Centrale 2012. Option Informatique.

Corrigé pour serveur UPS par JL. Lamard (jean-louis.lamard@prepas.org)

I Tri rapide d'un tableau.

I.A -

```
let echange i j v = let temp = v.(i) in v.(i) <- v.(j); v.(j) <- temp;;
echange : int -> int -> 'a vect -> unit = <fun>
```

I.B - On peut par exemple employer le tri par sélection qui consiste à rechercher le plus petit élément du tableau, à l'échanger avec le premier puis à recommencer sur le sous-tableau des $n-1$ derniers éléments etc ... □

```
let tri_selection v =
  let n = vect_length v and min = ref 0 in
    for i=0 to n-2 do
      min := i;
      for j=i+1 to n-1 do
        if v.(j) < v.(!min) then min := j
      done;
      echange i !min v
    done
;;
tri_selection : 'a vect -> unit = <fun>
```

Lors de la boucle d'indice i l'algorithme effectue $n-1-i$ comparaisons d'où un nombre total de comparaisons de $(n-1)+(n-2)+\ldots+1 \sim \dfrac{n^2}{2}$ en notant n la longueur du tableau. □

I.C.1) et 2)

Première méthode

Soit un (sous-)tableau $v(a,b)$ constitué des éléments d'indice entre a et b d'un tableau v que l'on veut séparer suivant le pivot $v(p_0)$ avec $a \leqslant p_0 \leqslant b$.

On parcourt le (sous-)tableau pour k de 0 à $n-1$ (avec $n = b-a+1$) en gérant un pointeur p de manière à maintenir l'invariant de boucle suivant : à la fin de la boucle k :

1/ $a \leqslant p \leqslant a+k$

2/ les $k+1$ premiers éléments du tableau c'est à dire ceux d'indice compris entre a et $a+k$ sont rangés de sorte ceux (s'il y en a) d'indice entre a et $p-1$ sont inférieurs ou égaux à $v(p)$ et ceux (s'il y en a) d'indice entre $p+1$ et $a+k$ sont strictement supérieurs à $v(p)$.

Si on établit un tel algorithme, à la fin de la boucle $n-1$ le tableau sera bien séparé suivant la condition exigée et la variable p contiendra l'indice du pivot. En outre il se réalise bien en place.

1/ Initialisation : il suffit d'échanger $v(p_0)$ et $v(a)$ (si $p_0 \neq a$!) et de faire pointer p sur a.

2/ Supposons l'étape $(k-1)$ réalisée avec $1 \leqslant k \leqslant n-1$.

a) Si $v(a+k) > v(p)$ alors il n'y a rien à faire !

b) Sinon :

Si $p = a+k-1$ on échange $v(p)$ et $v(p+1)$ puis on incrémente p

Sinon on échange $v(a+k)$ et $v(p+1)$ puis on échange $v(p)$ et $v(p+1)$ et enfin on incrémente p.

Chaque tour de boucle est à complexité temporelle bornée car il se compose d'une comparaison et au plus d'une autre comparaison, de deux échanges et d'une incrémentation.

La complexité temporelle est donc bien en $O(n)$. □

```
let separation1 v a b = let n = b-a+1 and p = ref a in
  for k=1 to n-1 do
    if v.(a+k) <= v.(!p) then match ( !p = a+k-1 ) with
      | true -> echange !p (!p+1) v;
                incr p
      | _    -> echange (a+k) (!p+1) v;
                echange !p (!p+1) v;
                incr p
  done;
  !p
;;
separation1 : 'a vect -> int -> int -> int = <fun>
```

Seconde méthode

Comme précédemment si $p_0 \neq a$ on commence par se ramener au cas où le pivot est le premier terme du tableau en échangeant $v(a)$ et $v(p_0)$.
On initialise ensuite deux pointeurs min et max respectivement à $a+1$ et b et on maintient l'invariant de boucle :

1/ $v(a)$ ne bouge pas.
2/ si $a+1 \leqslant k < min$ alors $v(k) \leqslant v(a)$
3/ si $max < k \leqslant b$ alors $v(a) < v(k)$

au cours de la boucle

Tant que $max - min > 0$:
Si $v(min) \leqslant v(a)$: incrémenter min
sinon échanger $v(min)$ et $v(max)$ puis décrémenter max.

À la sortie $min = max$ et deux cas:
1/ $v(min) \leqslant v(a)$: alors on échange $v(a)$ et $v(min)$ et on renvoie min.
2/ sinon on échange $v(a)$ et $v(min-1)$ et on renvoie $min-1$ (valable même si $min - 1 = a$ auquel cas il suffit de renvoyer a).

On remarque que cette étape finale peut être incluse dans la boucle en allant jusqu'au cas $max - min = 0$ et qu'il suffit à la sortie d'échanger $v(a)$ et $v(max)$ et de renvoyer max.

Cette boucle se termine en n tours car $max - min$ diminue d'une unité à chaque tour . Elle s'effectue sur place. Chaque tour est à temps borné donc la complexité temporelle de l'algorithme est en $O(n)$.

```
let separation2 v a b = let min = ref (a+1) and max = ref b in
  while (!max - !min ) >= 0 do
    if (v.(!min) <= v.(a)) then incr min
    else begin echange !min !max v; decr max end
  done;
  echange a !max v;
  !max
;;
separation2 : 'a vect -> int -> int -> int = <fun>
```

I.C.3)

```
let rec tri_rapide_aux v a b = match (b-a) with
  | n when n <= 0 -> ()
  | _ -> let p = separation v a b in
           tri_rapide_aux v a (p-1);
           tri_rapide_aux v (p+1) b
;;
tri_rapide_aux : 'a vect -> int -> int -> unit = <fun>

let tri_rapide v = tri_rapide_aux v 0 (vect_length v-1);;
tri_rapide : 'a vect -> unit = <fun>
```

II Étude de complexité.

Notons que l'équation de complexité du tri rapide est d'une manière générale $T(n) = P(n) + T(r_n) + T(s_n)$ où $P(n)$ est le temps de séparation, r_n et s_n les longueurs des deux sous-vecteurs à gauche et à droite du pivot après séparation. Notons que les deux algorithmes proposés de séparation effectuent n comparaisons.

II.A - Supposons v déjà trié par ordre croissant (ou décroissant car l'étude serait la même).
Il est impossible de répondre à la question sans préciser l'algorithme utilisé pour la séparation. En effet si à la première étape on a évidemment $r = 0$ et $s = n-1$ il n'en va pas forcément de même par la suite car le sous-vecteur de droite n'est pas forcément trié !

• Si on sépare par la première méthode, il est facile de voir que c'est bien le cas (et donc tout au long de l'algorithme) de sorte que l'équation de complexité en comparaisons est alors $C(n) = n + C(n-1)$.

Donc classiquement $C(n) \sim \dfrac{n^2}{2}$ et dans ce cas le tri rapide n'a rien de rapide ! □

• Si on sépare par la seconde méthode le sous-vecteur de droite obtenu à la première étape n'est pas trié et on ne peut rien dire de r_k et s_k pour $k \leqslant n-1$ (à part bien sûr $r_k + s_k = k-1$). En fait le membre de droite devient de plus en plus "anarchique" et la complexité du tri est probablement bien moindre que le quadratique et c'est a priori tout ce qu'on peut dire.

Il semble bien en tout cas que la seconde méthode de séparation soit bien meilleure ce que nous allons vérifier expérimentalement.

• Étude expérimentale.

Le programme suivant mesure le temps du tri rapide pour un vecteur de longueur n dont les éléments sont classés par ordre croissant de 0 à $n-1$.

```
let mesure n =
  let v = make_vect n 0 in
    for i=1 to n - 1 do v.(i) <- i done;
    let t = sys__time() in
      tri_rapide v;
      sys__time() -. t
;;
mesure : int -> float = <fun>
```

Avec la première fonction de séparation j'obtiens (en arondissant les temps renvoyés au dixième de seconde) pour les valeurs successives de n : 3000, 6000 et 12000 les temps suivants : 1.1, 4.6 et 17.9. La vérification expérimentale du quadratique est quasi parfaite !

Avec la seconde fonction de séparation on obtient pour les mêmes valeurs de n les temps suivants : 0.2, 0.5 et 1.3 (et 3.6 pour $n = 24000$). Déjà ces temps sont beaucoup plus faibles en valeur absolue et la croissance est manifestement moindre que le quadratique. Expérimentalement elle semble de l'ordre de $n^{1.4}$. □

II.B.1) Dans ce cas l'équation de complexité en comparaisons devient $C(2n) = 2n + C(n-1) + C(n)$ et comme C est clairement croissante (et que $C(n-1)$ est peu différent de $C(n)$ pour n grand) un majorant "raisonnable" de $C(n)$ vérifie la relation $M(2n) = 2n + 2M(n)$. □

II.B.2) Si $n = 2^k$ en notant $m_k = M_{2^k}$ il vient $m_k = 2^k + 2m_{k-1}$ donc par une itération évidente $m_k = k2^k + 2^k m_0$. Or $m_0 = M_1 = 0$ de sorte que $M_n = n\log_2 n$ lorsque $n = 2^k$. □

II.B.3) Si on admet (ce qui est "raisonnable") que la suite $\big(M(n)\big)_{n\in\mathbb{N}}$ est croissante il vient :

$k_n 2^{k_n} \leqslant M(n) \leqslant (k_n+1)2^{k_n+1}$ avec $k_n = \mathrm{Int}(\log_2 n)$.

Or lorsque $n \to +\infty$ il en va de même de k_n de sorte que lorsque $n \to +\infty$ on a $(k_n+1)2^{k_n+1} = \Theta\big(k_n 2^{k_n}\big)$ et on a également $k_n 2^{k_n} = \Theta\big(n\log_2 n\big)$ et finalement $M(n) = \Theta\big(n\log_2 n\big)$ □

III Recherche d'une pseudo-médiane.

III.A - Dans le tableau, en place.

III.A.1)

```
let mediane v i j k =
  if (v.(i)-v.(j))*(v.(i)-v.(k)) < 0 then i
  else if (v.(j)-v.(i))*(v.(j)-v.(k)) < 0 then j
  else k
;;
mediane : int vect -> int -> int -> int -> int = <fun>
```

III.A.2) L'algorithme suivant procède principalement d'une boucle sur r initialisé à 1.

À chaque tour r est multiplié par 3.

Au rang r on ne considérera que les éléments en position (position = indice + 1) $k * r$ avec $1 \leqslant k \leqslant (n/r)$

Le tour de rang r sera lui-même une boucle sur k où on prend la médiane des éléments en position kr, $(k+1)r$ et $(k+2)r$ que l'on place en position $(k+2)r$ pour k démarrant à 1 et tant que $(k+2)r \leqslant n$

À chaque tour sur k on augmente k de 3.

La boucle sur r se poursuit tant que $r < n$

À la fin la pseudo-médiane est placée en dernière position du vecteur.

```
let pseudo_mediane v =
  let n = vect_length v and r = ref 1 in
    while !r < n do
      let k = ref 1 in
        while (!k+2)*(!r) <= n do
          echange ((!k+2)*(!r)-1) (mediane v (!k*(!r)-1) ((!k+1)*(!r)-1) ((!k+2)*(!r)-1)) v;
          k := 3+(!k)
        done;
      r := 3*(!r)
    done
;;
pseudo_mediane : int vect -> unit = <fun>
```

III.B - À l'aide d'un arbre ternaire.

III.B.1)

```
let mediane3 a b c =
  if (a-b)*(a-c) < 0 then a
  else if (b-a)*(b-c) < 0 then b
  else c
;;
mediane3 : int -> int -> int -> int = <fun>
```

III.B.2)

```
let construire3 t1 t2 t3 =
  let a = racine t1 and b = racine t2 and c = racine t3 in
    let m = mediane3 a b c in N(m,t1,t2,t3)
;;
construire3 : ternaire -> ternaire -> ternaire -> ternaire = <fun>
```

III.B.3)

```
let rec construire v i j = match j-i+1 with
  | 1 -> F v.(i)
  | _ -> let k = (j-i+1)/3 in
           let t1 = construire v i (i+k-1)
           and t2 = construire v (i+k) (i+2*k-1)
           and t3 = construire v (i+2*k) (i+3*k-1) in
             construire3 t1 t2 t3
;;
construire : int vect -> int -> int -> ternaire = <fun>
```

III.B.4)

```
let pseudo_mediane2 v = racine (construire v 0 ((vect_length v)-1));;
pseudo_mediane2 : int vect -> int = <fun>
```

III.C - Étude théorique de l'algorithme.

III.C.1) La question est ambigüe car on dispose de deux algorithmes. Notons déjà qu'il est clair qu'ils renvoient bien la même pseudo-médiane. Notons $n = 3^K$ le nombre d'éléments du tableau.

• Algorithme dans le tableau : la boucle sur r comporte K tours. La boucle sur k consiste en le calcul de 3^{K-r} médianes et autant d'échanges. Son temps est donc proportionnel à 3^{K-r}.
La complexité temporelle est donc proportionnelle à $3^{K-1} + 3^{K-2} + \ldots + 1 = \frac{1}{2}(3^K - 1) = \frac{n-1}{2}$.
La complexité temporelle est donc linéaire. □

• Algorithme avec l'arbre ternaire : L'équation de complexité est en notant $t(k) = T(3^k)$: $t(k) = 3t(k-1) + \alpha$ où α est le temps constant d'exécution de `construire3`. Donc $t(k) = (1 + 3 + 3^2 + \ldots + 3^{K-1})\alpha + 3^K\beta = \Theta(3^K)$.
Là encore la complexité temporelle est linéaire. □

Ainsi si besoin dans la suite on raisonnera sur l'algorithme avec l'arbre ternaire où la nature récursive est claire.

III.C.2) Soit le prédicat $\mathcal{P}_k$:«dans un tableau de longueur 3^k il a au moins 2^k éléments majorés au sens large par l'élément renvoyé par l'algorithme par arbre ternaire»
$\mathcal{P}_0$ est clairement vrai. Supposons $\mathcal{P}_{k-1}$ vrai avec $k \geqslant 1$ et considérons un tableau de longueur 3^k.
Avec les notations de l'algorithme les sous-vecteurs v_1, v_2 et v_3 (dont la "réunion" constitue l'ensemble des éléments de v) correspondants aux feuilles des arbres t_1, t_2 et t_3 vérifient l'hypothèse de récurrence.

Notons a_i les racines respectives de ces trois arbres. La valeur renvoyée est la médiane a des a_i. Supposons (ce qui ne restreint en rien la généralité) que $a_1 \leqslant a_2 \leqslant a_3$ de sorte que $a = a_2$. Il y a 2^{k-1} éléments au moins de v_1 et v_2 inférieurs respectivement à a_1 et a_2 donc à a.
Donc il y a bien 2^k éléments au moins de v inférieurs ou égaux à a ce qui établit la propriété. □

Remarque : par une démonstration en tous points analogues on prouverait qu'il existe au moins 2^k éléments minorés par la valeur retournée.

III.C.3) On va commencer par construire un tel vecteur constitué uniquement de 0 et de 1 puis, dans un deuxième temps comme l'énoncé suggère (ce n'est pas vraiment clair) que les éléments sont deux à deux distincts, on va en déduire un vecteur convenant sans doublon.

- On construit itérativement sur la profondeur un arbre ternaire de la manière suivante :

On part de $F(0)$ puis on passe d'une profondeur i à la profondeur $i+1$ en remplaçant chaque feuille $F(0)$ par l'arbre $N\big(0, F(0), F(0), F(1)\big)$ et chaque feuille $F(1)$ par l'arbre $N\big(1, F(1), F(1), F(1)\big)$.
Soit t_k l'arbre ainsi construit de profondeur k et v le vecteur constitué des feuilles de cet arbre lues dans l'ordre de gauche à droite. Par construction même l'arbre ternaire construit à partir de ce vecteur v est l'arbre t_k. La valeur retournée par l'algorithme est donc 0. Or cet arbre contient (par une récurrence immédiate) exactement 2^k fois la valeur 0. □

- Soit $\widetilde{v}$ un vecteur obtenu à partir de v en remplaçant les 0 par des entiers strictement négatifs deux à deux distincts et les 1 par des entiers strictement positifs deux à deux distincts et soit $\widetilde{t_k}$ l'arbre ternaire construit à partir de $\widetilde{v}$.

Par récurrence descendante immédiate sur la profondeur il vient que si un noeud de t_k est égal à 0 (resp. 1) alors le noeud correspondant de $\widetilde{t_k}$ est strictement négatif (resp. strictement positif).
En particulier sa racine i.e. la valeur retournée par l'algorithme est strictement négative. Or cet arbre contient exactement 2^k éléments strictement négatifs. Il admet donc au plus 2^k éléments majorés au sens large par la racine. Mais par la question précédente il en admet au moins 2^k.
On vient donc ainsi de construire un vecteur de longueur 3^k n'ayant aucun doublon et admettant exactement 2^k éléments majorés au sens large par la valeur retournée par l'algorithme. □

III.C.4) Immédiat compte-tenu de la question III.C.2 et de sa remarque car alors $n^{\ln 2/\ln 3} = (3^k)^{\ln 2/\ln 3} = 2^k$. □

III.C.5) Soit v un vecteur de longueur n et k l'unique entier tel que $3^k \leqslant n < 3^{k+1}$.
On peut appliquer l'algorithme sur un sous-vecteur v' constitué de 3^k valeurs de v (pas forcément consécutives). Notons a' sa sous-médiane. On a au moins 2^k éléments de v' donc de v supérieurs et autant d'inférieurs à a'.
Or $n^{\ln 2/\ln 3} < 2^{k+1} = 2 \times 2^k$ de sorte qu'on a au moins $\dfrac{1}{2} n^{\ln 2/ln3}$ éléments de v supérieurs et inférieurs à la pseudo-médiane a' de v' qui est bien de ce fait une $\dfrac{\ln 2}{\ln 3}$-pseudo-médiane de v. □

L'implémentation ne demande strictement aucune modification : la fonction `construire` s'applique à un vecteur de longueur quelconque car à chaque appel récursif elle "oublie" éventuellement le ou les deux derniers éléments de $v(i,j)$ selon la congruence modulo 3 de $j-i+1$.
Elle renvoie donc bien la pseudo-médiane a' d'un tel sous-vecteur v'. □

III.D - Extension du principe de l'algorithme.

III.D.1) Pour un vecteur de longueur 5^k on procède récursivement de la même manière par le calcul de la médiane de blocs de 5 éléments successifs. Comme à chaque fois on a 3 éléments majorés et minorés au sens large par cette médiane, on prouve comme précédemment qu'il y a au moins 3^k éléments majorés et minorés par la valeur retournée qui de ce fait est une $\ln 3/\ln 5$-pseudo-médiane du vecteur.

De la même manière que ci-dessus cet algorithme s'applique en fait à un vecteur de longueur quelconque en retournant bien une $\ln 3/\ln 5$-pseudo-médiane. □

Cet algorithme est linéaire de la même manière que précédemment avec les blocs de 3.

III.D.1) En procédant avec des blocs de $2N+1$ éléments on obtient de la même manière un algorithme linéaire qui renvoie une $\ln(N+1)/\ln(2N+1)$-pseudo-médiane.
Or $\ln(2N+1) \sim \ln N \sim \ln(N+1)$ lorsque N tend vers $+\infty$ donc $\ln(N+1)/\ln(2N+1) \xrightarrow[N\to+\infty]{} 1$.
Ainsi en choisissant N assez grand on obtient bien une $(1-\varepsilon)$-pseudo-médiane avec une complexité temporelle linéaire. □

IV Gain apporté par la pseudo-médiane.

IV.A -) L'équation de complexié est $C(n) = \Theta(n) + C(p_1) + C(p_2)$ où $\Theta(n)$ est le temps de calcul de la pseudo-médiane, p_1 et p_2 les longueurs des deux "moitiés".

On sait que $p_1+p_2=n$, $p_1 \geqslant K_1\sqrt{n}$ et $p_2 \geqslant K_2\sqrt{n}$. Supposons par exemple $K_1 \leqslant K_2$
Le cas le plus défavorable est intuitivement (cela resterait à justifier rigoureusement ...) celui où les longueurs des deux "moitiés" sont respectivement de $K_1\sqrt{n}$ et $n-K_1\sqrt{n}$ tout au long des appels récursifs.
Ce qui fournit l'inégalité $C(n) \leqslant \Theta(n)+C(K_1\sqrt{n})+C(n-K_1\sqrt{n})$.
Or le tri rapide est au plus quadratique de sorte que $C(K_1\sqrt{n})=O(n)$ et en admettant que $C(n-K_1\sqrt{n})$ est asymptotiquement du même ordre que $C(n-\sqrt{n})$ (raisonnable ?) on obtient ainsi on obtient bien "raisonnablement" l'inégalité $C(n) \leqslant C(n-\sqrt{n})+Kn$. □

IV.B.-)• La suite $(\alpha_k)_{k\in\mathbb{N}}$ est clairement définie, à valeurs positives ou nulles et décroissante. Elle admet donc une limite $\ell \geqslant 0$.
Supposons $\alpha_k \geqslant 1$ pour tout k. Alors $\ell \geqslant 1$ et $\alpha_{k+1}=\alpha_k-\sqrt{\alpha_k}$ pour tout k donc $\ell=\ell-\sqrt{\ell}$ ce qui est contradictoire avec $\ell \geqslant 1$.
Il en découle que la suite est nulle à partir d'un certain rang et vérifie plus précisément :
$n>\alpha_1>\alpha_2>\ldots>\alpha_{k_0}>1>\alpha_{k_0+1}>0$ avec $\alpha_{k+1}=f(\alpha_k)$ pour $1 \leqslant k \leqslant k_0$ puis $\alpha_k=0$ pour $k>k_0+1$
en notant $f(x)=x-\sqrt{x}$.
Dans la première phase de décroissance stricte, à chaque itération α_k diminue de $\sqrt{\alpha}_k$. Donc tant que $\alpha_k \geqslant n/2$ à chaque itération la valeur diminue d'au moins $\sqrt{n/2}$.
Soit $k \geqslant \sqrt{n/2}$. Supposons $\alpha_k > n/2$. Alors au cours des k premières itérations la valeur de la suite a diminué d'au moins $k\times\sqrt{n/2} \geqslant n/2$ depuis sa valeur n de départ. Donc $\alpha_k \leqslant n-(n/2)=n/2$. Contradiction.
Ainsi pour $k \geqslant \sqrt{n/2}$ on a bien $\alpha_k \leqslant n/2$. □

• La relation $C(n) \leqslant Kn+C(n-\sqrt{n})$ s'écrit $C(\alpha_0) \leqslant K\alpha_0+C(\alpha_1)$ et par itération on obtient
$C(n) \leqslant K(\alpha_0+\alpha_1+\ldots+\alpha_k)+C(\alpha_{k+1}) \leqslant K(k+1)n+C(\alpha_{k+1})$ pour tout entier k (car $\alpha_i \leqslant n$ pour tout i).
En particulier avec $k=\text{Int}\left(\sqrt{n/2}\right)+1$ il vient $C(n) \leqslant K\Big(\text{Int}(\sqrt{n/2})+2\Big)n+C(n/2)$
car $\alpha_{k+1} \leqslant n/2$ et C est croissante.
Or $\text{Int}(\sqrt{n/2})+2=O(\sqrt{n})$ donc on a bien $C(n)=C(n/2)+O(n^{3/2})$. □

IV.C - Notons $x_k=C(2^k)$ et $a=2^{3/2}$. Il vient $x_k \leqslant x_{k-1}+Ka^k \leqslant \ldots \leqslant x_0+K(a^k+a^{k-1}+\ldots+a) \underset{\text{DEF}}{=} X_k$

Or $X_k=x_0+aK\dfrac{a^k-1}{a-1}=O(a^k)$ car $a>1$.

Ainsi $C(2^k)=O\big((2^k)^{3/2}\big)$ i.e. il existe $M>0$ tel que $C(2^k) \leqslant M\times(2^k)^{3/2}$ pour tout entier k.
Soit désormais n quelconque et k tel que $2^{k-1} \leqslant n < 2^k$. En admettant que C est croissante il vient
$C(n) \leqslant M\times(2^k)^{3/2}=2^{3/2}M\times(2^{k-1})^{3/2} \leqslant 2^{3/2}M\times n^{3/2}=O(n^{3/2})$

Ainsi en appliquant le tri rapide avec une 1/2-pseudo-médiane, dans le pire des cas on peur "raisonnablement" espérer une complexité en $O(n^{3/2})$ au lieu d'une complexité quadratique. □

IV.D - Essayons de trouver un majorant raisonnable de $C(n)$ lorsqu'on emploie l'algorithme du tri rapide utilisant une α-pseudo-médiane (avec $\alpha \geqslant 1/2$).

• Une première idée consiste à adapter le IV.A) ce qui fournit $C(n) \leqslant \Theta(n)+O(n^{2\alpha})+C(n-n^\alpha)$ donc
$C(n) \leqslant C(n-n^\alpha)+Kn^{2\alpha}$
Mais ce majorant est trop pessimiste et sans intérêt car lorsque $\alpha \to 1^-$ il augmente vers le quadratique !
Il convient donc de remplacer l'analyse du IV.a) par une analyse plus fine.

• On va chercher si on peut trouver un réel $\beta<2$ tel que $C(n)=O(n^\beta)$ dans le pire des cas. Nous allons admettre (raisonnable ?) son existence et chercher alors sa valeur possible.
Le IV.a) s'adapte alors en $C(n) \leqslant \Theta(n)+O(n^{\alpha\beta})+C(n-n^\alpha)$
De manière à nous ramener à une inégalité de même nature que dans le IV.A) : $C(n) \leqslant C(n-n^\alpha)+Kn$
nous faisons l'hypothèse supplémentaire que $\alpha\beta \leqslant 1$ et nous allons regarder si cette hypothèse est contradictoire ou non avec les conséquences que nous en tirons.

On introduit la suite (α_k) définie comme dans le IV.B) en remplaçant $\sqrt{x}$ par x^α. Comme précédemment elle décroît strictement jusqu'à atteindre une valeur inférieure à 1 puis est stationnaire en 0.
Dans la phase de décroissance, tant que $\alpha_k \geqslant n/2$ elle décroît à chaque itération d'au moins $(n/2)^\alpha$.
Donc pour $k \geqslant (n/2)^\gamma$ avec $\gamma=1-\alpha$ on a $\alpha_k \leqslant n/2$.
On en déduit comme précédemment $C(n) \leqslant K\Big(\text{Int}((n/2)^\gamma)+2\Big)n+C(n/2)$ et finalement
$C(n)=C(n/2)+O(n^{1+\gamma})=C(n/2)+O(n^{2-\alpha})$

On en déduit, par le même méthode que dans IV.C), que $C(n)=O(n^{2-\alpha})$.

Ainsi on trouve $2-\alpha$ comme valeur de β et cette valeur est bien cohérente avec les hypothèses faites à savoir $\beta<2$ et $\alpha\beta \leqslant 1$ car $\alpha^2-2\alpha+1=(1-\alpha)^2 \geqslant 0$

En conclusion on peut "raisonnablement" penser que la complexité du tri rapide avec une α-pseudo-médiane est dans le pire des cas en $O(n^{2-\alpha})$ donc en $O(n^{1+\varepsilon})$ pour $\alpha = 1-\varepsilon$. □

On remarque avec satisfaction que pour $\alpha = 1/2$ on retrouve bien le résultat précédent.
Pour $\alpha = \ln 2/\ln 3$ on obtient donc une complexité au pire de l'ordre de $O(n^{1.37})$ □

Complément : Étude expérimentale.

On va implémenter effectivement l'algorithme de tri rapide avec une $\ln 2/\ln 3$-pseudo-médiane puis faire des comparaisons de temps de calcul avec le tri rapide sans pseudo-médiane.

On commence par adapter la fonction `construire` pour renvoyer l'arbre ternaire dont les noeuds ont pour étiquette non pas la valeur des médianes successives mais leur indice dans le tableau initial.

```
let construire3_bis v t1 t2 t3 =
  let a = racine t1 and b = racine t2 and c = racine t3 in
    let m = mediane v a b c in N(m,t1,t2,t3)
;;

let rec construire_bis v i j = match j-i+1 with
  | n when n <=2  -> F i
  | _ -> let k = (j-i+1)/3 in
    let t1 = construire_bis v i (i+k-1)
    and t2 = construire_bis v (i+k) (i+2*k-1)
    and t3 = construire_bis v (i+2*k) (i+3*k-1) in
      construire3_bis v t1 t2 t3
;;
construire_bis : int vect -> int -> int -> ternaire = <fun>
```

La fonction suivante sépare $v(a,b)$ suivant sa pseudo-médiane et renvoie l'indice de cette médiane :

```
let separation3 v a b =
  let m = racine ( construire_bis v a b) in
    echange a m v;
    separation2 v a b
;;
separation3 : int vect -> int -> int -> int = <fun>
```

Le tri rapide s'écrit alors comme précédemment :

```
et rec tri_rapide3_aux v a b = match (b-a) with
  | n when n <= 0 -> ()
  | _ -> let p = separation3 v a b in
           tri_rapide3_aux v a (p-1);
           tri_rapide3_aux v (p+1) b
;;

let tri_rapide3 v = tri_rapide3_aux v 0 (vect_length v-1);;
tri_rapide3 : int vect -> unit = <fun>
```

On mesure alors les temps de calcul avce la fonction `mesure` écrite en II.A).

• *Avec la première fonction de séparation.*
Rappelons que l'on a vérifié qu'alors, sans pseudo-médiane, la complexité expérimentale était bien quadratique et on avait $t = 17.9$ pour $n = 12000$.
Avec l'utilisation de la pseudo-médiane on obtient pour les valeurs de n : 12000, 24000,48000 et 96000 les temps suivants : 0.2, 0.5, 1, 2.1 !!!
Progrès en valeur absolue considérable et la complexité ne semble pas éloignée pas loin du linéaire ...

• *Avec la seconde fonction de séparation.*
Sans pseudo-médiane on avait noté complexité de l'ordre de $n^{1.4}$ et pour $n = 24000$ on avait $t = 3.6$.
Avec l'utilisation de la pseudo-médiane on obtient pour les valeurs de n : 24000,48000 et 96000 les temps suivants : 0.4, 0.9, 2.1.
On obtient environ les mêmes temps qu'avec la première fonction de séparation, ce qui est normal. Et là encore le progrès en valeur absolue est considérable. La complexité est environ de l'ordre de $n^{1.2}$.

• En conclusion conformément à l'étude théorique le progrès dû à l'utilisation d'une pseudo-médiane est très sensible. □