# Notion de complexité

## Algorithmique

NSI1

25 février 2022

# Principe

- on veut résoudre un problème ;

# Principe

- on veut résoudre un problème ;
- celui-ci a une *taille* $n \in \mathbf{N}$ ;

# Principe

- on veut résoudre un problème ;
- celui-ci a une *taille* $n \in \mathbf{N}$ ;
- on met au point un (ou des) algorithme(s) donnant la solution du problème ;

# Principe

- on veut résoudre un problème ;
- celui-ci a une *taille* $n \in \mathbf{N}$ ;
- on met au point un (ou des) algorithme(s) donnant la solution du problème ;
- on se pose la question suivante :

# Principe

- on veut résoudre un problème ;
- celui-ci a une *taille* $n \in \mathbf{N}$ ;
- on met au point un (ou des) algorithme(s) donnant la solution du problème ;
- on se pose la question suivante :

Mon algorithme est-il *efficace* ?

# Principe

- on veut résoudre un problème ;
- celui-ci a une *taille* $n \in \mathbf{N}$ ;
- on met au point un (ou des) algorithme(s) donnant la solution du problème ;
- on se pose la question suivante :

Mon algorithme est-il *efficace* ?

S'il y a plusieurs algorithmes, y en a-t-il un plus efficace que les autres ?

# Exemple

- on dispose d'une liste d'entiers et on veut savoir s'ils sont tous pairs ou non ;

# Exemple

- on dispose d'une liste d'entiers et on veut savoir s'ils sont tous pairs ou non ;
- notons $n$ la longueur de la liste, c'est la *taille du problème* ;

# Exemple

- on dispose d'une liste d'entiers et on veut savoir s'ils sont tous pairs ou non ;
- notons $n$ la longueur de la liste, c'est la *taille du problème* ;
- intéressons-nous à 3 algorithmes écrits en PYTHON.

# Algorithme 1

```
def tous_pairs1(lst: list) -> bool:
    resultat = True
    for x in lst:
        if x % 2 == 1:
            resultat = False
    return resultat
```

# Algorithme 2

```
def tous_pairs2(lst: list) -> bool:
    for x in lst:
        if x % 2 == 1:
            return False
    return True
```

# Algorithme 3

```
def tous_pairs1(lst: list) -> bool:
resultat = True
for x in lst:
if x % 2 == 1:
resultat = False
return resultat
```

# Questions

Quel programme semble le plus efficace / inefficace ...

# Questions

Quel programme semble le plus efficace / inefficace ...

- en termes d'opérations ?

# Questions

Quel programme semble le plus efficace / inefficace ...

- en termes d'opérations ?
- en termes de mémoire ?

# Réponses

Il n’y a pas de réponse « absolue » :

# Réponses

Il n'y a pas de réponse « absolue » :

- Qu'est-ce qu'on considère comme une « opération » ?

# Réponses

Il n'y a pas de réponse « absolue » :

- Qu'est-ce qu'on considère comme une « opération » ?
- Comment fonctionne *concrètement* PYTHON ?

# Réponses

Il n'y a pas de réponse « absolue » :

- Qu'est-ce qu'on considère comme une « opération » ?
- Comment fonctionne *concrètement* PYTHON ?
  (gestion de la mémoire / procédés de calculs)

# La complexité temporelle

# Protocole

Pour évaluer l'efficacité en terme de nombre d'opérations :

# Protocole

Pour évaluer l'efficacité en terme de nombre d'opérations :

- On se met d'accord sur ce qu'on considère comme *opération élémentaire* (OPEL).

# Protocole

Pour évaluer l'efficacité en terme de nombre d'opérations :

- On se met d'accord sur ce qu'on considère comme *opération élémentaire* (OPEL).
- On estime approximativement le temps que prend une OPEL en machine.

# Protocole

Pour évaluer l'efficacité en terme de nombre d'opérations :

- On se met d'accord sur ce qu'on considère comme *opération élémentaire* (OPEL).
- On estime approximativement le temps que prend une OPEL en machine.
- Seules les OPEL sont considérées comme coûteuses en temps et sont comptabilisées.

# Protocole

Pour évaluer l'efficacité en terme de nombre d'opérations :

- On se met d'accord sur ce qu'on considère comme *opération élémentaire* (OPEL).
- On estime approximativement le temps que prend une OPEL en machine.
- Seules les OPEL sont considérées comme coûteuses en temps et sont comptabilisées.
- Les autres opérations sont *négligées*.

# Complexité moyenne

On peut « imaginer » une fonction $c_M$ (au sens mathématique du terme) qui

# Complexité moyenne

On peut « imaginer » une fonction $c_M$ (au sens mathématique du terme) qui

- serait définie pour toute taille $n$ du problème ;

# Complexité moyenne

On peut « imaginer » une fonction $c_M$ (au sens mathématique du terme) qui

- serait définie pour toute taille $n$ du problème ;
- donnerait le nombre moyen d'OPEL nécessaires pour résoudre un problème de taille $n$.

# Complexité moyenne

On peut « imaginer » une fonction $c_M$ (au sens mathématique du terme) qui

- serait définie pour toute taille $n$ du problème ;
- donnerait le nombre moyen d'OPEL nécessaires pour résoudre un problème de taille $n$.

Cette *complexité moyenne* est très rarement calculable (calculs trop compliqués).

# Complexité dans le pire des cas

Plus simple mais tout aussi utile que $c_M$.

# Complexité dans le pire des cas

Plus simple mais tout aussi utile que $c_M$.

On cherche pour une taille $n$ donnée le nombre maximal $c(n)$ d'OPEL pour résoudre un problème de cette taille.

# Ordre de grandeur

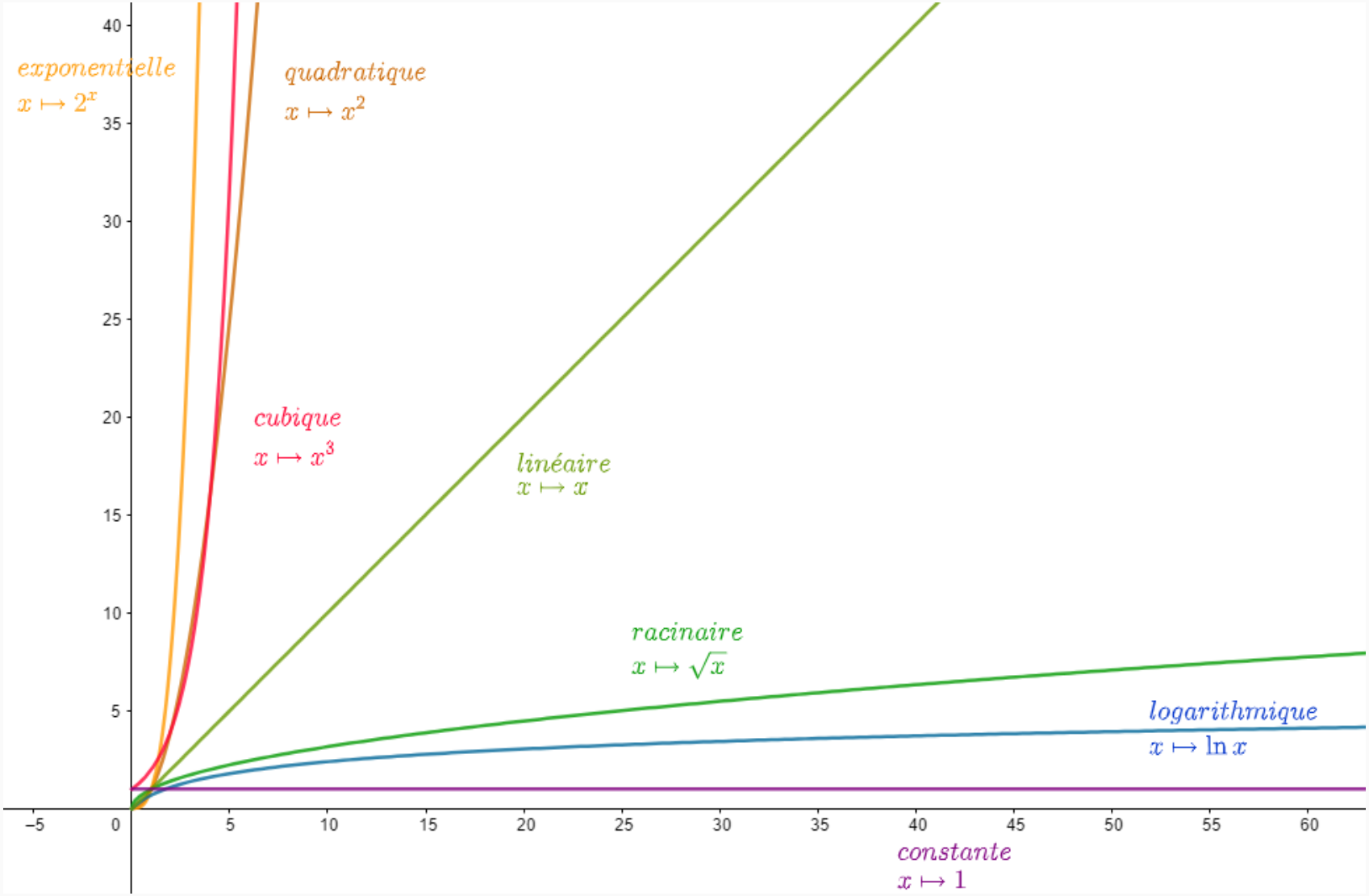
Quand $n$ augmente, en général, $c(n)$ augmente.

# Ordre de grandeur

Quand $n$ augmente, en général, $c(n)$ augmente.

Mais à quelle vitesse ?

# Vitesses de croissance : graphique



# Vitesses de croissance : tableau

Dans la première ligne on a indiqué différentes valeurs de $n$.

| complexité | 5 | 10 | 20 | 50 | 250 | 1 000 | 10 000 | 1 000 000 |
|---|---|---|---|---|---|---|---|---|
| constante | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns | 10 ns |
| logarithmique | 10 ns | 10 ns | 10 ns | 20 ns | 30 ns | 30 ns | 40 ns | 60 ns |
| racinaire | 22 ns | 32 ns | 45 ns | 71 ns | 158 ns | 316 ns | 1 µs | 10 µs |
| linéaire | 50 ns | 100 ns | 200 ns | 500 ns | 2,5 µs | 10 µs | 100 µs | 10 ms |
| quadratique | 250 ns | 1 µs | 4 µs | 25 µs | 625 µs | 10 ms | 1 s | 2,8 h |
| cubique | 1,25 µs | 10 µs | 80 µs | 1.25 ms | 156 ms | 10 s | 2,7 h | 316 ans |
| exponentielle | 320 ns | 10 µs | 10 ms | 130 jours | $10^{59}$ ans | ... | ... | ... |

# Bilan

La complexité, c'est important !

# Retour sur nos algorithmes

# Convention

On décide qu'une OPEL est un accès à un élément d'une liste, ou bien une multiplication.

# Algorithme 1

```
def tous_pairs1(lst: list) -> bool:
    resultat = True
    for x in lst:
        if x % 2 == 1:
            resultat = False
    return resultat
```

# Algorithme 1

```
def tous_pairs1(lst: list) -> bool:
    resultat = True
    for x in lst:
        if x % 2 == 1:
            resultat = False
    return resultat
```

Quand `lst` est de taille $n$, il faut $n$ OPEL.

# Algorithme 1

```
def tous_pairs1(lst: list) -> bool:
    resultat = True
    for x in lst:
        if x % 2 == 1:
            resultat = False
    return resultat
```

Quand `lst` est de taille $n$, il faut $n$ OPEL.
L'algorithme est de complexité linéaire.

# Algorithme 2

```
def tous_pairs2(lst: list) -> bool:
    for x in lst:
        if x % 2 == 1:
            return False
    return True
```

# Algorithme 2

```
def tous_pairs2(lst: list) -> bool:
    for x in lst:
        if x % 2 == 1:
            return False
    return True
```

Dans le meilleur des cas il faut 1 OPEL, $n$ dans le pire des cas.

# Algorithme 2

```
def tous_pairs2(lst: list) -> bool:
    for x in lst:
        if x % 2 == 1:
            return False
    return True
```

Dans le meilleur des cas il faut 1 OPEL, $n$ dans le pire des cas.
En moyenne il en faut environ 2.

# Algorithme 3

```
def tous_pairs3(lst: list) -> bool:
    produit = 1
    for x in lst:
        produit *= x
    if produit % 2 == 1:
        return False
    else:
        return True
```

# Algorithme 3

```
def tous_pairs3(lst: list) -> bool:
    produit = 1
    for x in lst:
        produit *= x
    if produit % 2 == 1:
        return False
    else:
        return True
```

Puisqu'on parcourt toute la liste, il faut $n$ OPEL...

# Algorithme 3

```
def tous_pairs3(lst: list) -> bool:
    produit = 1
    for x in lst:
        produit *= x
    if produit % 2 == 1:
        return False
    else:
        return True
```

Puisqu'on parcourt toute la liste, il faut $n$ OPEL...

On en rajoute $n$ pour les multiplications...

# Algorithme 3

```
def tous_pairs3(lst: list) -> bool:
    produit = 1
    for x in lst:
        produit *= x
    if produit % 2 == 1:
        return False
    else:
        return True
```

Puisqu'on parcourt toute la liste, il faut $n$ OPEL...
On en rajoute $n$ pour les multiplications...
C'est encore un peu « léger » : plus $n$ augmentent plus les multiplications prennent du temps...

# Algorithme 3

```
def tous_pairs3(lst: list) -> bool:
    produit = 1
    for x in lst:
        produit *= x
    if produit % 2 == 1:
        return False
    else:
        return True
```

Puisqu'on parcourt toute la liste, il faut $n$ OPEL...
On en rajoute $n$ pour les multiplications...
C'est encore un peu « léger » : plus $n$ augmentent plus les multiplications prennent du temps...et de la mémoire !

# Exemple : la fonction `maximum`

```
def maximum(lst: list) -> int:
    n = len(lst)
    result = lst[0]
    for i in range(n):
        if lst[i] > result:
            result = lst[i]
    return result
```

# OPEL

On convient qu'une OPEL est

- une affectation ;
- une comparaison.

# Comptabilisation des OPEL

```
def maximum(lst : list) -> int:
    n = len(lst) # 1 OPEL
    result = lst[0] # 1 OPEL
    for i in range(n): # 1 OPEL
        if lst[i] > result: # 1 OPEL
            result = lst[i] # 1 OPEL
    return result
```

# Comptabilisation des OPEL

```
def maximum(lst : list) -> int:
    n = len(lst) # 1 OPEL
    result = lst[0] # 1 OPEL
    for i in range(n): # 1 OPEL
        if lst[i] > result: # 1 OPEL
            result = lst[i] # 1 OPEL
    return result
```

Dans la boucle **for** , qui est itérée $n$ fois, il y a 3 OPEL.

# Comptabilisation des OPEL

```
def maximum(lst : list) -> int:
    n = len(lst) # 1 OPEL
    result = lst[0] # 1 OPEL
    for i in range(n): # 1 OPEL
        if lst[i] > result: # 1 OPEL
            result = lst[i] # 1 OPEL
    return result
```

Dans la boucle **for** , qui est itérée $n$ fois, il y a 3 OPEL.
Ainsi, la complexité temporelle de la fonction est $3n + 2$.

# Conclusion

Quand $n$ est grand, le « $+2$ » n'a guère d'importance et donc cette complexité est presque proportionnelle à $n$.

# Conclusion

Quand $n$ est grand, le « $+2$ » n'a guère d'importance et donc cette complexité est presque proportionnelle à $n$.

On dira donc qu'elle est *de l'ordre de n*, ou encore *linéaire*.