Chapter 1

Algorithmes

L'EDVAC est le successeur de l'ENIAC; contrairement à ce dernier il fonctionnait en binaire. Il utilisait des cartes perforées comme support d'entrée et de sortie (les cartes perforées sortantes étaient imprimées hors ligne sur une tabulatrice).

Considéré comme l'un des premiers ordinateurs entièrement électronique, il occupait 45,5 m^2 pour un poids de 7 850 kg mais sa mémoire n'était que de 5,5 kilo-octets.

John Von Neumann a participé à l'élaboration de cette machine avec John Eckert et John Mauchly. On doit l'expression **architecture de Von Neumann** à son travail sur cette machine: *First Draft of a Report on the EDVAC*. A noter qu'il attribuait lui-même à Alan Turing ce modèle de calculateur. Grace Hopper a créé le premier compilateur sur une évolution de l'EDVAC: l'UNIVAC.

EDVAC vers 1949

1.1 Approche de la complexité d'un algorithme.

1.1.1 Définition

Le traitement de certains algorithmes complexes peut nécessiter du temps et de la ressource machine: c'est ce qu'on appelle le coût de l'agorithme. Ce coût est calculable et reflète son efficacité: plus l'algorithme est coûteux, moins il est efficace, et pour une même tâche certains algorithmes se révèlent plus coûteux que d'autres.

Il existe deux types de complexité :

Complexité spatiale : permet de quantifier l'utilisation de la mémoire.

Complexité temporelle : permet de quantifier la vitesse d'exécution.

Nous nous intéressons à la complexité temporelle.

1.1.2 Calcul de la complexité

On peut mesurer le temps de calcul de façon expérimentale en chronométrant mais il est intéressant d'avoir une estimation théorique, en déterminant le nombre d'opérations significatives que fait l'algorithme.

Mode de calcul

Réaliser un calcul de complexité en temps revient à compter le nombre d'opérations élémentaires: affectation, calcul arithmétique ou logique, comparaison... effectuées par le programme.

On fera l'hypothèse que toutes les opérations élémentaires sont à égalité de coût, soit 1 *unité* de temps.

Exemple k = 5 * m : 1 multiplication + 1 affectation = 2 unités .

Il faut tenir compte de la taille des données passées en paramètre ainsi que de la façon dont se répartissent ses différents valeurs. On distinguera donc deux formes de complexité en temps :

- La complexité dans le meilleur des cas : c'est la situation la plus favorable (Exemple : recherche d'un élément situé à la première position d'une liste).
- La complexité dans le pire des cas : c'est la situation la plus défavorable, (Exemple: recherche d'un élément dans une liste alors qu'il n'y figure pas).

On envisagera toujours le pire des cas: on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé.

1.1.3 Exemple détaillé

L'exemple 1 prend en compte toutes les opérations élémentaires, nous verrons que ce n'est pas nécessaire.

Exemple 1

```
def factorielle(n):

    fact = 1 # initialisation: 1

    i = 2 # initialisation: 1

    while i <= n: # itérations: au plus n-1 et 1 test à chaque tour

        fact = fact * i # multiplication + affectation: 2

        i = i + 1 # addition + affectation: 2

    return fact # renvoi d'une valeur: 1
```

On a aussi un test à chaque itération à prendre en compte. Le nombre total d'opérations est donc :

$$1 + 1 + (n-1)5 + 1 = 5n - 2$$

Par conséquent: la complexité de cet algorithme est une fonction linéaire de n, son ordre de grandeur est noté: $O(n)$ (ou $\mathcal{C}_{factorielle}(n) = O(n)$)

1.2 Ordre de grandeur

1.2.1 Définition

On se rend compte que le calcul exact de la complexité va vite devenir impossible, c'est pourquoi on se contente d'un ordre de grandeur, ce qui reste pertinent, l'objectif étant de pouvoir comparer les algorithmes. On évalue l'efficacité d'un algorithme en donnant l'ordre de grandeur du nombre d'opérations qu'il effectue lorsque la taille du problème qu'il résout augmente. On parle ainsi d'algorithme linéaire, quadratique, logarithmique, etc. Les notations de Landau sont un moyen commode d'exprimer cet ordre de grandeur.

Trois situations sont décrites par ces notations. La plus fréquente, la notation O introduite ci-dessous, donne une majoration de l'ordre de grandeur. La force des notations de Landau réside dans leur concision.

Le calcul de cette complexité a donc comme résultat une fonction mathématique dépendant de la taille de la donnée qu'on donne sous forme simplifiée.

La complexité est notée *O(f(n))* où le grand O signifie ordre de grandeur. f est la fonction mathématique qui est la quantité d'information manipulée dans l'algorithme. *Voir Annexes*

Exemple: Soit un algorithme qui compte de 1 à n et qui affiche les valeurs correspondantes.

- Vous allez utiliser une boucle allant de 1 à n. Il faudra faire n passages pour tout afficher et donc vous aller manipuler n fois l'information. La fonction mathématique donnant le coût sera alors $f(n) = n$: la complexité est alors linéaire et vous la noterez *O(n)*.
- Si dans le même algorithme vous décidez de faire une seconde boucle dans la première, pour afficher par exemple une table de multiplication : la première boucle va toujours de 1 à n, la seconde va aussi de 1 à n.

  Au total vous obtenez n fois n boucles, donc n^2 boucles. La complexité est donc $f(n) = n^2$ et vous la noterez $O(n^2)$. Le coût de l'algorithme augmente au carré du nombre d'informations.
- Si vous rajoutez en plus une une opération dans la première boucle, elle aura un coût que vous pouvez prendre en compte. Si vous ajoutez une multiplication et qu'elle a un coût de 1, alors la complexité final est de $n \times (n + 1)$ soit $n^2 + n$. Cependant, si vous faites une courbe **pour de grandes valeurs de** n et que vous comparez avec celle de n^2, vous remarquerez que cet ajout devient négligeable.

Au final, l'algorithme conserve une complexité $O(n^2)$.

Lorsque l'on s'intéresse à la complexité $C(n)$ (n est la taille de l'entrée) d'une fonction, c'est bien souvent pour les grandes valeurs de n qu'il est pertinent de connaître $C(n)$, pour comparer vis à vis d'autres fonctions réalisant le même calcul. On cherche donc un comportement asymptotique de n, qu'on rapportera aux fonctions usuelles : logarithmes, puissances, exponentielles.

Comme pour le calcul effectué pour la factorielle, la complexité peut parfois être calculée finement, mais le plus souvent on se contente d'un ordre de grandeur. Il en existe à connaître:

- $O(1)$: complexité constante .
- $O(log(n))$: complexité logarithmique (recherche dichotomique).
- $O(n)$: complexité linéaire (recherche séquentielle d'une occurence.
- $O(n.log(n))$: complexité quasilinéaire.
- $O(n^2)$: complexité quadratique (tri par insertion).

1.2.2 Exemple détaillé pour deux boucles imbriquées

Détaillons les cas possibles en présence de deux boucles imbriquées: Soit n la taille d'une donnée. Dans les codes suivants, on désigne un nombre fixe d'opérations.

1. La boucle interne ne dépend pas de n.

```
for i in range(n):
    'q opérations'
    for j in range(k):
        'r opérations'
```

   - Dans la boucle interne, il y a k passages soit $k \times r$ opérations pour cette boucle.
   - Dans la boucle externe il y a q opérations, la boucle externe boucle donc sur $q + k \times r$ opérations. Cette valeur ne dépend pas de n et en notant $\alpha = q + k\times$ r pour simplifier, nous obtenons αn opérations au total, **le coût est donc linéaire.**

2. Deuxième cas: on boucle jusque n également dans la deuxième boucle.

```
for i in range(n):
    'q opérations'
    for j in range(n):
        'r opérations'
```

   Il y a cette fois $n \times r$ opérations dans la seconde boucle et le résultat devient:

   $$n(q + r \times n) = nq + rn^2 \text{ . } \textbf{Le coût est quadratique.}$$

3. Dernier cas: on boucle jusque i dans la seconde boucle. Le calcul est un peu plus délicat.

```
for i in range(n):
    'q opérations'
    for j in range(i):
        'r opérations'
```

   Il y a $q + i \times r$ opérations avec i variant de 0 à $n - 1$:

$$\begin{aligned} & q + (q + r) + (q + 2r) + ... + (q + (n - 1) \times r) && (1.1)\\ &= nq + (r + 2r + ...(n - 1) \times r) && (1.2)\\ &= nq + r(1 + 2 + ... + n - 1) && (1.3)\\ &= nq + r \times (n - 1)\frac{1 + (n - 1)}{2} && (1.4)\\ &= nq + r \times \frac{n(n - 1)}{2} && (1.5)\\ & && (1.6) \end{aligned}$$

   Nous obtenons une complexité de type $\alpha n^2 + \beta n$ c'est à dire **quadratique**.

Exemple 2 Voici une fonction où les boucles sont imbriquées, on considère le nombre d'affectation comme opération élémentaire à dénombrer.

```
pour i allant de 1 à n faire
    pour j allant de 1 à i faire
        S ⟵ S + 1
    fin
fin
```

Algorithme 1 : Deux boucles imbriquées

- On compte une affectation dans la boucle d'indice j: il s'agit donc d'additionner sur i tours de boucles une opérations élémentaire de coût 1; soit $\sum_{j=1}^{i} 1 = i$ affectations.
- Et en tenant compte de la boucle d'indice i:

$$\sum_{i=1}^{n}\left(\sum_{j=1}^{i} 1\right) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

La complexité de cet algorithme est fonction de n:

$$\mathcal{C}(n) = \frac{n(n+1)}{2}$$

Par conséquent: $\mathcal{C}_{somme}(n) = \Theta(n^2)$

1.2.3 Exercices

Exercice 1 Détailler le calcul de la complexité de la fonction `index` en tenant compte de toutes les opérations et affectations. En donner un ordre de grandeur.

```python
def index(elem, L):
    """
    retourne l'index de l'occurrence elem

    """
    i = 0
    continuer = True
    long =  len(L)
    while i < long and continuer:
        if L[i] == elem:
            continuer = False
        else:
            i = i + 1
    return i
```

Exercice 2 Voici une fonction pour convertir des secondes:

```python
def convertir(nb_sec):
    h = nb_sec // 3600
    m = (nb_sec - 3600*h) // 60
    s = nb_sec % 60
    return h,m,s
```

Donner le coût en temps en fonction de `nb_sec`.

Exercice 3

1. Quel est le rôle de cette fonction ?
2. En estimer la complexité.

```
def mystere(n):
  if n%2==0:
     res = 1
  else:
     res = -1
  return res
```

1.3 Terminaison d'un algorithme

Lorsqu'on se trouve en présence d'une boucle while, celle-ci est susceptible de diverger, on a alors une boucle infinie. Nous serons donc amenés à vérifier la terminaison de cette boucle. Une technique possible est celle du variant de boucle.

Définition: Un variant de boucle est une quantité positive, à valeurs dans $\mathbb{N}$, dépendant des variables de la boucle, qui décroît strictement à chaque passage dans la boucle vers une valeur satisfaisant la condition d'arrêt.

Exemples

1. $n - i$ pour une boucle du type `while i<n` et i croissante.
2. $j - i$ pour une variable j décroissante et i croissante.

Méthode:

1. Si, pour une boucle donnée, on peut exhiber un variant de boucle, alors le nombre de passages dans la boucle est finie.
2. Si, pour un algorithme donné, on peut exhiber, pour toute boucle de l'algorithme, un variant de boucle, alors l'algorithme s'arrête en temps fini.

Attention: vérifier la *terminaison* ne signifie pas vérifier la *correction* de l'algorithme (la validité) qui consiste à vérifier que l'on aura le résultat attendu.

Exercices Voici deux algorithmes, justifier leur terminaison à l'aide d'un variant de boucle.

1.
```
a = int(input("a : "))
b = int(input("b : "))
m = 0
while b > 0:
   m += a
   b -= 1
print("a*b = ", m)
```

2.
```
a = int(input("a : "))
b = int(input("b : "))
i = 0
m = 0
while i < b:
   m += a
   i += 1
print("a*b = ", m)
```

1.4 Notes et références

1. Nous n'utiliserons pas les deux autres notations: la notation Ω donne une minoration, et la notation ϑ donne deux bornes sur l'ordre de grandeur. A tire informatif, voici une définition plus formelle d'un ordre de grandeur en "grand O" :

   $$\text{Une fonction } T(n) \text{ est en } O(f(n)) \text{ si : } \exists n_0 \in \mathbb{N}, \exists c \in \mathbb{R}, \forall n \geqslant n0 : |T(n)| \leqslant c|f(n)|$$

   Autrement dit, à partir d'un certain rang n_0, T peut toujours être approchée par f à une constante près.

Exemple: Soient les suites $u_n = 20n + 1$ et $v_n = 5n + 7$ alors $\lim_{x \to +\infty} \frac{u_n}{v_n} = 4$. La suite est bornée et par conséquent $u_n = O(v_n)$.

2. Voir *Eléments d'algorithmique-D. Beauquier, J. Berstel, Ph. Chrétienne* téléchargeable ici: `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf`

3. Voir `https://interstices.info/la-theorie-de-la-complexite-algorithmique/`