Chapter 1

Algorithmique: Recherche dichotomique

Les deux activités suivantes ont pour thème la résolution d'équation. La première, assez basique, peut être retravaillée en écrivant une fonction.

1.1 Activité 1: approximation de $\sqrt{2}$

Le programme suivant donne une approximation de $\sqrt{2}$ par la méthode de dichotomie. On utilise le carré de m pour la déterminer. Nous savons que la fonction *carré* est croissante sur $[0;+\infty[$, si $m^2 < 2$ c'est que $m < \sqrt{2}$: m peut donc devenir la borne inférieure de l'intervalle de recherche.

```
1  n = int(input("Entrez la précision voulue:"))
2  borne_inf, borne_sup = 1, 2
3  while borne_sup - borne_inf > 10**(-n):
4      m = (borne_sup + borne_inf) / 2
5      if m**2 < 2:
6          borne_inf = m
7      else:
8          borne_sup = m
9
10 print("Une approximation de racine carrée de 2 est:", m)
```

1. Expliquer ce que font la première et la dernière ligne. Pourquoi est-il préférable éviter ce genre de pratiques ?
2. Implémenter une fonction `racine_2` renvoyant une valeur de $\sqrt{2}$ avec une précision choisie par l'utilisateur.

1.2 Activité 2: Approximation d'un solution d'équation.

Soit f la fonction définie sur $\mathbb{R}$ par $f(x) = x^2 + x - 1$.

1. Représenter par la méthode de votre choix $\mathcal{C}_f$.
2. On se propose de déterminer une valeur approchée α de l'équation $f(x) = 0$ sur $[0;1]$.

3. Utiliser l'algorithme 1 pour compléter le tableau de suivi des variables:

Algorithme 1

```
1  a ⟵ 0
2  b ⟵ 1
3  pour k allant de 1 à 10 faire
4  |  m ⟵ (a + b)/2
5  |  si f(m) × f(a) ⩾ 0 alors
6  |  |  a ⟵ m
7  |  sinon
8  |  |  b ⟵ m
9  |  fin
10 fin
```

k	m	a	b
		0	1
1	0,5	0,5	1
2			
3			
...	...	...	...
10			

4. Quel est le rôle de cet algorithme ? Interpréter les dernières valeurs de a et b obtenues.
5. En déduire une valeur de α.
6. Implémenter ce programme.

1.3 Algorithme de recherche naïf

Des volumes importants de données sont susceptibles d'être traitées par les ordinateurs. Des algorithmes efficaces sont alors nécessaires pour réaliser ces opérations comme, par exemple, la sélection et la récupération des données.

Les algorithmes de recherche entrent dans cette catégorie. Leur rôle est de déterminer si une donnée est présente et, le cas échéant, d'en indiquer sa position, pour effectuer des traitements annexes. La recherche d'une information dans un annuaire illustre cette idée. On cherche si telle personne est présente dans l'annuaire afin d'en déterminer l'adresse.

Plus généralement, c'est l'un des mécanismes principaux des bases de données : à l'aide d'un identifiant, on souhaite retrouver les informations correspondantes.

1.3.1 Premier algorithme

```
def recherche1(tab, val):
    res = False
    n = len(tab)
 for i in range(n):
     if tab[i] == val:
         res == True
 return res
```

Avec une affectation avant la boucle et une boucle de 0 à $n-1$, on effectue donc $2n+4$ opérations élémentaires dans le pire des cas. Le résultat peut dépendre de ce qu'on considère comme *opération élémentaire* mais au final, comme tout algorithme ayant cette forme, la complexité est linéaire : le temps de recherche est proportionnel à la longueur de la liste.

Cependant avec cette méthode, on n'exploite pas le caractère ordonné du tableau, savoir que telle valeur du tableau n'est pas la valeur recherchée n'apprend absolument rien sur les autres valeurs du tableau.

1.4 Algorithme de recherche dichotomique

Dans cette famille d'algorithmes, la recherche dichotomique permet de traiter efficacement des données représentées dans un tableau de façon ordonnée.

L'idée centrale de cette approche repose sur l'idée de réduire de moitié l'espace de recherche à chaque étape : on regarde la valeur du milieu et si ce n'est pas celle recherchée, on sait qu'il faut continuer de chercher dans la première moitié ou dans la seconde. Plus précisément, en tenant compte du caractère trié du tableau, il est possible d'améliorer l'efficacité d'une telle recherche de façon conséquente en procédant ainsi :

1. on détermine m, élément au milieu du tableau ;
2. si c'est la valeur recherchée, on s'arrête avec un succés ;
3. sinon, deux cas sont possibles :
   (a) si m est plus grand que la valeur recherchée, comme la tableau est trié, cela signifie qu'il suffit de continuer à chercher dans la première moitié du tableau ;
   (b) sinon, il suffit de chercher dans la moitié droite.
4. on répète cela jusqu' à avoir trouvé la valeur recherchée, ou bien avoir réduit l'intervalle de recherche à un intervalle vide, ce qui signifie que la valeur recherchée n'est pas présente.

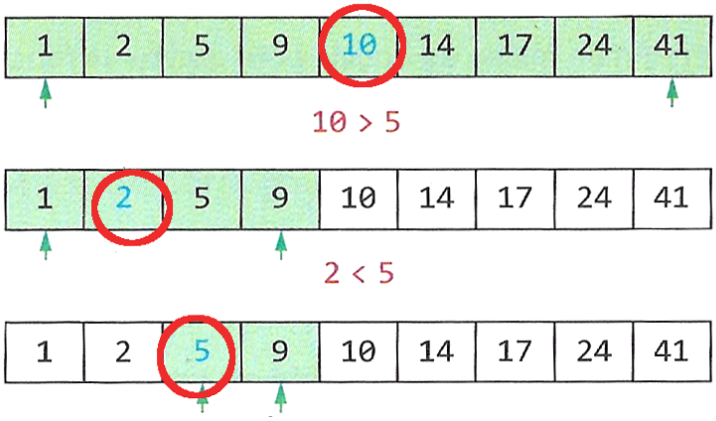
À chaque étape, on coupe l'intervalle de recherche en deux, et on en choisit une moitié. On dit que l'on procède par dichotomie, du grec *dikha* (en deux) et *tomos* (couper). Une implémentation en Python est la suivante :

```
1  def recherche_dichotomique(tab, val):
2          gauche = 0
3          droite = len(tab) - 1
4          while gauche <= droite:
5                  milieu = (gauche + droite) // 2
6                  if tab[milieu] == val:
7                      return milieu
8                  elif tab[milieu] > val:
9                      droite = milieu - 1
10                 else:
11                     gauche = milieu + 1
12         return -1
```

Note : la valeur renvoyée par l'algorithme est de type *entier*, c'est pourquoi elle renvoie -1 si la valeur recherchée n'est pas dans le tableau.

Figure 1.1: Illustration David Revoy CC-BY 4.0

Exemple On recherche 5 dans la liste, `gauche` et `droite` sont marqués par les flèches, `milieu` est entouré.



1.4.1 Analyse de l'algorithme

Pour s'assurer que le programme ci-dessus fonctionne correctement, il faut se poser deux questions importantes :

- Le programme renvoie-t-il bien un résultat ? Comportant une boucle non bornée, est-on sûr d'en sortir à un moment donné ?
- La réponse renvoyée par le programme est-elle correcte ? Ici, il y a deux sortes de réponses :
  - soit on a obtenu `-1`, auquel cas la valeur recherchée ne doit pas être présente dans le tableau ;
  - soit on a obtenu un entier positif ou nul qui correspond à la position de la valeur recherchée dans le tableau (ou, en cas de répétition, l'une des positions).

Nous allons traiter ces deux questions et en donner des réponses rigoureuses dans la suite, prouvant ainsi que le comportement de notre programme est bien celui espéré, ainsi qu'étudier sa complexité au dernier paragraphe.

1.4.2 Terminaison du programme

La fonction `recherche_dichotomique` contient une boucle non bornée `while`, et pour être sûr de toujours obtenir un résultat, il faut s'assurer que le programme termine. Pour prouver que c'est bien le cas, nous allons utiliser un **variant de boucle**.

Variant de boucle Il s'agit d'une quantité entière qui :

1. doit être positive ou nulle pour rester dans la boucle ;
2. doit décroître à chaque itération.

Si l'on arrive à trouver une telle quantité, il est évident que l'on va nécessairement sortir de la boucle au bout d'un nombre fini d'itérations, puisque un entier positif ne peut décroître infiniment.

Preuve de la terminaison Pour le cas qui nous occupe, un variant est très facile à trouver : il s'agit de la largeur de la quantité `droite - gauche`. La condition de boucle étant `gauche <= droite`, cela correspond exactement à ce que notre variant soit positif ou nul.

Montrons maintenant que le variant décroit strictement lors de l'exécution du corps de la boucle.

On commence par définir `milieu`:

```
milieu = (gauche + droite) // 2.
```

En particulier, on a alors `gauche` $\leqslant$ `milieu` $\leqslant$ `droite`

Ensuite, trois cas sont possibles.

- si `tab[milieu] == val`, on sort directement de la boucle à l'aide d'un return. La terminaison est assurée.
- si `tab[milieu] > val`, on modifie la valeur de gauche et `droite-gauche` diminue.
- sinon, on modifie `droite` et `droite-gauche` diminue également.

  Dans les deux cas, le variant a strictement décru.

Ayant réussi à exhiber un variant pour notre boucle, nous avons prouvé qu'elle termine bien.

Note : *exhiber* signifie que l' on met un objet en valeur, et cela suffit à conclure. C'est assez rare d'utiliser le mot *exhiber*, profitez-en !

1.4.3 Exercices

A la main

1. Ecrivez les étapes de l'algorithme pour:
   (a) `recherche_dichotomique([0, 1, 1, 2, 3, 5, 8, 13, 21, 23], 7)`.
   (b) `recherche_dichotomique([15, 16, 18, 19, 23, 24, 28, 29, 31, 33], 28)`.
   (c) `recherche_dichotomique([15, 16, 18, 19, 23, 24, 28, 29, 31, 33], 16)`.
2. [**Complexité:**] Au maximum, combien de tours de boucle ont été effectués avec ces listes de 10 valeurs ?

Avec un compteur

1. Générer une liste de 10 éléments pris aléatoirement entre 0 et 100 par compréhension.
2. Modifier la fonction `recherche_dichotomique` pour afficher le nombre de boucle effectuées par l'algorithme et le lancer sur des tableaux de différentes tailles : 10, 100 etc. (utiliser la question précédente pour générer ces listes).

Exemple La fonction `recherche_dichotomique` renvoie un tuple avec l'indice de l'élément `val` et le nombre de boucles effectuées.

```
>>> recherche_dichotomique([15, 16, 18, 19, 23, 24, 28, 29, 31, 33], 18)
(2, 3)
```

Note : On peut retrouver les valeurs maximales de nombre de tours en écrivant une fonction `nb_tours(n)` qui renvoie le plus petit entier k tel que:

$$2^k > n.$$

c'est à dire le nombre maximal de valeurs examinées par la recherche dichotomique dans un tableau de taille n où ne figure pas la valeur recherchée. Cette fonction est appelé logarithme en base 2 de n notée $\log_2(n)$.

3-Mesurer le temps d'execution Le module `timeit` de Python permet de prendre une mesure du temps d'exécution d'une fonction, en secondes. Ce module fournit une fonction `timeit` qui fonctionne comme indiqué ci-dessous :

```
import timeit

def ma_fonction():
    # Code de votre fonction ici
    pass

# Mesurer le temps d'exécution
temps_execution = timeit.timeit(ma_fonction, number=1)
print("Temps d'exécution:", temps_execution, "secondes")
```

Remarque On peut affiner l'utilisation de ce module mais cela est inutile ici:

- `setup` permet de préciser à timeit les modules à charger pour permettre l'exécution correcte de la fonction (y compris donc le module qui contient la fonction à mesurer),
- `stmt` – pour statement, instruction en anglais – est l'appel de fonction qui sera mesurée (donc avec ses paramètres),
- `number` est le nombre de fois où l'instruction stmt sera exécutée. Le temps mesuré sera le temps cumulé pour toutes ces exécutions.

Remarquez que le code Python des deux paramètres `setupet, stmt` sont donnés sous forme d'une chaîne de caractères. Par exemple, après avoir importé le module `timeit` : `from timeit import timeit`

- on peut mesurer le temps d'exécution de la recherche dichotomique sur une liste (triée) de taille 10;

- on peut mesurer le temps d'exécution du tri par sélection sur une liste mélangée de taille 10;

```
timeit(setup='from TP import recherche_dichotomique',
       stmt='recherche dichotomique([15, 16, 18, 19, 23, 24, 28, 29, 31, 33], 20)',
       number=100)
```

Le résultat obtenu représente le temps total mis pour exécuter 100 fois :

```
recherche dichotomique([15, 16, 18, 19, 23, 24, 28, 29, 31, 33])
```

1.4.4 Complexité

Pour mesurer cette complexité, on s'intéresse au nombre de valeurs du tableau `tab` qui ont été examinées pendant l'exécution de la boucle `while`. Ce dernier étant de taille n. Le temps d'exécution de la fonction est proportionnel à ce nombre.

Plaçons nous dans le pire cas où la valeur n'apparaît pas dans la liste. Si on exécute correctement l'algorithme avec une liste d'entiers de 1 à 100, vous avez pu vérifier dans l'exercice 2-3 qu'il nous faudra au plus 7 tours pour vérifier que la valeur n'est pas dans l'intervalle. Cette valeur est de 20 itérations pour une liste d'un million de valeurs.

Logarithme d'un nombre Ces valeurs s'expliquent par le fait que:

$$2^7 > 100 \text{ ou } 2^{20} > 10^6$$

Le nombre d'étape est proportionnel au nombre de fois où l'on peut diviser n par 2.

C'est le nombre entier k le plus petit tel que: $2^k > n$.

Cette fonction mathématique est bien connue: c'est le **logarithme de n en base 2** noté $\log_2(n)$. On dit aussi que c'est le nombre de chiffre dans la représentation binaire de n.

L'algorithme de recherche dichotomique est de complexité logarithmique

Notation de Landau: $\mathcal{C}(n) = O(\log_2(n))$.

1.5 Références

- Documents Éduscol