Diviser pour régner

M. Tellene

1 Création de l'environnement de travail

Avant de commencer le TP, vous allez créer votre environnement de travail. Pour ce faire vous allez dans votre dossier personnel. Une fois arrivé, créer un dossier « TP_diviser_regner ». C'est dans ce dossier que vous sauvegarderez les exercices de ce TP.

Il existe beaucoup de stratégies en programmation, l'une d'entre elle est « diviser pour régner ». Il est possible de résumer cette stratégie de la manière suivante : si un problème est trop compliqué, coupons le en deux problèmes plus petits.

Nous allons traiter deux exemples, : la recherche dans une liste et le calcul de puissance.

2 Recherche dans une liste

Supposons que vous ayez un livre de 100 pages. Vous voulez l'ouvrir à la page 62. Voici ce que vous pouvez faire :

1. La méthode naïve : faire défiler toutes les pages jusqu'à la page 62

   Combien de pages allez-vous examiner pour arriver à la page 62 ? .....

   Cela semble-t-il une méthode optimale ? .....

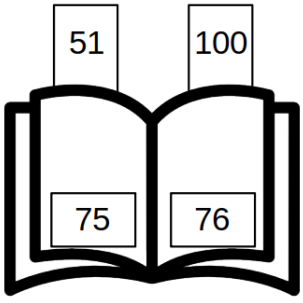
   La méthode suivante est un peu plus proche de la réalité :

2. La stratégie diviser pour régner :
   — On ouvre le livre en plein milieu. On arrive aux pages 50-51. La page 62 est-elle située avant ou après ces pages ? .....

   Pour clarifier les choses, voici un petit livre : 🕮 que nous agrémenterons de marque page et de numéros au fil de notre raisonnement.

   — Entre les pages 51 et 100, quelle sont les pages du milieu ? .....

   Nous représenterons cela de la manière suivante :

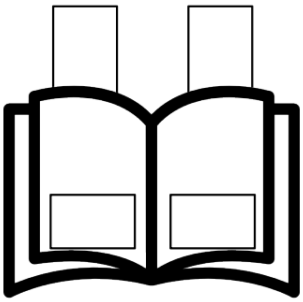


On a placé 2 marque-pages pour symboliser que nous avons à continuer notre recherche entre les pages 51 et 100, en regardant les pages du milieu, c'est-à-dire les pages 75 et 76.

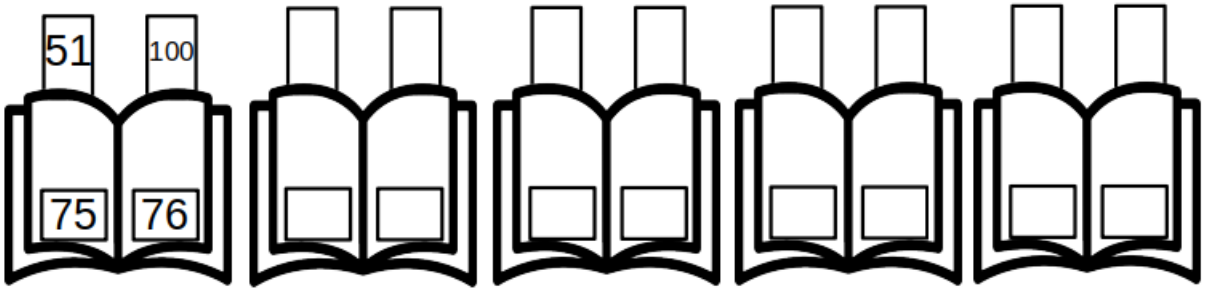
On se pose donc la question suivante : la page 62 est-elle avant ou après ces pages ? .....

Quelles sont les pages situées au milieu des valeurs que vous venez de trouver : .....

Mettre à jour le dessin suivant :



Voici quelques dessins supplémentaires à compléter jusqu'à ce qu'apparaisse la page 62 :



De manière plus formelle, on symbolise notre livre par une liste de la forme suivante :

```
[1, 2, 3, 4, 5, ... 97 , 98 , 99 , 100]
```

Dans le tableau suivant on récapitule les différentes étapes qui mènent à la page 62 :

Partie gauche du livre	Partie droite du livre	Où est le milieu ?
`[1, ... , 50]`	`[51, ... , 100]`	$(51+100)//2 = 75$
`[51, ... , 75]`	`[76, ... , 100]`	$(51+75)//2 = 63$

Évidement, on ne pense pas à tout ça quand on cherche un page dans un livre, mais comme nous faisons des algorithmes, nous devons être capables de résumer ce que l'on veut faire en de petites étapes simples et faciles à implémenter.

3. Comparaison du nombre d'étapes :

— Nombre d'étapes avec la méthode naïve : .....

— Nombre d'étapes avec la méthode « diviser pour régner » : .....

3 Recherche dans une liste

Nous voulons faire une fonction qui prenne en argument une liste `L` composée de nombre et un nombre `x`. Cette fonction renvoie `True` si `L` contient `x`, `False` sinon. Compléter la fonction suivante pour qu'elle réponde au cahier des charges.

```
def recherche_naive(L,x):
    """Cette fonction recherche la présence de x dans la liste L, en
    balayant toute la liste de gauche à droite et renvoie True si elle
    trouve la valeur et False sinon"""
    for i in range (len(L)):

    if ..... == ..... :

        return .....

    return .....
```

3.1 Cas d'une liste non-triée

Si la liste est non triée, on ne peut pas savoir si le nombre que l'on cherche est « plus vers le début » ou « plus vers la fin ». On est donc obligé de tester toutes les valeurs.

3.2 Cas d'une liste triée

Si la liste est triée, on peut utiliser la stratégie « diviser pour régner » vue précédemment, appliquée aux listes.

Testons-la sur un exemple :

— `L = [1, 2, 2, 4, 4, 9, 9, 9, 11, 12, 13, 14, 15, 20, 29, 30, 30]`

— `x = 14`

— C'est une liste de longueur 17. on notera $d = 0$ et $f = 16$ les indices du début et de la fin de notre liste.

$$\underset{\substack{\uparrow\\ d=0}}{[\ 1,} \quad 2, \quad 2, \quad 4, \quad 4, \quad 9, \quad 9, \quad 9, \quad \underset{\substack{\uparrow\\ 8}}{\mathbf{11},} \quad \underset{\substack{\uparrow\\ 9}}{\mathbf{12},} \quad 13, \quad 14, \quad 15, \quad 20, \quad 29, \quad 30, \quad \underset{\substack{\uparrow\\ f=16}}{30\ ]}$$

On va représenter notre stratégie « diviser pour régner » en des étapes successives pour rechercher si 14 est présent dans cette liste :

1. Recherche des valeurs du milieu entre d et f : $\dfrac{d+f}{2} = \dfrac{0+16}{2} = 8$

   — Les valeurs du milieu sont les numéros 8 et 9.

   — 14 peut-il être présent dans la liste entre les indices 0 et 8 : Non car `L[8] = 11 < 14`

   — 14 peut-il être présent dans la liste entre les indices 9 et 16 : Peut être

— On va donc continuer à chercher entre 9 et 16
— On actualise les valeurs d et f : $d = 9$, $f = 16$

$$[\ 1,\ 2,\ 2,\ 4,\ 4,\ 9,\ 9,\ 9,\ 11,\ \underset{\uparrow\ d=9}{12},\ 13,\ 14,\ 15,\ 20,\ 29,\ 30,\ \underset{\uparrow\ f=16}{30}\ ]$$

2. Recherche des valeurs du milieu entre d et f : $\frac{d+f}{2}$ = ...
   — Les valeurs du milieu sont donc ..... et .....
   — 14 peut-il être présent dans la liste entre les indices 9 et ..... : .....
   — 14 peut-il être présent dans la liste entre les indices ..... et 16 : .....
   — On va donc continuer à chercher entre ..... et .....
   — On actualise les valeur d et f : d = ..., f = ...

$$[\ 1,\ 2,\ 2,\ 4,\ 4,\ 9,\ 9,\ 9,\ 11,\ 12,\ 13,\ 14,\ 15,\ 20,\ 29,\ 30,\ 30\ ]$$

3. Recherche des valeurs du milieu entre d et f : $\frac{d+f}{2}$ = ...
   — Les valeurs du milieu sont donc ..... et .....
   — 14 peut-il être présent dans la liste entre les indices .....et ..... : .....
   — 14 peut-il être présent dans la liste entre les indices .....et ..... : .....
   — On va donc continuer à chercher entre ..... et .....
   — On actualise les valeur d et f : d = ..., f = ...

   On a `L[d] = 13`. On a donc bien trouvé la valeur 14 dans la liste.

$$[\ 1,\ 2,\ 2,\ 4,\ 4,\ 9,\ 9,\ 9,\ 11,\ 12,\ 13,\ 14,\ 15,\ 20,\ 29,\ 30,\ 30\ ]$$

Nous sommes donc arrivé sur notre valeur en 4 étapes au lieu de 12. Pas mal non? Passons à l'algorithme :

— Tout d'abord en langage « naturel » :

1. Je regarde la valeur au milieu de f et d, que je note id_milieu
2. Si `L[id_milieu] = x` alors mon nombre est dans la liste, on renvoie `True`
3. Si `L[id_milieu] < x` alors je remplace d par .....
4. Si `x < L[id_milieu]` alors je remplace f par .....

On continue tant que f ..... d

Si on n'a toujours pas `L[id]_milieu = x`, alors on renvoie : .....

— Voici l'algorithme en langage Python à compléter :

```
def recherche_dicho(L,x):
    """Cette fonction renvoie True si le nombre x est dans la liste L et
    False sinon"""
    if len(L) == 0:     #Ici, on traite le as de la liste vide
        return False

    d = ........

    f = ........

    id_milieu = (f+d)//2
    # id_milieu correspond à l'indice du milieu, il faut parfois
    # l'arrondir d'où l'utilisation de //

    while ......................... :

        if ......................... :
            return True

        elif ......................... :

            f = ........................

        elif ......................... :

            d = ........................

        ........................

    return ........................
```

4 Implémentation et comparaison des performances

1. Récupérer le fichier **rech_tableau_trie.py**
2. Compléter la fonction `recherche_naive`
3. Tester la fonction `recherche_naive`. Pour cela, voici ce que vous devez taper et obtenir dans la console :

```
>>> test_in_liste(recherche_naive,LISTE)
True
>>> test_not_in_liste(recherche_naive,LISTE)
False
```

   La première fonction indique que pour toute valeur de la liste, la fonction `recherche_naive` renvoie `True`. La deuxième fonction indique que pour toute valeur qui n'est pas dans la liste `LISTE`, la fonction `recherche_naive` renvoie `False`.

4. Compléter et tester la fonction `recherche_dicho` (*cf.* point 4)
5. On passe à l'étape d'évaluation des performances des fonctions. Il s'agit de voir quelle fonction est la plus rapide à exécuter. Pour cela, dé-commenter la dernière fonction. Puis exécuter la fonction `eval_des_perf(10)`. Vous obtiendrez quelque chose de ce genre-là :

```
>>> eval_des_perf(10)
```

```
Teste de la méthode naive :
Les tests de la méthode naïve ont pris : 0.00018262863159179688 secondes
---------------------------
Teste de la méthode diviser pour régner :
Les tests de la méthode diviser pour régner ont pris :
    0.00016570091247558594 secondes
```

Il est possible que vous obteniez des valeurs différentes, pas d'inquiétude. Cela indique le temps de calcul de la fonction sur une liste de longueur 10. A vous de faire des tests et de compéter le tableau suivant :

N	temps de la recherche naïve	temps de la recherche dichotomique
10	0.00018	0.00016
100		
1000		
100 000		
500 000		
1 000 000		

5 Calcul de puissance

1. On sait que $3^{10} = 59049$. (en python 3**10). Voici une première méthode pour calculer ce nombre :
   — $3^2 = 3 \times 3 = 9$
   — ...
   — $3^{10} = 3 \times 3^9 = 3 \times 19683 = 59049$

   Combien d'opérations cela nécessite-t-il? .....

   Algorithme en python à compléter :

```
def puissance_naive(a,n):
    """Calcule a^n en remultipliant par a successivement"""
    resultat = 1

    while ..... :

        resultat = .....

        n = .....

    return resultat
```

2. Reprenons l'exemple précédent, seriez-vous capable de calculer 3^{10} en 4 opérations?

   En faisant des mises au carré successives, voila ce que l'on obtient :

$$3^{10} = (3^2)^5 = (3^2)^4 \times (3^2)^1 = \left((3^2)^2\right)^2 \times (3^2)$$

Une petite remarque :

$$3^{10} = \left((3^2)^2\right)^2 \times (3^2) = 3^{2^3+2^1}$$

On reconnaît l'écriture binaire de 10 : $10 = 2^3 + 2^1 = 1010_2$

L'algorithme suivant, assez dur à trouver, est inspiré par l'algorithme qui permet de trouver une décomposition en base 2 :

Soit a un nombre et n un entier positif. On veut calculer a^n avec la méthode précédente :

`resultat` est initialisé à 1 — Si n est pair, on remplace a par a^2 et on divise n par 2 — Sinon n est impair, on remplace `resultat` par `resultat x a` et on enlève 1 à n — On recommence tant que $n > 0$ On renvoie `resultat`

Pour bien comprendre, appliquer cet algorithme au calcul de 2^{10} en complétant les éléments ci-dessous :

— $a = 2$, $n = 10$, $resultat = 1$

— $a =$ ....., $n =$ ....., $resultat =$ .....

— $a =$ ....., $n =$ ....., $resultat =$ .....

— $a =$ ....., $n =$ ....., $resultat =$ .....

— $a =$ ....., $n =$ ....., $resultat =$ .....

— $a =$ ....., $n =$ ....., $resultat =$ .....

— $a =$ ....., $n =$ ....., $resultat =$ .....

Compléter ensuite l'algorithme suivant :

```
def puissance_rapide(a,n):
    resultat = 1

    while n>0:

        ......................

            ......................

            ......................

        ......................

            ......................

            ......................

    return resultat
```

6 Implémentation et contrôle des performances

1. Récupérer le fichier `puissance.py`
2. Compléter et tester la fonction `puissance_naive`. Pour cela, voici ce que vous devez taper et obtenir dans la console :

```
>>> test(puissance_naive)
True
```

3. Compléter et tester la fonction `puissance_rapide` (*cf. point précédent)*
4. Enfin, dé-commenter la dernière fonction et compléter le tableau pour voir quelle fonction est la plus efficace.

a	n	temps de calcul de la puissance naïve	temps de calcul de la puissance rapide
2	10		
2	100		
2	1 000		
2	500 000		
2	1 000 000		