, en parallèle on calculera le nombre de comparaisons.
À nouveau pour simplifier le calcul, on calcule tout d'abord les complexités pour des listes L telles que $\text{longueur}(L) = n$ avec $n = 2^k$ avec $k \geq 0$ (on généralisera ensuite les formules pour n entier quelconque ≥ 1 en remarquant que n peut être encadré par 2 puissances de 2).
On pose, respectivement, $ATe(k)$, $AIe(k)$ et $APe(k)$ le nombre maximum d'appels pour toute liste L de longueur 2^k de TriListe(), ImpairListe() et PairListe() et $AFe(k)$ le nombre maximum d'appels pour toute paire de listes dont la somme des longueurs vaut 2^k de Fusion(), le « e » signifiant « exposant ».
On pose $CTe(k)$ le nombre maximum de comparaisons pour toute liste L de longueur 2^k de TriListe() et $CFe(k)$ le nombre maximum de comparaisons pour toute paire de listes dont la somme des longueurs vaut 2^k de Fusion(). On rappelle que ImpairListe() et PairListe() ne comportent aucune comparaison.
Pour $k \geq 1$, on a $\text{longueur}(\text{ImpairListe}(L)) = \text{longueur}(\text{PairListe}(L)) = 2^{k-1}$ le nombre d'appels est :

$$ATe(k) = 1 + AFe(k) + \big(1 + ATe(k-1) + 1 + AIe(k)\big) + \big(1 + ATe(k-1) + 1 + APe(k)\big)$$

Comme dans ce cas les longueurs des listes sont paires on a $AIe(k) = APe(k)$ (cf. V 6) et on peut simplifier en :

$$ATe(k) = 1 + AFe(k) + 2\big(2 + ATe(k-1) + AIe(k)\big)$$

$$= 1 + \big(2^k - 1\big) + 2\big(2 + ATe(k-1) + AIe(k)\big) = 2^k + 2\big(2 + ATe(k-1) + AIe(k)\big).$$

Ce qui donne :

$$ATe(k) = 2^k + 2\big(2 + ATe(k-1) + 2^{k-1}\big) = 4 + 2^{k+1} + 2\times ATe(k-1).$$

Le nombre de comparaisons est :

$$CTe(k) = CFe(k) + 2\times CTe(k-1).$$

D'après II 6, cela donne :

$$CTe(k) = 2^k - 1 + 2\times CTe(k-1).$$

D'où le tableau suivant :

k	n	$n\times k$	$ATe(k)$	$CTe(k)$	$\frac{ATe(k)}{n\times k}$	$\frac{CTe(k)}{n\times k}$
0	1	0	0	0	non def	non def
1	2	2	8	1	4	0,5
2	4	8	28	5	3,5	0,625
3	8	24	76	17	3,167	0,708
4	16	64	188	49	2,938	0,766
5	32	160	444	129	2,775	0,806
6	64	384	1020	321	2,656	0,836
7	128	896	2300	769	2,567	0,858
8	256	2048	5116	1793	2,498	0,875

Tableau 1

On peut remarquer sur les deux dernières colonnes que les deux complexités semblent en effet quasi proportionnelles à $n\times k$. Pour en être sûr d'un point de vue formel, on détermine maintenant les formules closes pour $CTe(k)$ et $ATe(k)$:

1. $CTe(k)$
   - $CTe(0) = 0$,
   - $CTe(1) = 2^1 - 1 + 2\times(0) = 1$,
   - $CTe(2) = 2^2 - 1 + 2\times(1) = 2^2 + 1$,
   - $CTe(3) = 2^3 - 1 + 2\times\big(2^2 + 1\big) = 2^3 + 2^3 + 2 - 1 = 2\times 2^3 + 1$,
   - $CTe(4) = 2^4 - 1 + 2\times\big(2\times 2^3 + 1\big) = 2^4 + 2\times 2^4 + 2 - 1 = 3\times 2^4 + 1$,
   - $CTe(5) = 2^5 - 1 + 2\times\big(3\times 2^4 + 1\big) = 2^5 + 3\times 2^5 + 2 - 1 = 4\times 2^5 + 1$,
   - …

   On peut en déduire que la formule close est de la forme $CTe(k) = (k-1)\times 2^k + 1$ pour tout $k \in \mathbb{N}$. Démontrons par récurrence qu'il s'agit effectivement de la bonne formule :
   On pose PCe la Propriété sur le nombre de Comparaisons :
   $\forall k \in \mathbb{N}, PCe(k)$: pour toute liste L, $\big(\text{longueur}(L) = 2^k\big) \rightarrow \big(CTe(k) = (k-1)\times 2^k + 1\big)$.

   <u>Initialisation</u> :
   $PCe(0)$ est vraie, car $\text{CTe}(0) = (0-1)\times 2^0 + 1 = 0$.

   <u>Induction</u> :
   Il faut montrer : $\forall k \in \mathbb{N}, PCe(k) \vDash PCe(k+1)$ (hérédité simple).

   On rappelle la définition de $PCe(k+1)$:
   $PCe(k+1)$: pour toute liste L, $\big(\text{longueur}(L) = 2^{k+1}\big) \rightarrow \big(CTe(k+1) = k\times 2^{k+1} + 1\big)$

Soit L une liste d'entiers telle que $\text{longueur}(L) = 2^{k+1}$ avec $k+1 \geq 1$. On exécute donc le second « sinon » et $CTe(k+1) = 2^{k+1} - 1 + 2\times CTe(k)$. Comme $k \geq 0$, on peut appliquer l'hypothèse d'hérédité, ce qui donne :

$$CTe(k+1) = 2^{k+1} - 1 + 2\times\left((k-1)\times 2^k + 1\right)$$

Donc

$$CTe(k+1) = 2^{k+1} - 1 + (k-1)\times 2^{k+1} + 2 = k\times 2^{k+1} + 1$$

et $PCe(n+1)$ est vérifiée.

Conclusion :
$PCe(0)$ est vraie et $\forall k \in \mathbb{N}$, l'hérédité simple est vérifiée. On en déduit donc que $\forall k \in \mathbb{N}$, $PCe(k)$ est vraie.

Donc $\text{CTe}(k) \in \theta\left(k\times 2^k\right)$ où 2^k est la longueur de la liste.

2. $ATe(k)$
   - $ATe(0) = 0$,
   - $ATe(1) = 4 + 2^2 + 2\times ATe(0) = 4 + 2^2 + 0 = 4 + 4 = 8$,
   - $ATe(2) = 4 + 2^3 + 2\times ATe(1) = 4 + 2^3 + 2\times(4+4) = 4\times(1+2) + (2+2)\times 2^2$
     $= 4\times(2^2 - 1) + 4\times 2^2 = (2\times 2 + 4)\times 2^2 - 4$,
   - $ATe(3) = 4 + 2^4 + 2\times ATe(2) = 4 + 2^4 + 2\times\left((2\times 2 + 4)\times 2^2 - 4\right)$
     $= 4\times(1-2) + (2+4+4)\times 2^3 = (2\times 3 + 4)\times 2^3 - 4$,
   - $ATe(4) = 4 + 2^5 + 2\times ATe(3) = 4 + 2^5 + 2\times\left((2\times 3 + 4)\times 2^3 - 4\right)$
     $= 4\times(1-2) + (2+6+4)\times 2^4 = (2\times 4 + 4)\times 2^4 - 4$,
   - $ATe(5) = 4 + 2^6 + 2\times ATe(4) = 4 + 2^5 + 2\times\left((2\times 4 + 4)\times 2^4 - 4\right)$
     $= 4\times(1-2) + (2+8+4)\times 2^5 = (2\times 5 + 4)\times 2^5 - 4$,
   - ...

   On peut en déduire que la formule close est de la forme $ATe(k) = (2k+4)\times 2^k - 4$ pour tout $k \in \mathbb{N}$. Démontrons par récurrence qu'il s'agit effectivement de la bonne formule :
   On pose PAe la Propriété sur le nombre d'Appels :
   $\forall k \in \mathbb{N}, PAe(k)$: pour toute liste L,
   $\left(\text{longueur}(L) = 2^k\right) \rightarrow \left(ATe(k) = (2k+4)\times 2^k - 4\right)$.

   Initialisation :
   $PAe(0)$ est vraie, car $\text{ATe}(0) = (0+4)\times 2^0 - 4 = 0$.

   Induction :
   Il faut montrer : $\forall k \in \mathbb{N}$, $PAe(k) \vDash PAe(k+1)$ (hérédité simple).

   On rappelle la définition de $PAe(k+1)$:
   $PAe(k+1)$: pour toute liste L,
   $\left(\text{longueur}(L) = 2^{k+1}\right) \rightarrow \left(ATe(k) = (2(k+1)+4)\times 2^{k+1} - 4\right)$

   Soit L une liste d'entiers telle que $\text{longueur}(L) = 2^{k+1}$ avec $k+1 \geq 1$. On exécute donc le second « sinon » et $ATe(k+1) = 4 + 2^{k+2} + 2\times ATe(k)$. Comme $k \geq 0$, on peut appliquer l'hypothèse d'hérédité, ce qui donne :

$$ATe(k+1) = 4 + 2^{k+2} + 2\times\left((2k+4)\times 2^k - 4\right)$$

   Donc

$$ATe(k+1) = 4\times(1-2) + (2+2k+4)\times 2^{k+1} = (2(k+1)+4)\times 2^{k+1} - 4$$

   et $PAe(n+1)$ est vérifiée.

   Conclusion :
   $PAe(0)$ est vraie et $\forall k \in \mathbb{N}$, l'hérédité simple est vérifiée. On en déduit donc que $\forall k \in \mathbb{N}$, $PAe(k)$ est vraie.

Donc $\mathrm{ATe}(k) \in \theta\left(k \times 2^k\right)$ où 2^k est la longueur de la liste.

La généralisation à toute liste de longueur pouvant être différente d'une puissance de 2 s'obtient en observant que pour tout $n \in \mathbb{N}^*$, $\exists k \in \mathbb{N}$, $2^k \leq n < 2^{k+1}$. On a donc :

$$CTe(k) \leq CT(n) \leq CTe(k+1)$$

et

$$ATe(k) \leq AT(n) \leq ATe(k+1).$$

Donc CT et AT sont bien dans $\theta(\log(n) \times n)$.