université
PARIS-SACLAY

Algorithmique II
DIU - EIL

Christine Froidevaux

chris@lri.fr

Juin 2019

UNIVERSITÉ PARIS SUD

PLAN

- **Terminaison**
- Correction
- Méthodes algorithmiques
  - Diviser pour régner
  - Algorithmes gloutons
  - Programmation dynamique
- Tris

Terminaison

But : s'assurer que le programme / l'algorithme termine au sens qu'il n'y a pas de boucle infinie.
Cette étude est souvent liée au calcul de la complexité en temps.

Algorithme itératif avec boucle for :

Nombre fini de passages dans la boucle : il suffit de vérifier que le corps de la boucle contient un nombre fini d'instructions, sachant que le corps d'une boucle peut être une boucle…

Ex 1 : fonction2 qui écrit les carrés des entiers (cours 1)

Ex2 : tri sélection (cours 1)

Ex3 : algorithme naïf puissance(a,n)

!! Attention Ne pas modifier la variable du **for** dans le corps de la boucle (selon les langages de programmation cela peut avoir de curieux effets) **!!**

Terminaison

Algorithme itératif avec boucle while :

La condition d'arrêt porte sur la comparaison de variables entières : **variant de boucle**

Propriété fondamentale : Toute suite entière à valeurs positives strictement décroissante ne peut prendre qu'un nombre fini de valeurs.

➔ Il s'agit donc d'identifier une suite de valeurs entières liée à la condition de continuation de la boucle.

Ex du tri insertion :

```
while (j >=0 and t[j]>v):
        t[j+1] = t[j]
        j = j -1
```

Variant de boucle : j.

La suite des valeurs prises par j décroit strictement et est bornée par 0.

Cas particuliers…

Ex1 : La condition d'arrêt porte sur des éléments d'une suite réelle dont on sait qu'elle va dépasser à partir d'un certain rang le seuil de condition indiqué :

```
Def serieHarmBornee():
    S=0
    n= 1
    while (S<=2):
        S=S+(1/n)
        n=n+1
    return(S)
```

Cas particuliers…

Ex2 : Suite de Syracuse, définie par son premier terme u_0 (entier >0) et par une relation de récurrence selon que u_n est pair ou non.

Si $u_0 = 2$ on arrive tout de suite à 1, mais on ne peut que conjecturer (Collatz, 1932) que quel que soit le u0 de départ, on arrive à 1 en un temps fini.

Cas particuliers…

Ex3 : ***Multiplication de deux entiers naturels a et b***

Cpt := 1 ; produit := a ;

Tant que cpt ≤ b faire produit := produit + a

fintantque

Terminaison ? Comment corriger ? Est-ce correct ?

Cas particuliers…

Ex4 : Boucle qui porte sur une **condition d'égalité entre flottants**

```
Def diff(n,p):
    print('n = ',n)
    print('p = ',p)
    while n!=p:
        n=n-1.0
        print('n = ', n)
    print('fin')
```

#diff(10,3) finit rapidement

#appeler diff(10.2,3.2) engendre une boucle infinie..

Terminaison de programmes récursifs

- On utilise aussi ici la propriété fondamentale : Toute suite entière à valeurs positives strictement décroissante ne peut prendre qu'un nombre fini de valeurs ➔ on examine la suite des valeurs prises par les arguments de l'algorithme récursif.

Ex1 : factorielle(n).

Soit u_p la suite des arguments successifs de la fonction.

$u_0 = n$ et $u_{p+1} = u_p - 1$

La suite est une suite d'entiers strictement décroissante. Elle ne prend donc qu'un nombre fini de valeurs.

PLAN

- Terminaison
- **Correction**
- Méthodes algorithmiques
  - Diviser pour régner
  - Algorithmes gloutons
  - Programmation dynamique
- Tris

Correction : invariant de boucle

1) Un ***invariant de boucle*** est une propriété (ou un ensemble de propriétés) qui relie les variables de l'algorithme et qui ne change pas tout au long de la boucle. C'est donc une propriété qui est vraie :

- Avant la première exécution de la boucle
- A chaque tour de la boucle
- Et donc en sortie de boucle.

Il reste alors à vérifier que le but poursuivi par l'algo est bien atteint, en utilisant cette propriété.

2) L'invariant de boucle peut aussi être une quantité numérique qui demeure inchangée après chaque passage dans la boucle.

➔ Difficile de trouver un bon invariant (pas de méthode générale) : repose sur la compréhension de l'algo … et l'intuition.

➔ La propriété peut être démontrée par récurrence

Invariant de boucle : exemples

Multiplication de deux entiers a et b :

```
def mult(a,b):
    cpt=0
    produit=0
    while cpt < b:
        produit= produit + a
        cpt = cpt+1
    return produit
```

Variant de boucle : b-cpt

Invariant de boucle : à la fin de la k-ième itération …

Invariant de boucle : exemple 1

Multiplication de deux entiers a et b :

```
def mult(a,b):
    cpt=0
    produit=0
    while cpt < b:
        produit= produit + a
        cpt = cpt+1
    return produit
```

Invariant de boucle : à la fin de la k-ième itération **produit = k * a**

Invariant de boucle : exemple 2

Division euclidienne

Input : deux entiers $a \geq 0$, $b > 0$

Output : deux entiers q et r tels que $a = b*q + r$, $0 \leq r < b$

```
def divEuclid(a,b):
#retourne le quotient de a par b dans q et le reste dans a
    assert b >0 # le programme s'arrête si la précondition n'est pas vérifiée
    q=0
    while a >= b:
        a=a-b
        q=q+1
    return (q,a)
# divEuclid(17,4)
```

Terminaison et correction

Terminaison

Variant de boucle : **a** ; a est un entier positif ou nul qui décroit strictement (car b>0) jusqu'à devenir plus petit que b (b inchangé). La boucle while s'arrête donc.

Correction : ***Invariant*** de boucle « expr = **q*b + a** » a toujours la même valeur à chaque passage n°k dans la boucle.

k=0 : $q_0=0$, $a=a_0$, $expr_0 = a_0$, $b_0 = b$

Hyp : au k-ième passage : $q_k * b_k + a_k = a_0$

Au (k+1-ième) passage : $a_{k+1} = a_k - b_k$; $q_{k+1} = q_k+1$; $b_{k+1} = b_k$

$q_{k+1} * b_{k+1} + a_{k+1} = (q_k+1) * b_k + (a_k - b_k) = q_k * b_k + a_k = a_0$

A la sortie du while, après p itérations : $a_p < b_p$ et on a : $a_0 = q_p * b_p + a_p$

Avec $b_p = b$; a_p est le reste de la division euclidienne de a_0 par b, et q_p son quotient.

Recherche dichotomique

```
def dichotomie(t,v):
#renvoie l'indice d'une occurrence de v dans t si v est présent et sinon retourne false
    g = 0
    d = len(t)-1
    while g <= d:
# quand g = d, il reste une case, il faut continuer pour voir si v se trouve dans la case !
        mil = (g+d) // 2
        if t[mil] == v:
            return mil
        elif t[mil] < v:
            g = mil+1 # Ici le +1 évite que g reste le même (sinon boucle infinie possible)
        else:
            d = mil-1 # Ici le -1 évite que d reste le même (sinon boucle infinie possible)
    return False
```

Tests : cas où l'élément n'est pas présent, est en 1ere place, est en dernière place, est en plein milieu

Recherche dichotomique : complexité

Soit n = len(t). On compte le nombre de comparaison d'un élément de tableau avec v (== et <) dans le pire des cas.

Il y en 2 par passages dans la boucle while. On maximise ce nombre en considérant que v ne figure pas dans t (on ne sort pas de la boucle while par un return mais parce que g > d).

On divise en 2 à chaque fois l'intervalle sur lequel on recherche v

Par récurrence, montrons qu'après k itérations de la boucle while :

$$\mathbf{d-g < (n/2^k)}$$

- Rappel : si $2^k \leq n < 2^{k+1}$, alors $k = \lfloor \log_2 n \rfloor$

Recherche dichotomique : complexité

Au départ : d-g = n-1 et k = 0. OK

Hyp : $d-g < (n/2^k)$ et $g \leq d$

A la k+1-ème itération : g et d deviennent g' et d' tels que :

Soit $g' = \lfloor (g+d)/2 \rfloor + 1$ et d'=d, soit $d' = \lfloor (g+d)/2 \rfloor - 1$ et g'= g

Cas 1 : $d'-g' = d - (\lfloor (g+d)/2 \rfloor + 1) \leq d - ((g+d)/2) = (d - g)/2$

$<_{\text{hyp rec}} (n/2^k)/2 = (n/2^{k+1})$

Cas 2 : $d'-g' = \lfloor (g+d)/2 \rfloor - 1 - g \leq ((g+d)/2) - g = (d-g)/2$

$<_{\text{hyp rec}} (n/2^k)/2 = (n/2^{k+1})$

Après k itérations, on a **$d-g < (n/2^k)$**

Si on fait p itérations avec p tel que $n<2^p$, alors d-g <1, i.e. d=g et on fait alors au plus une dernière itération.

Complexité en $O(\log_2 n)$

Recherche dichotomique : terminaison

- **Terminaison** :

L'expression **d-g** est un ***variant de boucle***. Soit d_k, g_k, les valeurs de d et g au k-ième passage dans la boucle.

On a : $\mathbf{d_k - g_k < (n/2^k)}$, donc $\mathbf{d_{k+1} - g_{k+1} < d_k - g_k}$

On a donc une suite d'entiers positifs ou nuls décroissante strictement, elle prend donc un nombre fini de valeurs et se termine lorsque $d_k - g_k = 0$;

Recherche dichotomique : correction

- **Correction** :

Le tableau étant trié en ordre croissant, on a ***l'invariant de boucle*** :

« Si v est présent dans t, il figure entre les indices g et d inclus, $0 \leq g \leq d \leq n-1$ »

Preuve en raisonnant sur le nombre k d'itérations effectuées.

k = 0 : g = 0, d= n-1 ; si v est dans t, v est à un indice entre 0 et n-1

Hyp : On suppose que v figure dans $t[g_k..d_k]$. On fait une itération.

A la fin de la (k+1)ième itération (on n'est pas sorti de la boucle par un return) :

$mil = \lfloor (g_k+d_k)/2 \rfloor$;

si v> t[mil], alors $v \in t[g_{k+1}..d_{k+1}]$ avec g_{k+1}= mil+1, $d_{k+1} = d_k$, car tableau trié

Si v < t[mil], alors $v \in t[g_{k+1}..d_{k+1}]$ avec d_{k+1}= mil-1 et $g_{k+1}=g_k$, car tableau trié.

Récurrence établie.

A la sortie de la boucle pour un certain k_0, on a :

$v \in t[g_{k0}..d_{k0}]$, donc on n'a pas $g_{k0}>d_{k0}$; on est sorti par un return, avec v = t[mil]

Tri insertion

```
def TriInsertion(t):
    for i in range (1, len(t)):
        v = t[i]
        j = i -1
     while (j >=0 and t[j]>v):
        t[j+1] = t[j]
        j = j -1
     t[j+1] = v
```

Exemple :

Tri du tableau t : [4,2,5,1,3] ; len(t) = 5

En italique : cartes non encore distribuées

0	1	2	3	4	indices de t
4	**2**	*5*	*1*	*3*	**i = 1, v = 2**
4	4	*5*	*1*	*3*	**j = 0**
2	4	*5*	*1*	*3*	j = -1
2	4	**5**	*1*	*3*	**i = 2, v = 5**
2	4	5	*1*	*3*	**j = 1**
2	4	**5**	*1*	*3*	j = 1
2	4	5	**1**	*3*	**i = 3, v = 1**
2	4	5	**5**	*3*	**j = 2**
2	4	**4**	5	*3*	**j = 1**
2	**2**	4	5	*3*	**j = 0**
1	2	4	5	*3*	j = -1
1	2	4	5	**3**	**i = 4, v = 3**
1	2	4	5	**5**	**j = 3**
1	2	4	**4**	5	**j = 2**
1	2	4	4	5	**j = 1**
1	2	**3**	4	5	j = 1

Correction du tri par insertion

- **Terminaison** : len(t) = n

On fait n-1 passages dans la boucle for ; on fait au plus i passages dans la boucle while, et i ≤n-1. On exécute un nombre fini d'instructions.

- **Correction** : on utilise un invariant de la boucle for : soient a_0, a_1, …, a_{n-1} les éléments rangés dans le tableau t initial.

P(i) : « Les éléments t[0], t[1], …, t[i] sont triés » : t[0] ≤ t[1] ≤ … ≤ t[i] (pour 0 ≤ i ≤n-1)

- P(0) vraie ; P est vraie avant d'entrer dans la boucle
- Hyp : P vraie avant le ième passage dans la boucle for : on a P(i-1)
- Au ième passage, on distingue plusieurs cas sur v = t[i] :

1) v ≥ t[i-1] : on ne rentre pas dans la boucle, par hyp de récurrence, tous les éléments sont triés jusqu'à i-1 et donc maintenant jusqu'à i :

   t[0] ≤ t[1] ≤ … ≤ t[i]= v

Correction du tri par insertion

Hyp (suite) P(i-1) vraie : « Les éléments t[0], t[1], …, t[i] sont triés » : $a_0 = t[0] \leq a1 = t[1] \leq \ldots \leq a_{i-1} = t[i-1]$

2) v < t[i-1] : on entre dans la boucle while ; à sa sortie

- **Si j ≥ 0** : t[j] = $a_j \leq v$; au tour précédent dans le while, on avait $a_{j+1} > v$ et on avait recopié vers la droite t[j+1] et décalé les autres éléments à droite jusqu'à l'indice i : on a donc dans t les éléments $a_0, a_1, \ldots, a_j, a_{j+1}, a_{j+1}, a_{j+2}, \ldots, a_{i-1}$. Par hypothèse de récurrence :

  $a_0 \leq a_1 \leq \ldots, a_j \leq a_{j+1} \leq a_{j+1} \leq a_{j+2} \leq \ldots \leq a_{i-1}$.

  L'affectation t[j+1] = v range les éléments dans l'ordre :

  $a_0, a_1, \ldots, a_j, v, a_{j+1}, a_{j+2}, \ldots, a_{i-1}$, ce qui est l'ordre croissant.

- Si j < 0 (en fait j= -1) : le tableau t contient les éléments $a_0, a_0, a_1, \ldots, a_j, a_{j+1}, a_{j+2}, \ldots, a_{i-1}$. L'affectation t[j+1] = v range v en t[0] et tous les éléments sont en ordre croissant : $v \leq a_0 \leq a_1 \leq \ldots, a_j \leq a_{j+1} \leq a_{j+2} \leq \ldots \leq a_{i-1}$, avec a_{i-1} en t[i].

La propriété P est vraie à la fin du ième passage dans la boucle for : P(i) est un invariant de boucle. Or l'lgorithme se termine à la (n-1) itération et à la fin les éléments t[0], t[1], …, t[n-1] sont triés. cqfd

Correction : cas récursif

Méthode : on montre généralement par récurrence que l'algo récursif fait bien ce qu'il doit faire. Selon le type de récursivité, on aura une récurrence simple ou forte….

Ex1 : factorielle(n) calcule bien n!

Rem : la terminaison et la correction se montrent souvent ensemble

Ex2 :
```
def puissance_rapide(x,n):
    if n==0:
        return 1
    else:
        r = puissance_rapide(x, n//2)
        if n%2==0: # n pair
            return(r*r)
        else:
            return(x*r*r)
```

Exponentiation rapide

On montre par une ***récurrence forte*** que la propriété suivante est vraie :

P(n) : « puissance_rapide(x,n) termine et renvoie la valeur x^n »

P(0): vérifiée car on retourne 1 et x^0 = 1 (on admet que 0^0 = 1)

Hyp : on suppose P(k) vérifiée pour $0 \leq k < n$.
Montrons P(n) :

puissance_rapide(x,n) appelle puissance_rapide(x, $\lfloor n/2 \rfloor$) et stocke le résultat dans r. Soit $p = \lfloor n/2 \rfloor$. Comme $n>0$, $p < n$, et par hyp de récurrence, l'appel de puissance_rapide(x, p) termine et renvoie x^p.

Si n pair, n = 2p ; x^n = x^p * x^p, cad, r *r

Si n impair, n = 2p+1 ; x^n = x^p * x^p * x, cad, r *r *x

P(n) est vérifiée.

PLAN

- Terminaison
- Correction
- Méthodes algorithmiques
  - **Diviser pour régner**
  - Algorithmes gloutons
  - Programmation dynamique
- Tris

Méthodes algorithmiques

Différentes méthodes de concevoir des algorithmes :

- **Algorithme glouton**

C'est un algorithme qui produit une solution pas à pas en faisant à chaque étape un choix qui maximise un critère local. Il existe une notion formelle (matroïdes) qui permet d'établir l'optimalité ou la no optimalité de l'algorithme

Exs : pb du sac à dos, du rendu de monnaie.

- **Programmation dynamique**

On décompose un pb en sous-pbs tels que la solution optimale du pb principal s'obtientt à partir des solutions optmales des sous-pbs

Exs : nombre de Fibonacci, alignement de séquences, rendu de monnaie

Diviser pour régner

Divide and conquer

Principe :

- ***Diviser*** : on décompose un problème de taille n en plusieurs sous-problèmes de même nature que le problème initial, ayant des tailles strictement plus petites que celle du problème initial.
- ***Régner*** : on résout chaque sous-problème ; il faut connaître des tailles pour lesquelles le problème peut être résolu sans division (problèmes élémentaires).
- ***Combiner*** : puis on rassemble les solutions des sous-problèmes pour obtenir une solution globale.

Remarque : toutes les étapes n'existent pas forcément

Diviser pour régner

Divide and conquer

1) **Recherche dichotomique** : étape de décomposition seulement

2) **Tri fusion** :

(i) On découpe le tableau de taille n à trier en deux sous-tableaux de taille n/2 ; (ii) on trie (récursivement) chacun des 2 sous-tableaux; (iii) on fusionne les deux sous-tableaux triés (fonction fusion)

➔ 2 appels récursifs suivis d'un traitement

3) **Tri rapide :**

(i) On choisit un élément du tableau considéré comme pivot et on le place dans le tableau de sorte que à gauche du pivot tous les éléments lui sont inférieurs, et à sa droite, tous les éléments lui sont supérieurs (ii) o appelle récursivement le tri rapide sur le sous-tableau de gauche et sur le sous-tableau de droite

➔ un traitement suivi de 2 appels récursifs

Tri fusion

Canevas de tri_fusion du tableau t entre les indices g et d :
si d > g alors
    m = (g + d) / 2
    tri_fusion(T, g, m)
    tri_fusion(T, m + 1, d)
    fusionner(T,g,m,d)

Exo : trier t = [8,4,10,12, 36, 7, 2, 9]. Dessiner au tableau l'arbre des appels récursifs.

- C(n) (resp. f(n)) la complexité en nombre de comparaisons de trifusion (resp. fusion): C(n) = 2 C(n/2) + f(n) si n>1 ; C(1) = 0

Pire des cas : f(n) = n-1 et **C(n) = O(n log n)**

Intuition : L'arbre des appels récursifs est de profondeur $\log_2 n$, et si on considère tous les sous-tableaux d'un même niveau, on voit qu'on a fait à peu près n comparaisons pour l'ensemble des fusions, d'où O(n log n) (cf l'exemple).

PLAN

- Terminaison
- Correction
- Méthodes algorithmiques
  - Diviser pour régner
  - Algorithmes gloutons
  - Programmation dynamique
- **Tris**

TRIS

- Soit L une liste de n éléments. A chaque élément est associée une clé telle que les clés soient toutes comparables entre elles. On va trier les éléments selon leurs clés. En pratique, on confond les éléments avec les clés.
- On stocke L dans un tableau et on trie sur place en faisant des échanges ou des déplacements, en fonction des résultats des comparaisons des clés.

- Complexité en temps :
  - Nombre de déplacements d'éléments,
  - Nombre de comparaisons entre clés.

- Résultat **d'optimalité** pour le nombre de comparaisons : il n'y a pas d'algorithme qui opère par comparaisons et transferts, ***sans autre information sur les clés***, qui dans le pire des cas soit de complexité en nombre de comparaisons d'ordre strictement plus petit que $\Theta(n \log n)$.

TRIS

- **Méthodes par sélection** :

Choisir un élément, le mettre à sa place définitive et continuer

Exs : tri sélection, tri bulles, tri rapide (placement du pivot)

- **Méthodes par insertion** :

C'est la méthode du joueur de cartes. Etant donné une partie triée des clés, mettre une nouvelle clé à sa place dans cette partie.

Exs: tri insertion séquentielle, tri insertion dichotomique

- **Méthode diviser pour régner** : tri fusion, tri rapide

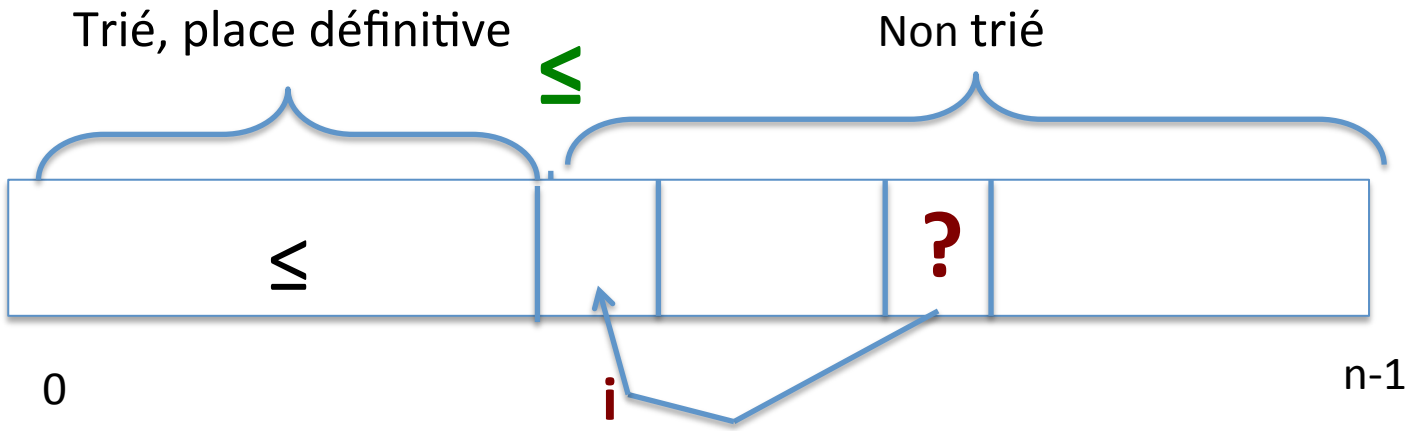
- **Choix d'un tri** :

Plus ou moins grande complexité en temps (comparaisons ? Transferts d'éléments ?), tris « simples », stables (l'ordre des éléments égaux est conservé), progressifs (à l'étape i on a le début du tableau trié et on peut commencer un autre traitement)

- Choisir un tri en fonction de la configuration du tableau : déjà trié, trié ordre inverse, presque trié…

Principe du tri par sélection

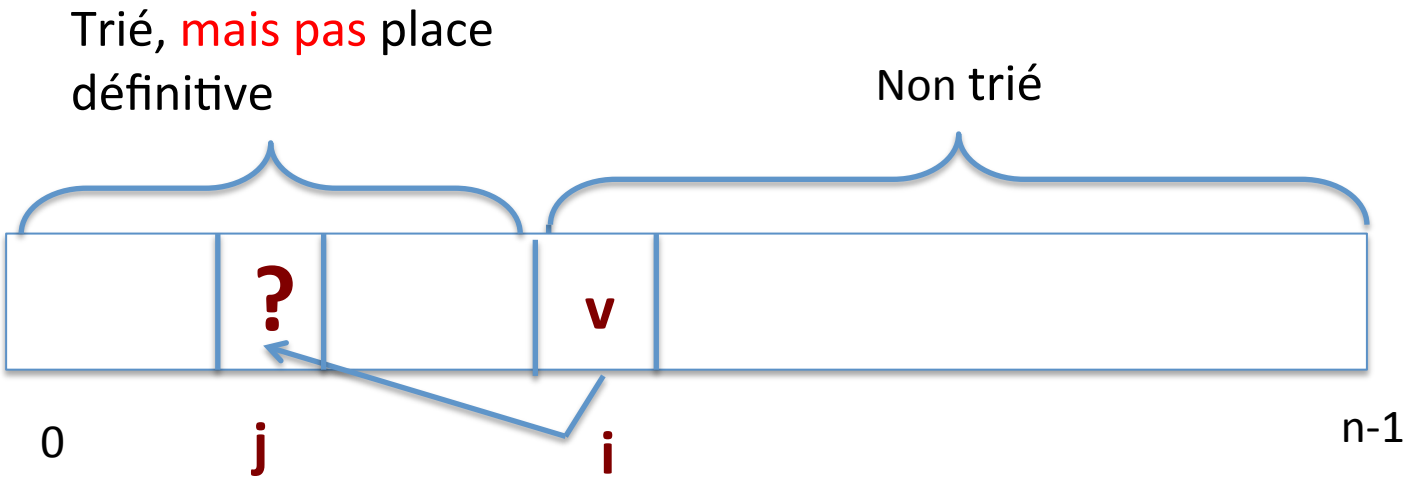
- Choisir ***le plus petit élément de la partie non encore triée ;*** le placer en place i (c'est sa place définitive) et continuer sur le reste du tableau :



Tri insertion

Principe : méthode du joueur de cartes (ou du professeur ramassant et triant les copies)

A la ième étape, on insère l'élément t[i] à sa place j parmi les éléments à sa gauche qui sont triés entre eux, mais pas forcément à leur place définitive. Pour cela on décale successivement d'une place vers la droite, les éléments déjà triés, de la place i-1 à la place j.



Faire « tourner à la main » sur un exemple

Complexité des tris

On s'intéresse aux comparaisons entre éléments de tableau, et aux déplacements des éléments du tableau

Algorithme	Au pire comparaisons	En moyenne comparaisons	Au pire déplacements	En moyenne déplacements
Tri sélection	O(n^2)	O(n^2)	O(n)	O(n)
Tri insertion (séquentielle)	O(n^2)	O(n^2)	O(n^2)	O(n^2)
Tri insertion dichotomique	O(n log n)	O(n log n)	O(n^2)	O(n^2)
Tri bulles	O(n^2)	O(n^2)	O(n^2)	O(n^2)
Tir rapide	O(n^2)	O(n log n)	O(n^2)	O(n log n)
Tri fusion	O(n log n)	O(n log n)	O(n log n)	O(n log n)

Corrigé TD2

Exercice : recherche du minimum et du maximum

On se propose de chercher parmi n éléments d'une liste représentée par un tableau, l'indice d'un plus grand élément et l'indice d'un plus petit élément. Les éléments sont supposés comparables pour une certaine relation d'ordre

On utilise une méthode récursive qui consiste à diviser la liste en deux sous-listes, à calculer récursivement les indices des minimum et maximum sur chacune des sous-listes, puis à comparer les éléments correspondants pour conclure.

1. Donnez un algorithme MMR (MinMaxRec) résolvant le problème par cette méthode. On précisera la structure de données choisie et on donnera la spécification de l'algorithme.

Exercice : recherche récursive du minimum et du maximum dans un tableau (suite)

2) Faites tourner votre algorithme sur la liste des 8 éléments suivants : 10, 20, 5, 8, 1, 30, 7 et 5.

Vous donnerez l'arbre des appels récursifs, en numérotant dans l'ordre les appels effectués.

Indiquez à chaque étape les valeurs des variables et le nombre de comparaisons qu'effectue l'algorithme.

3) Etudiez la complexité en temps de votre algorithme, en nombre de comparaisons entre éléments. On donnera une équation de récurrence en fonction du nombre d'éléments n, et on la résoudra pour $n = 2^k$.

4) Proposez des jeux de tests.

Algorithme récursif min et max

```
def MMR(t,g,d):
#retourne deux entiers min et max qui sont les indices de l'élément minimal et de l'élément maximal dans le tableau t
    if g==d:
        return (g,d)
    elif d ==g+1:
        if t[d] < t[g]:
            return (d,g)
        else:
            return (g,d)
    else:  # on considère une partie de tableau de plus de 2 éléments
        v1,w1 =MMR(t,g, (g+d)//2)
        v2,w2 =MMR(t,((g+d)//2)+1,d)
        if t[v1] <t[v2]:
            x=v1
        else:
            x=v2
        if t[w1] >t[w2]:
            y=w1
        else:
            y=w2
    return(x,y)
#premier appel : MMR(t,0, len(t)-1)
```

PGCD : terminaison et correction

```
def PGCD(a,b):
#a et b sont entiers strictement positifs
    x=a
    y=b
    while x!=y:
        if x > y:
            x=x-y
            print(‘x = ‘, x)
        else:
            y=y-x
    return x
```

- Variant de boucle : max(x,y) est un entier >0 au départ (car a et b sont >0) e tà la fin de chaque itération
- Invariant de boucle : PGCD(x,y)

Résulte de la propriété : PGCD(a,a) = a et PGCD(a,b) = PGCD(a-b, a) si a>b et = PGCD(a, b-a) sinon

Mystere (exo3)

```
Def mystere(n) :
    r = 2
    i = 0
    while i < n :
        r = r * r
        i = i + 1
    return r
```

1. Que pensez-vous que calcule cette fonction ?
2. Donner une preuve formelle de terminaison
3. Proposer un invariant de boucle et prouver la correction de votre fonction par rapport au résultat de la question 1.

Mystère : corrigé

1) On conjecture que Mystère(n) calcule $2^{2\hat{}n}$

2) Terminaison : si n = 0, on n'entre pas dans la boucle, fini. Sinon, on considère la suite n-i.

3) Invariant de boucle : $r_k = 2^{2\hat{}k}$

Cas de base : $r_0 = 2 = 2^{2\hat{}0} = 2^1$

Hyp de récurrence : $r_{k-1} = 2^{2\hat{}(k-1)}$. A la fin de la k-ième itération, $r_k = r_{k-1} * r_{k-1}$.

On a donc : $r_k = (2^{2\hat{}(k-1)}) * (2^{2\hat{}(k-1)}) = 2^{2\hat{}(k-1)) + 2\hat{}(k-1))} = 2^{2\hat{}k}$

Ecriture binaire d'un entier naturel

```
def binaire(n):
    if n==0:
        return'0'
    ch=' '
    while n > 0:
        ch = str(n%2) + ch
        n = n//2
    return ch
```

Complexité (meilleur et pire des cas) : nombre de passages = $\lceil \log_2(n) \rceil + 1 = O(\log(n))$

Variant de boucle : n, n décroit strictement (n = n//2)

Invariant : Inv = $((2^k) * n_k) + m_k$