Chapitre 7

Algorithmes appliqués à des intervalles
Dichotomie et intégration numérique

« Le mystère crée l'émerveillement et l'émerveillement
est la base du désir de l'humanité de comprendre.»
Neil Armstrong

Objectifs du chapitre : (extrait du Bulletin officiel 2013)
2. Algorithmique et programmation I
Seconde partie du semestre 1

Contenus	**Précisions et commentaire**
Recherche par dichotomie dans un tableau trié.	Les questions de précision du calcul sont en lien avec la partie 1.b.
Recherche par dichotomie du zéro d'une fonction continue et monotone.	Les questions de précision du calcul sont en lien avec la partie 1.b
Méthodes des rectangles et des trapèzes pour le calcul approché d'une intégrale sur un segment.	

Les principales capacités développées dans cette partie de la formation sont :

- comprendre un algorithme et expliquer ce qu'il fait,
- modifier un algorithme existant pour obtenir un résultat différent,
- concevoir un algorithme répondant à un problème précisément posé,
- expliquer le fonctionnement d'un algorithme,
- écrire des instructions conditionnelles avec alternatives, éventuellement imbriquées,
- justifier qu'une itération (ou boucle) produit l'effet attendu au moyen d'un invariant,
- démontrer qu'une boucle se termine effectivement,
- s'interroger sur l'efficacité algorithmique temporelle d'un algorithme.

1) Introduction

Cette nouvelle partie introduit de nouveaux algorithmes simples et universels et va permettre d'aborder une utilisation de Python dans le cadre de la résolution de problèmes classiques en mathématiques et qui se retrouvent fréquemment dans les calculs en sciences. Nous allons surtout nous focaliser sur les intervalles et la manière de les découper. Le mot intervalle doit être pris au sens large. Une liste comportant des valeurs triées par ordre croissante contient un nombre discret de valeur dans un intervalle donné.

2) La dichotomie

a) Principe général

Définition :
La **dichotomie** d'un ensemble consiste en un partage de celui-ci en deux sous-ensembles distincts.
Utiliser la dichotomie dans un algorithme s'inspire du principe « diviser pour mieux régner ». Cela signifie qu'en divisant un intervalle, on se retrouve avec deux parties plus petites, donc plus faciles à analyser. D'autant que rien n'interdit ensuite de continuer la division…

En informatique, la division en deux parties distinctes nécessite juste deux petites précautions :

- **Première précaution :**

Si l'on travaille sur des listes ou des tableaux, l'indice des cases doit nécessairement rester entier. Il conviendra donc de choisir la division euclidienne pour appliquer la dichotomie.

Exemple :

```
>>> liste=[2,4,6]
>>> m=len(liste)//2
>>> m
1
>>> liste[m]
4
>>> m=len(liste)/2
>>> m
1.5
>>> liste[m]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not float
>>>
```

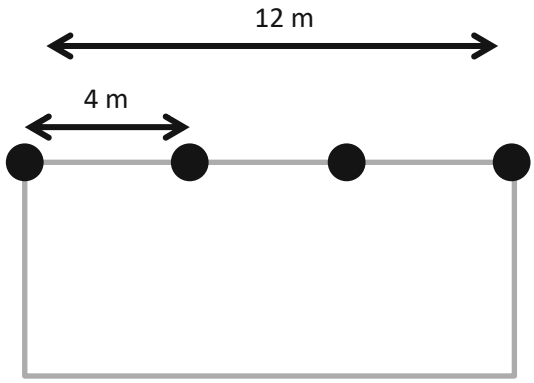
● **Deuxième précaution :**
Si l'on souhaite faire une recherche dichotomique dans un tableau, il faut veiller à ce que ce tableau soit déjà trié par ordre croissant ou décroissant, peu importe.

Conclusion :
Le découpage d'un problème en problèmes plus petits, donc plus simples, est une technique très utile et souvent très efficace.

b) Le découpage en intervalles réguliers

On peut découper un intervalle en plusieurs parties en définissant dès le départ le nombre d'intervalles souhaité.

Activité 1 : Planter des piquets
On a un rectangle de 12 m de large et l'on souhaite planter des piquets tout le long.
On souhaite découper cette longueur en 3 intervalles, alors chacun de ces intervalles fera 4 m. Au passage, on remarquera que l'on aura un piquet de plus que le nombre d'intervalle.



Mathématiquement, sur un axe (O, x), $[a; b]$ sera divisé en n intervalles de longueur h grâce à la formule :

$$h = \frac{(b-a)}{n}$$

En reprenant le raisonnement des piquets, on aura donc créé n intervalles et $n+1$ points.

3) Recherche dans un tableau trié par dichotomie

a) Principe de l'algorithme

➢ **Conception de l'algorithme de recherche :**
On va chercher un élément dans un tableau de taille n. En Python, on modélise un tableau par une liste par exemple mais peu importe pour le moment, on cherche un algorithme général. L'idée est de diviser successivement ce tableau et de

comparer la valeur de l'élément de la case du milieu à la valeur recherchée. Si l'élément recherché a une valeur supérieure à celui de la case du milieu, alors on prendra pour notre nouvelle recherche uniquement la partie droite du tableau qui a été divisé. Si l'élément cherché a une valeur inférieure à l'élément du milieu, alors on prendra la partie gauche du tableau pour la nouvelle recherche. On remarque alors que l'on peut appliquer notre nouvelle recherche sur un tableau deux fois plus petit et ainsi de suite. La recherche sera terminée lorsque le tableau n'aura plus qu'une seule case : ce sera alors celle de l'élément cherché s'il est dans le tableau.

Remarque :
La séparation en deux parties d'un tableau de nombre impair pour les indices de cases n'est pas un problème. On aura simplement une des deux parties avec une case de plus que l'autre mais cela n'affecte en rien la recherche du résultat comme on le verra dans l'exemple ci-après.

➢ **Pseudo-code :**

```
fonction rechercheDichotomique(tab,elt):
      g ← 0
      d ← longueur(tab)-1
    ①tant que g ≤ d boucler:
           m ← (g+d)//2
          ②si tab[m]=elt alors:
                retourne VRAI
           fin si
          ③si tab[m]<elt alors:
                g ← m+1
          ④sinon:
                d ← m-1
           fin si
      fin tant que
      retourne FAUX
fin fonction
```

l'algorithme ne fonctionne que sur un tableau déjà trié

élément à rechercher dans le tableau

division entière

l'élément est bien présent dans le tableau

Cela n'est possible que parce que le tableau est trié

Puisque l'élément n'est pas à l'indice m, autant prendre le point de départ du nouveau parcours du tableau l'indice m+1

Puisque l'élément n'est pas à l'indice m, autant prendre le point de départ du nouveau parcours du tableau l'indice m–1

si cette instruction est exécutée, c'est que l'élément n'a pas pu être trouvé ou que la longueur du tableau est nulle

Une variante consiste à renvoyer l'indice de la case qui contient l'élément cherché et si l'élément n'est pas dans le tableau, aucune valeur n'est renvoyée :

```
fonction rechercheDichotomique(tab,elt):
      g ← 0
      d ← longueur(tab)
    ①tant que g ≤ d boucler:
          m ← (g+d)//2
         ②si tab[m]=elt alors:
               retourne m
          fin si
         ③si tab[m]<elt alors:
               g ← m+1
         ④sinon:
               d ← m-1
          fin si
      fin tant que
      retourne RIEN
fin fonction
```

Remarques :

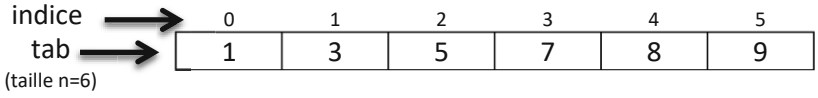
- En Python RIEN se traduit par None.
- Ici, on ne crée pas deux nouveaux tableaux deux fois plus petits à chaque tour de boucle. On ne divise que l'étendue des indices à parcourir. C'est donc un algorithme qui ne consomme pas plus de place mémoire que celle déjà utilisée pour stocker le tableau initial.

➢ **Exécution sur un exemple :**

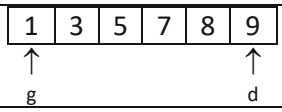
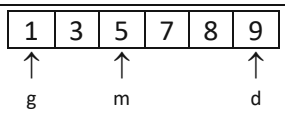
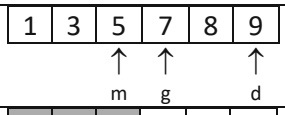
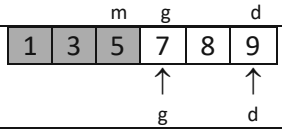
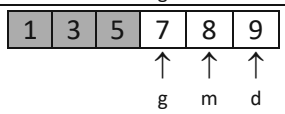
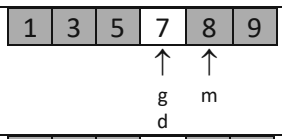
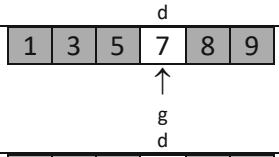
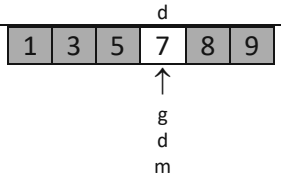
Il est important de s'entraîner à exécuter pas à pas un programme sans l'aide de l'ordinateur. Voici un exemple concret.

On va se baser sur le programme qui renvoie l'indice de la case de l'élément cherché et RIEN si l'élément n'est pas dans le tableau.

Tableau initial :

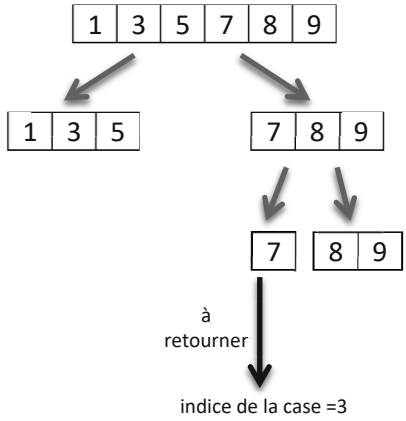


Nous cherchons l'indice d'un élément de valeur 7, s'il existe.

partie du code exécuté	élément cherché (elt)	commentaires	tableau tab
début de la fonction	7	• g ← 0 • d ← 5	
①	7	g ≤ d? ⇒ 0 ≤ 5 est VRAI	
	7	• m ← (5+0)//2 Donc m ← 5//2 ⇒ m ← 2	
②	7	tab[2]=7? ⇒ 5 = 7 est FAUX	
③	7	tab[2]<7? ⇒ 5<7 est VRAI	
	7	• g ← 3 d reste à 5	
①	7	g ≤ d? ⇒ 3 ≤ 5 est VRAI	
	7	• m ← (5+3)//2 Donc m ← 8//4 ⇒ m ← 4	
②	7	tab[4]=7? ⇒ 8 = 7 est FAUX	
③	7	tab[4]<7? ⇒ 8<7 est FAUX	
④	7	• d ← 3 g reste à 3	
①	7	g ≤ d? ⇒ 3 ≤ 3 est VRAI	
	7	• m ← (3+3)//2 Donc m ← 6//2 ⇒ m ← 3	

②	7	`tab[3]=7?` $\Rightarrow$ `7 = 7 est VRAI`	
	7	`retourne 3`	

La recherche dans le tableau peut se voir ainsi :



b) Complexité de l'algorithme

Le calcul de la complexité peut se faire de deux façons équivalentes ici. On va écrire les deux afin de réviser les notions vues au chapitre précédent.

➢ **1re façon :**
On a n=`longueur(tab)`. Il est fortement conseillé, pour éviter les confusions, de n'utiliser la lettre n que pour la mesure du nombre de données, ce qui est bien dans ce cas la longueur du tableau puisque chaque case représente une donnée.
Trouver la complexité, c'est trouver le nombre d'itérations que va effectuer la boucle `tant que`. Cela se comprend facilement car si cette boucle possède par exemple 5 instructions élémentaires, alors sa complexité serait alors $5 \times n$, ce qui serait suffisant pour trouver sa complexité en $O(n)$ dans l'exemple évoqué.
$\rightarrow$ Il nous faut donc évaluer le nombre de tour de boucle effectuée dans le `tant que` et la complexité de l'algorithme apparaîtra alors.

• Avant d'entrer dans la boucle, on a g=0 et d=n-1. On a donc $d - g = n - 1$
• A la fin de la 1re itération, on a $d - g = \frac{n-1}{2}$
• A la fin de la kième itération, on a $d - g = \frac{n-1}{2^k}$

Or, le nombre total d'itérations effectuées est donné par la condition ①, c'est-à-dire que la boucle se termine dès que g ≥ d, soit d-g ≤ 0, ou encore d-g < 1 (d et g sont des entiers).

On doit donc avoir, pour que la boucle se termine, k tel que : $\frac{n-1}{2^k} < 1$.

On a donc : $n - 1 < 2^k$

$\Leftrightarrow k > log_2(n-1)$

La complexité sera alors :

• Dans le meilleur cas, c'est-à-dire celui ou dès le premier tour de boucle l'élément central est celui que l'on cherche. Dans ce cas, il n'y a rien à faire et la complexité est indépendante de la taille des données. Elle est à temps constant $O(1)$.
• Dans le pire cas, elle est en $O(log_2(n))$, ce qui est un algorithme particulièrement efficace.

➢2e façon :

On peut introduire une suite numérique. On ne doit pas oublier (voir chapitre précédent) que la relation i=i+1 dans une boucle signifie :

i (de l'itération k courante) = i(de l'itération précédente k-1)+1.

C'est donc une relation de récurrence qui permet de définir une suite numérique.

• On pose : $U_0 = d - g = n - 1$ (avant d'entrer dans la boucle pour la 1re itération)
• $U_1 = \frac{d-g}{2} = \frac{U_0}{2^1}$
• $U_2 = \frac{U_1}{2} = \frac{U_0}{2^2}$

…

• $U_k = \frac{U_{k-1}}{2} = \frac{U_0}{2^k}$

Avec le même raisonnement que dans l'autre solution, U_k<1 est la condition pour que la boucle se termine.

D'où $\frac{n-1}{2^k} < 1$ soit $k > log_2(n-1)$

On retrouve la complexité de l'algorithme en $O(log_2(n))$.