# Chapitre 2
# Algorithmes classiques

Simon Dauguet
*simon.dauguet@gmail.com*

10 octobre 2024

Nous allons ici voir certaines structures d'algorithmes classiques qui reviennent souvent et donc qu'il est bon d'avoir vu et pratiqués.

## Table des matières

# 1 Algorithmes de type seuil

## 1.1 Principe général

Dans un algorithme de type seuil, on calcule une valeur jusqu'à atteindre la négation d'une condition (on ne connaît *a priori* pas à quel moment elle est atteinte). Le cas typique est le calcul des valeurs d'une suite (qui est suffisamment "proche" de sa limite, ou d'un terme numéro de terme particulier).

Les algorithmes de type seuil sont généralement caractérisés par l'utilisatin de boucle `while` : on ne s'arrête de calculer que lorsque le seuil est atteint (ou dépasser).

**Exemple 1.1 :**
On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par $\forall n \in \mathbb{N}$, $u_n = 3 \times 2^n + 7$. Et on cherche le premier indice pour lequel le terme $u_n$ va dépasser un certain seuil `s`.

```
1  def seuil_u(s:int) -> int :
2      k=0
3      u=7
4      while u<s :
5          k = k+1
6          u = 3*2**k+7
7      return(k)
```

**Exercice 1 :**
On cherche à déterminer le plus petit entier $n$ tel que 2024 divise $n!$.

1. Proposer une solution mathématique, puis un code python qui réponde au problème.
2. Proposer une fonction `divFacto(s:int) -> int`, qui généralise la question et renvoie le plus petit entier $n$ tel que $s|n!$.

## 1.2 Algorithme de Héron d'Alexandrie - Approximation de racines par la méthode de la tangente de Newton

L'algorithme de Héron d'Alexandrie est un algorithme ancien d'extraction de racine très efficace. Il s'agit d'un cas particulier de la méthode de la tangente de Newton.

La méthode de la tangente de Newton consiste en la construction explicite d'une suite qui converge vers un $0$ d'une fonction. Dans le cas de la méthode de Newton, la fonction à laquelle on applique cette méthode doit vérifier certaines conditions mathématiques pour qu'il puisse y avoir convergence de la suite construite. Nous ne attarderons pas sur les détails de ces conditions car nous ne sommes pas dans un cours de maths.

Le cas de l'algorithme de Héron d'Alexandrie est l'application de cette méthode à des fonction particulières qui vérifient les hypothèses de convergence de la méthode de Newton, d'une part, et dont

le zéro est la racine carrée d'un réel, d'autre part. Donc l'algorithme de Héron d'Alexandrie exploite la méthode de Newton pour donner une approximation de la racine carrée d'un réel strictement positif.

**Proposition 1.1 (Méthode de Newton) :**
Soit $a < b$ et $f : [a, b] \to \mathbb{R}$ de classe $\mathcal{C}^2$ tel que $\exists r \in ]a, b[$ tel que $f(r) = 0$ et $\forall x \in [a, b]$, $f'(x) \neq 0$.

Alors il existe $\eta > 0$ tel que la suite $(x_n)_{n\in\mathbb{N}}$ définie par $x_0 \in [r - \eta, r + \eta]$ et

$$\forall n \in \mathbb{N},\ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

converge vers $r$.

*Démonstration :*
On pose la fonction $g : x \mapsto x - \frac{f(x)}{f'(x)}$. On montre alors $g(x) = x \iff f(x) = 0$. Donc les points fixes de $g$ sont les zéros de $f$.

Par ailleurs $f$ s'annule en $r$ par hypothèse. $f'$ est continue et ne s'annule pas, donc par TVI, elle est de signe stricte constant sur $[a, b]$. Donc $f$ est strictement monotone. Donc injective. Donc $r$ est l'unique zéro de $f$ sur $[a, b]$.

De plus, $g$ est de classe $\mathcal{C}^1$ sur $[a, b]$. En particulier, $g'$ est continue. Donc par continuité en $r$, $\forall \tau \in ]0, 1[$, $\exists \eta > 0$, $\forall x \in [r - \eta, r + \eta] \subset [a, b]$, $|g'(x)| = |g'(x) - g'(r)| \leq \tau$. Donc $g$ est $1/2$-contractante sur un voisinage de $r$.

De plus, par l'IAF, $\forall x \in [r - \eta, r]$, $\exists c \in ]x, r[$ tel que $|g(x) - g(r)| \leq |x - r|/2 < |x - r| \leq \eta$. De même à droite de $r$. Donc $\forall x \in [r - \eta, r + \eta]$, $|g(x) - r| \leq \eta$. Donc $g([r - \eta, r + \eta]) \subset [r - \eta, r + \eta]$. Donc $[r - \eta, r + \eta]$ est un intervalle stable sur lequel $g$ est contractante.

Donc la suite $(x_n)$ est bien définie, dans $[r - \eta, r + \eta]$ et $|x_{n+1} - r| = |g(x_n) - r| \leq |x_n - r|$. Donc par récurrence facile, $|x_n - r| \leq |x_0 - r|/2^n \leq \eta/2^n$. Donc la suite $(x_n)$ converge vers $r$. □

**Remarque :**
On notera que l'hypothèse $f'(x) \neq 0$ sur l'intervalle implique que $f$ s'annule et change de signe en $r$. En effet, $f'$ étant de signe constant, $f$ est donc monotone. Et donc, de fait, elle doit changer de signe lorsqu'elle s'annule.

On pourrait adapter un peu les hypothèses pour accepter le cas où $f$ s'annule sans changer de signe.

**Remarque :**
En pratique, il faudra donc se placer sur un petit intervalle autour du zéro cherché. Autrement dit, il faut déjà une bonne approximation du zéro de la fonction.

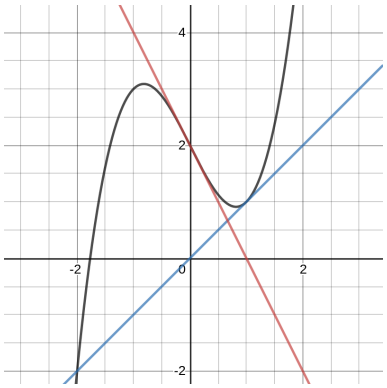
L'idée générale étant qu'en se plaçant suffisamment proche du zéro de la fonction, elle aura un bon comportement qui assurera la convergence de la suite $(x_n)$. Même dans le cas où $f$ ne change pas de signe, ou qu'elle n'est pas tout à fait $\mathcal{C}^2$ etc.

**Remarque :**
L'hypothèse du petit intervalle autour du zéro est fondamentale et est la source de beaucoup de problème. Tout va dépendre en fait du choix de l'intervalle autour de la racine. Et c'est ce qui est délicat encore une fois.

Il se pourrait par exemple, qu'en choisissant un valeur $x_0$ un peu trop loin, la suite $(x_n)$ oscille entre deux valeurs. On pourrait même avoir une valeur de la suite qui sorte de l'intervalle de définition de la fonction $f$.

Le problème vient essentiellement du fait que si l'intervalle est "trop gros", la fonction $f$ pourra avoir de grande variations sur cet intervalle. On pourrait tomber aussi sur un point où la dérivée s'annule.

**Contre-exemple :**
On considère la fonction $f : x \mapsto x^3 - 2x + 2$. Alors elle a un unique zéro qui est dans l'intervalle $[-2, 1]$. Mais si on se place sur $[-2, 2]$ par exemple, et qu'on démarre avec $x_0 = 0$ ou $x_0 = 1$, alors la suite $x_n$ oscillera une fois sur deux entre $0$ et $1$ et ne convergera donc pas.




Définition 1.1 (Algorithme d'Héron d'Alexandrie) :
Soit $a > 0$. On considère la fonction $f : x \mapsto x^2 - a$. Alors $f$ s'annule une unique fois sur $\mathbb{R}_+$ en $\sqrt{a}$.

En appliquant la méthode de Newton à cette fonction $f$, on obtient la suite

$$\forall n \in \mathbb{N},\ x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right).$$

Cette suite converge vers $\sqrt{a}$.

**Exemple 1.2 :**
Pour calculer une valeur approchée de $\sqrt{3}$, on pose alors $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = \frac{1}{2}(u_n + 3/u_n)$. Pour avoir une valeur approchée à $\varepsilon$ près de $\sqrt{3}$, on cherche le premier entier $n$ tel que $|u_{n+1} - u_n| < \varepsilon$.

```
def heron3(epsilon:float) -> float :
    u=1
    v=-epsilon
    while abs(u-v) >= epsilon :
        v = u
        u = (u+3/u)/2
    return(u)
```

**Exercice 2 :**
On souhaite avoir une valeur approchée de $\pi$.

1. Écrire une fonction `Pi(epsilon:float) -> float` qui prend en argument un flottant `epsilon` et renvoie une valeur approchée de $\pi$ à `epsilon` près.
2. Modifier cette fonction pour renvoyer le nombre d'itérations faites pour avoir cette valeur approchée.

# 2 Algorithmes de type somme

Le principe général des algorithme de type consiste en le parcours (entier ou partiel) d'une liste, explicitement ou implicitement. Comme par exemple pour faire la sommes des termes d'une liste.

## 2.1 Somme des termes d'une suite

Clairement, pour sommer tous les termes d'une liste, il faut parcourir la liste en entier.

```
def somme(tab: list) -> int:
    som = 0
    for i in range( len(tab) ):
        som = som + tab[i]
    return(som)
```

**Exercice 3 :**
Proposer une fonction `parfait(n:int) -> bool` qui renvoie si `True` si `n` est un entier parfait ou non. Un entier parfait est un entier qui est la somme de ses diviseurs strictes.

### 2.2 Maximum d'une liste de termes

La recherche du maximum d'une liste nécessite de parcourir également la liste en entier. Et donc est un algorithme de type somme. Il consiste en la sauvegarde d'un terme et la comparaison successive avec tous les termes de la liste.

Pour initialiser le processus de comparaison, on peut commencer par conserver le premier terme de la liste, si celle-ci n'est pas vide, ou bien prendre $-\infty$ (avec la syntaxe `-float("inf")`).

**Exemple 2.1 :**

```
1 def maximum(tab: list) -> int:
2     vmax = -float('inf')          # ou tab[0]
3     for i in range( len(tab) ): # parcours de la liste en entier
4         if vmax < tab[i]:        # comparaison du terme maximal actuel avec le nouveau terme
  de la liste
5             vmax = tab[i]        # sélection de ce nouveau terme s'il est plus grand que
  celui conservé jusque là
6     return vmax
```

**Exercice 4 :**
Proposer une fonction `nbMax(tab: list) -> int`, qui renvoie le nombre de fois qu'apparaît le maximum dans `tab`.

## 3 Algorithmes de type recherche

### 3.1 Cas général

Un algorithme de recherche d'une valeur dans une liste est relativement similaire aux algorithmes de type somme, à la différence près qu'on ne parcours pas nécessairement une portion de la liste en entier. Il y a un parcours de liste mais qui s'arrête dès qu'une condition est atteinte (et donc qui peut s'arrêter avant la fin de la liste).

Lors du parcours du tableau, si la valeur du tableau vaut celle recherchée :

- on renvoit `True` et on s'arrête.
- Si non, on continue.
- Enfin, si tout le tableau a été vérifié et que la valeur recherchée n'a pas été trouvée, on renvoie `False`.

Ainsi le tableau sera parcouru entièrement lorsque la valeur cherchée est en dernière position ou si elle n'existe pas.

**Exemple 3.1 :**

```python
def recherche(val:object, tab:list)->bool:
    for i in range( len(tab) ):
        if val == tab[i]:
            return True
    return False
```

**Exercice 5 :**
Proposer une fonction `tabCroissant(tab: list) -> bool`, qui précise si les valeurs de `tab` sont trié dans l'ordre croissant.

## 3.2 Recherche dichotomique

Le principe de la recherche dichotomique (de *díka* : en deux, et *tomós* : section, coupure) consiste à faire une recherche dans une plage de valeur en coupant en deux l'intervalle sur lequel on travail à chaque étape.

### 3.2.1 Recherche d'un terme dans un tableau trié

Considérons le tableau trié croissant `tab = [1, 1, 4, 5, 5, 6, 7]` et choisissons pour valeur pivot la valeur milieu du tableau.

a) On cherche la valeur $7$ dans le tableau.

- On considère tout le tableau et on compare la valeur cherchée $7$ avec la valeur milieu $5$. Puisque $5 < 7$ la valeur cherchée ne peut être qu'à droite.
- On considère donc le sous-tableau `[5, 6,7]` et on compare la valeur cherchée $7$ avec la valeur milieu $6$. Puisque $6 < 7$ la valeur cherchée ne peut être qu'à droite.
- On considère donc le sous-tableau `[7]` dont la valeur milieu est justement $7$ : la valeur $7$ a été trouvée.

b) On cherche la valeur $2$ dans le tableau.

- On considère le tableau initial et on compare la valeur cherchée $2$ avec la valeur milieu $5$. Puisque $5 > 2$ la valeur cherchée ne peut être qu'à gauche.
- On considère donc le sous-tableau `[1, 1,4]` et on compare la valeur cherchée $2$ avec la valeur milieu $1$. Puisque $1 < 2$ la valeur cherchée ne peut être qu'à droite.
- On considère donc le sous-tableau `[4]` et on compare la valeur cherchée $2$ avec la valeur milieu $4$. Puisque $4 < 2$ la valeur cherchée ne peut être qu'à gauche, mais puisqu'il n'existe plus de valeur, la valeur $2$ n'a pas été trouvée.

On peut donc écrire l'algorithme en pseudo-code :

| Tant que le tableau sélectionné n'est pas vide, |
|---|
| Comparer la valeur cherchée à celle du milieu, |
| Si la valeur cherchée est plus petite, sélectionner le sous-tableau de gauche |
| Si la valeur cherchée est plus grande, sélectionner le sous-tableau de droite |
| Si la valeur cherchée est égale, on a trouvé : arrêt |

et donc :

```
1  def rechercheDicho(tab:list) -> bool :
2      btrouve = False
3      indg = 0
4      indd = len(tab)-1
5      while (not btrouve) and (indd-indg >= 0):
6          indm = (indg + indd) // 2
7          if val < tab[indm]:
8              indd = indm-1
9          elif val > tab[indm]:
10             indg = indm+1
11         else :
12             btrouve = True
13     return(btrouve)
```

**Exercice 6 :**
Proposer une fonction `rechercheDicho(tab: list, val: int) -> tuple`, qui à partir d'un tableau trié croissant renvoie un couple (booléen, entier) qui précise si la valeur est dans le tableau et auquel cas le premier indice où l'on peut trouver une occurrence de `val` dans `tab`.

### 3.2.2 Recherche du zéro d'une fonction par le TVI

**Proposition 3.1 (Principe de dichotomie) :**
Soit $a < b$ et $f : [a, b] \to \mathbb{R}$ continue telle que $f(a)f(b) \leq 0$.
On définit les deux suites $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ par récurrence de la manière suivante :

- $a_0 = a$, $b_0 = b$
- Pour tout $n \in \mathbb{N}$, on définit $a_{n+1}$ et $b_{n+1}$ de al manière suivante :
  - Si $f(a_n)f(\frac{a_n+b_n}{2}) \leq 0$, alors $a_{n+1} = a_n$ et $b_{n+1} = \frac{a_n+b_n}{2}$
  - Si $f(\frac{a_n+b_n}{2})f(b_n) \leq 0$, alors $a_{n+1} = \frac{a_n+b_n}{2}$ et $b_{n+1} = b_n$.

Alors $(a_n)_{n\in\mathbb{N}}$ est croissante, $(b_n)_{n\in\mathbb{N}}$ est décroissante et elles sont adjacentes. En particulier, elles convergent et ont la même limite $\ell$ vérifiant $f(\ell) = 0$.

**Remarque :**
On notera qu'il n'y a pas de problème de définition dans les deux suites $(a_n)$ et $(b_n)$. Il faut montrer par récurrence que $\forall n \in \mathbb{N}$, $f(a_n)f(b_n) \leq 0$. Et dans ce cas là, on a $f(a_n)f(\frac{a_n+b_n}{2}) \leq 0$ ou $f(\frac{a_n+b_n}{2})f(b_n) \leq 0$ puisque le changement de signe doit automatiquement se faire quelque part,

donc soit sur la moitié de gauche, soit de droite de l'intervalle. Éventuellement les deux en même temps si la fonction s'annule plusieurs fois en changeant de signe.

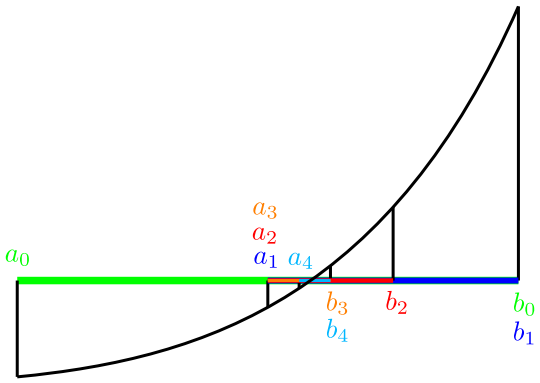
Le théorème fondamental du principe de dichotomie est le théorème des valeurs intermédiaires. Ce qui nécessite d'avoir une fonction continue et qui s'annule en changeant de signe.

**Remarque :**
L'algorithme de dichotomie permet de trouver UN zéros de la fonction.

**Remarque :**
L'inconvénient de la méthode de dichotomie est d'avoir déjà une idée, même approximative de l'endroit où se trouve le zéros cherché. Il faut un intervalle où se trouve le zéro, ce qui n'est pas forcément évident a priori.

Autrement dit, il faut déjà avoir une idée de la valeur du zéro pour chercher le zéro. Il existe des problèmes pour lesquels on peut montrer qu'il y a des solutions mais sans pouvoir avoir de précision sur la localisation (à un coût raisonnable en terme d'effort) sur ces solutions.

Il faut donc déjà, d'une certaine manière, connaître la solution pour la trouver.




```
1  def dicho(f:"function", a:float, b:float, epsilon:float) -> float:
2      """Algorithme de dichotomie.
3      f -- fonction continue sur [a,b]
4      a et b -- réels
5      epsilon -- réel>0, critère d'approximation. """
6      if f(a)*f(b) > 0 or epsilon <= 0 :
7          print("Erreur sur les hypothèse du principe de dichotomie.")
8      c,d = a,b
9      while (d-c) > 2*epsilon :
10         m = (c+d)/2
11         if f(c)*f(m) <= 0 :
12             d = m
13         else :
14             c = m
15     return((c+d)/2)
```

**Remarque :**
On préférera renvoyer $(c+d)/2$ plutôt que $m$ pour être sur d'avoir une approximation à $\varepsilon$ près du

zéro de $f$. En effet, entre $c$ et $d$, a priori, l'un des deux est une meilleure approximation que l'autre. L'un est à distance $< \varepsilon$ du zéro et l'autre est à distance $> \varepsilon$ (mais $< 2\varepsilon$) du zéro. Mais on ne sait pas lequel. En revanche, on est sur que $m$ est à distance $< \varepsilon$ du zéro.

**Exemple 3.2 :**
On peut tester l'algorithme pour trouver une valeur approchée de $\pi$ :

```
>>> zeroDicho(math.sin,1,4,0.0001)
3.141510009765625
>>> math.pi
3.141592653589793
```

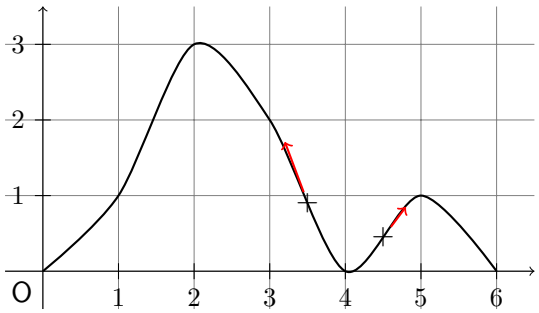
# 4 Algorithmes gloutons

Définition 4.1 :
Un algorithme glouton (*greedy algorithm*) est un algorithme qui, à chaque itération, fait un choix local optimal pour résoudre le problème traité.

**Proposition 4.1 :**
L'algorithme glouton ne renvoie pas toujours la meilleure solution (l'optimal global).

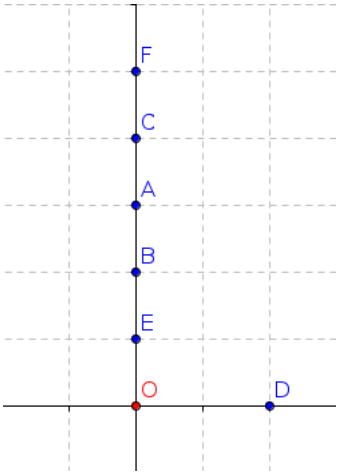
**Contre-exemple :**

On suppose que l'on cherche le maximum d'une courbe et qu'à chaque itération notre critère soit de choisir d'aller vers la plus grande pente. Si on part de l'abscisse $x = 3,5$, on trouvera pour maximum 3, qui est un maximum **global**. Mais si on part de l'abscisse $x = 4,5$, on trouvera pour maximum 1, qui est un maximum **local**.



**Exemple 4.1 :**

On souhaite parcourir les points de la liste ci-contre, en partant du point $O$ et ne passant qu'une seule fois par chaque point. L'ordre de parcours n'est pas important.
En utilisant un algorithme glouton, on choisir, à chaque étape, le point le plus proche qui n'a pas encore été visité.
Clairement, ce chemin n'est pas optimal (la distance totale parcourue n'est pas la plus petite).




Les algorithmes gloutons fournissent des solutions à beaucoup de problèmes classiques (voir TD et TP) dont le problème du voyageur (plus court chemin), le rendu monnaie, la coloration d'un graphe etc.