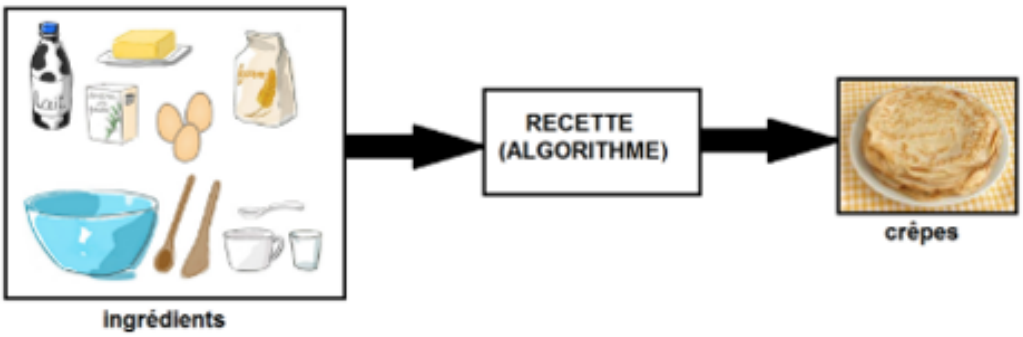
Algorithmie

1 Qu'est ce qu'un algorithme ?

1.1 Définition

Définition 5.1

Un algorithme est une procédure pas-à-pas de résolution d'un problème s'exécutant en un temps fini.



Un délicieux algorithme :-)

La plupart des algorithmes transforment des données d'entrée en des données de sortie.

1.2 Les points essentiels d'un algorithme

La preuve : Il faut être en mesure de prouver que l'algorithme résoud bien le problème pour lequel il a été écrit

L'arrêt : Il faut prouver que l'algorithme s'arrête en un nombre fini d'étapes, et ceci dans tous les cas de figure.

Le choix des structures de données : La façon d'organiser et d'utiliser les données influence l'écriture et la performance de l'algorithme.

Son efficacité : Il est très important d'avoir une idée du comportement de l'algorithme lorsque la taille des données passée en argument augmente.

1.3 Le pseudo code

1.3.1 Définition

En programmation, le pseudo-code, également appelé LDA (pour Langage de Description d'Algorithmes) est une façon de décrire un algorithme en langage presque naturel, sans référence à un langage de programmation en particulier.
L'écriture en pseudo-code permet de comprendre les difficultés de la mise en œuvre de l'algorithme et de développer une démarche structurée dans la construction de celui-ci.
En effet, son aspect descriptif permet de décrire avec plus ou moins de détail l'algorithme, permettant de ce fait de commencer par une vision très large en s'affranchissant de la syntaxe des langages de programmation.

Il n'existe pas de réelle convention standardisée pour le pseudo-code. Nous utiliserons le pseudo-code utilisé à L'Ecole Supérieure d'Informatique.
Les mots-clés seront écrits en majuscules et soulignés Les variables ne contiendront aucun espace, aucun caractère accentué, ne commenceront pas par un chiffre et seront écrites en minuscules Les commentaires seront précédés par //.

1.3.2 Le pseudo-code

Afficher un texte : ECRIRE "Veuillez entrer les nom et prénom"

Afficher une variable ECRIRE nom,prenom

Afficher une variable et du texte ECRIRE "Les contenus de val1 et val2",val1," et ",val2

Déclarer les variables et affectation (internes et externes) :

```
    val1:entier     // déclaration de variable , ici val1
                    //est du type entier
val2:reel
flag:logique
nom,prenom:chaine
val1 <- 5    // Affectation interne, ici val1 prend  //la valeur 5
val2 <-5.5
flag <- VRAI
LIRE nom, prenom  // Affectation externe:  //l'utilisateur entre les valeurs
```

Les opérateurs arithmétiques :

*	multiplication
/	division
+	addition ou concaténation
-	soustraction
MOD	reste d'une division
DIV	division entière
**	exposant

>	plus grand que (greater than)
<	plus petit que (less than)
>=	plus grand ou égal à (greater than or loose-equals)
<=	plus petit ou égal à (less than or loose-equals)
=	égalité
!=	inégalité

Les opérateurs logiques :

ET	et
OU	ou inclusif
XOU	ou exclusif
NON	inverse de

Les boucles conditionnelles

```
SI (val1 > 5) ALORS
   ECRIRE "La valeur",val1,"est supérieure à 5"
FINSI
SI (val1 > 5) ALORS
   ECRIRE "La valeur",val1,"est supérieure à 5"
SINON
   ECRIRE "La valeur",val1,"est égale ou inférieure à 5"
FINSI
SI (val1 > 5) ALORS
   ECRIRE "La valeur",val1,"est supérieure à 5"
SINONSI (val1=5) ALORS
   ECRIRE "La valeur",val1,"est égale à 5"
SINON
   ECRIRE "La valeur",val1,"est inférieure à 5"
FINSI
```

Les boucles :

```
TANT QUE (cpt < val1)
      ECRIRE "Tour n°",cpt
   Cpt <- Cpt + 1
FINTANT

POUR(i DE 0 A 5 PAR 1) FAIRE
    ECRIRE "Tour n°",i
FINPOUR
```

Les tableaux :

```
montab1:tableau[0,15]  de entier
// pour déclarer un tableau (une liste  // en python)
montab2:tableau[0,5]  de chaine
montab3:tableau[0,9]  de reel
montab4:tableau[0,126]  de logique

montab:tableau[0,5] de entier   //pour initialiser le tableau avec des 0
i:entier
i <- 0
TANT QUE(i <= 5) FAIRE
 montab[i] <-0
 i <-i + 1
FINTANT


POUR(i DE 0 A 5 PAR 1 FAIRE)     // pour parcourir un tableau
 ECRIRE "Veuillez entrer une valeur"
 LIRE montab[i]
FINPOUR
  //Affichage du tableau
POUR(i DE 0 A 5 PAR 1 FAIRE)
 ECRIRE montab[i]
FINPOUR
```

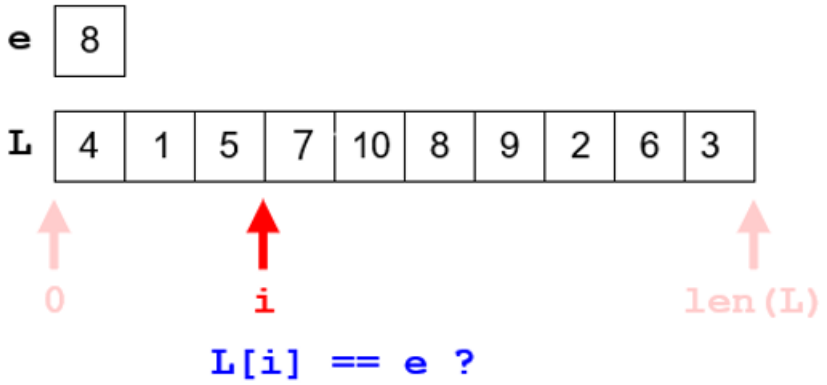
Les fonctions ou procédures :

```
FONCTION politesse()      //Procédure sans arguments
ECRIRE "Bonjour"
FIN FONCTION

politesse()    // pour appeler la fonction ou la procédure politesse

FONCTION calcul(val1:entier, val2:entier)
   res:entier
   res = val1 + val2
   RETOURNER res
FIN FUNCTION

val1, val2, res:entier   //Programme principal
LIRE val1, val2
res = calcul(val1,val2)
```

2 Recherche d'occurrences

2.1 Algorithme de recherche d'occurrences

La recherche d'une occurrence !consiste à déterminer la position (l'indice i) d'un élément e présent dans la liste L. On parle d'occurrence de e dans L.
Exemple : Je recherche l'occurrence de 8 dans la liste L :



Exercice 5.1

Ecrire un algorithme de recherche d'une occurrence utilisant une boucle POUR.
L'algorithme est une fonction qui prend en argument un tableau d'entiers et une valeur entière, et qui renvoie l'occurrence de cette valeur (FAUX si la valeur n'est pas présente dans le tableau).

Exercice 5.2

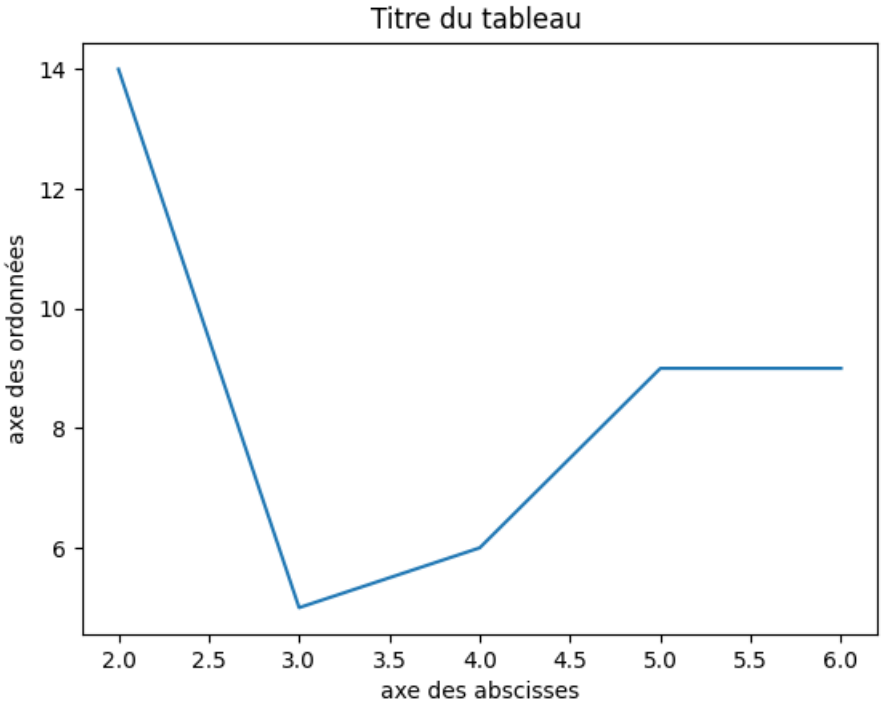
Ecrire un algorithme de recherche d'une occurrence en utilisant une boucle TANT QUE.
L'algorithme est une fonction qui prend en paramètres un tableau d'entiers et une valeur entière, et qui renvoie l'occurrence de cette valeur (FAUX si la valeur n'est pas présente dans le tableau)

2.2 Durée d'exécution

Exercice 5.3

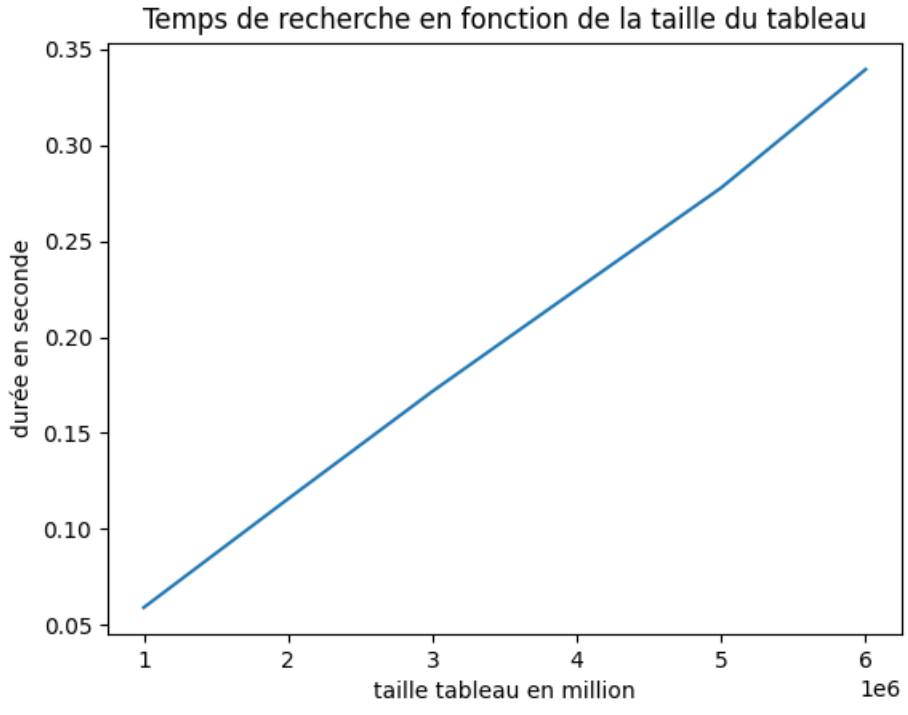
1. Construire une liste de 1 million d'élements, avec des entiers de 0 à 999999.
2. Appliquer un des algorithmes précédents à cette liste et calculer le temps d'exécution pour la recherche d'un élément qui n'est pas dans le tableau (afin d'être sûr qu'on parcourt le tableau en entier). Garder la valeur en mémoire.
3. Recommencer les deux questions précedentes pour une liste de 2 000 000 , puis 3 000 000 etc jusqu'à 6 000 000. Garder les résultats.
4. Observer ce code qui permet l'affichage du graphique ci-après.

```
import matplotlib.pyplot as plt
plt.title('Titre du tableau ')
plt.plot([2,3,4,5,6],[14,5,6,9,9])
plt.xlabel('axe des abscisses')
plt.ylabel('axe des ordonnées')
plt.show()
```



Exemple d'utilisation de matplotlib

5. Placer les points obtenus aux questions précédentes sur un graphique, afin de visualiser la durée d'exécution de la fonction recherche en fonction de la taille de la liste.
   Vous devriez obtenir un graphique comme celui-ci :



On en déduit, de façon empirique, que le temps de recherche est linéaire par rapport à la taille du tableau.

2.3 Compléxité temporelle

Nous allons essayer de formaliser la notion précédente et d'estimer le coût en temps de cet algorithme.

☞ Nous supposerons que chaque opération (affectation, comparaison, opération arithmétique, ...) a un coût de 1 « unité » de temps.

Reprenons un des deux algoritmes, celui avec la boucle pour par exemple :

```
1 FONCTION OCCURRENCE(T :tableau d'entiers, element :entier)
2 |  POUR i DE O A longueur(T)-1 FAIRE
3 |  |  SI T[i] = element ALORS
4 |  |  |  RENVOYER i
5 |  RENVOYER False
```

On suppose que la taille du tableau est n :
ligne 2 : au pire n affectations pour i plus 1 interrogation de longueur de tableau
ligne 3 : au pire, n comparaisons

Soit $T(n)$ le nombre d'opérations. $T(n) = 2n + 1$. On a donc une complexité en $\mathcal{O}(n)$, soit une complexité linéaire.

Exercice 5.4

1. Créer l'algorithme de la fonction **maximum(T)** qui reçoit en argument un tableau d'entiers et qui retourne le maximum du tableau.
2. Calculer la complexité temporelle de cette algorithme.
3. Implémenter cet algorithme en Python

Exercice 5.5

1. Créer l'algorithme de la fonction **minimum(T)** qui reçoit en argument un tableau d'entiers et qui retourne le minimum du tableau.
2. Calculer la complexité temporelle de cette algorithme.
3. Implémenter cet algorithme en Python

3 Tri par sélection

3.1 Principe

Principe : on cherche le minimum des éléments non triés et on le place à la suite des éléments triés

VIDÉO Un animation sur le tri sélection - Cliquez ici !

3.2 L'algortithme

Savoir-Faire 5.1

Savoir Construire l'algorithme correspondant au tri par sélection

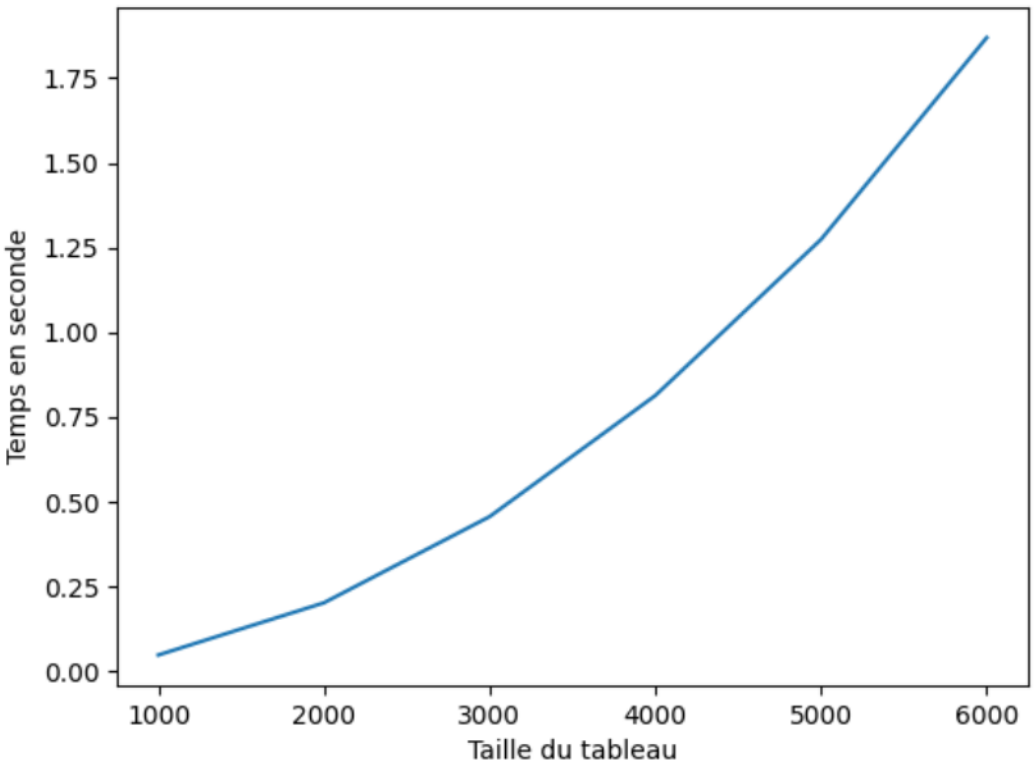
3.3 Implémentation en Python

3.4 Complexité temporelle

3.4.1 Etude empirique

Exercice 5.6

1. Construire une liste de 1000 élements, avec des entiers de 0 à 999 rangés dans l'ordre décroissant.
2. Appliquer la fonction tri_selection à cette liste et calculer le temps d'exécution. Garder la valeur en mémoire.
3. Recommencer les deux questions précedentes pour une liste de 2 000 , puis 3 000 etc jusqu'à 6 000. Garder les résultats.
4. Placer les points obtenus aux questions précédentes sur un graphique, afin de visualiser la durée d'exécution de la fonction recherche en fonction de la taille de la liste.
   Vous devriez obtenir un graphique comme celui-ci :



Durée du tri sélection en fonction de la taille du tableau.

3.4.2 Meilleur des cas - Pire des cas

Pour analyser la complexité de cet algorithme, nous allons analyser le nombre de comparaisons effectué ainsi que le nombre d'échange lors du tri.
Or, le nombre de comparaisons à effectuer ne dépend pas de la liste (du fait qu'elle soit triée ou non).
En conséquence, la complexité en moyenne et dans le pire des cas sera identique.
Notons S_n le nombre de comparaisons pour déterminer le minimum dans un tableau de longueur n et T_n le nombre de comparaisons dans l'algorithme du tri sélection.

Calculer T_n et en déduire la complexité moyenne et dans le pire des cas ?

3.5 Pré et post conditions

3.6 Terminaison

3.7 Validité de l'algorithme

3.7.1 Invariant de boucle

Pour prouver la correction, nous utilisons un invariant de boucle :
Pour chaque i, et à la fin de la boucle, on a :
"la liste [0 ;i+1[est triée par ordre croissant, et les éléments de la liste [i+1 ;n[sont tous supérieurs à tous les éléments de la liste [0 ;i+1["

3.7.2 Démonstration

Démonstration 5.1

Étape 1 : La propriété est-elle vraie pour i=0 ? :

Étape 2 : On suppose la propriété vraie pour l'entier k. Est-elle vraie pour l'entier k+1 ?

Etape 3 : Et pour le dernier passage dans la boucle ?

Étape 4 : Conclusion

4 tri par insertion

4.1 Principe

On insère un à un les éléments parmi ceux déjà trié.
☞ Le ***tri par insertion*** est aussi appelé "tri naturel" : c'est souvent ce tri que nous utilisons naturellement pour trier un jeu de cartes par exemple.

VIDÉO Une animation pour comprendre le tri par insertion - Cliquez ici !

Exercice 5.7
Après avoir étudié la vidéo, donner les différentes étapes du tri par insertion pour le tableau suivant : $T = [8, 5, 1, 9, 7]$

4.2 L'algortithme

Savoir-Faire 5.2

Savoir Construire l'algorithme correspondant au tri par insertion

```
1 FONCTION OCCURRENCE(T :tableau d'entiers, element :entier)
2     POUR i DE 1 A longueur(T) − 1 FAIRE
3         x ← T[i]
4         j ← i
5         TANT QUE j > 0 and T[j − 1] > x FAIRE
6             t[j] ← t[j − 1]
7             j ← j − 1
8         t[j] ← x
```

4.3 Implémentation en Python

4.4 Complexité temporelle

Savoir-Faire 5.3

Savoir déterminer la complexité du tri par insertion

Dans le pire des cas, le tableau est rangé dans l'ordre décroissante.

Soit $T(n)$ Le nombre de comparaisons.

Calculer $T(n)$, et en déduire la complexité.

***

4.5 Terminaison

La boucke POUR ne pose aucun problème !

En revanche, il faut faire attention à la boucle TANT QUE qui peut ne jamais terminer !

La condition de la boucle TANT QUE "est $j < 0$ and $T[j-1] > x$".

La boucle s'arrêtte quant l'une ou l'autre des conditions n'est plus vraies.

Ici, on décrémente j à chaque fois de, donc nous sommes certains que la condition $j > 0$ ne sera plus vraie à partir d'une certaine étape !

☞ Notre algorithme s'arrêtera donc !

4.6 Validité de l'algorithme

Démonstration 5.2

☛ Invariant de boucle

Considérons la propriété P(i) : "Le tableau T[0],T[1],...,T[i-1] est trié ".

Étape 1 : La propriété est-elle vraie pour i=0 ? :

Étape 2 : On suppose la propriété vraie pour l'entier k. Est-elle vraie pour l'entier k+1 ?

Etape 3 : Et pour le dernier passage dans la boucle ?

Étape 4 : Conclusion

5 Recherche dichotomique

5.1 Principe

Considérons le tableau trié [5,7,12,14,23,27,35,40,41,45].

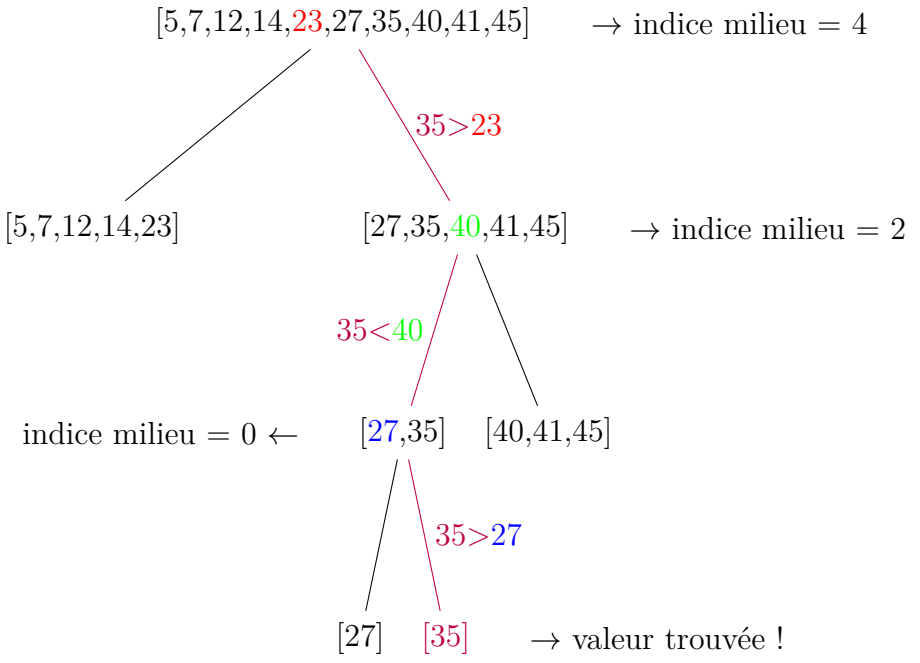
Si on veut trouver une valeur, 35 par exemple, il est possible d'utiliser l'algorithme de recherche d'une occurrence. Dans le pire des cas, on parcourt le tableau en entier. Dans notre exemple, il faut faire 7comparaisons.

Mais comme la liste de départ est triée, la recherche ***dichotomique*** permet d'améliorer la performance de la recherche.

On souhaite rechercher l'entier 35 :

- Si la liste est vide, la recherche est finie et on renvoir False
- Sinon, on cherche la valeur la plus "centrale" dans la liste et on la compare avec l'élément recherché.

- Si la valeur est celle recherchée, la recherche est finie, et on renvoie True
- Si la valeur est strictement inférieure à l'élément recherchée, reprendre la recherche avec la la seconde moitié de la liste
- Si la valeur est strictement supérieure à l'élément cherché, reprendre la recherche avec la première moitié de la liste.



☞ Il suffit ici de 3 tours de boucles.

5.2 Algorithme

Exercice 5.8
Créer une fonction **recherche_ dichotomique(liste,element)** qui prend en paramètre un tableau d'entiers et un entier, et renvoie True si l'élément est dans la liste, False sinon

5.3 Complexité

5.3.1 Etude théorique

Considérons un tableau de taille n.

☞ On admet que la partie entière de $log_2(n)$ augmentée de 1 donne le nombre de chiffres de l'écriture binaire de n.

Exemples

- $log_2(4) = 2$, et l'écriture binaire de 4 (100) contient 3 chiffres.
- $log_2(64) = 6$, et l'écriture binaire de 64 (1000000) contient 7 chiffres.
- $log_2(24) \simeq 4.58$, et l'écriture binaire de 24 (11000) contient 5 chiffres.

A chaque étape de la recherche dichotomique, on divise la longueur de la liste par 2 (à 1 près).

Or, diviser un entier par 2 revient à supprimer le bit de poids faible dans son écriture binaire (toujours à 1 près !)

- 24 est représenté en binaire par 11000. 12 est représenté en binaire par 1100. (on a supprimé le dernier chiffre).
- et 6 est représenté par 110 (on a supprimé le dernier chiffre).
- 3 est représenté par 11 (on a supprimé le dernier chiffre).
- si on continue le procédé en enlevant une nouvelle fois le dernier chiffre, on arrive à 1 comme écriture binaire, qui correspond au nombre 1. C'est bien la moitié de 3, à 1 près.

Ainsi, si la longueur du tableau est n, et puisqu'on divise par 2 la longueur du tableau à chaque étape, on enlève le bit de poids faible à l'écriture binaire de n à chaque étape.
Comme l'écriture binaire de n comporte $log_2(n) + 1$ chiffres, il faudra au maximum $log_2(n)$ étapes !

Ainsi, le nombre maximal d'étapes pour la recherche dichotomique dans un tableauu de longueur n est $log_2(n)$.

Autrement dit, le nomnre maximum de comparaisons dans un tableau de longueur$n$ est $log_2(n)$.
on a donc une complexité temporelle proportionnelle à $log_2(n)$. On parle de complexité logarithmique.
On note $\mathcal{O}(log_2(n))$.

5.3.2 Illustration sur des exemples

Exercice 5.9
Compléter le tableau suivant, vous comprendrez l'intérêt de la recherche dichotomique par rapport à une recherche séquentielle...Bien sûr, il faut que le tableau soit trié !!

Taille de la liste	0	1	2	4	128	1 000	10 000	100 000	1 000 000	n
Recherche séquentielle										n
Recherche dichotomique										$log_2(n)$

***

5.4 Terminaison

La boucle TANT QUE peut poser problème, et il faut s'assurer que dans tous les cas, la boucle n'est pas infinie.
Considérons comme variant de boucle la longueur du tableau en cours.

Exercice 5.10
Montrer que ce variant décroît strictement et finira par être strictement inférieur à 1.