Algorithmique & programmation

Chapitre 5 : Arbres

Tri par tas

Définition d'un tas

- Tout vecteur V[1..n] peut être considéré comme la représentation d'un arbre binaire complet au sens large, on va se servir de cet arbre binaire complet pour expliquer le tri d'un vecteur par la méthode du tas
- Un tas
  - Un tas est un arbre binaire complet (au sens large) dans lequel tout nœud non terminal vérifie

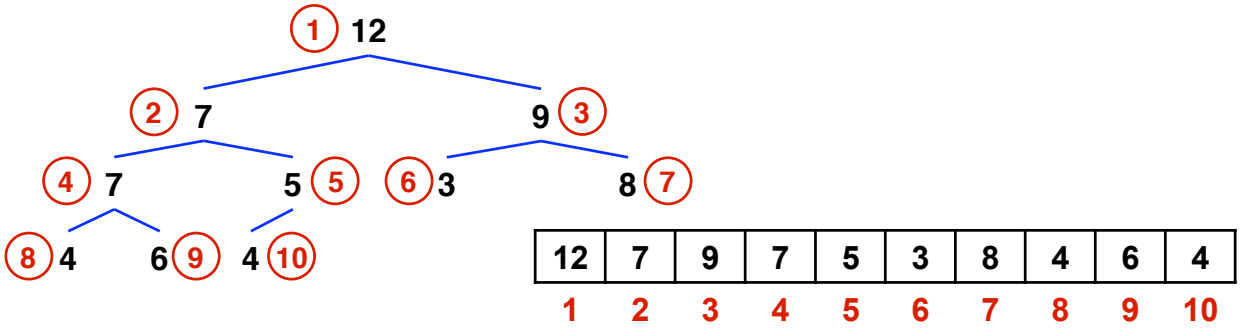
nœud↑. info ≥ max(nœud↑.gauche↑.info, nœud↑. droit↑.info)

  - Si un seul fils (obligatoirement fils gauche) :

nœud↑.info ≥ nœud↑.gauche↑.info

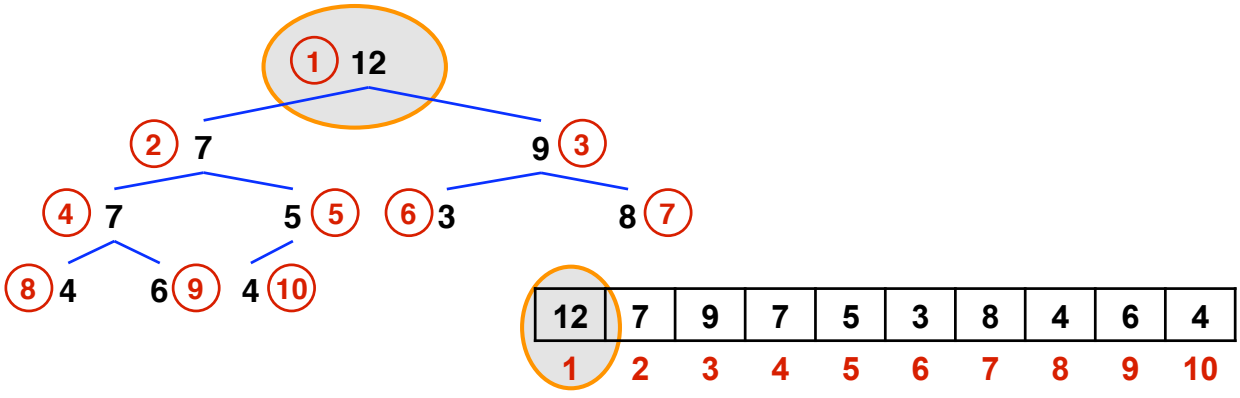
- « valeur du père ≥ valeurs du ou des fils »
- une feuille est un tas

Représentation vectorielle



- Traduction sur la représentation V[1..n]
  - si i ≥ 1, 2×i+1 ≤ n *{le nœud i a deux fils}*
    - V[i] ≥ V[2×i] *{fils gauche}*
    - V[i] ≥ V[2×i+1] *{fils gauche}*
  - sinonsi i ≥ 1, 2×i ≤ n *{le nœud i a un fils}*
    - ou V[i] ≥V[2×i] *{fils gauche}*
  - sinon i ≥ 1, 2×i > n *{le nœud i n'a pas de fils}*

Propriété d'un tas



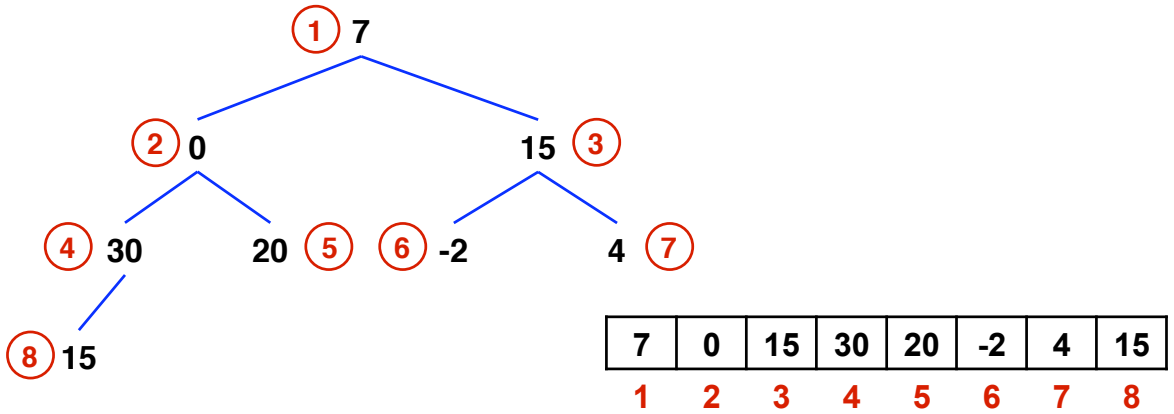
- La valeur associée à la racine est le maximum de tous les éléments
  - c'est donc le premier élément du vecteur
  - **V[1] ≥ V[1..n]**

Algorithme de tri d'un vecteur

- 2 parties
  1. Transformation de V[1..n] pour qu'il devienne un tas
     - V[1] ≥ V[1..n]
  2. Utilisation de cette propriété pour trier le vecteur
- Analogie avec le tri bulle
  1. faire remonter le plus grand du vecteur encore à trier
     - remontée dans un arbre de profondeur $\log_2 n$
     - *dans tribulle la profondeur est n*
  2. le mettre à sa place

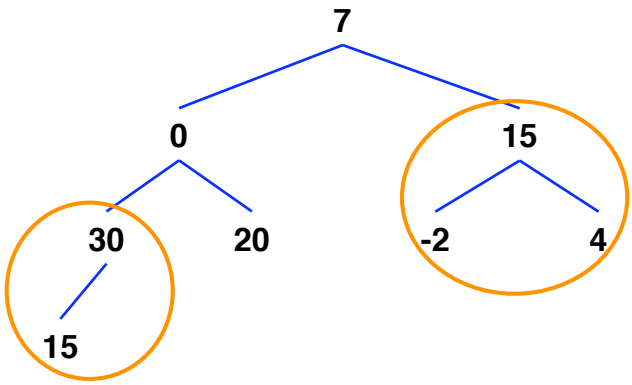
Pourquoi passer par un arbre ?

- L'arbre n'est ici qu'un objet virtuel nécessaire à la compréhension des actions entreprises sur le vecteur
- Exemple



Construction d'un tas : exemple

- faire un parcours ascendant par niveau en transformant si nécessaire les sous-arbres rencontrés pour qu'ils deviennent tas
- Premier niveau



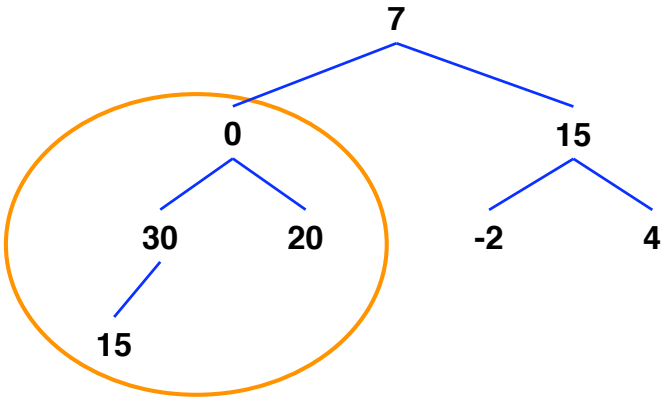
Construction d'un tas : exemple

- Premier niveau

  - sont des tas

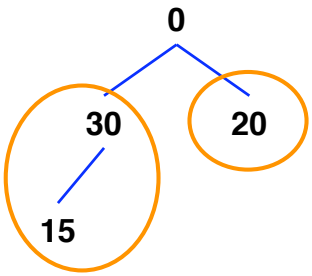
Construction d'un tas : exemple

- Deuxième niveau



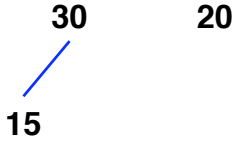
Construction d'un tas : exemple

- Deuxième niveau



  - ce n'est pas un tas

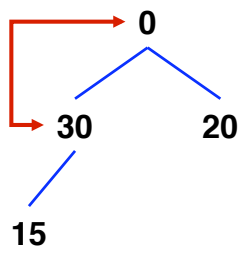
- Par contre



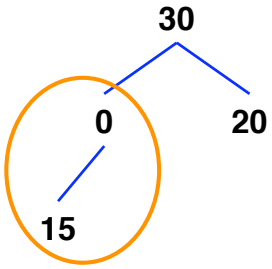
  - sont des tas

Construction d'un tas : exemple

- Deuxième niveau *suite*



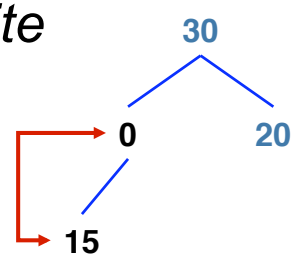
  - on sélectionne le plus grand des deux fils pour le permuter avec la racine
  - on obtient



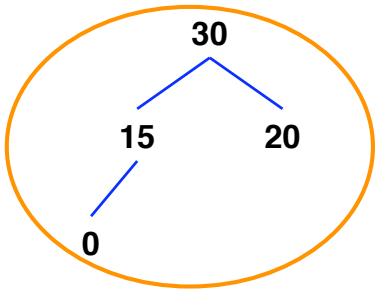
  - le sous-arbre modifié est-il un tas ? **non !**

Construction d'un tas : exemple

- Deuxième niveau *suite*



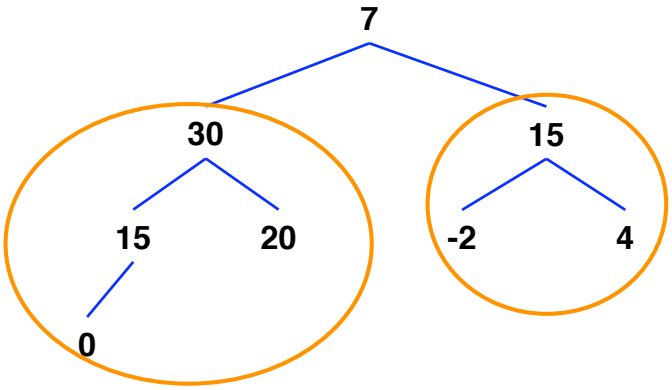
  - on sélectionne le plus grand des deux fils pour le permuter avec la racine
  - on obtient



  - le sous-arbre modifié est un tas !

Construction d'un tas : exemple

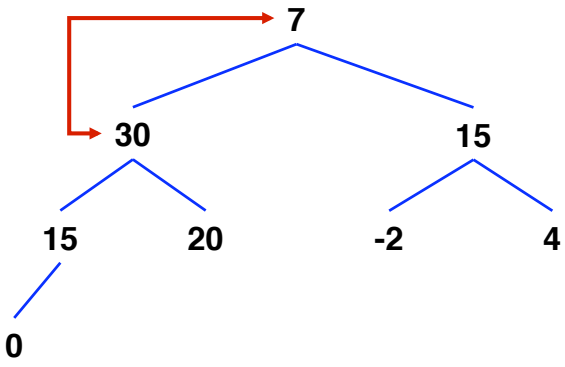
- Deuxième niveau *fin*
  - l'arbre finalement obtenu est :



- Cet arbre n'est pas un tas
  - mais on sait que les sous-arbres sont des tas

Construction d'un tas : exemple

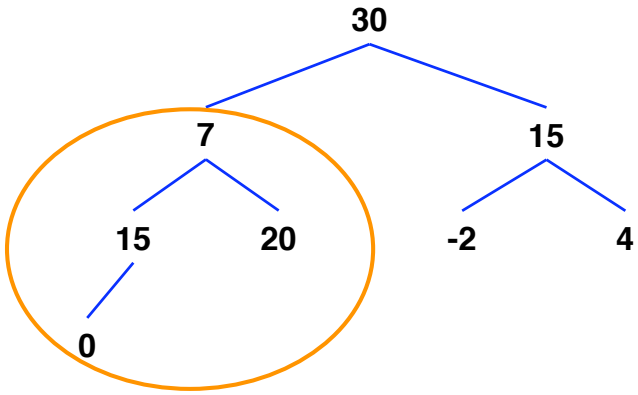
- Troisième niveau



  - On permute 30 et 7

Construction d'un tas : exemple

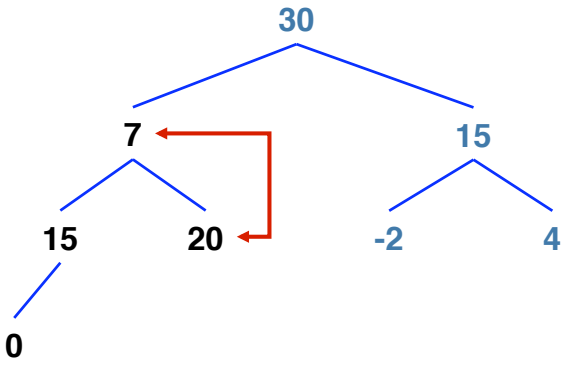
- Troisième niveau *suite*



- Le sous arbre modifié est-il un tas ?
  - **non !**

Construction d'un tas : exemple

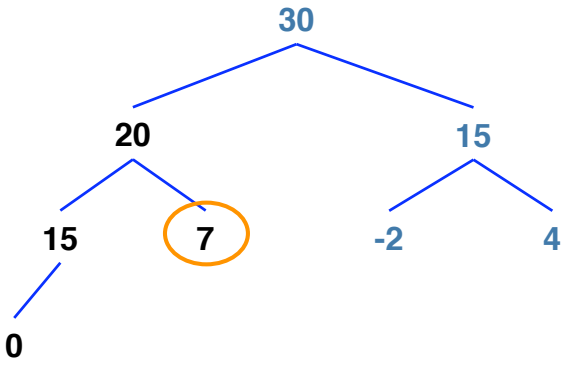
- Troisième niveau *suite*



- On permute 7 et 20

Construction d'un tas : exemple

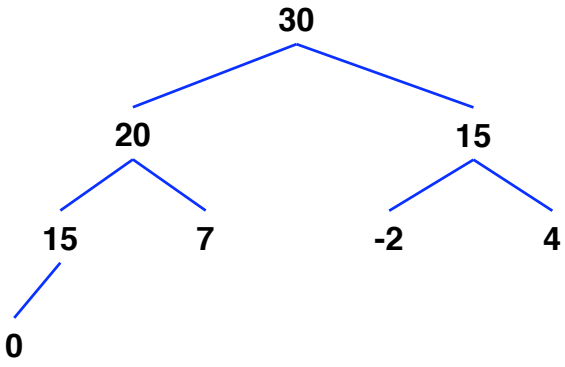
- Troisième niveau *suite*



- Le sous arbre modifié est-il un tas ?
  - **oui !**

Construction d'un tas : exemple

- Troisième niveau *suite*



- L'arbre est un tas
  - **C'est terminé !**

Vers l'algorithme de **construiretas**

- On appelle arbre i (ou sous-arbre i)
  - l'arbre (ou sous-arbre) dont la racine est à l'indice i dans le vecteur V

- 2 parties :
  1. parcours des sous-arbres
  2. pour chaque sous-arbre sélectionné, le transformer pour qu'il vérifie la propriété du tas, sachant que tous ses sous-arbres vérifient cette propriété.

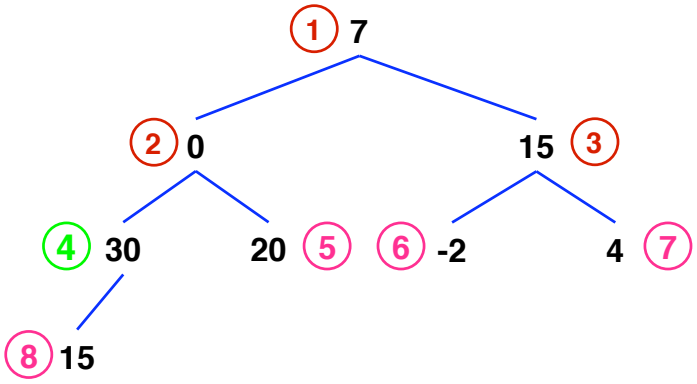
Vers l'algorithme de **construiretas**

- Supposons connue la procédure **constas**

procédure constas (dr V[i..n] :vecteur) ;
spécification $\{n > 0 ,1 \leq i, tas\ (V[i+1..n])\} \rightarrow \{tas\ (V[i..n])\}$

- qui transforme le sous-arbre i pour qu'il vérifie la propriété du tas
  - sachant que tous les sous-arbres j
    - tels que j $\in$ [i+1..n]
  - vérifient cette propriété

Parcours des sous-arbres



- Chaque feuille vérifie la propriété du tas,
  - **V[(n div 2)+1..n]** ne contient que des feuilles,
- On peut donc affirmer
  - tas(V[(n div 2)+1..n])
- On commencera donc le parcours par le sous-arbre **n div 2**

Algorithme de **construiretas**

procédure construiretas (dr V[1..n] : vecteur) ;
spécification *{n > 0}* ➔ *{tas (V[1..n])}*
    i : entier ;
debproc
    i := n div 2 ; *{tas (V[i+1..n])}*
    tantque i ≥ 1 faire
        constas (V[i..n]) ; *{tas (V[i..n])}*
        i := i - 1 ; *{tas (V[i+1..n])}*
    finfaire ;
    {(i = 0, tas (V[i+1..n])) ➔ {tas (V[1..n])}
finproc ;

Algorithme de **construiretas**

- ou encore, en utilisant une instruction pour

procédure construiretas (dr V[1..n] : vecteur) ;
spécification *{n > 0}* ➔ *{tas (V[1..n])}*
    i : entier ;
debproc
    pour i := n div 2 bas 1 faire
        constas (V[i..n]) ;
    finfaire ;
finproc ;

Vers l'algorithme de **constas**

procédure constas (dr V[i..n] :vecteur) ;
spécification $\{n > 0 ,1 \leq i, tas\ (V[i+1..n])\}$ ➔ $\{tas\ (V[i..n])\}$

- Schéma récursif

➤ 2×i > n ⇨ *l'arbre est une feuille, tas(V[i..n])* ✳

➤ 2×i ≤ n ⇨

filsmax = indice du plus grand des 2 fils ou bien
indice du fils unique

➤➤ V[filsmax] ≤ V[i]
⇨ *(tas (V[i+1..n]) , V[i] ≥ V[filsmax) ⇨ tas (V[i..n])* ✳

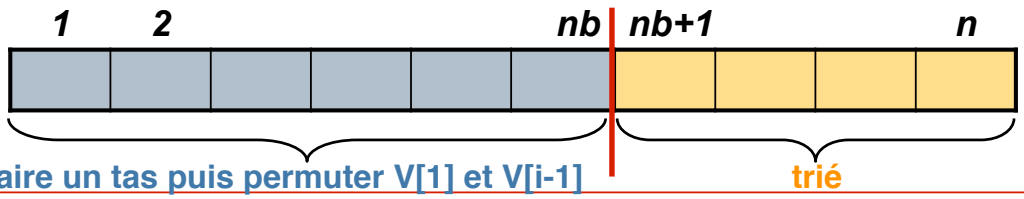
➤➤ V[filsmax] > V[i]
⇨ permut (V[filsmax], V[i]) ; constat(V[filsmax..n]) ;

procédure constas

```
procédure constas (dr V[i..n] : vecteur) ;
spécification {n > 0 , i ≥ 1 , tas (V[ i+1..n])}è {tas (V[i..n])}
    filsmax : entier;
debproc
    filsmax := 2 * i ;          {filsmax = indice du fils gauche, s'il existe}
    si filsmax ≤ n alors        {il y a au moins un sous-arbre}
        si filsmax < n alors    {il y a deux sous-arbres}
          si V[filsmax+1] >V[filsmax] alors
             filsmax := filsmax + 1 ;
          finsi ;               {filsmax = indice du plus grand des 2 fils}
        finsi ;
        si V[filsmax] > V[i] alors
            permut (V[filsmax], V[i]) ;
            constas (V[filsmax..n]) ;
        finsi ;
    finsi ;
finproc ;
```


Vers le tri par tas

- Rappel de l'analogie avec tribulle
  - en construisant un tas sur V[1..n] on fait remonter max(V[1..n]) à la place V[1]
  - on peut alors permuter V[1] et V[n] et le plus grand est alors à sa place
  - à l'étape i
    - V[nb+1..n] est trié
    - construire un tas sur V[1..nb]
    - permuter V[1] et V[nb]
      - alors V[nb..n] est trié



Vers le tri par tas

- Schéma itératif *(initialisation & cas)*

Hypothèse trié (V[nb+1..n]),
V[nb+1..n] ≥ V[1..nb],
tas (V[2..nb])

- nb = 1 ⇨ *trié (V[2..n]), V[2..n] ≥ V[1]*
  ⇨ *trié (V[1..n])* ✳
- nb > 1 ⇨
  constat (V[1..nb]);
  ⇨ *tas (V[1..nb]), V[1] ≥ V[1..nb]*
  permut (V[1], V[nb]) ;
  i := i - 1 ; ➡ ***H***

Vers le tri par tas

- Schéma itératif *(itération)*

- nb = 1 ⇨ *trié (V[2..n]), V[2..n] ≥ V[1]*
  ⇨ *trié (V[1..n])* ✳
- nb > 1 ⇨
  constat (V[1..nb]);
  ⇨ *tas (V[1..nb]), V[1] ≥ V[1..nb]*
  permut (V[1], V[nb]) ;
  i := i - 1 ; ➡ ***H***

Itération tantque (nb > 1) faire …

Vers le tri par tas

Schéma itératif *(initialisation)*

Hypothèse trié (V[nb+1..n]),
V[nb+1..n] ≥ V[1..nb],
tas (V[2..nb])

Initialisation construiretas(V[1..n]) ;
permut (V[1], V[n]) ;
nb := n-1 ; ➡ ***H***

Vers le tri par tas

Schéma itératif

Hypothèse trié (V[nb+1..n]),
V[nb+1..n] ≥ V[1..nb],
tas (V[2..nb])

- nb = 1 ⇨ *trié (V[2..n]), V[2..n] ≥ V[1] ⇨ trié (V[1..n])* ✳
- nb > 1 ⇨

constat (V[1..nb])
⇨ *tas (V[1..nb]), V[1] ≥ V[1..nb]*
permut (V[1], V[nb]) ;
i := i - 1 ; ➡ ***H***

Itération tantque (nb > 1) faire …

Initialisation construiretas(V[1..n]) ;
permut (V[1], V[n]) ;
nb := n-1 ; ➡ ***H***

procédure tritas

procédure tritas (dr V[1..n] : vecteur) ;
spécification *{n > 1}* ➔ *{V[1..n] trié}*
nb : entier;
debproc
construiretas (V[1..n]) ;
permut (V[1], V[n]) ;
nb:=n-1 ; *{trié (V[nb+1..n]), V[nb+1..n] ≥ V[1..nb], tas (V[2..nb])}*
tantque nb > 1 faire
constas (V[1..nb]) ; *{tas (V[1..nb])}*
permut (V[1], V[nb]) ; *{trié (V[nb..n]) ,V[nb..n] ≥ V[1..nb-1]}*
nb:=nb-1 ; *{trié (V[nb+1.. n]), V[nb+1..n] ≥ V[1..nb], tas (V[2..nb])}*
finfaire ;
{nb=1, trié (V[nb+1..n]), V[nb+1..n] ≥ V[1..nb])
⇒ (trié (V[2..n]) , V[2..n] ≥ V[1]) ⇒ trié (V[1..n])}
finproc ;

procédure tritas

- **pour** et explicitation de **construiretas**

procédure tritas (dr V[1..n] : vecteur) ;
spécification *{n > 1}* ➔ *{V[1..n]trié}*
nb : entier ;
debproc
pour i := n div 2 bas 1 faire
constas (V[i..n]) ;
finfaire ;
permut (V[1], V[n]) ;
pour nb := n - 1 bas 2 faire
constas (V[1..nb]) ;
permut (V[1], V[nb]) ;
finfaire ;
finproc ;

Conclusion

- Tri très performant
  - équivalent au tri par segmentation (Quicksort)
  - coût de l'ordre de n $\log_2 n$
- Rappel sur le tribulle
  - coût de l'ordre de n^2

Visualisation des performances