• La méthode de dichotomie existe depuis l'Antiquité. Le mot dichotomie vient du grec et signifie division en deux parties.

• Le principe est de partager la tâche que l'on doit effectuer en deux parties et de poursuivre l'algorithme sur l'une des deux parties, en utilisant à nouveau ce principe de dichotomie. La dichotomie est aussi utilisée dans certaines preuves mathématiques.

• Dans ce cours, nous allons étudier les deux algorithmes dichotomiques qui sont au programme : **la recherche dichotomique dans une liste triée** et **l'exponentiation rapide**. L'avantage principal de ces algorithmes est d'améliorer la complexité des algorithmes classiques effectuant les mêmes tâches.

1 Recherche dichotomique dans une liste triée

1.1 L'algorithme classique de recherche d'un élément dans une liste

• Il est très fréquent, pour gérer l'immense masse de données dont nous disposons, de devoir rechercher la présence ou non d'un élément dans une liste. Voici une fonction réalisant ceci :

```
def recherche(L, a):
    """renvoie l'indice de l'élément a dans la liste L, s'il est présent et renvoie False sinon"""
    n = len(L)
    for i in range(n):
        if L[i] == a:
            return(i) # on renvoie l'indice de l'élément a
    return(False) # si on a parcouru la liste sans trouver, on renvoie False
```

• On remarque que notre fonction renvoie *False* quand la liste est vide, ce qui est bien le résultat souhaité.

• L'instruction a *in* L permet aussi de tester si l'élément a est dans la liste L mais le but de ce paragraphe est de programmer par nous-même cette tâche.

• Pour évaluer la complexité de notre algorithme, il s'agit de compter le nombre de comparaisons entre $L[i]$ et a en fonction de la longueur de la liste qui est notée n. Dans le pire des cas, si l'élément a n'est pas dans la liste L ou s'il est placé en dernière position, nous devons parcourir toute la liste et effectuer ainsi n comparaisons. On retiendra donc que :

Le coût de la recherche d'un élément dans une liste de taille n est en $O(n)$

De plus, le fait que les éléments de la liste soient triés, par exemple dans l'ordre croissant, ne changera pas cette complexité avec cet algorithme.

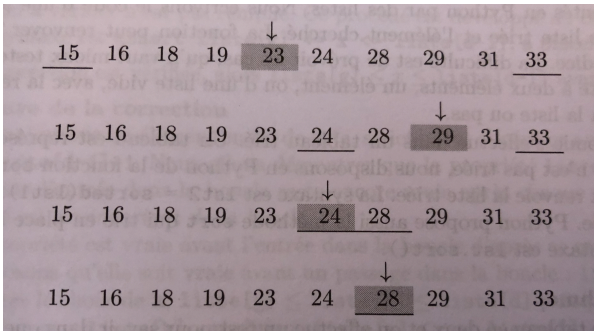
• L'intérêt de la méthode dichotomique que nous allons à présent étudier est d'accélérer grandement cette recherche.

1.2 Explication du principe

• La méthode de la recherche dichotomique n'est pas applicable qu'à une condition préalable que l'on supposera vérifiée dans toute la suite :

la liste est triée dans l'ordre croissant

• Voyons le principe sur un exemple. On considère la liste triée $L = [15, 16, 18, 19, 23, 24, 28, 29, 31, 33]$ et on cherche si l'élément 28 est dans cette liste.



▶ On compare 28 avec l'élément central de la liste, ici 23, comme $28 > 23$, on sait que 28 se situe éventuellement dans la partie droite de la liste.

▶ On se concentre alors sur la sous-liste $[24, 28, 29, 31, 33]$, on prend le milieu, 29, et comme $28 < 29$, on sait que 28 se situe éventuellement dans la sous-liste de gauche $[24, 28]$.

▶ Le milieu de la sous-liste $[24, 28]$ est 24 et comme $24 < 28$, on sait que 28 se situe éventuellement dans la sous-liste de droite réduite à $[28]$.

▶ On compare enfin 28 avec le milieu de la liste $[28]$, comme c'est égal, on a trouvé 28 et on renvoie son indice. Si l'on avait cherché l'élément 27 le procédé aurait été identique, seule la conclusion change.

• On remarque que lorsque l'on considère le milieu de la liste, on arrondit à l'indice inférieur. À chaque étape, la nouvelle liste que l'on regarde est au moins de longueur deux fois moindre que la liste précédente.

1.3 L'algorithme

• Le principe se comprend bien mais la façon de l'écrire n'est pas aisée car il faut faire attention aux indices utilisés et aux cas particuliers comme celui d'une liste vide ou d'une liste réduite à un élément. D'ailleurs, comme l'écrivait Donald Knuth, le pionnier de l'algorithmique :

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky, and many good programmers have done it wrong the first few times they tried.

• Voici une version de l'algorithme de recherche dichotomique dans une liste triée :

```
1  def dicho(L, a):
2      """recherche si l'élément a est présent dans la liste triée dans l'ordre croissant L et renvoie un
       indice si c'est le cas et False sinon"""
3      g = 0
4      d = len(L) - 1
5      # g et d sont les indices des bornes de la sous-liste dans laquelle on cherche notre élément
6      # ces bornes vont évoluer à chaque étape, selon que l'on choisisse la sous-liste de gauche ou de
       droite
7      while g <= d: # tant que la sous-liste examinée n'est pas vide
8          m = (g + d) // 2 # l'indice du milieu du tableau
9          if L[m] == a: # on a trouvé a
10             return(m) # on renvoie l'indice et cela termine la fonction
11         elif L[m] < a: # a se situe peut-être dans la partie de droite
12             g = m + 1 # on change la borne de gauche de notre sous-liste pour considérer la sous-liste de
       droite
13         else: # a se situe peut-être dans la partie de gauche
14             d = m - 1 # on change la borne de droite de notre sous-liste pour considérer la sous-liste de
       gauche
15     return(False) # si on arrive ici, c'est que l'élément n'a pas été trouvé
```

• Cet algorithme est à parfaitement comprendre et à savoir réécrire, vous aurez l'occasion de travailler avec en TD et en TP. Poursuivons l'étude théorique de cet algorithme en justifiant la terminaison, la correction et en étudiant la complexité.

1.4 Validité et complexité

Dans ce paragraphe, nous allons résumer les résultats à connaître sur cet algorithme de recherche dans une liste triée. Les preuves rigoureuses de ces résultats sont à comprendre également mais elles sont écrites à la fin de ce document dans la partie Annexe.

1.4.1 Complexité

La complexité de l'algorithme de recherche d'un élément dans une liste triée est logarithmique, en $O(\log_2(n))$ si n désigne la longueur de la liste.

• On peut expliquer ce résultat, sans rigueur, de la façon suivante. Prenons une liste de longueur $n = 2^p$ avec $p \in \mathbb{N}$. Dans le pire des cas, on cherche un élément qui n'appartient pas à la liste initiale. À chaque passage dans la boucle *while* la taille de notre liste est divisée au moins par 2. Ainsi, il faut environ p étapes pour se retrouver avec la liste vide et sortir de la boucle *while*. Or $p = \log_2(n)$ donc il y a environ $\log_2(n)$ passages dans la boucle d'où la complexité en $O(\log_2(n))$.

• La base dans laquelle on écrit le logarithme n'a aucune importance, puisque tous les logarithmes sont égaux à une constante près, ainsi on peut juste écrire que la complexité est $O(\log(n))$.

• C'est bien entendu un gain très important par rapport à l'algorithme classique qui est en $O(n)$ mais il faut garder en tête que notre méthode demande que la liste soit triée au préalable.

1.4.2 Terminaison

• C'est lié à la complexité étudiée précédemment, en effet le nombre de passages dans la boucle *while* est fini donc l'algorithme se termine. Pour être plus rigoureux, on peut démontrer ceci exhibant un variant de boucle :

La quantité $d - g + 1$ qui représente la taille de la sous-liste que l'on examine pour trouver l'élément a est un variant de boucle.

1.4.3 Correction

• On peut montrer que l'algorithme renvoie le résultat attendu en trouvant un invariant de boucle.

Notre invariant de boucle est :

$$\mathcal{P} \ : \text{"si } a \in L \text{ alors } a \text{ se situe entre les indices } g \text{ et } d \text{ (inclus)"}$$

• On démontre dans la partie Annexe que cette propriété est vraie initialement et reste vraie à chaque itération de la boucle. Lorsque l'on sort de la boucle, on a $g > d$ ainsi on en déduit que a n'est pas dans L, ce qui justifie la valeur $False$ que l'on renvoie alors.

2 Exponentiation rapide

• Le but du problème est de calculer a^n où a est un flottant et $n \in \mathbb{N}^*$. On cherche la complexité de ce problème en terme de multiplications selon la donnée en entrée : n. En effet, on suppose que l'opération $a \times a$ ne dépend pas de la valeur de a.

2.1 Méthode naïve

• Cela consiste à utiliser la définition $a^n = \underbrace{a \times a \times ... \times a}_{n \text{ fois}}$. On a l'algorithme suivant :

```
def exp1(a, n):
    P = 1
    for i in range(n):
        P = P * a
    return(P)
```

• Cet algorithme présente une complexité en $O(n)$. Nous allons voir qu'il est possible de l'améliorer.

2.2 Amélioration : exponentiation rapide

• L'algorithme qui suit est basé sur la remarque :

$$\begin{cases} a^n = (a^2)^{\frac{n}{2}} & \text{si } n \text{ est pair} \\ a^n = a(a^2)^{\frac{n-1}{2}} & \text{si } n \text{ est impair} \end{cases}$$

On peut écrire une fonction qui utilise ce principe :

```
def exporapide(a, n):
    r = 1
    while n > 0:
        if n % 2 == 1: # cas où n est impair
            r = r * a # on se ramène alors à un exposant pair en multipliant par a
        a = a * a # a^n=(a^2)^(n//2)
        n = n // 2
    return(r)
```

• Voilà un exemple qui décrit le fonctionnement de l'algorithme pour $a = 2$ et $n = 10$.

$$\begin{cases} r = 1 \\ a = 2 \\ n = 10 \end{cases} \longrightarrow \begin{cases} r = 1 \\ a = 4 \\ n = 5 \end{cases} \longrightarrow \begin{cases} r = 4 \\ a = 16 \\ n = 2 \end{cases} \longrightarrow \begin{cases} r = 4 \\ a = 256 \\ n = 1 \end{cases} \longrightarrow \begin{cases} r = 1024 \\ a = 256 \\ n = 0 \end{cases}$$

Vous pouvez vérifier qu'avec ce procédé, nous avons effectué 6 multiplications.

2.3 Validité et complexité

Nous avons démontré dans l'exercice 5 du cours 4 que cet algorithme se termine et donne le bon résultat. Je vous y renvoie pour les détails de ces preuves. En résumé, on a :

• L'algorithme se termine car n est un variant de boucle puisque c'est un entier qui décroit strictement à chaque passage dans la boucle.

• L'expression $r \times a^n$ est un invariant de boucle car cette quantité ne varie pas lors d'un passage dans la boucle. Comme initialement elle vaut a^n et qu'à la fin de la boucle, c'est-à-dire quand $n = 0$, elle vaut r alors $r = a^n$ et on renvoie bien le résultat attendu.

• Enfin, en ce qui concerne la complexité de cet algorithme, elle est également en $O(\log(n))$ où n est l'exposant. Nous ne ferons pas la preuve détaillée de ceci car l'idée est la même que dans l'algorithme de recherche d'un élément dans une liste triée. En effet, à chaque itération de la boucle, la puissance n devient $n//2$ donc elle est au minimum divisée par 2 (tout comme la longueur de notre tableau est au minimum divisée par 2). Cette complexité logarithmique est bien meilleure que la complexité linéaire de l'algorithme naïf.

Remarques : i) C'est avec cette méthode d'exponentiation rapide que Python effectue des élévations à une puissance.

ii) Si l'on considère la décomposition binaire de n :

$$n = \sum_{i=0}^{d} d_i 2^i \text{ avec : } \forall i \in [\![0, d]\!],\ d_i \in \{0, 1\}$$

on a :

$$a^n = a^{\sum_{i=0}^{d} d_i 2^i} = \prod_{i=0}^{d} a^{d_i 2^i} = \prod_{i=0}^{d} (a^{2^i})^{d_i}$$

Cette écriture met en évidence le fait que le nombre de multiplications à effectuer est de l'ordre de grandeur du nombre de chiffres dans l'écriture en binaire de n, c'est-à-dire de $\log_2(n)$.

3 Annexe

On considère l'algorithme de recherche d'un élément dans une liste triée que nous avons donné dans le paragraphe 1.3. Pour étudier la complexité de cet algorithme, il est important de savoir combien de passages dans la boucle *while* nous allons effectuer, en fonction de la longueur, n de la liste. Ce nombre de passages est variable car la fonction peut se terminer dès le premier passage dans la boucle dans le cas où $L[m] = a$, c'est pour cela que nous allons examiner le nombre de passages dans le pire des cas.

3.1 Complexité

• La présence de la boucle *while* ne nous permet pas de savoir facilement à l'avance combien de passages dans la boucle vont être effectués. On se place dans le pire des cas, c'est-à-dire celui où l'élément recherché ne se trouve pas dans le tableau et est supérieur à tous les éléments du tableau, ce qui fait que l'on choisira systématiquement la sous-liste de droite qui est de longueur supérieure ou égale à la sous-liste de gauche.

• Pour comprendre le principe, plaçons-nous dans le cas idéal où le nombre d'éléments de notre liste est de la forme $2^p - 1$ avec $p \in \mathbb{N}^*$. Dans l'algorithme, on retire l'élément central et on partage les $2^p - 2$ éléments restants en deux sous-listes ayant donc chacune $2^{p-1} - 1$ éléments. On recommence ensuite le procédé avec la sous-liste de gauche ou de droite. La puissance de 2 mise en jeu diminue de 1 à chaque étape ainsi au bout de p étapes la sous-liste étudiée est vide (ce qui correspond à $g > d$) et on sort de la boucle *while*.

• Par exemple, si l'on commence avec une liste contenant 127 éléments, on passe à 63, 31, 15, 7, 3, 1 et enfin 0 élément. C'est d'ailleurs facile à vérifier en ajoutant la commande $print(d-g+1)$ entre les lignes 7 et 8 de notre algorithme, ceci en remarquant que $d-g+1$ est bien le nombre d'éléments de la sous-liste commençant à l'indice g et se terminant à l'indice d.

• Dans le cas général, en notant toujours n le nombre d'éléments de la liste, on se donne $p \in \mathbb{N}$ tel que : $2^p - 1 < n \leq 2^{p+1} - 1$. **Dans le pire des cas, la boucle** *while* **est parcourue au plus** $p+1$ **fois.** En effet, on sait à présent que dans le pire des cas, il y a $p+1$ itérations pour une liste de longueur $2^{p+1} - 1$ et donc nécessairement moins d'itérations pour une liste de longueur moindre. En résumé :

si $2^p - 1 < n \leq 2^{p+1} - 1$ **alors il y a au plus** $p+1$ **passages dans la boucle while**

• De façon, plus explicite, on a :

$$2^p - 1 < n \leq 2^{p+1} - 1 \Leftrightarrow 2^p < n + 1 \leq 2^{p+1} \Leftrightarrow p < \log_2(n) \leq p + 1$$

Ceci implique que p est la partie entière de $\log_2(n)$ ou la partie entière de $\log_2(n) + 1$. Dans les deux cas, on en déduit que :

l'algorithme se termine

• Au sein d'une boucle *while*, le nombre d'opérations est fini et il y a $\lfloor \log_2(n) \rfloor$ ou $\lfloor \log_2(n)+1 \rfloor$ passages dans la boucle.

La complexité est logarithme en $O(\log_2(n))$

3.1.1 Terminaison

Montrons, comme annoncé dans le paragraphe 1.4.2, que la quantité $d - g + 1$ est un variant de boucle. Cette quantité est bien un entier naturel puisque d et g restent des entiers naturels. De plus, $d - g + 1$ décroit strictement à chaque passage dans la boucle. En effet, notons d' et g' les nouvelles valeurs de d et g après une itération de la boucle et montrons que $d' - g' + 1 < d - g + 1$. Il y a deux cas :

• Soit $g' = m + 1$ avec $m = (d + g)//2$ et $d' = d$ (le cas du $elif$) et :

$$d' - g' + 1 = d - ((d + g)//2 + 1) + 1 < d - g + 1$$

car $(d + g)//2 + 1 > g$ puisque $d \geq g$.

• Soit $g' = g$ et $d' = m - 1$ avec $m = (d + g)//2$ (le cas du $else$) et :

$$d' - g' + 1 = (d + g)//2 - 1 - g + 1 < d - g + 1$$

car $(d + g)//2 - 1 < d$ puisque $d \geq g$.

La décroissante stricte est bien démontrée et nous sommes bien en présence d'un variant de boucle. La boucle *while* se termine.

3.1.2 Correction

On utilise l'invariant de boucle donné dans le cours :

$\mathcal{P}$: "si $a \in L$ alors a se situe entre les indices g et d (inclus)"

• Avant de rentrer dans la boucle, on a $g = 0$ et $d = len(L) - 1$ et il est clair que si a est dans la liste alors son indice est entre 0 et $len(L) - 1$.

• On suppose la propriété vraie au début du corps de la boucle, montrons qu'elle reste vraie à la fin de la boucle. Il y a trois cas à examiner :

▶ Si $L[m] = a$, on a trouvé l'élément et on renvoie son indice donc la fonction renvoie le bon résultat et notre invariant de boucle ne sert pas.

▶ Si $L[m] < a$, comme notre liste est triée, si l'élément a est dans la liste alors c'est nécessairement dans la partie droite de la liste donc encore entre les indices g et d avec g qui a changé de valeur puisque $g = m + 1$. La propriété reste donc vérifiée.

▶ Si $L[m] > a$, comme notre liste est triée, si l'élément a est dans la liste alors c'est nécessairement dans la partie gauche de la liste donc encore entre les indices g et d avec d qui a changé de valeur puisque $d = m - 1$. La propriété reste donc vérifiée.

• Lorsque l'on sort de la boucle, on a $g > d$ donc si a est dans la liste, il se situe entre les indices g et d donc dans la liste vide. Ceci n'est pas possible donc a n'est pas dans la liste et on renvoie $False$ comme attendu.