Algorithmes Classiques

1. Algorithmes sur les entiers et les puissances

1.1 Calcul de puissance

Algorithme naïf

In [1]:
```
def puissance(a, n):
    res = 1
    for k in range(n):
        res = res * a
    return res
```

In [2]:
```
puissance(2, 10)
```

Out[2]: 1024

In [3]:
```
puissance(1 + 1/1000, 1000)
```

Out[3]: 2.7169239322355985

Exponentiation rapide en récursif

In [4]:
```
def puissance_rr(a, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return puissance_rr(a * a, n // 2)
    else:
        return a * puissance_rr(a * a, n // 2)
```

In [5]:
```
puissance_rr(2, 10)
```

Out[5]: 1024

In [6]:
```
puissance_rr(1 + 1/1000, 1000)
```

Out[6]: 2.71692393223552

Exponentiation rapide en itératif

In [7]:
```
def puissance_ri(a, n):
    b = a
    res = 1
    while n != 0:
        if n % 2 == 1:
            res = res * b
        b = b * b
        n = n // 2
    return res
```

In [8]:
```
puissance_ri(2, 10)
```

Out[8]: 1024

In [9]:
```
puissance_ri(1 + 1/1000, 1000)
```

Out[9]: 2.7169239322355203

In [10]:
```
%timeit puissance(1 + 1/1000, 1000)
%timeit puissance_rr(1 + 1/1000, 1000)
%timeit puissance_ri(1 + 1/1000, 1000)
```

```
43.2 µs ± 545 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
2.07 µs ± 17.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
1.48 µs ± 44.2 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

1.2 Evaluation d'un polynôme

Version naïve

In [11]:
```
def eval_tres_naif(p, x):
    res = 0
    for k in range(len(p)):
        res = res + p[k] * puissance(x, k)
    return res
```

In [12]:
```
eval_tres_naif([1, 2], 3) == 1 + 2 * 3
```

Out[12]: True

In [13]:
```
eval_tres_naif([1, 2, 3], 4) == 1 + 2 * 4 + 3 * (4**2)
```

Out[13]: True

In [14]:
```
def eval_naif(p, x):
    res = 0
    value = 1
    for k in range(len(p)):
        res = res + p[k] * value
        value = value * x
    return res
```

In [15]:
```
eval_naif([1, 2], 3) == 1 + 2 * 3
```

Out[15]: True

In [16]:
```
eval_naif([1, 2, 3], 4) == 1 + 2 * 4 + 3 * (4**2)
```

Out[16]: True

Algorithme de Horner

In [17]:
```
def eval_horner(p, x):
    res = 0
    n = len(p) - 1
    for k in range(n + 1):
        res = res * x + p[n - k]
    return res
```

In [18]:
```
eval_horner([1, 2], 3) == 1 + 2 * 3
```

Out[18]: True

In [19]:
```
eval_horner([1, 2, 3], 4) == 1 + 2 * 4 + 3 * (4**2)
```

Out[19]: True

In [20]:
```
n = 1000
p = [1.0 for k in range(n + 1)]
x = 0.9
%timeit eval_tres_naif(p, x)
%timeit eval_naif(p, x)
%timeit eval_horner(p,x)
```

```
20.7 ms ± 212 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
98.4 µs ± 293 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
100 µs ± 455 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

1.3 Décomposition d'un entier en base b

In [21]:
```
def decomposition_base(b, n):
    res = []
    while n != 0:
        res.append(n % b)
        n = n // b
    return res
```

In [22]:
```
decomposition_base(10, 345)
```

Out[22]: [5, 4, 3]

In [23]:
```
decomposition_base(2, 10)
```

Out[23]: [0, 1, 0, 1]

1.4 Calcul du pgcd

Calcul du pgcd en récursif

In [24]:
```
def pgcd_r(a, b):
    if b == 0:
        return a
    else:
        return pgcd_r(b, a % b)
```

In [25]:
```
pgcd_r(3, 7)
```

Out[25]: 1

In [26]:
```
pgcd_r(15, 21)
```

Out[26]: 3

Calcul du pgcd en itératif

In [27]:
```
def pgcd_i(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

In [28]:
```
pgcd_i(3, 7)
```

Out[28]: 1

In [29]:
```
pgcd_i(15, 21)
```

Out[29]: 3

2. Algorithmes sur les listes

2.1 Recherche du maximum et du minimum

In [30]:
```
def maximum(t):
    res = t[0]
    for k in range(len(t)):
        if t[k] > res:
            res = t[k]
    return res
```

In [31]:
```
maximum([2, 6, -1])
```

Out[31]: 6

In [32]:
```
def minimum(t):
    res = t[0]
    for k in range(len(t)):
        if t[k] < res:
            res = t[k]
    return res
```

In [33]:
```
minimum([2, 6, -1])
```

Out[33]:
```
-1
```

2.2 Calcul de la moyenne

In [34]:
```
def moyenne(t):
    n = len(t)
    res = 0
    for k in range(n):
        res = res + t[k]
    res = res / n
    return res
```

In [35]:
```
moyenne([10, 12, 9])
```

Out[35]:
```
10.333333333333334
```

2.3 Recherche dans une liste

Version rapide, simple et efficace. What else?

In [36]:
```
def recherche(t, x):
    for k in range(len(t)):
        if t[k] == x:
            return True
    return False
```

In [37]:
```
recherche([2, 3, 7], 3)
```

Out[37]:
```
True
```

In [38]:
```
recherche([2, 3, 7], -1)
```

Out[38]:
```
False
```

Version avec un seul return, mais un peu plus lente

In [39]:
```
def recherche_v0(t, x):
    res = False
    for k in range(len(t)):
        if t[k] == x:
            res = True
    return res
```

In [40]:
```
recherche_v0([2, 3, 7], 3)
```

Out[40]:
```
True
```

In [41]:
```
recherche_v0([2, 3, 7], -1)
```

Out[41]:
```
False
```

Version avec un seul return, rapide, mais utilisant les subtilités du "and" paresseux

In [42]:
```
def recherche_v1(t, x):
    n = len(t)
    k = 0
    while k < n and t[k] != x:
        k = k + 1
    return k != n
```

In [43]:
```
recherche_v1([2, 3, 7], 3)
```

Out[43]: True

In [44]:
```
recherche_v1([2, 3, 7], -1)
```

Out[44]: False

2.4 Recherche dichotomique dans une liste triée dans l'ordre croissant

Une version avec plusieurs return. Les autres versions sont très dangereuses.

In [45]:
```
def recherche_dichotomique(t, x):
    g = 0
    d = len(t) - 1
    while g <= d:
        m = (g + d) // 2
        if x < t[m]:
            d = m - 1
        elif x > t[m]:
            g = m + 1
        else:
            return True
    return False
```

In [46]:
```
recherche_dichotomique([2, 3, 7], 2)
```

Out[46]: True

In [47]:
```
recherche_dichotomique([2, 3, 7], -1)
```

Out[47]: False