Algorithmique : Introduction aux méthodes de recherche

Chapitre 15

Table des matières

1 Performance algorithmique

1.1 Introduction

Un algorithme doit proposer une *solution à un problème donné*, *rapidement*, en utilisant le *moins d'espace mémoire possible*. Si un algorithme conduit à une solution au bout d'un temps « infini », on peut considérer, qu'en pratique, ils ne propose aucune solution[1][2].

Remarque. Un problème peut généralement être résolu de plusieurs façons différentes, correspondant à différents algorithmes.

La *performance d'un algorithme* porte sur deux aspects : *la durée du calcul* et *la quantité de mémoire* nécessaires à la résolution du problème.

Malheureusement ces deux points s'opposent. Il est souvent nécessaire d'occuper davantage la mémoire pour gagner en temps de calcul, ou d'écrire plus d'instructions, et donc faire plus de calculs, pour aboutir à une gestion de la mémoire optimale.

1.2 Notion de complexité algorithmique (Culture)

La *complexité algorithmique* donne des informations sur *la durée du calcul* nécessaire à la résolution du problème. *Plus la* complexité algorithmique *est petite, moins de calculs sont effectués et plus l'algorithme est performant (sous réserve que la gestion de l'espace mémoire utilisé par l'implémentation de l'algorithme ne constitue pas un problème).*

Il existe plusieurs méthodes pour analyser la performance d'un algorithme, comme :

L'analyse moyenne. Elle consiste à évaluer la durée moyenne des calculs à effectuer.

L'analyse pessimiste (ou du pire). C'est l'analyse la plus courante (et celle dont nous allons parler dans la suite de ce cours). Elle consiste à *évaluer le nombre de calculs à effectuer dans le scénario le moins favorable.*

Par exemple, si un problème nécessite de parcourir toute une liste afin de rechercher une donnée, dans cette analyse, on considère que la donnée se trouve en dernière position ou est absente.

Lorsqu'on utilise le résultat de cette analyse, *aucune mauvaise surprise ne peut intervenir, le pire cas à été envisagé à l'avance.*

Les complexités sont très diverses d'un algorithme à l'autre. On peut néanmoins les regrouper en quelques grandes familles :

Les algorithmes *logarithmiques*. Leur notation est de la forme $O(\log N)$. *Ces algorithmes sont très performants en temps de traitement. Le nombre de calculs dépend du logarithme du nombre de données initiales à traiter.*

Sur la **Figure 1.**, on peut constater que, plus le nombre de données N à traiter est important, moins le nombre de calcul à effectuer augmente rapidement (la valeur de la dérivée est de moins en moins grande).

1. C'est l'une grande différence entre l'informatique et les mathématiques : en informatique, on effectue *réellement* les calculs !

2. Ce n'est pas toujours une mauvaise nouvelle ! Certains choix en cryptographie reposent sur l'idée que « casser » la protection nécessite une durée de calcul trop grande, *pour le matériel dont on dispose aujourd'hui.*

La fonction logarithme à utiliser dépend du problème étudié mais, comme la complexité est définie à un facteur près, la *base du logarithme* n'a pas d'importance.

La *complexité logarithmique* apparait dans les problèmes dans lesquels l'ensemble des données peut être décomposé en deux parties égales, qui sont elles-mêmes décomposées en 2, ...[3] Le logarithme à utiliser est alors la fonction réciproque de $f : x \mapsto 2^x$, c'est à dire $\log_2$ (aussi appelé *logarithme entier*[4]).

Les algorithmes *linéaires*. Leur notation est de la forme $O(N)$. *Ces algorithmes sont rapides. Le nombre de calculs dépend, de manière linéaire, du nombre de données initiales à traiter.*

La complexité linéaire apparaît dans les problèmes dans lesquels on parcourt l'ensemble des données pour réaliser une opération (recherche d'une valeur, par exemple).

Les algorithmes *linéaires* et *logarithmiques*. Leur notation est de la forme $O(N \log N)$.

Cette complexité apparaît dans des problèmes dans lesquels on découpe répétitivement les données en deux parties, que l'on parcourt complètement ensuite.

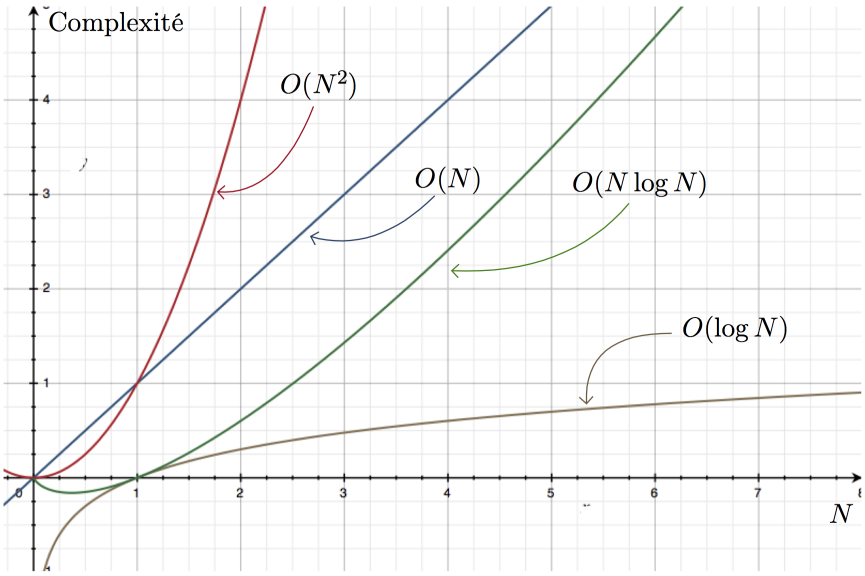
Les algorithmes de type *puissance*. Leur notation est de la forme $O(N^k)$ où k est la puissance.

Une *complexité quadratique* apparait, par exemple, lorsqu'on parcourt un tableau à deux dimensions.

Les algorithmes *exponentiels* ou *factoriels*. Leur notation est de la forme $O(e^N)$ ou $O(N!)$. *Ce sont les algorithmes les plus complexes. Le nombre de calculs augmente de façon exponentielle ou factorielle en fonction du nombre de données à traiter.* Un algorithme de complexité exponentielle traitera, dans le pire des cas, un ensemble de 10 données en effectuant 22 026 calculs ; un ensemble de 100 données en effectuant $2{,}688 \cdot 10^{43}$ calculs !!!

On dit généralement que les problèmes produisant ce type d'algorithmes sont « non calculables ».

On rencontre ce type d'algorithmes dans les problématiques liées à la programmation de fonctions humaines comme la vision, la reconnaissance des formes ou l'intelligence artificielle.



3. Cf. *Dichotomie*, partie 2.2.

4. Le *logarithme entier* d'un nombre x supérieur ou égal à 1 est *le nombre de fois qu'il faut le diviser par deux pour obtenir un nombre inférieur ou égal à 1.*

Complexité	**Durée pour $N = 10^6$**
Logarithmique : $O(\log N)$	10 ns
Linéaire : $O(N)$	1 ms
Quadratique : $O(N^2)$	1/4 h
Polynomiale : $O(N^k)$	30 ans si $k = 3$
Exponentielle : $O(2^n)$	plus de 10^{300000} milliards d'années

Tableau 1. *Odres de grandeur des durées d'exécution d'un problème de taille 10^6 sur un ordinateur à un milliard d'opérations par seconde (« Informatique pour tous en CPGE », éditions Eyrolles).*

2 Quelques méthodes de recherche

La recherche est une opérations fondamentale dans un programme qui gère un ensemble de données. La recherche traite deux informations distinctes : la *clé* et la *donnée*. *La clé est l'information utilisée dans la recherche pour localiser la donnée.*

2.1 La recherche séquentielle

La *recherche séquentielle* est une méthode élémentaire qui consiste à comparer la clé recherchée à toutes les autres clés. *Rechercher une valeur entière dans une liste d'entiers se résume à parcourir chaque élément de la liste et à vérifier s'il s'agit de l'élément recherché*. Dans cet exemple, la clé et la donnée recherchée sont confondues.

Remarque. La complexité algorithmique d'une recherche séquentielle est $O(N)$.

2.1.1 Algorithme itératif

La boucle utilisée dans une recherche séquentielle est une boucle `TANT QUE` avec un double critère d'arrêt : on s'arrête quand on a trouvé la donnée ou quand on a atteint la fin de la liste.

```
Fonction recherche_sequentielle_iterative(tab, nb, val_rech)
Déclarations
        Paramètre tab            Tableau d'Entiers
        Paramètres nb, val_rech  Entier
        Variable i               Entier
        Variable trouvé          Booléen
        Constante NON_TROUVÉ = -1
Début
        i ← 0
        trouvé ← FAUX
        # boucle de recherche
        Tant que ((NON trouvé) ET (i<nb)) Faire
                Si (tab[i] = val_rech) Alors
                        trouvé ← VRAI
                Sinon
                        i ← i + 1
```

```
                FinSi
        FinTantQue
        Si (trouvé = VRAI) Alors
                retourner i
        Sinon
                retourner NON_TROUVÉ
        FinSi
Fin
```

Remarque. Cet algorithme ne trouve que la première occurence de la valeur recherchée[5] (si elle est présente).

2.1.2 Algorithme récursif

```
Fonction recherche_sequentielle_recursive(tab, nb, val_rech, i)
Déclarations
        Paramètre tab               Tableau d'Entiers
        Paramètres nb, val_rech, i  Entier
        Constante NON_TROUVÉ = -1
Début
        Si i = nb Alors
                retourner NON_TROUVÉ
        Sinon Si (tab[i] = val_rech) Alors
                retourner i
        Sinon
                retourner recherche_sequentielle_recursive(tab, nb, val_rech, i+1)
        FinSi
Fin
```

2.2 La recherche dichotomique

Dès que le nombre de données devient important, la recherche séquentielle n'est plus envisageable. Sa complexité en $O(N)$ pénalise les autres traitements qui l'utilisent. Il faut diminuer le temps de traitement par l'emploi d'algorithmes plus performants, comme la *recherche dichotomique*. Cet algorithme est basé sur le principe de « *diviser pour résoudre* ». *On divise l'ensemble de recherche en deux sous-ensembles égaux. On détermine ensuite dans quel sous-ensemble doit se trouver la clé de recherche, puis on poursuit la recherche dans ce sous-ensemble.*

Avertissement. Le préalable à cette méthode de recherche est de disposer d'*un ensemble trié de données*[6], car la détermination du sous-ensemble dans lequel se poursuit la recherche se fait par comparaison entre la valeur recherchée et les valeurs de début et de fin du sous-ensemble.

Remarque. La division par deux de l'ensemble de données de recherche à chaque appel indique que sa complexité est en $O(\log N)$.

Exemple. Recherche dans un annuaire comprenant $66{\cdot}10^6$ entrées :

- $66{\cdot}10^6$ tests dans le pire des cas pour la recherche séquentielle ;

5. Cf. exercice 2 pour une amélioration du résultat.

6. La méthode de recherche séquentielle fonctionne que l'ensemble soit trié ou pas.

– $\log_2(66 \cdot 10^6) = 26$ tests dans le pire des cas pour la recherche dichotomique !!!

2.2.1 Algorithme récursif

```
Fonction recherche_dichotomique_recursive(tab, début, fin, val_rech)
Déclarations
        Paramètre tab                   Tableau d'Entiers
        Paramètres début, fin, val_rech  Entier
        Variable milieu, résultat       Entier
        Constante NON_TROUVÉ = -1
Début
        milieu ← (début + fin) DIVISION ENTIÈRE 2
        Si (val_rech = tab[milieu]) Alors
                résultat ← milieu
        Sinon Si (début >= fin) Alors
                résultat ← NON_TROUVÉ
        Sinon Si (val_rech < tab[milieu]) Alors
                résultat ← recherche_dichotomique_recursive(tab, début, milieu-1, val_rech)
        Sinon
                résultat ← recherche_dichotomique_recursive(tab, milieu+1 ,fin, val_rech)
        FinSi
        retourner résultat
Fin
```

Remarque. `début`, `fin` et `milieu` doivent être des entiers puisque ce sont des indices.

2.2.2 Algorithme itératif

```
Fonction recherche_dichotomique_iterative(tab, début, fin, val_rech)
Déclarations
        Paramètre tab                    Tableau d'Entiers
        Paramètres début, fin, val_rech  Entiers
        Variable milieu                  Entier
        Constante NON_TROUVÉ = -1
Début
        Tant que (début <= fin) Faire
                milieu ← (début + fin) DIVISION ENTIÈRE 2
                Si (val_rech = tab[milieu]) Alors
                        retourner milieu
                Sinon Si (val_rech < tab[milieu]) Alors
                        fin ← milieu - 1
                Sinon
                        début ← milieu + 1
                FinSi
        FinTantQue
        retourner NON_TROUVÉ
Fin
```

2.3 Utilisation de la recherche dichotomique en mathématiques

La méthode de la dichotomie peut être appliquée à la résolution de l'équation $f(x)=0$.

Théorème des valeurs intermédiaires. Le raisonnement à mettre en œuvre s'appuie sur le théorème des *valeurs intermédiaires* :

Soit $f\colon [a,b] \longrightarrow \mathbb{R}$ une fonction continue sur un segment.

Si $f(a)\cdot f(b) \leqslant 0$, alors il existe $\ell \in [a,b]$ tel que $f(\ell)=0$.

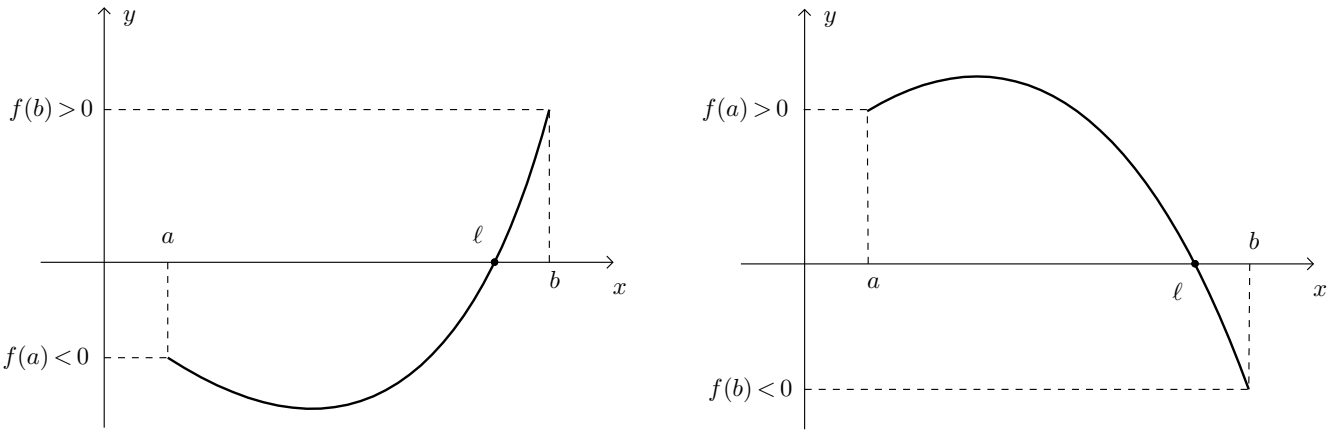


Figure 1. *Illustration du théorème des valeurs intermédiaires.*

2.3.1 Principe de résolution du problème

1. On choisit un intervalle $[a,b]$ dans lequel on pense que la fonction f s'annule *une seule fois* (il peut donc être nécessaire que cet intervalle soit petit).

2. **Étape 1.** On déterminer les coordonnées du *point milieu de l'intervalle* : $\frac{a+b}{2}$.

   a. Si $f(a)\cdot f\left(\frac{a+b}{2}\right) \leqslant 0$, le théorème des valeurs intermédiaires permet de conclure qu'il existe un point $c \in \left[a, \frac{a+b}{2}\right]$ tel que $f(c)=0$.

   b. Si $f(a)\cdot f\left(\frac{a+b}{2}\right) \geqslant 0$, le théorème des valeurs intermédiaires permet de conclure qu'il existe un point $c \in \left[\frac{a+b}{2}, b\right]$ tel que $f(c)=0$.

   On est donc parvenu, à l'issu de cette étape, à définir un intervalle dans lequel $f(x)=0$, de longueur moitié par rapport à l'intervalle initial.

3. **Étapes suivante.** On répète le processus mis en œuvre dans l'**Étape 1** pour l'intervalle $\left[a, \frac{a+b}{2}\right]$ ou $\left[\frac{a+b}{2}, b\right]$.

4. **Étapes suivantes.** On continue à répéter le processus jusqu'à ce que l'intervalle soit de longueur inférieure à une longueur choisie.

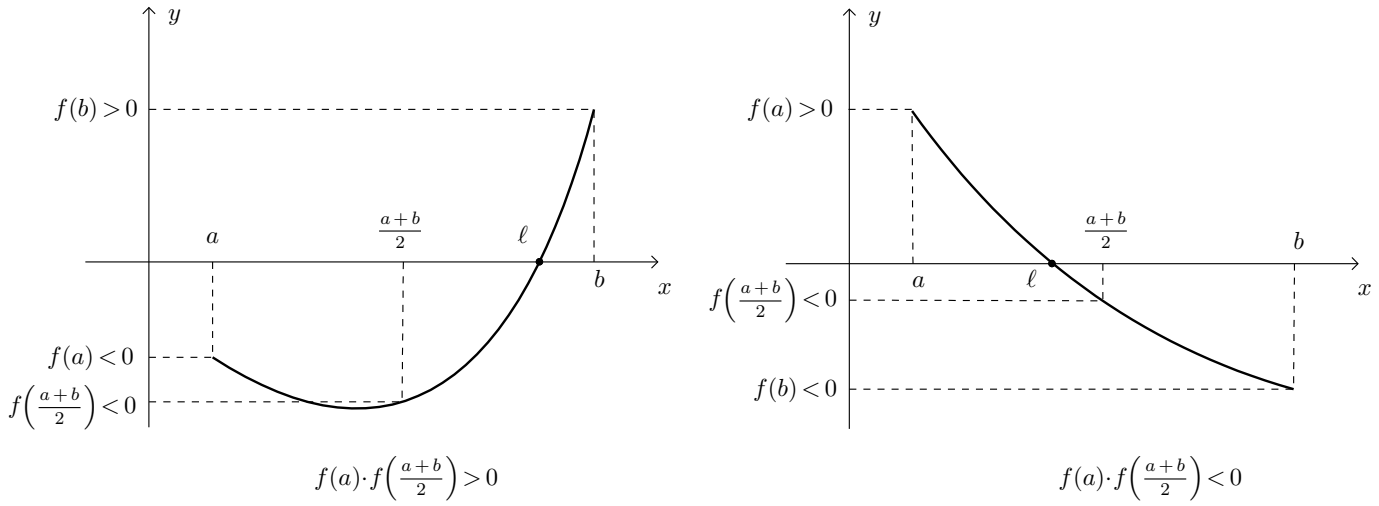


Figure 2. *Étape 1.*

2.3.2 La méthode converge-t-elle réellement ?

1. *Construction des suites (a_n) et (b_n) qui évoluent comme les bornes de l'intervalle, et (x_n) qui est un zéro de f pour chaque intervalle.*

   Au rang 0. On pose $a_0 = a$, $b_0 = b$. Comme $f(a)\cdot f(b) \leqslant 0$, il existe une solution, notée x_0, de l'équation $f(x) = 0$ dans l'intervalle $[a_0, b_0]$.

   Au rang 1. Si $f(a_0)\cdot f\left(\frac{a_0+b_0}{2}\right) \leqslant 0$, alors on pose $a_1 = a_0$ et $b_1 = \frac{a_0+b_0}{2} < b_0$, sinon on pose $a_1 = \frac{a_0+b_0}{2} > a_0$ et $b_1 = b_0$.

   Comme $f(a_1)\cdot f(b_1) \leqslant 0$, il existe une solution, notée x_1, de l'équation $f(x) = 0$ dans l'intervalle $[a_1, b_1]$.

   Au rang n. Si $f(a_n)\cdot f\left(\frac{a_n+b_n}{2}\right) \leqslant 0$, alors on pose $a_{n+1} = a_n$ et $b_{n+1} = \frac{a_n+b_n}{2} < b_n$, sinon on pose $a_{n+1} = \frac{a_n+b_n}{2} > a_n$ et $b_{n+1} = b_n$.

   Comme $f(a_{n+1})\cdot f(b_{n+1}) \leqslant 0$, il existe une solution, notée x_{n+1}, de l'équation $f(x) = 0$ dans l'intervalle $[a_{n+1}, b_{n+1}]$.

2. *Étude des suites construites* :

   Par construction :

   - $a_n \leqslant x_n \leqslant b_n$;
   - (a_n) est une suite croissante, (b_n) une suite décroissante ;

   donc la limite de $(b_n - a_n)$ tend vers 0 lorsque n tend vers $+\infty$: $\lim (b_n - a_n) \xrightarrow[n\to+\infty]{} 0$. *Les suites (a_n) et (b_n) sont adjacentes. Elles admettent donc une même limite, ℓ.*

   Par application du théorème des gendarmes, on peut aussi en conclure que $\lim (x_n) \xrightarrow[n\to+\infty]{} \ell$ et, comme f est une fonction continue, $f(\ell) = \lim_{n\to+\infty} f(x_n) = \lim_{n\to+\infty} 0 = 0$.

   En conclusion, les suites (a_n) et (b_n) tendent vers la valeur ℓ, solution de $f(x) = 0$.

Remarque. *En pratique, on arrête le processus dès que* $b_n - a_n = \frac{b-a}{2^n}$ *est inférieur à la précision souhaitée.*

2.3.3 Exemple : Approximation de $\sqrt{10}$

- On définit la fonction f définie par $f(x) = x^2 - 10$. f est une fonction continue sur $\mathbb{R}$ qui s'annule en $\sqrt{10}$ et $-\sqrt{10}$.
- On limite l'étude à l'intervalle $[3;4]$ puisque : $f(3) = 3^2 - 10 = -1 < 0$ et $f(4) = 4^2 - 10 = 6 > 0$, $\sqrt{10} \in [3;4]$.
- On écrit le processus itératif :

  → On pose $a_0 = 3$ et $b_0 = 4$. $f(a_0) \leqslant 0$ et $f(b_0) \geqslant 0$.

  $\frac{a_0+b_0}{2} = 3{,}5$ donc $f\left(\frac{a_0+b_0}{2}\right) = f(3{,}5) = 3{,}5^2 - 10 = 2{,}25 \geqslant 0$.

  $f(a_0) \cdot f\left(\frac{a_0+b_0}{2}\right) \leqslant 0$, on en déduit que $\sqrt{10}$ est dans l'intervalle $[3; 3{,}5]$.

  → On pose $a_1 = 3$ et $b_1 = 3{,}5$.

  $\frac{a_1+b_1}{2} = 3{,}25$ donc $f\left(\frac{a_1+b_1}{2}\right) = f(3{,}25) = 0{,}5625 \geqslant 0$.

  $f(a_1) \cdot f\left(\frac{a_1+b_1}{2}\right) \leqslant 0$, on en déduit que $\sqrt{10}$ est dans l'intervalle $[3; 3{,}25]$.

  → On pose $a_2 = 3$ et $b_2 = 3{,}25$.

  $\frac{a_2+b_2}{2} = 3{,}125$ donc $f\left(\frac{a_2+b_2}{2}\right) = f(3{,}125) = -0{,}23... \leqslant 0$.

  $f(b_2) \geqslant 0$, f s'annule sur l'intervalle $[3{,}125; 3{,}25]$.

  À ce stade, on a donc démontré que : $3{,}125 \leqslant \sqrt{10} \leqslant 3{,}25$[7].

2.3.4 Précision du résultat

La longueur de l'intervalle $[a_n, b_n]$ est $\frac{b-a}{2^n}$ puisqu'il a été nécessaire de découper n fois l'intervalle de départ en deux pour parvenir à cet intervalle.

On peut alors rechercher *le nombre d'itérations nécessaires pour atteindre une précision choisie* lors de la phase préliminaire de l'étude : il suffit de rechercher le nombre n tel que $\frac{b-a}{2^n} \leqslant 10^{-N}$:

$$\begin{aligned} (b-a)\,10^N &\leqslant 2^n \\ \log(b-a) + \log(10^N) &\leqslant \log(2^n) \\ \log(b-a) + N &\leqslant n\log(2) \\ n &\geqslant \frac{\log(b-a)+N}{\log(2)} \end{aligned}$$

Exemple. Si $b - a \leqslant 1$, une précision de :

- 10^{-10} ($\sim$ 10 décimales) nécessite 34 itérations ;

7. Il faudrait bien sûr poursuivre le processus pour une plus grande précision.

- 10^{-100} ($\sim$ 100 décimales) nécessite 333 itérations ;
- 10^{-1000} ($\sim$ 1000 décimales) nécessite 3322 itérations.

Ajouter une décimale nécessite donc 3 ou 4 itérations supplémentaires.

3 Exercices

Exercice 1. Écrire un programme qui, dans un premier temps, crée une liste comportant 1 000 nombres entiers (compris entre 1 et 10 000) tirés au hasard et qui, ensuite, implémente les algorithmes de recherche séquentielle pour vérifier si un nombre entier, entré par l'utilisateur, est présent dans la liste. Le retour doit correspondre à la position dans la liste du nombre (retourner -1 si le nombre n'est pas présent).

Exercice 2. Écrire un programme inspiré du programme précédent, qui retourne *toutes les positions* du nombre recherché (s'il est présent plusieurs fois).

Exercice 3. Écrire un programme qui, dans un premier temps, crée une liste comportant 1 000 nombres entiers *triés* (formule pour remplir la liste : $\text{tab}[i] = 3 \times i + 2$) et qui, ensuite, implémente l'algorithme de recherche dichotomique récursif, pour vérifier si un nombre entier, entré par l'utilisateur, est présent dans la liste. Le retour doit correspondre à la position dans la liste du nombre (retourner -1 si le nombre n'est pas présent).

Exercice 4. Même exercice que l'exercice précédent mais cette fois le programme doit implémenter l'algorithme de recherche dichotomique itérative.

Exercice 5. Écrire un programme qui retourne la valeur approchée de la racine cubique de n'importe quel nombre.

Exercice 6. Écrire un programme de devinette : l'utilisateur choisit un nombre au hasard (compris entre 1 et 50) et l'ordinateur doit découvrir ce nombre. Faire en sorte que le nombre de tentatives de l'ordinateur lui permette de trouver la bonne valeur à coup sur (choisir de façon pertinente ce nombre).