Rechercher

Rechercher une valeur dans un tableau de nombres
Rechercher **une** lettre dans une chaîne de caractères

Plan

Rechercher dans un tableau 1D
- rechercher une valeur dans un tableau de nombres
- rechercher une lettre dans une chaîne de caractères

Recherche séquentielle
- plusieurs algorithmes possibles
- terminaison, correction, complexités

Profiter de valeurs ordonnées (triées) : la recherche dichotomique
- rechercher une valeur dans un tableau de nombres triés
- un nombre = une clé d'un objet plus complexe

Illustration d'algorithmes de complexité différentes pour résoudre un même problème

Recherche séquentielle

Rechercher une valeur dans un tableau de nombres
Rechercher une lettre dans une chaîne de caractères

Recherche séquentielle dans un tableau

Problèmes : on cherche une valeur v dans un tableau t de taille n
- si elle est présente, on peut vouloir retourner (1) un booléen ou (2) sa position
- si elle est absente, on a un traitement prédéfini dans le cas 2

Analyse

Les données :
- un tableau t de longueur n et contenant des valeurs d'un certain type : entier, flottant, caractères
- la valeur v cherchée, elle est du type des éléments du tableau

La sortie :
- cas 1 : un booléen
- cas 2 : un entier

Principe d'une première solution avec une boucle pour :

```
trouve= Faux
pour  i de 0 à  n-1 faire
      si  t[i] == v  alors
            trouve = Vrai
      fin si
fin pour
... // retourne booléen ou position
```

Recherche séquentielle dans un tableau de nombres

```
//en-tête
fonction appartient(t : tab. d'entiers, n : entier, v : entier) retourne booléen
```

```
fonction appartient(t : tab. d'entiers, n : entier, v : entier) retourne booléen
// retourne vrai si v est un entier présent dans le tableau t, retourne faux sinon
      trouve : booléen = Faux          // la réponse Vrai ou Faux avec une valeur initiale
début
      pour i de 0 à n-1 faire
            si t[i] == v alors
                  trouve = Vrai
            fin si
      fin pour
      retourner trouve
fin fonction
```

```
// appel
declare
      t [5] : tableau d'entiers = [11, 4, 9, 3, 12]
      res : booléen
début
      res = appartient(t, 5, 9)
      afficher (res)              // Vrai
fin
```


Solution avec boucle `pour` : terminaison

Terminaison

- la boucle pour effectue exactement n itérations et l'indice i varie de 0 à n-1
- à chaque itération de i qui varie entre 0 et n-1, on accède à la valeur de t[i]
- cette valeur t[i] existe car t est un tableau de longueur n, d'indices variant entre 0 et n-1

Donc la boucle pour termine.

Cas de la fonction qui retourne le booléen trouve :
- trouve existe et vaut Faux avant la première exécution de la boucle pour
- trouve est *éventuellement* modifiée en Vrai lors de l'exécution de la boucle pour
- trouve existe et a une valeur Faux ou Vrai qui est retournée à l'issue de la boucle pour

Donc la fonction termine en retournant une valeur booléenne.

```
trouve= Faux
pour  i de 0 à  n-1 faire
      si t[i] == v alors
            trouve = Vrai
      fin si
fin pour
... // retourne booléen ou position
```


Solution avec boucle `pour` : correction

Correction sans introduire d'invariant

1. si la valeur v est présente dans le tableau t, il existe un indice j compris entre 0 et n-1 tel que t[j] == v
- la boucle pour effectue exactement n itérations et son indice i varie de 0 à n-1
- quand i == j (0 ≤ i, j < n), t[i] == v donc trouve est modifié et vaut Vrai
- La valeur Vrai de trouve n'est pas modifiée pour i > j et i < n (mais la variable trouve peut l'être)
- donc trouve == Vrai une fois la boucle pour terminée quand v est présent dans t.

2. si la valeur v est *absente* du tableau t, il existe *aucun* indice i compris entre 0 et n-1 tel que t[i] == v
- trouve vaut Faux avant la première itération de la boucle pour
- la boucle pour effectue exactement n itérations et son indice i varie de 0 à n-1
- il existe *aucun* indice i compris entre 0 et n-1 tel que t[i] == v donc la valeur de trouve n'est pas modifiée lors de l'exécution de la boucle pour
- donc trouve == Faux une fois la boucle pour terminée quand v est absent de t.

Dans les deux cas, trouve a bien la valeur booléenne attendue.

```
trouve= Faux
pour  i de 0 à  n-1 faire
      si t[i] == v alors
            trouve = Vrai
      fin si
fin pour
... // retourne booléen ou position
```


Solution avec boucle `pour` : complexité en temps

Rappel du modèle d'analyse de la complexité
- chaque instruction a le même coût
- exécution séquentielle

On veut mesurer le temps d'exécution comme une fonction de la *taille du problème*
- quand le nombre d'éléments est grand, la taille de chaque élément devient négligeable devant la taille totale et ce quelque soit la taille des élément
- la taille du problème est n : le nombre d'éléments du tableau t
- on cherche donc à exprimer le temps T comme une fonction T(n) : complexité en temps

```
1.  trouve = Faux
2.  pour  i de 0 à  n-1 faire
3.        si t[i] == v alors
4.              trouve = Vrai
5.        fin si
6.  fin pour
7.  retourne trouve
```

ligne	*instruction*	*nb*	*temps total*
1	affectation	1	1
2	addition	n+1	n+1
2	affectation	n+1	n+1
2	comparaison	n+1	n+1
3	comparaison	n	n
4	affectations	au plus n	0 ≤ t ≤ n
7	retour	1	1

4 n + 5 ≤ Total ≤ 5 n + 5

Solution avec boucle **pour** : complexité en temps

Complexité en temps
- on cherchait : $T(n) = f(n)$
- on compte : $4n+5 \leq T(n) \leq 5n+5$
- quand n est grand, T(n) est minorée par une fonction linéaire de n
- quand n est grand, T(n) est majorée par une fonction linéaire de n
  - . Quand n est grand, T(n) est donc équivalent à une fonction linéaire de n
  - . La notation de cette équivalence asymptotique : $T(n) = \theta(n)$
- on a le même résultat en comptant seulement la comparaison t[i] == v

On retient que :

l'algorithme de recherche séquentielle d'une valeur dans un tableau de longueur n est d'une complexité en temps linéaire par rapport à n

Est-ce "toujours" vrai ? Est-ce que ce n'est pas une conséquence d'un choix d'écriture de l'algorithme : de la boucle pour ?

Recherche séquentielle : une autre écriture

On cherche une valeur v dans un tableau t de taille n
- si elle est présente, on peut retourner (1) un booléen ou (2) sa position
- si elle est absente, on a un traitement prédéfini dans le cas 2

En effet, on n'a pas besoin de parcourir tout le tableau

boucle pour → boucle tantque

```
i= 0;
trouve= Faux
tantque (i < n) et (trouve == Faux) répéter
    si t[i] == v alors
        trouve = Vrai
    sinon
        i =i+1
    fin si
fin tantque
... // retourne booléen ou position
```


Exemple dans le cas d'une chaîne de caractères

Même principe : une chaîne de caractères est un tableau 1D de caractères

```
fonction appartient(s : tab. de caractères, n : entier, c : caractère) retourne booléen
// pour les chaine de caractères, version avec un seul retourne
// retourne vrai si c est un caractère présent dans la chaîne s, retourne faux sinon
    i : entier  = 0              //itérateur
    trouve : booléen = Faux  // la réponse avec une valeur initiale
début
    tantque (i < n) et (trouve == Faux) répéter
        si s[i] == c alors
            trouve = Vrai
        sinon
            i =i+1
        fin si
    fin tantque
    retourner Faux
fin fonction
```


Recherche séquentielle avec boucle **tantque**

Complexité en temps :
a-t-on amélioré le temps d'exécution ?

```
i= 0;
trouve= Faux
tantque (i < n) et (trouve == Faux) répéter
    si t[i] == v alors
        trouve = Vrai
    sinon
        i =i+1
    fin si
fin tantque
```

- OUI si v est présent en début du tableau, le parcours (la boucle) se termine avant l'itération i==n, et ce indépendamment de n
  - . un "meilleur cas" est obtenu en temps constant par rapport à n

- NON si v est en fin de tableau ou absent
  - . un "pire cas" est obtenu en un temps toujours majoré par k.n

La complexité T(n)
- est majorée par une fonction linéaire de la taille n du tableau
- et peut aussi être égale à une fonction constante (par rapport à n)

La recherche séquentielle est au pire linéaire en n, ce qu'on note T(n) = O(n)

Autres écritures de la recherche séquentielle avec tantque

Le "et" est paresseux
- sa valeur est retournée dès que possible
- le second argument n'est pas évalué si le premier argument est faux
  - a et b == faux "*dès que*" a == faux
- cela garantit aucun accès au delà de la taille du tableau :

```
i = 0
tantque (i < n) et (t[i] != v) répéter  // t[n] n'est jamais évalué !!
        i = i+1
fin tantque
... // maj booléen ou position
```


Autres écritures de la recherche séquentielle avec pour

Dans une fonction, le retourne termine l'appel de la fonction
- peut permettre de quitter une boucle avant la condition d'arrêt
- aussi valable pour les boucles pour
- plusieurs retourne sont nécessaires dans l'algorithme
- attention dans ce cas lors de l'analyse de la terminaison et le la complexité

```
fonction appartient (t : tab. d'entiers, n : entier, v : entier) retourne booléen
    i : entier                      //itérateur
début
    pour i de 0 à n-1 faire         // ne s'exécute pas n fois exactement !!!!!
        si t[i] == v alors
            retourner Vrai
        fin si
    fin pour
    retourner Faux
fin fonction
```


Autres écritures de la recherche séquentielle avec pour

Autre variante avec la réponse trouve et 2 retourne

```
fonction appartient (t : tab. d'entiers, n : entier, v : entier) retourne booléen
    i : entier                      //itérateur
    trouve : booléen = Faux         //réponse avec initialisation si v absent dans t
début
    pour i de 0 à n-1 faire         // ne s'exécute pas n fois exactement !!!!!
        si t[i] == v alors
            trouve = Vrai
            retourner trouve
        fin si
    fin pour
    retourner trouve
fin fonction
```


Recherche séquentielle : exercices

Compléter la version avec le "et"
- pour appartient qui retourne un booléen
- pour position qui retourne la première position de v

Prouver la terminaison des différentes versions de ces recherches
- facile avec pour et deux retourne

Identifier et prouver les invariants de boucle

La propriété suivante est un invariant de l'algorithme avec la variante pour avec retourne qui prouve que la valeur Faux est correctement retournée par trouve.
(P) : au début de chaque itération i, si v est présent dans t alors il est présent dans le sous-tableau t[i, n-1]

Une autre écriture de la recherche séquentielle :
- une version avec sentinelle, linéaire dans le pire cas mais avec un coefficient linéaire (la constante de la complexité cachée dans le O) moins important.

Complexité de la recherche séquentielle

Complexité en temps
- dépend de la longueur du tableau n
- se mesure par le nombre de comparaisons avec la valeur cherchée
  t[i] == v (ou s[i] == c)

Le nombre de comparaisons dépend de la position de v dans t :
- meilleur cas : 1 comparaison
- pire cas : n comparaisons
- dans le cas général : inférieure ou égale à n

La complexité en temps de la recherche séquentielle dans un tableau de n valeurs est au pire linéaire en n : T(n) = O(n)

Recherche dichotomique

Rechercher une valeur dans un tableau de nombres triés

... et ainsi améliorer la complexité de la recherche

Dichotomie = division en 2

Dichotomie :
division d'un concept en deux concepts contraires couvrant l'étendue du concept

Divide-and-conquer ou diviser pour régner
- diviser le problème en des problèmes similaires et de taille moins importante (ou plus simple) et ce de façon répétée (*récursive*) jusqu'à obtenir un problème suffisamment simple dont on connait la solution
- un *principe général* (paradigme) pour définir un algorithme récursif
- un algorithme récursif est un algorithme qui "s'appelle" → objet d'un chapitre suivant

La recherche par dichotomie dans un tableau trié est un algorithme "divide and conquer" où chaque division réduit la recherche à un ensemble de taille moitié, l'autre ensemble n'étant plus considéré.
- ce qui justifie une écriture non récursive "facile" de la recherche dichotomique

Recherche dichotomique : principe

Un algorithme "diviser pour régner" où chaque division réduit la recherche à un ensemble de taille moitié, l'autre ensemble n'étant plus considéré
- diviser :
  . on partage en 2 par la moitié le tableau trié
- régner :
  . on compare la valeur cherchée à la valeur médiane du tableau
  . si besoin, on en déduit la moitié gauche ou droite du tableau qui contient la valeur cherchée
  . on recommence la recherche sur la "bonne moitié"

Terminaison
la taille de l'espace de recherche est diminuée par 2 à chaque itération
- exemple : n=64→32 → 16 → 8 → 4 → 2 → 1
- si n>0 est la taille de l'espace de recherche (nombre de valeurs, longueur du tableau), cette taille évolue comme la suite n, n/2, n/4, ..., $n/2^k$, ..., soit comme une suite géométrique de raison ½ < 1, et de premier terme non nul. Cette suite géométrique converge vers 1.
- Ainsi la dichotomie construit, à terme, un ensemble de 1 élément, un singleton, égal ou non à la valeur cherchée (résultat de la comparaison : étape 1 de la phase "régner").
- Donc la dichotomie termine.

Il faut maintenant assurer que la recherche est correcte.

Dichotomie : exemples

Je cherche la valeur 11 dans t = [1, 3, 4, 5, 6, 10, 11, 12, 13, 14] de longueur 10

n = 10, n÷2 = 5 t=[1, 3, 4, 5, 6, **10**, 11, 12, 13, 14]
t[5]=10 < 11 donc on restreint la recherche à la moitié droite de t :
t[6:9]=[11, 12, 13, 14] = **t1 sous-tableau de t** pour faciliter les notations ensuite

t1 de longueur 4 : n = 4, n÷2 = 2, t1 = [10, 11, **12**, 13]
t1[2] = 12 > 11 donc on on restreint la recherche la moitié gauche de t1 :
t1[0:1] = [10, 11] = **t2 sous-tableau de t**

t2 de longueur 2 : n = 2, n÷2 = 1, t2=[10, **11**]
t2[1] = 11 == 12 donc 11 est présent (et trouvé) dans le tableau t2, donc dans t

Remarque : on a trouvé la valeur cherché après 3 découpages, cad. 3 itérations

Dichotomie : une autre recherche dans le même tableau

Je cherche la valeur 1 dans t = [1, 3, 4, 5, 6, 10, 11, 12, 13, 14]

n = 10, n÷2 = 5 t=[1, 3, 4, 5, 6, 10, 11, 12, 13, 14]
t[5]=10 > 1 donc on restreint la recherche à la moitié gauche de t :
t[0:4]=[1, 2, 3, 5, 6] ici on travaille directement sur une tranche de t : un sous-tableau de t → gestion des indices

n = 5, n÷2 = 2, t=[1, 3, 4, 5, 6]
t[2] = 4 > 1 donc on restreint la recherche à la moitié gauche de t :
t[0:1] = [1, 3]

n = 2, n÷2 = 1, t=[1, 2]
t[1] = 2 > 1 donc on restreint la recherche à la moitié gauche de t :
t[0] = [1]

n = 1, n÷2 = 0, t=[0] est un singleton
t[0] = 1 == 1 donc 1 est présent dans le tableau t

Remarque : on a trouvé cette valeur après 4 découpages ; la recherche séquentielle l'aurait trouvé en 1 itération.

Dichotomie : une dernière recherche dans le même tableau

Je cherche la valeur 13 dans t = [1, 3, 4, 5, 6, 10, 11, 12, 13, 14]

n = 10, n÷2 = 5 t=[1, 3, 4, 5, 6, **10**, 11, 12, 13, 14]
t[5]=10 < 13 donc on restreint la recherche à la moitié droite de t :
t[6:9]=[11, 12, 13, 14]

n = 4, n÷2 = 2, t=[11, 12, **13**, 14]
t[2] = 13 == 13 donc 13 est présent dans le tableau t

Remarques
- on a trouvé cette valeur après 2 découpages ;
- la recherche séquentielle l'aurait trouvé en 9 itérations.
- la valeur 14 serait trouvée en une itération supplémentaire,
soit 3 itérations par dichotomie vs. 10 par recherche séquentielle

Recherche dichotomique : formalisation

Appelons pivot la valeur médiane de t

Pour profiter d'inégalités strictes, supposons que toutes les valeurs de t sont différentes.

Pour une recherche correcte, il faut utiliser le caractère trié des valeurs et expliciter que le choix de la partie gauche (les valeurs inférieures au pivot) ou droites (les valeurs supérieures au pivot) garantit que la valeur cherchée n'est pas dans la partie exclue (inégalités larges si non unicité des valeurs).

Notons g et d les indices gauche et droit du tableau t à chaque itération

Notons m l'indice du pivot : pivot == t[m] avec m = (g+d)÷2

On a : t[i] < pivot si g ≤ i <m
t[i] > pivot si m < i ≤ d

et cette propriété est conservée au fur et à mesure des itérations sur t[g,d] tant qu'on a pas trouvé v ou tant que t[g,d] n'est pas vide

Dichotomie : algorithme itératif

```
fonction dichotomie(v : flottant, t : tableau de flottants, n : entier) retourne booléen
// renvoie Vrai si la valeur v est présente dans le tableau t, et Faux sinon
    present : booleen = Faux
    g : entier = 0              // indice du premier élément du tableau t (à gauche)
    d : entier = n-1            // indice du dernier élément du tableau t (à droite)
début
    tantque (present == Faux) et (g <= d) faire  // si g > d alors t[g,d] est vide
        m = (g+d) ÷ 2                            // indice du pivot ≥ 0
        si t[m] == v alors
            present = Vrai                       // on a trouvé : le pivot est la valeur cherchée
        sinon si v < t[m] alors
            d = m-1                              // on cherchera dans la partie gauche de t
        sinon
            g = m+1                              // on cherchera dans la partie gauche de t
        fin si
    fin tantque
    retourne present                             // ou m si on cherche la position de v
fin fonction
```


Dichotomie : algorithme itératif

Remarques :

- la boucle tantque n'introduit aucun indice
- present est mis à jour si le pivot est la valeur cherchée
- les moitiés gauche ou droite du tableau sont choisies en modifiant, respectivement :
  . d = m-1 : indice "à gauche" du pivot
  . g = m+1 : indice "à droite" du pivot
- si g==d, m=(g+d)÷2 donne m==g==d et

```
present = Faux ; g=0; d=n-1
tantque (present == Faux) et (g <= d) faire
    m = (g+d) ÷ 2
    si t[m] == v alors
        present = Vrai
    sinon si v < t[m] alors
        d = m-1
    sinon
        g = m+1
    fin si
fin tantque
```

d=m-1 == g-1 < g donc t[g,d] est vide
g=m+1 == d+1 > d donc t[g,d] est vide

Recherche dichotomique : correction

```
present = Faux ; g=0; d=n-1
tantque (present == Faux) et (g <= d) faire
    m = (g+d) ÷ 2
    si t[m] == v alors
        present = Vrai
    sinon si v < t[m] alors
        d = m-1
    sinon
        g = m+1
    fin si
fin tantque
```

Invariant (P) : au début de l'itération,
si v est présent dans t alors il est dans t[g,d]

- Initialisation : au début de la première itération, t[g,d] = t[0,n-1] = t "tout entier". (P) est vraie.
- Conservation : au début de l'itération, (P) assure que si v est présent, il existe j tel que v==t[j] avec g ≤ j ≤ d.

. si j==(g+d)÷2==m alors ni g ni d sont modifiés par l'itération et (P) reste vraie au début de l'itération suivante.

. si v < t[m] alors d=m-1. On sait que t[i] < pivot pour g ≤ i <m et t[i] > pivot sinon.
Donc v présent est dans t[g,m-1]= t[g,d]. (P) reste vraie au début de l'itération suivante.

. si v> t[m] alors g=m+1. On sait que t[i] > pivot pour m < i ≤ d et t[i] < pivot sinon.
Donc v présent est dans t[m+1,d]= t[g,d]. (P) reste vraie au début de l'itération suivante.

- Terminaison : La boucle tantque se termine dans 2 cas.

. cas a : present==Vrai alors l'itération précédente a trouvé m tel que v==t[m] sans modifier g et d.
A la terminaison, v, présent dans t[g,d], est trouvé en t[m] et g ≤ m ≤ d donc (P).

. cas b : si present==Faux et g > d. g>d donc t[g,d] est vide donc v n'est pas présent dans t (t est vide !).
A la terminaison, on a la contraposée de (P) donc (P).

Contraposée : A => B <=> non B => non A

ici la contraposée de (P) : si v n'est pas présent dans t[g,d] alors v n'est pas présent dans t.

Recherche dichotomique : complexité

La complexité en temps de la recherche dichotomique est au pire logarithmique par rapport au nombre d'éléments du tableau :

$T(n) = O(\log n)$.

```
present = Faux ; g=0; d=n-1
tantque (present == Faux) et (g <= d) faire
    m = (g+d) ÷ 2
    si t[m] == v alors
        present = Vrai
    sinon si v < t[m] alors
        d = m-1
    sinon
        g = m+1
    fin si
fin tantque
```

D'où sort ce logarithme ?

- principe : la taille de la tranche de tableau où la recherche s'effectue diminue d'un facteur 2 à chaque itération :

$n \rightarrow n/2 \rightarrow n/4 \rightarrow .... \rightarrow n/2^k \rightarrow n/2^{k+1} \rightarrow ... \rightarrow 1$ (suite géométrique de raison ½)

- rappels : pour r>0, $r = 2^k \Leftrightarrow k = \log_2(r)$ $\log_2(r) = \ln(r) / \ln(2)$
- intuitivement : soit N tel que $2^{N-1} < n \leq 2^N$, alors $n/2^N \leq 1$ donc la taille de la tranche de tableau où la recherche s'effectue vaut 1 au bout de N = floor($\log_2$(n)) itérations.

Notation : floor(s) = L n I : partie entière par valeur inférieure

Recherche dichotomique : complexité

La complexité en temps de la recherche dichotomique est au pire logarithmique : T(n) = O(log n).

```
present = Faux ; g=0; d=n-1
tantque (present == Faux) et (g <= d) faire
        m = (g+d) ÷ 2
        si t[m] == v alors
                present = Vrai
        sinon si v < t[m] alors
                d = m-1
        sinon
                g = m+1
        fin si
fin tantque
```

Montrons qu'après k itérations de la boucle, on : $d - g < n / 2^k$.

Par récurrence sur k.

- Pour k=0 : avant la première itération, g=0, d=n-1
$d-g = n-1 < n/2^0$.

- Supposons $d - g < n/2^k$ et $g \leq d$ après l'itération k.
Après l'itération k+1 : m = floor(g+d/2)

rappel : x -1 < floor(x) ≤ x

. si g=m+1, $d - [floor(g+d/2)+1] \leq d - (g+d)/2 = (d - g) /2 < n /(2^k \times 2) = n/2^{k+1}$.

. si d= m-1, $[floor(g+d/2) -1] - g \leq g+d/2 -1 - g < g+d/2 - g = (d - g)/2 < n /(2^k \times 2) = n/2^{k+1}$.

Si $k \geq \log_2(n)$ alors $2^k \geq n$ et $d-g < n/2^k \leq 2^k/2^k = 1$.

Donc après $k \geq \log_2(n)$ itérations, d-g < 1 donc d-g ≤ 0 et la boucle s'arrête à l'itération k (si n est une puissance de 2) ou à la suivante k+1.

Le nombre d'itérations de la boucle est au plus de $\log_2(n)$ ou $floor(\log_2(n))+1$. Chaque itération prend un temps constant par rapport à n (où : on effectue une comparaison t[m]==v par itération). Donc le temps total est au pire majoré par $c \times \log_2(n)$. D'où T(n) = O(log n).

Rechercher : que retenir !

Deux algorithmes de recherche d'une valeur dans un tableau 1D
- cas général : la recherche séquentielle et ses multiples variantes (boucle pour tantque, ...)
- cas de la recherche dans un tableau trié : la recherche dichotomique

La complexité en temps de la recherche
- dépend de la longueur du tableau n
- se mesure par le nombre de comparaisons avec la valeur cherchée t[i] == v ou s[i] == c

Résultats fondamentaux
- La recherche séquentielle est de complexité au pire linéaire : T(n) = O(n)
- La recherche dichotomique dans un tableau trié est au pire logarithmique : T(n) = O(log n)

Au pire ? Le nombre de comparaisons effectuées dépend de la position de v dans t
- il y a **des** meilleurs cas : 1 seule comparaison donne le résultat.
- ces meilleurs cas dépendent des données : valeurs de t et valeur cherchée
- ces meilleurs cas ne sont pas les même selon l'algorithme
- il n'y a aucune donnée qui demande plus n comparaisons dans le traitement séquentiel ou plus de $\log_2 n$ (+1) comparaisons dans le traitement dichotomique

Et pour finir !

On a aussi vu :

- des preuves de terminaison, de correction, des invariants de boucle
- des analyses de complexité en temps
- des variantes d'écritures d'une même idée algorithmique
- une méthode générale : *divide and conquer , divisier pour régner,* qui conduit naturellement à des algorithmes récursifs (que nous verrons bientôt)
- que ces algorithmes s'appliquaient aussi bien à des tableau de nombres, des tableaux de n'importe quoi du moment qu'il sont numérotés, et à des chaînes de caractères où on cherche un caractère

On n'a pas signalé que la complexité en espace était constante par rapport à la taille du tableau

Correction exercice recherche séquentielle

```
pour i de 0 à n-1 faire
        si t[i] == v alors
                trouve = Vrai
                retourner trouve
        fin si
fin pour
retourner trouve
```

On a montré que si trouve==Vrai alors v est bien dans le tableau t.

Invariant de la boucle pour qui prouve que la valeur Faux est correctement retournée par trouve.

(P) : au début l'itération i, si v est présent dans t alors il est présent dans le sous-tableau t[i, n-1]

Initialisation : avant la première itération, i==0. La sous-tableau t[0,n-1] est le tableau complet. (P) est bien vraie.

Conservation : supposons qu'au début de l'itération i, si v est présent dans t alors il est présent dans le sous-tableau t[i, n-1]. Si l'exécution atteint l'itération suivant, alors t[i] ≠ v. Donc si x est présent dans le tableau t[i, n-1] alors il apparait dans le sous-tableau t[i+1, n-1]. Ainsi (P) est vraie à l'itération suivante qui est d'indice i+1.

Terminaison : La boucle termine soit avec trouve==Vrai (cas déjà traité), soit avec i==n. Dans ce cas trouve==Faux *est la valeur retournée par l'algorithme.*

La contraposée de l'invariant de boucle est : si v n'est pas présent dans le tableau t, alors il n'est pas présent dans le sous-tableau t[i, n-1]. Ici i==n donc le sous-tableau t[n, n-1] est vide. Il ne peut pas contenir la valeur v cherchée. La contraposée de (P) nous dit qu'il n'est donc pas présent dans le tableau t. Il est donc correct de retourner la valeur Faux *dans ce cas.*