TP-Cours d'Informatique 08

# Dichotomie

ITC – PCSI$^2$ 2024-2025 – Lycée Fabert (Metz) – L. Parise – N.Wipf – S.Limal – D.Mengel

***Divide ut imperes*** : nombre d'algorithmes utilisent l'approche « **Diviser pour régner** » dont la méthode de dichotomie est une parfaite illustration (mais nous en verrons d'autres). Ces algorithmes ont tous en commun de séparer un problème en plusieurs **sous-problèmes similaires** au problème initial, mais de **taille** moindre donc plus faciles à traiter.

Le paradigme de programmation « **Diviser pour régner** » est par nature récursif (on peut appliquer la même méthode pour résoudre les sous-problèmes issus du problème initial) . Cependant, l'exécution d'algorithmes récursifs pouvant conduire à un dépassement de pile ou à des redondances d'appels (voir TP-Cours n°07), on en préfère parfois une écriture itérative.

## I. Principe de Dichotomie, recherche dichotomique dans une liste triée

Voici le **problème** abordé dans cette partie : étant donnés $x \in \mathbb{N}$ et une liste $a$ triée constituée d'entiers naturels distincts $a_0 < a_1 < \cdots < a_{n-1}$ avec $n \in \mathbb{N}^*$ (la longueur de $a$), on cherche à déterminer si $x$ est présent dans la liste $a$. Le cas échéant, nous souhaitons obtenir la position de $x$ dans $a$.

**Remarque :** ce problème peut facilement se généraliser en considérant un ensemble $E$ muni d'une relation d'ordre totale comme un ensemble de chaînes de caractères muni de l'ordre lexicographique.

**Principe de la recherche dichotomique ou recherche par subdivision binaire :** l'idée générale est de chercher l'élément $x$ dans la première ou la seconde moitié de $a$ en appliquant l'algorithme :

- trouver l'indice $m$ le plus *central* de la liste $a$.
- comparer la valeur *centrale* $a[m]$ à l'élément recherché $x$.
- Si la valeur est égale à l'élément, cesser la recherche car $x$ est dans $a$ en position $m$, sinon reprendre la procédure dans la **moitié gauche** de $a$ lorsque $x < a[m]$ ou dans la **moitié droite** dans le cas contraire.

### I.1. Création d'une liste triée et recherche linéaire

Dans un premier temps, nous allons écrire une fonction créant une liste triée et une fonction de recherche linéaire dans une liste. Ces deux fonctions nous seront utiles dans la suite du TP.

1. Définir la fonction listeTriee (n:**int**) → a: **list** qui renvoie une liste triée $a$ aléatoire de longueur $n \in \mathbb{N}^*$ formée des entiers $a_0 < a_1 < \cdots < a_{n-1}$ vérifiant :

$$a_0 \in [\![1;6]\!] \qquad \text{et} \qquad \forall i \in [\![1;n-1]\!],\ a_{i+1} - a_i \in [\![1;6]\!]$$

   **Indication :** on utilisera la fonction randint du module random.

2. En utilisant l'algorithme de recherche séquentielle dans un tableau vu dans le TP-Cours n°03, écrire la fonction rechercheLin(x:**int**, a: **list** ) → present:**bool**, pos:**int** qui renvoie un couple formé du booléen present indiquant si l'entier x est dans la liste a ainsi que la position pos que x **occupe ou devrait occuper dans** a.

On rappelle ici que si $a$ est une liste de longueur $n \in \mathbb{N}^*$, le nombre maximal de comparaisons effectuées entre $x$ et un élément de $a$ par un algorithme de recherche linéaire de $x$ dans $a$ est $n$. On dira que la **complexité au pire** d'un tel algorithme est de l'ordre de $n$, ce que l'on notera $O(n)$.

### I.2. Recherche dichotomique : exemples, algorithmes itératifs et récursifs

Nous allons décliner l'algorithme de dichotomie en utilisant deux indices entiers g (pour **gauche**) et d (pour **droite**) permettant de déterminer, à chaque étape de son exécution, la **zone active de recherche de x dans a**. Au début de l'algorithme, on pose donc g=0 et d=n−1. Ensuite,

- on définit l'indice *central* m=(g+d)//2 de la zone de a délimitée par les indices g et d.
- Si x et a[m] sont égales, la recherche est terminée car x est dans a en position $m$
- Si x<a[m] on indique que la recherche de x doit s'effectuer dans la moitié gauche de la zone active c'est-à-dire parmi les éléments a[g], a[g+1],..., a[m−1] en posant $d = m - 1$
- Si x>a[m] on indique que la recherche de x doit s'effectuer dans la moitié droite de la zone active c'est-à-dire parmi les éléments a[m+1],a[m+2],..., a[d] en posant $g = m + 1$.

Cette procédure se poursuit tant que g⩽d et que l'on n'a pas trouvé x dans a.

On pourra remarquer que cet algorithme se termine bien car l'expression entière d−g décroît strictement à chaque étape de celui-ci (on dira plus tard que d−g est un **variant de boucle**). La sortie de la boucle **while** est ainsi provoquée par le fait qu'on a trouvé x dans a ou que g>d, ce qui signifie que x n'est pas dans a.

Dans la suite, on donne 2 exemples de recherche dichotomique d'un entier dans une liste triée a. On cherche d'abord l'entier 21 qui n'est pas présent dans a, puis l'entier 22 qui se trouve bien dans a.

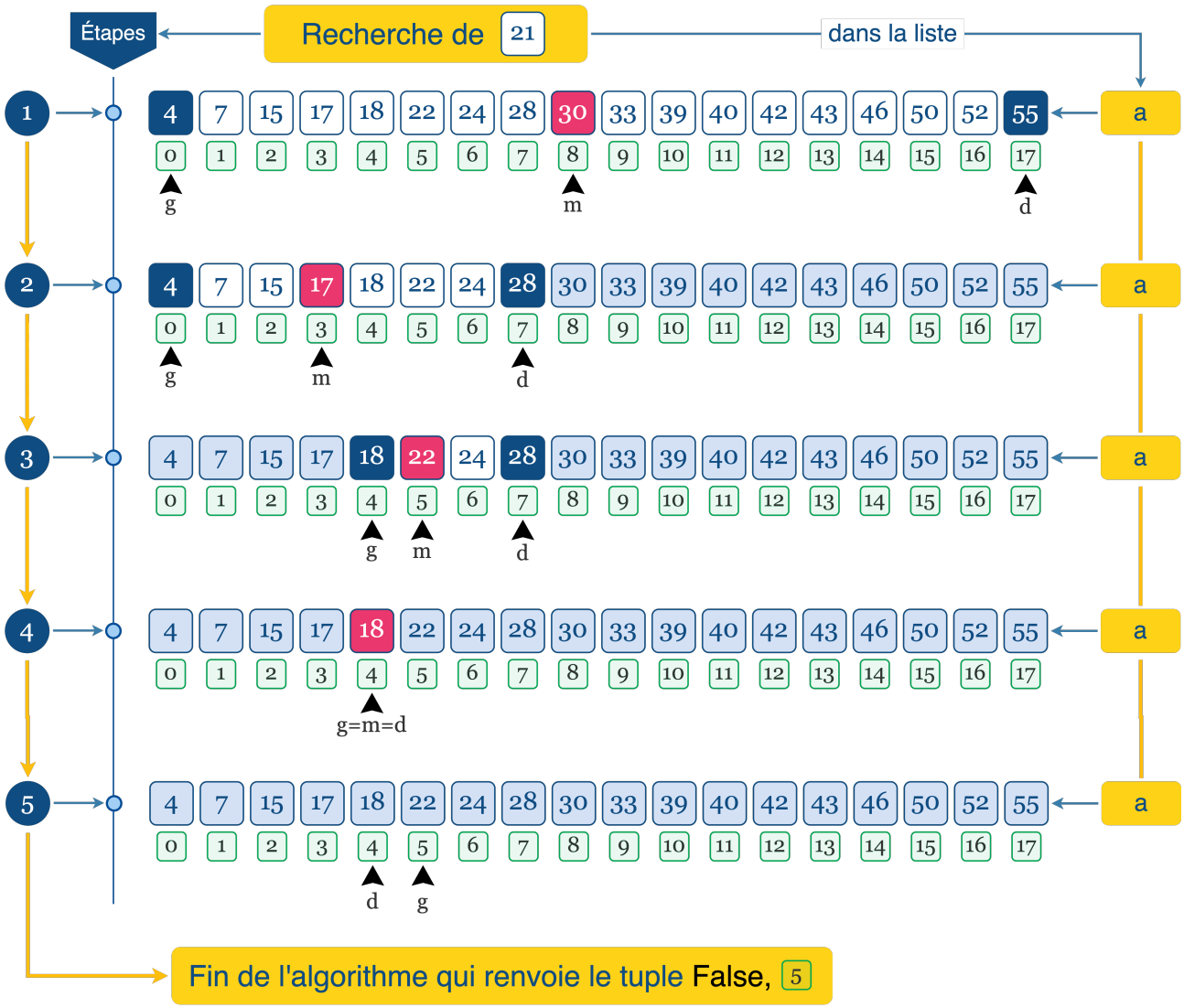



FIGURE 1 – *Exemple d'application de l'algorithme de recherche dichotomique dans une liste triée*

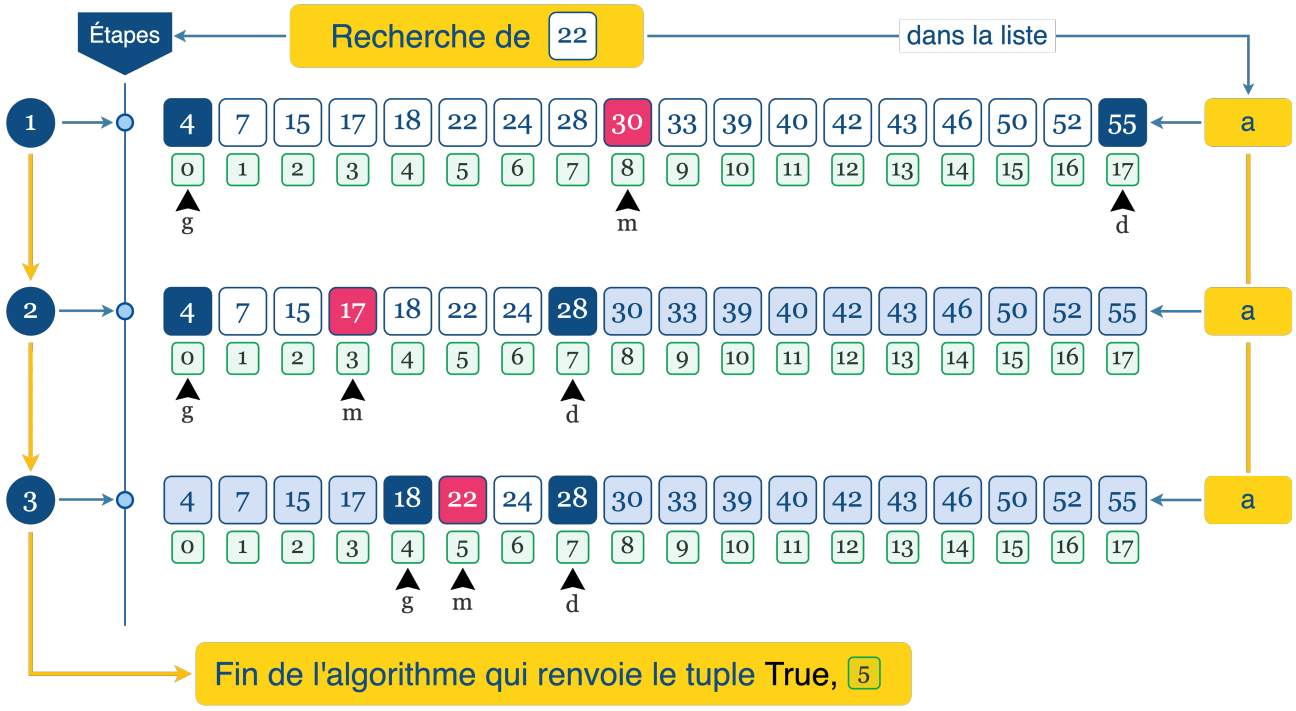



FIGURE 2 – *Exemple d'application de l'algorithme de recherche dichotomique dans une liste triée*

3. En suivant l'algorithme de recherche dichotomique, compléter le schéma suivant en vous aidant des deux exemples ci-dessus.

4. En utilisant l'algorithme itératif de recherche dichotomique décrit ci-dessus, écrire la fonction rechercheDicho(x:**int**, a:**list**) → present:**bool**, pos:**int** qui renvoie un couple formé du booléen present indiquant si l'entier x est dans la liste a ainsi que la position pos que x **occupe ou devrait occuper dans** a.
5. Écrire la fonction rechercheDicho2, version récursive naturelle de la fonction précédente.

**Application**. Nous allons utiliser maintenant une des deux fonctions précédentes pour insérer ou supprimer un élément d'une liste triée.

6. Écrire la fonction inserer(x:**int**, a:**list**) → action:**bool** qui insère l'entier x dans la liste triée a (et donc modifie potentiellement a) si x ne s'y trouve pas déjà. Cette fonction renverra un booléen noté ici action indiquant si l'opération a été effectuée. Avant d'utiliser votre environnement de développement Python préféré, compléter le code ci-dessous :

```
1  def inserer(x:int, a:list):
2      present, pos = rechercheDicho(x,a)
3      if present:
4          return False
5      else:
6
7
8
9
10
```

Listing 1 – Code de la fonction `inserer(x:int,a:list)→bool`

7. Écrire la fonction supprimer(x:**int**, a:**list**) → action:**bool** qui supprime l'entier x dans la liste triée a (et donc modifie potentiellement a) si x s'y trouve effectivement. Cette fonction renverra un booléen noté ici action indiquant si l'opération a été réalisée. Avant toute chose, compléter le code ci-dessous :

```
1  def supprimer(x:int, a:list):
2      present, pos = rechercheDicho(x,a)
3      if not present:
4          return False
5      else:
6
7
8
9
10
```

Listing 2 – Code de la fonction `supprimer(x:int,a:list)→bool`

### I.3. Complexité

La complexité en temps, au pire, relative au nombre de tests effectués sur des éléments de la liste d'entiers $a$ triée et de longueur $n \in \mathbb{N}^*$, de la fonction rechercheLin(x:**int**,a:**list**):→True de recherche linéaire de $x$ dans $a$ est $n$.

En effet, si l'entier $x$ cherché dans $a$ est le dernier élément de $a$ ou si $x$ n'est pas dans $a$, cette dernière fonction effectue les $n$ comparaisons possibles de $x$ aux éléments de $a$. On dira que **cette complexité au pire est de l'ordre de $n$, ce que l'on notera (au second semestre)** $O(n)$.

Intéressons-nous à présent à la complexité de la fonction de recherche dichotomique de $x$ dans $a$ rechercheDicho(x:**int**, a:**list**) → present:**bool**, pos:**int** dont on donne ici le code qui servira de corrigé :

```
1  def rechercheDicho(x,a):
2
3      g = 0
4      d = len(a)-1
5
6      while g<=d:
7          m = (g+d)//2
8
9          if a[m]==x:
10             return True,m
11         elif x<a[m]:
12             d = m - 1
13         else:
14             g = m + 1
15
16     return False, g
```

Listing 3 – Code de la fonction `rechercheDicho(x:int,a:list)→bool, int`

À chaque itération de la boucle **while**, on effectue 2 comparaisons de $x$ avec un élément de $a$. Nous allons donc compter le nombre d'itérations, au pire, de cette boucle, celui-ci étant proportionnel au nombre de comparaisons que l'on cherche à déterminer.

L'étude précise de cette complexité sera réalisée en détail au second semestre, mais on peut tout de même donner dès à présent des explications *presque rigoureuses* sur son calcul.

En partant d'un tableau de longueur $n \in \mathbb{N}^*$, la zone active de recherche de $x$ dans $a$ est **à peu près** de longueur $n/2$ (en fait $n/2$-1 **si** $n$ **est pair** dans l'algorithme précédent) après une itération de la boucle **while**.

Grosso modo, la longueur de la zone active de recherche est divisée par 2 à chaque itération de la boucle et toute itération peut conduire à la sortie de la fonction, si a[m]==x. Au pire, si l'algorithme se termine avec g==d parce que $x$ n'est pas dans $a$ ou parce qu'il faut réduire la recherche à une zone de formée d'un seul élément pour trouver $x$, on obtient donc $\frac{n}{2^k} = 1$ au bout des $k \in \mathbb{N}^*$ itérations de la boucle. Il s'ensuit

$$\ln n = k \ln 2 \qquad \text{soit} \qquad k = \frac{\ln n}{\ln 2} = \log_2(n)$$

On dira alors que la rechercheDicho est logarithmique et sera notée $O(\ln n)$.

## II. Recherche d'un zéro d'une fonction continue

On considère $(a, b) \in \mathbb{R}^2$ tel que $a < b$ et $f : [a; b] \longrightarrow \mathbb{R}$ continue vérifiant $f(a)f(b) \leqslant 0$. Autrement dit, on suppose que $f(a)$ et $f(b)$ sont de signes opposés. Le lemme du théorème des valeurs intermédiaires consiste alors à démontrer qu'il existe $c \in [a; b]$ tel que $f(c) = 0$ : on dit que $c$ est un zéro de $f$.

Dans cette partie, on cherche une valeur approchée d'un zéro de $f$ en construisant, selon un procédé dichotomique, **deux suites adjacentes** $(a_n)_{n\in\mathbb{N}}$ **et** $(b_n)_{n\in\mathbb{N}}$ qui convergent vers $c \in [a; b]$ vérifiant $f(c) = 0$. Voici cet algorithme :

1. On pose $a_0 = a$ et $b_0 = b$ de sorte que $f(a_0)f(b_0) \leqslant 0$.

2. Si on suppose que pour un certain $k \in \mathbb{N}$, $f(a_k)f(b_k) \leqslant 0$, alors on pose $c_k = \frac{1}{2}(a_k + b_k)$ et

   — soit $f(a_k)f(c_k) \leqslant 0$ et on pose $a_{k+1} = a_k$ et $b_{k+1} = c_k$,

   — soit $f(a_k)f(c_k) > 0$ et dans ce cas, $f(c_k)f(b_k) = \frac{f(c_k)}{f(a_k)} f(a_k)f(b_k) \leqslant 0$ et on pose $a_{k+1} = c_k$ et $b_{k+1} = b_k$

   Dans les deux cas précédents, on obtient $f(a_{k+1})f(b_{k+1}) \leqslant 0$, ce qui garantit que $f$ change de signe sur $[a_{k+1}; b_{k+1}]$ et par ailleurs, $b_{k+1} - a_{k+1} = \frac{1}{2}(b_k - a_k)$ : la longueur du segment $[a_{k+1}; b_{k+1}]$ est la moitié de celle de $[a_k; b_k]$.

On montre alors (dans le cours de mathématiques) que les suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ convergent vers une limite commune $c \in [a_k; b_k]$ pour tout $k \in \mathbb{N}$ puis que $f(c) = 0$.

La construction des premiers termes des suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ est illustrée sur le graphique ci-dessous :

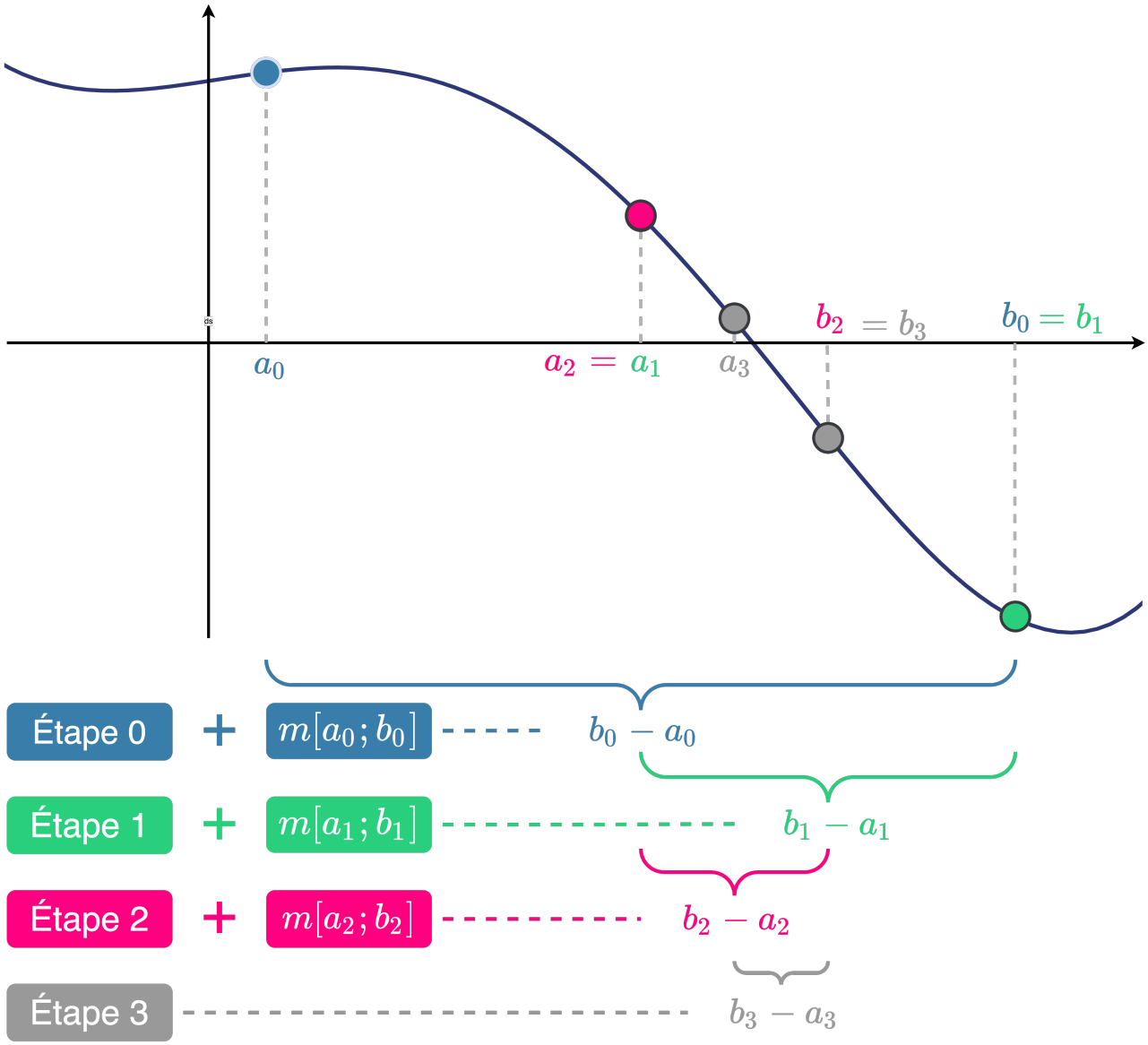
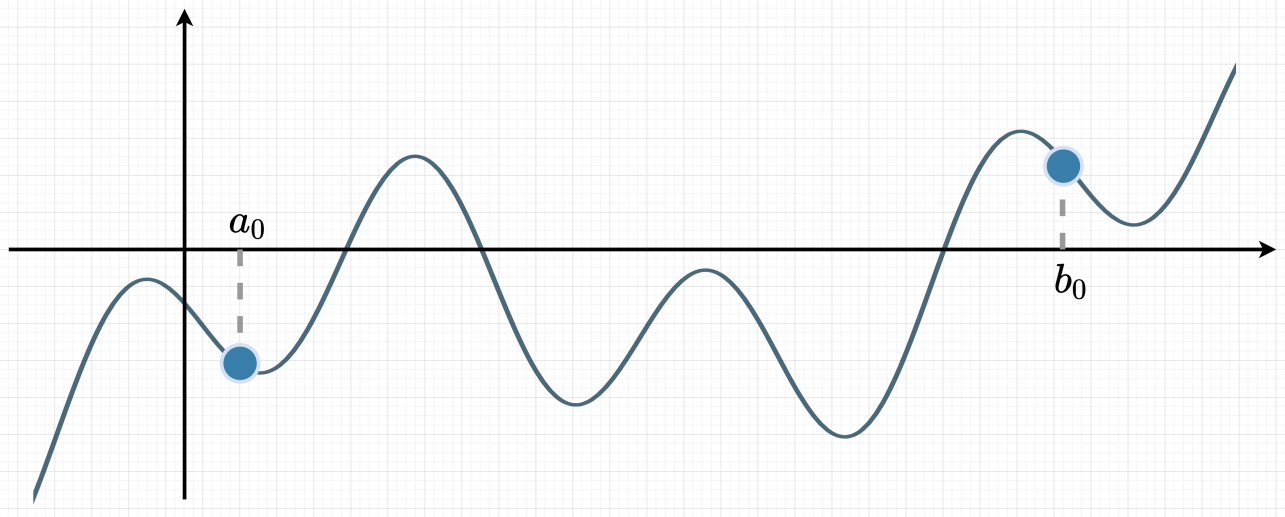



FIGURE 3 – *Exemple d'application de l'algorithme dichotomique de construction des premiers termes des suites* $(a_n)_{n\in\mathbb{N}}$ *et* $(b_n)_{n\in\mathbb{N}}$

8. Compléter le graphique de la page suivante en construisant les premiers termes des suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ pour respecter le même algorithme.




Pour tout $k \in \mathbb{N}$, $a_k \leqslant c \leqslant b_k$ donc on peut affirmer que pour $\varepsilon > 0$ fixé, $a_k$ ou $b_k$ sont des valeurs approchées de $c$ à $\varepsilon$ près dès lors que $b_k - a_k \leqslant \varepsilon$.

9. En utilisant l'inégalité précédente comme condition d'arrêt, et la version itérative de l'algorithme ci-dessus, écrire la fonction zeroDichotomique(f:function,a,b,eps:**float**)→**float** construisant les premiers termes des suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ pour renvoyer une valeur approchée à $\varepsilon > 0$ près d'un zéro de $f : [a; b] \to \mathbb{R}$ continue telle que $f(a)f(b) \leqslant 0$.

10. Donner une valeur approchée de $\sqrt{2}$ à $10^{-6}$ près en appliquant la fonction zeroDichotomique à $f : x \mapsto x^2 - 2$ sur $[0; 2]$.

C'est l'occasion de tester l'instruction **assert** ! L'instruction **assert** de Python est une aide au débogage qui permet de vérifier qu'une condition est bien remplie. Le cas échéant, le programme continue simplement le déroulement du flux d'exécution. Dans le cas contraire, **assert** lève une exception appelée AssertionError avec un message d'erreur facultatif. Observons son utilisation sur un exemple :

```
#import sys # facultatif pour éliminer le Traceback pénible
#sys.tracebacklimit=0
def symetrique(x:float):
    assert x!=0
    return 1/x
```

Test de la fonction symetrique dans la console :

```
>>> symetrique(4)
0.25
>>> symetrique(0)
AssertionError
```

Le message d'erreur étant assez peu informatif, il est possible d'ajouter un message à l'instruction **assert** précisant le type de problème rencontré :

```
def symetrique(x:float):
    assert x!=0, "le paramètre doit être un réel non nul"
    return 1/x
```

11. Ajouter une instruction **assert** au début de la fonction zeroDichotomique pour s'assurer de la condition $f(a)f(b) \leqslant 0$ est bien remplie lors de son appel.

12. Écrire la fonction zeroDichotomique2(f:function,a,b,eps:**float**)→**float**, version récursive de la fonction zeroDichotomique.