Chapitre 1

Algorithmes de tri et de recherche dichotomique

1.1 Trier

Durant la seconde guerre mondiale, Betty Holberton fut embauchée en tant que calculatrice, et fut vite choisie pour être l'une des six programmatrices de l'ENIAC, le premier ordinateur entièrement électronique. Elle y développe notamment la première routine de tri.

Le tri est un outil essentiel de l'informatique. Il existe de nombreux algorithmes, plus ou moins performants et plus ou moins complexes à mettre en œuvre mais ils sont fondamentaux pour gérer de grandes quantités de données et y accéder dans des délais très brefs.

La fonction d'un algorithme de tri est d'organiser un ensemble de données selon un ordre déterminé. Généralement les données à trier sont des nombres ou des chaînes de caractères qui peuvent être triés numériquement dans l'ordre croissant ou alphabétiquement.

Pour étudier les algorithmes de tri, on va travailler sur des tableaux de nombres entiers que l'on souhaite trier dans l'ordre croissant.

1.2 Tri par insertion

1.2.1 Principe

Le **tri par insertion** s'inspire de la manière dont la plupart des gens trient une poignée de cartes. On commence avec une main gauche vide et les cartes face contre table. On retire ensuite du paquet une carte à la fois, pour l'insérer à sa bonne place dans la main gauche. Pour trouver cette bonne place, on la compare avec chacune des cartes déjà présentes dans la main de droite à gauche.

1.2.2 Algorithme

Algorithme 1 : Tri par insertion

Données : *tableau*
$N \leftarrow taille(tableau)$
Pour *numElement allant de 1 à* $N-1$ **Faire**
  $element \leftarrow tableau[numElement]$
  $position \leftarrow numElement$
  Tant que $position > 0$ ***et*** $tableau[position-1] > element$ **Faire**
    $tableau[position] \leftarrow tableau[position-1]$
    $position \leftarrow position - 1$
  $tableau[position] \leftarrow element$

Exemple : appliquons cet algorithme au tableau $T = [8, 5, 1, 9, 7]$. On a $N = 5$.

Pour $numElement = 1$ **:** on a

- $element = 5$;
- $position = 1$;
- on exécute la boucle **Tant que** :

$position$	$T[position-1]$	$b_1 = (position > 0)$	$b_2 = (T[position-1] > 5)$	$b_1 \wedge b_2$	T
1	8	1	1	1	$[8,8,1,9,7]$
0	7	0	1	0	$[8,8,1,9,7]$

  l'une des deux conditions n'étant pas respectées, la boucle s'arrête ;
- on a finalement $T[position] = T[0] = element = 5$. Le tableau vaut donc à présent $T = [5, 8, 1, 9, 7]$.

Pour $numElement = 2$ **:** on a

- $element = 1$;
- $position = 2$;
- on exécute la boucle **Tant que** :

$position$	$T[position-1]$	$b_1 = (position > 0)$	$b_2 = (T[position-1] > 1)$	$b_1 \wedge b_2$	T
2	8	1	1	1	$[5,8,8,9,7]$
1	5	1	1	1	$[5,5,8,9,7]$
0	7	0	1	0	$[5,5,8,9,7]$

  l'une les deux conditions n'étant pas respectées, la boucle s'arrête ;
- on a finalement $T[position] = T[0] = element = 1$. Le tableau vaut donc à présent $T = [1, 5, 8, 9, 7]$.

Pour $numElement = 3$ **:** on a

- $element = 9$;
- $position = 3$;
- on exécute la boucle **Tant que** :

$position$	$T[position-1]$	$b_1 = (position > 0)$	$b_2 = (T[position-1] > 1)$	$b_1 \wedge b_2$	T
3	9	1	0	0	$[1,5,8,9,7]$

l'une les deux conditions n'étant pas respectées, la boucle ne démarre pas et le tableau n'est pas modifié.

Pour $numElement = 4$ **:** on a

- $element = 7$;
- $position = 4$;
- on exécute la boucle **Tant que** :

$position$	$T[position-1]$	$b_1 = (position > 0)$	$b_2 = (T[position-1] > 7)$	$b_1 \wedge b_2$	T
4	9	1	1	1	$[1,5,8,9,9]$
3	8	1	1	1	$[1,5,8,8,9]$
2	5	1	0	0	$[1,5,8,8,9]$

l'une les deux conditions n'étant pas respectées, la boucle s'arrête ;

- on a finalement $T[position] = T[0] = element = 7$. Le tableau vaut donc à présent $T = [1,5,7,8,9]$. Le tableau est finalement trié.

1.2.3 Terminaison de l'algorithme

Pour prouver que l'algorithme se termine, il faut étudier les deux boucles.

Boucle Pour : c'est une boucle bornée donc le nombre de passages est déterminé et fini.

Boucle Tant que : les valeurs prises constituent une suite d'entiers décroissante incluse dans la suite d'entiers de *position* à 1. Il y a donc au plus *position* passages dans cette boucle **Tant que**.

Les deux boucles se terminent donc l'algorithme se termine.

Remarque : si on note N la taille de notre tableau, on a N répétitions de la boucle **Pour** et au plus N répétitions de la boucle **Tant que** au sein de celle-ci. Ce qui fait en tout $N \times N = N^2$ répétitions de boucles. On dira que le coût de cet algorithme est dans le pire des cas quadratique.

1.2.4 Correction de l'algorithme

Définition 1.1. *Un **invariant de boucle** est une propriété qui est satisfaite avant et après chaque répétition de boucle.*

Proposition 1.1. *« Le tableau $T_n = [a_0, \dots, a_n]$ est trié pour tout $n \in [\![1\,;N-1]\!]$ » est un invariant de la boucle **Pour** de l'algorithme 1.*

Remarque : on a utiliser un raisonnement de type **récurrence** afin de montrer que cette proposition est vraie. Ce type de raisonnement repose sur deux chose : une **initialisation** : une proposition P_0 doit être vraie au rang 0 ; et une **hérédité** : si la proposition est vraie au rang k alors elle l'est au rang $k+1$, $P_k \implies P_{k+1}$. L'hérédité étend la véracité la proposition initiale à tous les autres rangs.

Démonstration.

Initialisation : Avant le premier passage dans la boucle, l'élément $T[0]$ est seul donc il est rangé dans l'ordre croissant. Donc le tableau T_0 est trié.

Hérédité : Supposons que pour $position = k$ quelconque, le tableau T_k soit trié. Lors du passage de $T_{k+1} = T_k + T[k]$ dans la boucle **Tant que** on a deux possibilités.

1. Si $T[k] \geq T_k[k-1]$, $T[k]$ reste à sa place et T_{k+1} est trié.
2. Sinon, on a $T[k] \leftarrow T_k[k-1]$ et on réitère la boucle **Tant que** avec les T_{k-j} pour $j \in [\![0\,;k]\!]$. On obtient en sortir de boucle **Tant que** un tableau de la forme

$$T_{k+1} = [T_k[0], \dots, T_k[position-1], T_k[position], \underbrace{T_k[position]}_{=T_{k+1}[position+1]}, \dots, T_k[k-1]],$$

avec donc $T_{k+1}[position+1] = T_k[position]$ par exécution de la boucle. Or par hypothèse de récurrence, on avait

$$T_k = [T_{k+1}[0], \dots, T_{k+1}[position-1], T_k[position], \dots, T_k[k-1]]$$

de trié. Ainsi T_{k+1} l'est aussi. Comme par condition d'exécution de la boucle **Tant que**, on a $T_k[j] \leq T_{k+1}[k]$ pour tout $j \in [\![0\,;position]\!]$ et $T_{k+1}[k] \leq T_k[j]$ pour tout $j \in [\![position\,;k]\!]$, on en déduit qu'en effectuant $T_{k+1}[position] \leftarrow T_{k+1}[k]$, on obtient le tableau trié

$$T_{k+1} = [T_k[0], \dots, T_k[position-1], T_{k+1}[k], T_k[position], \dots, T_k[k-1]].$$

Par récurrence, on en déduit que la proposition est vraie pour $T_{N-1} = T$. On en déduit que T est trié à la fin de l'algorithme. □

1.3 Tri par sélection

1.3.1 Principe

On recherche le plus petit élément m du tableau et on le place au premier indice du tableau. On réitère ensuite avec le tableau constitué des éléments suivants et ainsi de suite.

1.3.2 Algorithme

Algorithme 2 : Tri par sélection

```
Données : tableau
N ← taille(tableau)
Pour numElement allant de 0 à N − 2 Faire
    positionMin ← numElement
    Pour position allant de numElement + 1 à N − 1 Faire
        Si tableau[position] < tableau[positionMin] Alors
            positionMin ← position
    Si positionMin ≠ numElement Alors
        Échanger tableau[numElement] et tableau[positionMin]
```

Remarque : il est aussi possible de procéder avec le maximum à la place du minimum voire les deux en mêmes temps.

Exemple : appliquons cet algorithme au tableau $T = [8, 5, 1, 9, 7]$. On a $N = 5$.

Pour $numElement = 0$ **:** on a

- $positionMin = 0$;
- on exécute la boucle **Pour** :

$position$	$T[position] < T[positionMin]$	$positionMin$
1	$(5 < 8) = 1$	1
2	$(1 < 5) = 1$	2
3	$(9 < 1) = 0$	2
4	$(7 < 1) = 0$	2

- Comme $positionMin = 2 \neq 0 = numElement$, on échange $T[2]$ et $T[0]$. Ainsi on a présent $T = [1, 5, 8, 9, 7]$.

Pour $numElement = 1$ **:** on a

— $positionMin = 1$;

— on exécute la boucle **Pour** :

$position$	$T[position] < T[positionMin]$	$positionMin$
2	$(8 < 5) = 0$	1
3	$(9 < 5) = 0$	1
4	$(7 < 5) = 0$	1

— Pas d'échange à faire ici, on a effet $positionMin = numElement$. T reste donc identique.

Pour $numElement = 2$ **:** on a

— $positionMin = 2$;

— on exécute la boucle **Pour** :

$position$	$T[position] < T[positionMin]$	$positionMin$
3	$(9 < 8) = 0$	1
4	$(7 < 8) = 1$	4

— Comme $positionMin = 4 \neq 2 = numElement$, on échange $T[4]$ et $T[2]$. Ainsi on a présent $T = [1, 5, 7, 9, 8]$.

Pour $numElement = 3$ **:** on a

— $positionMin = 3$;

— on exécute la boucle **Pour** :

$position$	$T[position] < T[positionMin]$	$positionMin$
4	$(8 < 9) = 1$	4

— Comme $positionMin = 4 \neq 3 = numElement$, on échange $T[4]$ et $T[3]$. Ainsi on a présent $T = [1, 5, 7, 8, 9]$. Le tableau est maintenant trié.

1.3.3 Terminaison de l'algorithme

Comme les deux boucles de l'algorithme sont des boucles **Pour**, celles-ci sont bornées et donc l'algorithme se termine.

Comme pour le tri par insertion, si on note N la taille de notre tableau, on a N répétitions de la boucle **Pour** et au plus N répétitions de la deuxième boucle **Pour** au sein de celle-ci. Ce qui fait au maximum $N \times N = N^2$ répétitions de boucles. Le coût de cet algorithme est dans le pire des cas quadratique lui aussi.

1.3.4 Correction de l'algorithme

Proposition 1.2. *Soit $T = [a_0, \dots, a_{N-1}]$. « a_n est le minimum du tableau $T_n = [a_n, \dots, a_{N-1}]$ pour tout $n \in [\![0\,;N-1]\!]$ » est un invariant de la boucle **Pour** de l'algorithme 2.*

Démonstration. Soit $T = [a_0, \dots, a_{N-1}]$. On note $T_n = [a_n, \dots, a_{N-1}]$ pour tout $n \in [\![1\,;N-1]\!]$. L'ensemble $A_n = \{a_n, \dots, a_{N-1}\}$ étant une partie finie de $\mathbb{R}$, elle admet un plus petit élément : $\min(T_n)$. Si $a_n = \min(T_n)$, il reste à sa place, sinon il est échangé avec $\min(T_n)$. Dans les deux cas, on a à la fin de l'exécution de la boucle **Pour** :

$$a_n = T_n[n] \leq T_n[i],\ \forall i \in [\![n\,;N-1]\!]\,.$$

□

Soit $T = [a_0, \dots, a_{N-1}]$. À la fin de la première exécution de la boucle **Pour**, on a donc

$$a_0 \leq a_i, \forall i \in [\![1\,;N-1]\!]\,. \tag{1.1}$$

À la fin de la seconde exécution de la boucle **Pour**, on a donc

$$a_1 \leq a_i, \forall i \in [\![2\,;N-1]\!]\,. \tag{1.2}$$

D'après (1.1), on a donc $a_0 \leq a_1$. Par récurrence immédiate, on obtient à la fin de la dernière exécution de la boucle **Pour** que

$$a_0 \leq a_1 \leq \cdots \leq a_{N-1},$$

autrement dit que T est trié.

1.4 Recherche dichotomique

1.4.1 Principe

On considère un tableau trié dans l'ordre croissant. La recherche dichotomique consiste à y rechercher un élément ; si cet élément y est présent, elle renvoie sa position, sinon elle indique qu'il est absent du tableau. Elle procède par comparaison en déterminant quelle partie du tableau est susceptible de contenir notre élément en traitant des sous parties du tableau de plus en plus petites.

1.4.2 Algorithme

Algorithme 3 : Recherche dichotomique

Données : *tableau*, *element*
$bInf \leftarrow 0$
$bSup \leftarrow taille(tableau)$
Tant que $1 < bSup - bInf$ **Faire**
  $milieu \leftarrow (bInf + bSup)//2$
  Si $element < tableau[milieu]$ **Alors**
    $bSup \leftarrow milieu$
  Sinon
    $bInf \leftarrow milieu$
Si $tableau[bInf] = element$ **Alors**
  Sorties : $bInf$
Sinon
  Sorties : L'élément n'est pas présent dans le tableau.

Exemples : appliquons cet algorithme au tableau $T = [1, 3, 4, 6, 7, 9, 10]$.

Recherche de 9 :

$bInf$	$bSup$	$1 < bSup - bInf$	$milieu$	$9 < T[milieu]$
0	6	$(0 < 6 - 1) = 1$	3	$(9 < 6 = T[3]) = 0$
3	6	$(3 < 6 - 1) = 1$	4	$(9 < 7 = T[4]) = 0$
4	6	$(4 < 6 - 1) = 1$	5	$(9 < 9 = T[5]) = 0$
5	6	$(5 < 6 - 1) = 0$		

Recherche de 4 :

$bInf$	$bSup$	$1 < bSup - bInf$	$milieu$	$4 < T[milieu]$
0	6	$(0 < 6 - 1) = 1$	3	$(4 < 6 = T[3]) = 1$
0	3	$(0 < 3 - 1) = 1$	1	$(4 < 3 = T[1]) = 0$
1	3	$(1 < 3 - 1) = 1$	2	$(4 < 4 = T[2]) = 0$
2	3	$(2 < 3 - 1) = 0$		

Recherche de 8 :

$bInf$	$bSup$	$1 < bSup - bInf$	$milieu$	$4 < T[milieu]$
0	6	$(0 < 6 - 1) = 1$	3	$(8 < 6 = T[3]) = 0$
3	6	$(3 < 6 - 1) = 1$	4	$(8 < 7 = T[4]) = 0$
4	6	$(4 < 6 - 1) = 1$	5	$(8 < 9 = T[5]) = 1$
4	5	$(4 < 5 - 1) = 0$		

1.4.3 Terminaison de l'algorithme

Il s'agit de prouver que la boucle **Tant que** finit par s'arrêter, autrement dit que sa condition d'exécution devient fausse.

On note a_n et b_n les valeurs respectives de $bInf$ et $bSup$ à la n-ième itération de la boucle **Tant que**. On pose l_n la longueur de l'intervalle $[a_n\,;b_n]$; autrement dit $l_n = b_n - a_n$. On a donc avant la première itération $l_0 = b_0 - a_0$. Soit $n \in \mathbb{N}$, à la $n+1$-ième itération de la boucle **Tant que**, on a deux cas possibles :

1er cas : $bSup \leftarrow milieu$ **;** on a alors $a_{n+1} = a_n$ et $b_{n+1} = \dfrac{a_n + b_n}{2}$, d'où

$$l_{n+1} = b_{n+1} - a_{n+1} = \frac{a_n + b_n}{2} - a_n = \frac{b_n - a_n}{2} = \frac{1}{2} l_n.$$

2e cas : $bInf \leftarrow milieu$ **;** on a alors $a_{n+1} = \dfrac{a_n + b_n}{2}$ et $b_{n+1} = b_n$, d'où

$$l_{n+1} = b_{n+1} - a_{n+1} = b_n - \frac{a_n + b_n}{2} = \frac{b_n - a_n}{2} = \frac{1}{2} l_n.$$

Dans tous les cas, nous avons donc $l_{n+1} = \dfrac{1}{2} l_n$, autrement dit est une suite géométrique de raison $\dfrac{1}{2}$ et premier terme l_0. Comme la raison de la suite appartient à $]0\,;1[$, on en déduit que $l_n \to 0$ lorsque $n \to +\infty$. Il existe donc $n \in \mathbb{N}$ tel que

$$l_n < 1 \iff b_n - a_n < 1.$$

Il existe donc une itération de la boucle **Tant que** pour laquelle sa condition d'exécution devient fausse : $1 < bSup - bInf$. Cette boucle prend alors fin et l'algorithme se termine.

1.4.4 Correction de l'algorithme

Proposition 1.3. *Soient $T = [bInf, \ldots, bSup]$ un tableau trié et $x \in [T[bInf]\,;T[bSup]]$. On note a_n et b_n les valeurs de $bInf$ et $bSup$ avant la n-ième itération de la boucle* ***Tant que****. Alors « $x \in [T[a_n]\,;T[b_n]]$ » est un invariant de la boucle* ***Tant que*** *de l'algorithme 3.*

Démonstration. On procède par récurrence.

Initialisation : Par hypothèse, on a $x \in [T[bInf]\,;T[bSup]] = [T[a_1]\,;T[b_1]]$. La propriété est donc vraie pour $n = 1$.

Hérédité : On suppose que $x \in [T[a_n]\,;T(b_n]]$, montrons que $x \in [T[a_{n+1}]\,;T[b_{n+1}]]$. Lors de l'itération de la boucle **Tant que**, si on a

$$T\left[\left\lfloor \frac{a_n + b_n}{2} \right\rfloor\right] \leq x,$$

alors on pose $a_{n+1} = \left\lfloor \dfrac{a_n + b_n}{2} \right\rfloor$ et donc

$$T[a_{n+1}] \leq x \leq T[b_{n+1}] = T[b_n],$$

par hypothèse de récurrence, autrement dit $x \in [T[a_{n+1}]\,;T[b_{n+1}]]$. Sinon on pose $b_{n+1} = \left\lfloor \dfrac{a_n + b_n}{2} \right\rfloor$ et à nouveau par hypothèse de récurrence :

$$T[a_n] = T[a_{n+1}] \leq x \leq T[b_{n+1}],$$

et donc $x \in [T[a_{n+1}]\,;T[b_{n+1}]]$.

□

Si x est l'élément recherché et qu'il vérifie $x \in [T[bInf]\,;T[bSup]]$ avant l'exécution de la boucle **Tant que**, alors x appartient encore à cet intervalle avant la dernière itération de la boucle. À la fin de la boucle **Tant que**, $[T[bInf]\,;T[bSup]]$ est réduit à un seul élément car $bInf$ et $bSup$ sont des entiers et que $bInf \geq bSup - 1$. Soit ce dernier élément est x et il est trouvé, soit il ne l'est pas et on en déduit que x n'est pas dans le tableau.

1.5 Exercices

1.5.1 Démarrage

Exercice 1.1. Trier les tableaux suivants $T_1 = [4, 5, 1, 3]$ et $T_2 = [6, 1, 0, 4, 7]$ en exécutant à la main les algorithmes de tri par :

1. insertion ;
2. sélection.

Exercice 1.2. Exécuter l'algorithme de recherche par dichotomie sur le tableaux triés $[0, 4, 5, 8, 11, 13, 20]$ pour les valeurs suivantes :

1. 3 ;
2. 12 ;
3. 4.

1.5.2 Approfondissement

Exercice 1.3. [Trié ?] Écrire une fonction déterminant si un tableau est trié ou non. Elle renverra le résultat sous la forme d'un booléen. Quelle est la complexité de l'algorithme associé à cette fonction ?

Exercice 1.4. [Tri par insertion]

1. Écrire un programme Python effectuant un tri par insertion.
2. À partir du programme précédent, écrire un programme triant un tableau dans l'ordre décroissant sans créer un autre tableau.

Exercice 1.5. [Tri par sélection] Écrire un programme Python effectuant un tri par sélection.

Exercice 1.6. [Recherche dichotomique] Écrire un programme Python effectuant une recherche dichotomique sur un tableau trié. On ajoutera un test pour déterminer si la valeur recherchée appartient bien à l'intervalle formé par les bornes du tableau.

Exercice 1.7. [Médiane, quartiles...] Programmer des fonctions déterminant les médiane, quartiles, déciles et centiles d'une liste de valeurs. Les fonctions s'assureront qu'il est pertinent de calculer ces quantités.

1.5.3 Entraînement

Exercice 1.8. Trier les tableaux suivants $T_1 = [8, 2, 4, 5]$ et $T_2 = [5, 6, 3, 4, 0]$ en exécutant à la main les algorithmes de tri par :

1. insertion ;
2. sélection.

Exercice 1.9. Exécuter l'algorithme de recherche par dichotomie sur le tableaux triés $[-3, -1, 0, 5, 9, 10]$ pour les valeurs suivantes :

1. -1 ;
2. 9 ;
3. -2.