Chapitre 11

Complexité d'un algorithme

11.1 Introduction

Pour bien programmer, il faut d'abord éviter les erreurs de syntaxe, écrire du code clair, bien commenté, décomposer son programme en fonctions, et éventuellement les regrouper dans des modules que l'on pourra ré-employer.

Mais un bon programme doit surtout appliquer les bons algorithmes. De façon évidente certains algorithmes seront meilleurs que d'autres.

Un critère important de qualité d'un algorithme est sa **complexité** : la complexité d'un algorithme est un concept mathématique qui précise comment son temps d'exécution variera lorsque la taille de données auquel il s'applique augmente.

Dans ce cours nous allons définir ce qu'est la complexité d'un algorithme, montrer comment la calculer, et l'appliquer sur quelques exemple.

11.1.1 Exemple 1

• Ecrire une fonction `suiteFibonacci` qui prend en paramètre un entier `n` et retourne le terme u_n de la suite de Fibonacci :

$$u_0 = 0, \quad u_1 = 1, \quad \forall\, n \in \mathbb{N},\ u_{n+2} = u_n + u_{n+1}$$

```
def suiteFibonacci(n):
    u, v = 0, 1
    for k in range(n-1):
        u, v = v, u+v
    return v
```

Mesurons son temps d'exécution pour plusieurs valeurs du paramètre `n` :

```
In [2]: %timeit suiteFibonacci(10)
[...]    1.13 µs
In [3]: %timeit suiteFibonacci(100)
[...]    8.16 µs
In [4]: %timeit suiteFibonacci(1000)
[...]    106 µs
```

Son temps d'exécution est approximativement proportionnel à `n`. En effet, il est du même ordre que le nombre de passages dans la boucle. A chaque passage 1 addition et deux affectations sont exécutées, d'où une durée presque identique à chaque passage dans la boucle.

Maintenant on demande la code d'une fonction `sommeFibonacci` qui prend un paramètre `n` et renvoie la somme $\sum_{k=0}^{n} u_k$ des $n+1$ premiers termes de la suite de Fibonacci.

Beaucoup d'élèves codent ainsi :

```
def sommeFibonacci(n):
    S = 0
    for k in range(n+1):
        S += suiteFibonacci(k)
    return S
```

Le code est de mauvais : son temps d'exécution est de l'ordre de :

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$$

```
In [5]: %timeit sommeFibonacci(10)
[...]    9.99 µs
In [6]: %timeit sommeFibonacci(100)
[...]    434 µs
In [7]: %timeit sommeFibonacci(1000)
[...]    49.5 ms
```

Le temps d'exécution est approximativement proportionnel à n^2.

Codons plutôt ainsi :

```
def sommeFibonacci(n):
    u, v = 0, 1
    S = u+v
    for k in range(n-1):
        u, v = v, u+v
        S += v
    return S
```

Le temps d'exécution sera proportionnel à n (environ).

```
In [8]: %timeit sommeFibonacci(10)
[...]    1.58 µs
In [9]: %timeit sommeFibonacci(100)
[...]    14.3 µs
In [10]: %timeit sommeFibonacci(1000)
[...]    177 µs
```

On retrouve un temps d'exécution linéaire. Pour $n = 1000$, le temps d'exécution a été divisé par près de 300 !

L'écart serait encore plus important pour $n = 10000$, 100000, etc...

11.1.2 Exemple 2

Ecrire en python une fonction qui prend en argument une chaîne de caractère et détermine si le caractère 'a' est présent dans la chaîne. Analysons plusieurs solutions.

• Première solution :

```
def contienta(chaine):
    N = len(chaine)
    k = 0
    result = False
    while (result == False and k < N):
        if chaine[k] == 'a':              # boucle
            result = True                 # ayant une
        k += 1                            # durée = a
    return result
```

• L'algorithme consiste à parcourir les caractères de la chaîne tant qu'on n'a pas trouvé le caractère 'a'. Le résultat est stocké dans une variable booléenne valant initialement `False` et passant à `True` dès que le caractère `'a'` est lu.

• Le temps d'exécution du programme est du même ordre de grandeur que la première occurrence de 'a' dans la chaine. Il existe des constantes a et b, dépendant du temps d'éxécution d'opérations élémentaires, comme l'affectation ou l'incrémentation, tels que si k est l'indice de la première occurrence de `'a'` dans `chaine`, le temps d'exécution sera $\approx a.k + b$.

• Pour des chaines de taille N, le temps d'exécution sera au plus $a.N + b$ (lorsque le caractère 'a' est absent ou présent en dernière place). C'est le temps d'exécution dans le pire des cas pour des entrées de taille N. Il est ici linéaire au sens que le temps d'exécution a pour asymptote la droite oblique $N \mapsto (a.N + b)$.

• Dans le meilleur des cas, celui où `'a'` se trouve en début de chaine, le temps d'exécution sera `a+b`.

• Deuxième solution :

```
def contienta(chaine):
    n = chaine.count('a')    # Compte le nombre de 'a'
    return bool(n)    # True si n ≠ 0, False sinon.
```

• Le code est plus court ! Est-il meilleur ? Bien au contraire !

• On compte le nombre de fois où le caractère `'a'` apparaît dans `chaine` à l'aide de la méthode `.count()` des objets séquentiels. Il faut nécessairement lire tous les caractères de `chaine`, donc pour n'importe quelle chaîne de longueur N il faudra un temps d'exécution de la forme $c.N + d$ pour des constantes c et b.

• Dans le pire des cas le temps de calcul est encore linéaire en fonction de la taille des données.

• Mais la complexité dans le meilleur des cas et en moyenne est désormais aussi $c.N + d$. Elle était meilleure auparavant.

• Pour de grandes valeurs de N, c'est à dire pour des données de grande taille, notre précédent programme devrait être en moyenne deux fois plus rapide.

• Troisième solution :

```
def contienta(chaine):
    return ('a' in chaine)
```

• Le code est encore plus court. Mais le temps d'exécution en moyenne et dans le pire des cas pour une entrée de taille N sera comme dans le première exemple, de même ordre de qualité, linéaire.

L'utilisation de fonctions préprogrammées spécifiques améliore le temps d'exécution. Mais ici ce sont les constantes a et b qui sont optimisées. L'instruction `in` ne peut que lire les éléments de la chaine l'un après l'autre tant qu'elle n'a pas trouvé 'a', puisqu'il n' y a aucun moyen de savoir à priori si `'a'` figure dans `chaine`.

L'algorithme employé est d'aussi bonne qualité que dans la première solution. On ne peut pas faire mieux.

11.1.3 Approche intuitive de la complexité

• A l'exécution des deux programmes on ne verrait une différence pour des chaines de longueur petite, qu'en la mesurant (`%timeit` ou module `time`), l'exécution étant quasi-instantanée. Par contre pour des chaines de très grandes longueurs on pourrait percevoir une différence, et une relative lenteur de la deuxième solution par rapport aux deux autres.

• De nombreux systèmes informatiques manipulent des données de très grandes : Météorologie, gestion du trafic aérien, statistiques de la population, traitement d'image, serveurs, réseau sociaux, etc...

• Soit N_{max} la taille maximale des données pour que l'exécution d'un programme s'exécute efficacement dans un temps d'exécution acceptable. De quelle capacité informatique faudrait-il se doter pour doubler la capacité de traitement $2N_{max}$, pour la multiplier par 10 : $10N_{max}$?

• Si l'algorithme employé a un temps d'exécution linéaire, $a.N + b \approx a.N$, il suffit approximativement respectivement de doubler, ou de multiplier par 10 son parc informatique en pratiquant du calcul en parallèle, ou parallélisme.

• Si l'algorithme employé a un temps d'exécution quadratique $a.N^2 + b.N + c \approx a.N^2$, il suffit de multiplier approximativement son parc informatique par 4, ou 100 !

• Si l'algorithme employé a un temps d'exécution exponentiel, par exemple $a.2^N$ il faudrait élever les capacités du parc informatique au carré, ou à la puissance 10.

En effet : $2^{N_{max}} \leqslant K_{max} \implies 2^{2N_{max}} \leqslant K_{max}^2$ et $2^{10N_{max}} \leqslant K_{max}^{10}$.

C'est pourquoi une solution algorithme de complexité de temps de calcul linéaire en fonction de la taille des données présente de considérables avantages par rapport à une solution de complexité quadratique, pire : exponentielle, etc...

Comparons les temps d'exécution sur un ordinateur à 10 GHz (10 milliards d'opérations à la seconde) dans le pire des cas pour divers tailles de données et différentes fonction de complexités. La taille des données est N.

	$N = 20$	$N = 50$	$N = 100$	$N = 200$
$1000.N$	0.000002s	0.000005s	0.00001s	0.00002s
$100.N^2$	0.000004s	0.000025s	0.0001s	0.0004s
$10.N^3$	0.000008s	0.000125s	0.001s	0.008s
2^N	0.0001s	1,3 jours	-	-
3^N	0.35s.	23.10^6 années	-	-
$N!$	7,7 ans	-	-	-
- : supérieur à 100 milliards d'années				

11.2 Définition de la complexité

• On l'a vu puisque la taille des données numériques est bornée par leur représentation, le temps d'exécution de l'addition de 2 nombres, de leur multiplication, etc..., sont bornés par une constante dépendant de la machine et de l'interpréteur. Il en est de même si l'on effectue n'importe quelle opération mathématiques usuelles : `+,-,*,/,//,%`.

• Pour évaluer comment le temps d'exécution d'un programme évoluera en fonction de la taille N de ses données, on compte le nombre d'opérations dont le temps d'exécution ne dépend pas de N, qu'on exprime comme fonction de N.

• Le comportement asymptotique de cette fonction (linéaire, quadratique, exponentiel,...) définit la *complexité de l'algorithme*.

11.2.1 Opérations élémentaire

• On appelle opérations élémentaires toutes les opérations consistant en :

- une comparaison, un test, une opération logique,
- une opération arithmétique, une fonction mathématiques,
- l'accès à un élément d'une structure de donnée (liste, chaine, etc..), sa modification,
- l'affectation de variable,
- saisie/impression/retour d'une donnée à l'écran, ou dans un fichier.
- Toute instruction ou opération prédéfinie, basique, s'effectuant sur une donnée de taille fixée par le système, et dont le temps d'exécution est majoré par une constante ne dépendant que du système.

• Par exemple :

- Lire ou modifier un élément dans une liste est une opération élémentaire.
- Le calcul de la longueur d'une liste est une opération élémentaire (l'interpréteur la connait à priori).
- Les instructions `liste.count('a')` ou `sum(liste)` ne sont pas des opération élémentaires. Leur temps d'exécution dépend de la longueur de `liste`.

11.2.2 Fonctions de complexité

• Donné un algorithme ou un programme prenant en paramètre des données de taille N variable.

• Sa **Fonction de complexité (en temps) dans le pire des cas** de l'algorithme, est l'application $F_{max} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $F_{max}(N)$ est le nombre maximum d'opérations élémentaires nécessaires pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus défavorable).

• Sa **Fonction de complexité (en temps) dans le meilleur des cas** de l'algorithme, est l'application $F_{min} : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $F_{min}(N)$ est le nombre minimum d'opérations élémentaires nécessaires pour exécuter le programme sur une entrée de taille N (celui dans le cas le plus favorable).

• Sa **Fonction de complexité (en temps) en moyenne** de l'algorithme, est l'application $F_{moy} : \mathbb{N} \longrightarrow \mathbb{R}$ définie par $F_{moy}(N)$ est le nombre moyens d'opérations élémentaires effectuées lorsque l'on

exécute le programme sur les entrées de taille N.

• On ne s'intéresse pas au détail de la fonction de complexité, mais à son comportement asymptotique, sa croissance en fonction de N au voisinage de $+\infty$. Précisément on s'intéresse à l'ordre de la fonction de complexité :

11.2.3 Ordre d'une application

Définitions

• Soient f et g deux applications de $\mathbb{N}$ dans $\mathbb{R}_+$.

On note $f = O(g)$ si il existe une constante réelle $C > 0$ et $n_0 \in \mathbb{N}$ tels que pour tout $n \geqslant n_0$: $f(n) \leqslant C \times g(n)$.

$$f = O(g) \iff \exists C > 0, \exists n_0 \in \mathbb{N},\ \forall n \in \mathbb{N},\ n \geqslant n_0 \implies |f(n)| \leqslant C \times |g(n)|$$

Lorsque f/g existe : $f = O(g) \iff f/g$ est borné.
Notamment lorsque f/g a une limite finie.

• $f = \Theta(g)$, et on dit que f et g ont même ordre si : $f = O(g)$ et $g = O(f)$.

C'est le cas notamment lorsque f/g a une limite finie $\neq 0$ en ∞.

Exemples :

- $\ln(n) = O(n)$, $n = O(n^2)$, $\forall\, a > 0,\ n^a = O(\exp(n))$
- $n^2 + n + 1 = \Theta(n^2) = O(n^3) = O(n^4) = \cdots = O(e^n)$.
- $f = o(g) \implies f = O(g)$ et $f \neq \Theta(g)$
- $f = o(g) \implies f + g = \Theta(g)$
- $f \underset{+\infty}{\sim} g \implies f = \Theta(g)$

Propriétés :

- La relation O est un quasi ordre partiel : c'est une relation réflexive et transitive.
- La relation Θ est une relation d'équivalence : réflexive, symétrique, transitive.
- Compatibles avec l'addition : $O(f) + O(g) = O(f + g)$ $O(f) + O(f) = O(f)$
- Compatibles avec la multiplication :
  Soit $a > 0$ une constante : $O(a \cdot f) = O(f)$, $\Theta(a \cdot f) = \Theta(f)$, $O(f) \times O(g) = O(f \times g)$, $\Theta(f) \times \Theta(g) = \Theta(f \times g)$.

On a coutume de noter :

- $O(1)$ l'ordre d'une application bornée.
- $\Theta(\ln(n))$ l'ordre de $\ln(n)$.
- $\Theta(n)$ l'ordre des applications linéaires non nulles.
- $\Theta(n^2)$ l'ordre des applications quadratiques kn^2 , $k > 0$.
- $\Theta(f)$ l'ordre de f, c.à d. ensemble des applications de même ordre que f.

11.2.4 Complexité d'un algorithme

Si un algorithme a une fonction de complexité, respectivement :

- dans le pire des cas,
- dans le meilleur des cas,
- en moyenne,

$T(n)$ on dira que l'**algorithme est de complexité**, respectivement

- dans le pire des cas,
- dans le meilleur des cas,
- en moyenne)

bornée si $T(n)$ est majorée (ordre $O(1)$).

logarithmique si $T(n)$ a pour ordre $\Theta(\ln(n))$.

linéaire si $T(n)$ a pour ordre $\Theta(n)$.

quadratique si $T(n)$ a pour ordre $\Theta(n^2)$.

polynomiale si $\exists k \in N$, $T(n)$ a pour ordre $\Theta(n^k)$.

exponentielle si $\exists a > 1$, $T(n)$ a pour ordre $\Theta(a^n)$.

11.2.5 Calculs de complexité

• Cas d'école :
Compter le nombre d'occurrences d'un élément dans une liste de longueur variable N :

```
def compte(liste,a):
    compteur = 0
    for x in liste
        if x == a:
            compteur += 1
    return compteur
```

Pour une entrée de taille N le nombre d'opérations élémentaires est égal à $T(N) = 2 + 2.N + k$ où k est le nombre d'occurrence de `a` dans `liste`, $0 \leqslant k \leqslant N$.

La fonction de complexité dans le pire des cas est $F_{max}(N) = 2+3N$; l'algorithme est de complexité, dans le pire des cas, linéaire, d'ordre $\Theta(N)$.

La fonction de complexité dans le meilleur des cas est $F_{min}(N) = 2 + 2N$; l'algorithme est de complexité, dans le pire des cas, linéaire, d'ordre $\Theta(N)$.

Il n'est pas possible de faire mieux : pour compter le nombre d'occurrence de `a` il faut lire chaque élément de la liste soit au moins N opérations élémentaires.

Tout algorithme de la forme :

```
for x in liste :
    suite d'opérations élémentaires sans boucle ou sortie
    anticipée (break, return)
```

est de complexité dans le pire des cas, le meilleur des cas et en moyenne linéaire en fonction de la longueur de la liste.

• Autres cas d'école :
Recherche d'un élément dans une liste :

```
def find(liste,a):
    for x in liste :
        if x == a:
            return True
    return False
```

est de complexité dans le pire des cas, linéaire.

Il n'est pas possible de faire mieux (dans le pire des cas, celui où l'élément est absent, tous les élément doivent être lus au moins une fois, puisqu'on n'a aucune information à priori sur la position que pourrait occuper `a`, ainsi au moins N opérations).

de façon générale lorsque l'entrée est de taille N :

```
for i in [[1,N]]:
    for j in [[1,N]]:
        que des opérations élémentaires sans boucle ou
        sortie anticipée
```

est de complexité quadratique N^2.

Exemple : Calcul d'une somme double :

$$\sum_{0 \leqslant i,j \leqslant N} a_{i,j}$$

```
S = 0
for i in range(n):
    for j in range(n):
        S += a[i][j]
print(S)
```

est de complexité quadratique en moyenne et dans le pire et le meilleur des cas.

Parcours d'un tableau triangulaire :

```
for k in [[1,N]]:
    for j in [[1,k]]:
        que des opérations élémentaires sans boucle
        ou sortie anticipée
```

est de complexité quadratique N^2.

En effet supposons qu'il y ait C opérations élémentaires dans la boucle.

La fonction de complexité vaut dans chaque cas $C+2C+3C+\cdots NC = C\frac{N(N+1)}{2} = \frac{C}{2}(N^2+N)$ qui est d'ordre quadratique.

Ainsi les fonctions de complexité en moyenne, dans le meilleur et dans le pire des cas sont toutes quadratiques.

11.2.6 Exemple : Recherche d'un motif

Exemple : Fonction prenant en argument deux chaînes de caractère `chaine` et `motif` et qui détermine si la chaine `motif` apparaît comme sous-chaine dans `chaine`.

```
def contient(chaine, motif):
    N, n = len(chaine), len(motif)
    for k in range(N-n+1):
        if chaine[k:k+n] == motif:
            return True
    return False
```

Quelle est sa complexité en fonction de :

`N` longueur de `chaine`,

`n` longueur de `motif`

dans le pire et le meilleur des cas ?

```
def contient(chaine, motif):
    N, n = len(chaine), len(motif)    # O(1)
    for k in range(N-n+1):
        if chaine[k:k+n] == motif:    # Θ(n)
            return True               # O(1)
    return False                      # O(1)
```

• Dans le meilleur des cas, celui où `motif` se trouve en début de chaine :
Un seul passage dans la boucle for : complexité $\Theta(n)+2\cdot O(1) = \Theta(n)$.
La complexité dans le meilleur des cas est $\Theta(n)$.

• Dans le pire des cas, celui où `motif` se trouve en toute fin de chaine, ou n'apparaît pas :
`N-n+1` passages dans la boucle for :
complexité : $2\cdot O(1) + (N-n+1)\cdot\Theta(n) = \Theta(n(N-n))$.
La complexité dans le pire des cas est $\Theta(n(N-n))$.

• Lorsque `motif` est de taille bornée, la complexité en fonction de `N` est $O(1)$ dans le meilleur des cas et $O(N)$ dans le pire des cas.

11.3 Exemple : Recherche par dichotomie d'un élément dans une liste triée

11.3.1 Principe

• Exemple : recherche d'un nombre dans une liste triée.

Nous codons en TD comment rechercher par dichotomie un élément dans une liste triée dans le sens croissant (ou décroissant).

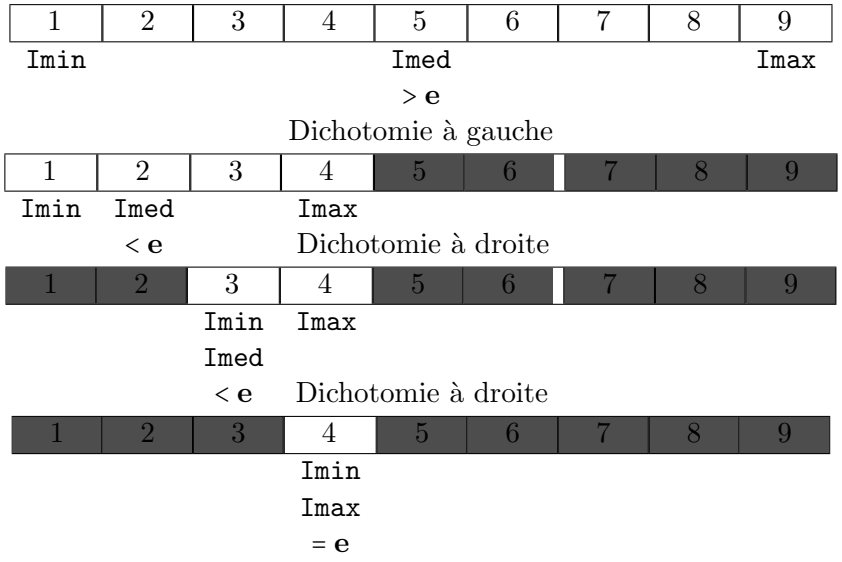
On peut toujours appliquer l'algorithme de recherche d'un élément dans une liste déjà vu, ou appelé par l'instruction `x in liste`. Mais il ne met pas à profit l'information donnée : la liste est ordonnée. Sa complexité est linéaire dans le pire des cas.

Une meilleur idée consiste à appliquer une recherche par dichotomie :

- Comparer l'élément recherché `e` avec l'élément en milieu de liste `m = L[len(L)//2]`.
- En cas d'égalité : l'élément est trouvé : sortie.
- Si `e<m` recommencer sur la première moitié de la liste.
- Si `e>m` recommencer sur la deuxième moitié de la liste.

Utiliser deux variables `Imin` et `Imax` de type entiers qui délimitent la partie du tableau à inspecter. `Imed` est l'indice du milieu :

Exemple : Recherche de **e = 4** dans le tableau suivant :



11.3.2 Code

Méthode par dichotomie :

```
def dich_search(l,e):
# Recherche dichotomique dans une liste croissante
    Imin, Imax = 0, len(l)-1
    while Imax - Imin >= 0:
        Imed = (Imin + Imax)//2
        if l[Imed] == e:        # Cas de succés
            return True
        elif l[Imed] < e:
            Imin = Imed+1         # dichotomie à droite
        else:
            Imax = Imed-1         # dichotomie à gauche
    return False    # Cas d'échec
```

11.3.3 Calcul de la complexité

• Pour une liste de taille N le nombre d'opérations élémentaires pour rechercher un élément par dichotomie est du même ordre que la profondeur de la dichotomie (chaque étape consistant en 1 calcul, 1 affectation, au plus 2 tests, et 1 opération élémentaire).

• Alors pour une liste de N éléments combien est au plus la profondeur de la dichotomie, pour trouver un élément, en fonction de N ?

• Il est plus facile de calculer la fonction réciproque : Si pour une liste au plus k profondeurs de dichotomies sont nécessaires pour chercher un élément, combien d'éléments contient cette liste approximativement ?

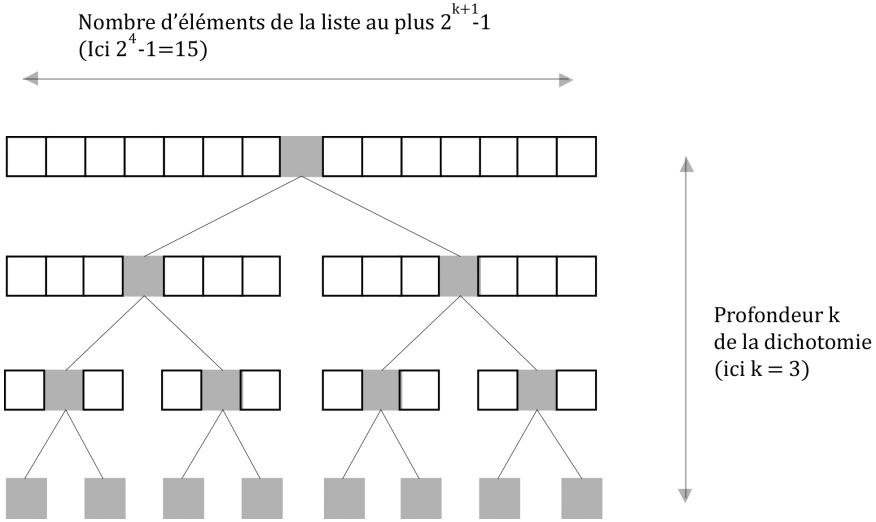
- A chaque dichotomie on sépare la liste en 2 listes de tailles égales ± 1.
- Donc si à l'étape $i+1$ la liste contient N_{i+1} éléments, alors à l'étape i la liste contient entre $2 \times N_{i+1}$ et $2 \times (N_{i+1}+1)$ éléments .
- A la dernière étape, dans le pire des cas, la liste ne contient plus qu'un élément.

Ainsi la liste était initialement de taille comprise entre 2^k et au plus $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$.

• Donc en composant par $\log_2$ (qui est croissante) :

$$\begin{aligned} 2^k \leqslant N \leqslant 2^{k+1} &\implies k \leqslant \log_2(N) \leqslant k+1 \leqslant 2k \\ &\implies k = O(\log_2(N)) \text{ et } \log_2(N) = O(k) \end{aligned}$$

La profondeur de dichotomie k a pour ordre $\Theta(\log_2(N))$.

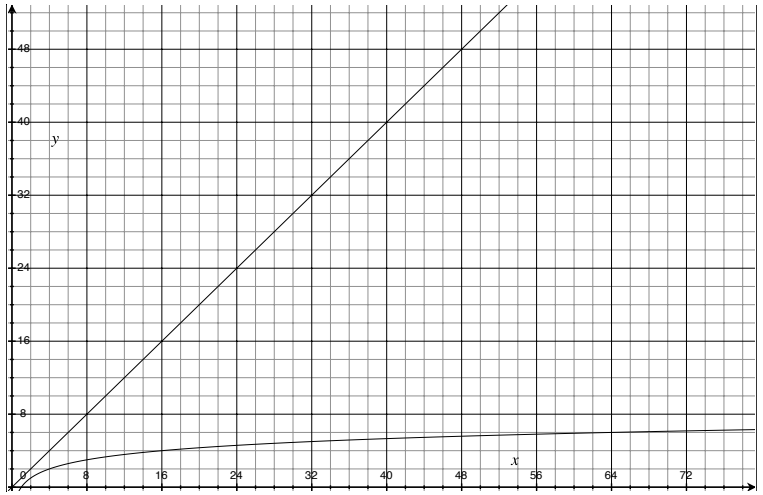


• Les cases grisées représentent les seuls éléments susceptibles d'être comparés avec l'élément recherché aux diverses étapes de la dichotomie.

Lors d'une implémentation, seules celles sur une branche (jusqu'en bas dans le pire des cas) seront comparées, alors que l'algorithme naïf peut comparer avec tous les éléments de la liste.

• La complexité d'une recherche par dichotomie dans une liste triée dans le sens croissant est dans le pire des cas :

$O(\log_2(N))$ **de complexité logarithmique**



• C'est beaucoup plus rapide par dichotomie que par une recherche naïve, linéaire :

11.3.4 Vérification : comparaison des temps d'exécution

```
from random import random
from time import clock
L = [random() for k in range(100000)]
L.sort()
#
a = clock()
recherche(L,0)    # Recherche non dichotomique
b = clock()
print("recherche non dichotomique",b-a,"secondes")
#
dich_search(L,0)
b = clock()
print("recherche dichotomique",b-a,"secondes")
```

Résultats :

```
recherche non dichotomique 0.010511999999998523 secondes
recherche dichotomique     1.2000000001677336e-05 secondes
```

C'est 1000 fois plus rapide par dichotomie pour une recherche dans un tableau de 100000 éléments !