TP (S1) 4 Algorithmes dichotomiques

1 Premiers pas
2 Recherche dans une liste triée
3 Étude du tri par insertion
4 Recherche de la tranche de somme maximale

Objectifs

- Mettre en place des méthodes dichotomiques, récursivement et impérativement.
- Mettre en évidence le gain en complexité d'un algorithme dichotomique comparé à un algorithme naïf.

Fichier externe ?

OUI `TP_Dicho.py` *(présent(s) dans le répertoire partagé de la classe)*

Commencez par récupérer et ouvrir le fichier `TP_Dicho.py` disponible dans le répertoire partagé de la classe, il contient le code des fonctions, généralement à compléter, qui seront étudiées dans les différents exercices de ce TP. Exécutez-en les lignes 3 à 6 (à faire à chaque ouverture de `Pyzo`) afin de disposer des bibliothèques nécessaires aux illustrations graphiques des différents résultats obtenus.

1. PREMIERS PAS

1.1. Principes et premiers exemples

Pour traiter un problème de « taille n », on distingue deux approches :

1. l'approche la plus naturelle, dite « naïve », qui consiste généralement à se ramener à un problème de taille $n-1$,
2. l'approche dichotomique qui, quand elle est possible, consiste à se ramener à un ou plusieurs problèmes de taille $n/2$.

On illustre ceci sur trois exemples :

- **[Exemple A : un petit jeu]** Soit $n \in \mathbb{N}$, considérons le jeu classique suivant :
  1. Le joueur 1 choisit un entier p au hasard entre 0 et n,
  2. le joueur 2 tente de trouver p avec le moins d'essais possible : pour cela, à chaque essai il propose un entier t (tentative), et le joueur 1 doit lui dire si t est inférieur, supérieur ou égal à t. On se donne les deux stratégies suivantes pour le second joueur :
     - **2.1)** *La méthode naïve :* le joueur 2 commence avec une première tentative $t_1 = 0$, puis tente $t_2 = 1$ en cas d'échec *etc.* jusqu'à trouver p.
     - **2.2)** *La méthode dichotomique :* le joueur 2 commence avec une première tentative « au milieu entre 0 et n » : $t_1 = \left\lfloor \frac{0+n}{2} \right\rfloor$;
       - ◇ si $t_1 = p$, la recherche s'arrête,
       - ◇ si $t_1 < p$ sa tentative suivante t_2 sera « au milieu entre t_1+1 et n » : $t_2 = \left\lfloor \frac{t_1+1+n}{2} \right\rfloor$,
       - ◇ sinon sa tentative suivante t_2 sera « au milieu entre 0 et t_1-1 » : $t_2 = \left\lfloor \frac{0+t_1-1}{2} \right\rfloor$ *etc.* jusqu'à trouver p.

  On devine bien que la méthode dichotomique permet, sauf coup de chance, de trouver p beaucoup plus rapidement que la naïve. En effet, et dans le pire des cas, il faudra de l'ordre de n tentatives ($n+1$ pour être exact) pour trouver p avec la méthode naïve, alors qu'il n'en faut que de l'ordre de $\log_2(n)$ avec la méthode dichotomique, ce résultat sera démontré formellement au deuxième semestre. La différence est énorme ! Dans le cas où $n = 2^{270} = (2^{10})^{27} \approx (10^3)^{27} = 10^{81}$, c'est-à-dire l'ordre de grandeur du nombre d'atomes de l'Univers, il faudra dans le pire des cas un nombre de tentatives équivalent au nombre d'atomes de l'Univers pour trouver p avec la méthode naïve, tandis qu'il n'en faudra que environ 270 pour le trouver avec la méthode dichotomique.
- **[Exemple B : calcul de x^n]** Soient $x \in \mathbb{R}$ et $n \in \mathbb{N}$. Pour calculer x^n il faut :
  - ◇ par l'approche naïve initialiser une variable à 1 (pour traiter le cas de x^0) puis la multiplier n fois par x,
  - ◇ par l'approche dichotomique, appelée ici l'exponentiation rapide (voir TP 3)

utiliser le fait que $x^0 = 1$ et, pour $n > 0$, la relation de récurrence :

$$x^n = \begin{cases} (x*x)^{n//2} & \text{si } n \text{ est pair} \\ (x*x)^{n//2} * x & \text{si } n \text{ est impair} \end{cases}$$

Là encore, la méthode naïve implique de faire de l'ordre de n multiplications pour obtenir la valeur x^n, tandis qu'il n'en faut que de l'ordre de $\log_2(n)$ avec l'exponentiation rapide. Là encore, ce résultat sera démontré formellement au deuxième semestre.

- **[Exemple C : résolution approchée de l'équation $f(x) = 0$]** Considérons une fonction f continue et ne s'annulant qu'une seule fois sur le segment S $= [a, b]$, notons alors x_0 l'unique solution de l'équation $f(x) = 0$ sur S. Pour obtenir un encadrement à $\varepsilon > 0$ près de x_0 on procède par dichotomie :
  - ◇ Si $b - a \leq \varepsilon$ alors la liste `[a,b]` donne un encadrement à ε près de x_0.
  - ◇ Sinon on note m le milieu de S : $m = \dfrac{a+b}{2}$. Il y a alors deux cas de figure :
    1. si f change de signe entre a et m alors $x_0 \in [a, m]$,
    2. sinon $x_0 \in [m, b]$.
  - ◇ On itère jusqu'à obtenir un encadrement ε près de x_0.

  La longueur du segment dans lequel on cherche x_0 est ainsi divisée par 2 à chaque étape. Le nombre d'étapes nécessaire pour avoir un encadrement à ε près de x_0 est donc le premier entier n pour lequel $\dfrac{b-a}{2^n} \leq \varepsilon$ soit $n \geq \log_2\left(\dfrac{b-a}{\varepsilon}\right)$.
  Notons qu'ici il n'y a pas d'approche naïve.

1.2. Implémentation récursive

La programmation récursive est particulièrement bien adaptée aux algorithmes dichotomiques comme l'illustre l'exercice ci-dessous.

1.2.1. Sans fonction auxiliaire

Exercice 1 **Implémentation récursive des exemples B et C** [Sol 1]

1. Complétez le code de la fonction `expo_rapide_rec` et testez cette fonction pour différentes valeurs de x et de n.
2. Complétez la fonction `approx_dicho(f,a,b,e)` qui, lorsque f est une fonction continue et s'annulant une unique fois sur le segment $[a, b]$, renvoie sous forme de liste un encadrement à `e` près de l'unique solution sur $[a, b]$ de l'équation $f(x) = 0$.

Donnez un encadrement à 10^{-4} près de l'unique solution sur $[0, 1]$ de l'équation $g(x) = 0$ où $g : x \mapsto x^5 + x - 1$.

1.2.2. Avec une fonction auxiliaire

Pour coder récursivement une méthode dichotomique, il est parfois nécessaire de passer par une fonction auxiliaire.

Prenons le jeu de l'exemple A où l'on souhaite calculer le nombre de tentatives pour trouver un certain entier p selon la méthode dichotomique : pour le coder récursivement, il faut

- passer en paramètre le nombre de tentatives déjà effectuées et renvoyer le nombre de tentatives lorsque l'on trouve le nombre,
- appeler la fonction récursivement en augmentant le nombre de tentatives de 1 si la tentative échoue, en initialisant le nombre de tentatives déjà effectuées à 0 lors du premier appel. Cependant, cette initialisation ne peut se faire dans le corps de la fonction récursive : si c'est le cas, ce nombre sera ré-initialisé à 0 à chaque appel ! Il faut donc coder une fonction dite « maitre » qui se chargera de cette initialisation.

Exercice 2 **Implémentation récursive de l'exemple A** [Sol 2] Complétez le code des fonctions `devine_dicho_aux` et `devine_dicho_rec` et testez cette dernière pour différentes valeurs de n et de p.

1.3. Implémentation impérative

Comme on l'a vu dans le TP (S1) 3, on est limité en programmation récursive par la taille de la pile d'appels. Il est donc préférable, lorsque c'est possible, de proposer une programmation impérative de nos algorithmes dichotomiques. La programmation impérative d'un algorithme dichotomique se fait généralement avec une boucle `while` : tant que l'on ne rencontre pas le ou les cas d'arrêt, on poursuit l'exploration.

Exercice 3 **Implémentation impérative des exemples B et C** [Sol 3]

1. Complétez le code de la fonction `approx`. Testez-la et comparez les résultats avec sa version récursive.

2. Proposer une version impérative de l'algorithme d'exponentiation rapide est plus difficile. L'idée est la suivante : un entier n peut s'écrire sous la forme $n = \sum_{i=0}^{k} b_i \cdot 2^i$, avec $b_i \in \{0,1\}$ pour tout $0 \leqslant i \leqslant k$ et k un entier. Alors :

$$x^n = x^{\sum_{i=0}^{k} b_i \cdot 2^i} = \prod_{i=0}^{k} \left(x^{2^i}\right)^{b_i}.$$

Il nous faut donc une variable qui va contenir les x^{2^i}, notée X_i dans la suite et élevée au carré à chaque étape, une autre qui va stocker le résultat (donc le produit), notée R_i dans la suite. Les b_i quant à eux se calculeront à l'aide des commandes de division euclidienne : dans la suite, ce seront successivement les valeurs de $N_i \% 2$.

Pour implémenter ce principe, on construit donc trois suites **finies** (N_i), (R_i) et (X_i) de la façon suivante :

- $N_0 = n$, $X_0 = x$ et $R_0 = 1$
- tant que $N_i > 0$, $N_{i+1} = N_i // 2$, $X_{i+1} = X_i * X_i$ et $R_{i+1} = R_i * X_i^{N_i \% 2}$.

On montre alors que :

1. l'algorithme se termine en nombre fini d'étapes. Effet la suite (N_i) est par construction une suite strictement décroissante d'entiers naturels, et une telle suite admet nécessairement un nombre fini de termes [1]. Il existe donc un entier naturel k tel que $N_k = 0$.
2. En sortie de boucle : $R_k = x^n$. On montre en effet facilement par récurrence que, pour tout $i \in [\![0, k]\!]$,
   $R_i * X_i^{N_i} = x^n$; on a bien alors $R_k = x^n$ puisque $N_k = 0$.

Ci-contre un exemple avec $n = 10$:

i	N_i	X_i	R_i
0	10	x	1
1	5	x^2	$R_0 * 1 = 1$
2	2	x^4	$R_1 * X_1 = x^2$
3	1	x^8	$R_2 * 1 = x^2$
4	0	x^{16}	$R_3 * X_3 = x^{10}$

Complétez le code de la fonction `expo_rapide`, non récursive, qui renvoie la valeur de x^n à l'aide de l'algorithme d'exponentiation rapide. Testez-la et comparez les résultats avec sa version récursive.

1. On dit que $\mathbb{N}$ muni de sa relation d'ordre usuelle est un ensemble bien fondé.

2. RECHERCHE DANS UNE LISTE TRIÉE

Notations :
Dans cette partie et dans les suivantes, on appellera *tranche* d'une liste (ou d'un tableau) `L` toute sous-liste de `L` constituée d'éléments consécutifs de `L`.

Une tranche sera caractérisée par le couple d'entiers (d, f) où d est son indice de début **inclus** et f son indice de fin **exclus**, c'est-à-dire `L[d:f]`. Lorsque $d \geq f$, la tranche est donc vide.

Par exemple si `L` est la liste `['a', 'b', 'c', 'd', 'e', 'f']` alors :

- la tranche $(1,4)$ de `L` est la liste `L[1:4] = ['b', 'c', 'd']`,
- la tranche $(0,3)$ de `L` est la liste `L[0:3] = L[:3] = ['a', 'b', 'c']`,
- la tranche $(3,6)$ de `L` est la liste `L[3:6] = L[3:] = ['d', 'e', 'f']`,
- la tranche $(3,4)$ de `L` est la liste `L[3:4] = ['d']`,
- La tranche $(3,3)$ de `L` est la liste `L[3:3] = []`.

On s'intéresse ici à la recherche d'un élément noté `e` dans un tableau ou une liste `L` de longueur n. La première idée, naïve, consiste à tester tous les éléments un par un jusqu'à trouver `e` ou jusqu'à épuiser la liste (voir exercice 3 du TP2); une telle approche demandera en moyenne de faire $n/2$ comparaisons avant de détecter ou non l'élément.

2.1. Version récursive

Dans le cas où la liste est triée, on peut appliquer une méthode de recherche dichotomique que l'on détaille ici dans le cas où la liste est triée par ordre croissant :

- **[Cas terminal]** si la liste est vide. Dans ce cas `e` n'appartient pas à la liste et on renvoie `False`.
- **[Cas non terminal]** si la liste n'est pas vide.
  Notons alors m l'indice de l'élément central de `L` : `m = n//2`, trois cas sont alors possibles :
  1. si `L[m] == e`, alors `e` appartient à la liste et on renvoie `True`,
  2. si `L[m] > e`, alors `e` est à chercher dans la tranche $(0, m)$ de `L`,
  3. si `L[m] < e`, alors `e` est à chercher dans la tranche $(m+1, n)$ de `L`.

Là encore, cette méthode est bien plus efficace que la naïve puisqu'elle ne demandera en moyenne que $\log_2(n)$ comparaisons, sa limitation étant de ne s'appliquer qu'à des listes triées.

Exercice 4 **Implémentation récursive** [Sol 4] Complétez le code de la fonction récursive `present_dicho_rec` et testez-la.

2.2. Version impérative

Pour implémenter impérativement cet algorithme, on procède ainsi :

- On initialise deux variables d et f qui représentent la tranche de `L` dans laquelle chercher `e`.
- Tant que la tranche n'est pas vide, on note m l'indice de l'élément central de la tranche : `m = (d+f)//2`, trois cas sont alors possibles :
  1. si `L[m] == e`, alors `e` appartient à la liste et on interrompt la boucle en renvoyant `True`,
  2. si `L[m] > e`, alors on met à jour d et f pour chercher `e` dans la tranche (d, m) de `L`,
  3. si `L[m] < e`, alors on met à jour d et f pour chercher `e` dans la tranche $(m+1, f)$ de `L`.
- À l'issue de la boucle, on peut renvoyer `False` puisque celle-ci ne va à son terme que si `e` n'est pas présent dans `L`.

Exercice 5 **Implémentation impérative** [Sol 5] Complétez le code de la fonction `present_dicho`. Testez-la et comparez-la à sa version récursive.

3. ÉTUDE DU TRI PAR INSERTION

On appelle *tri* tout algorithme permettant de trier par ordre croissant ou décroissant une collection d'objets, ils seront étudiés plus en détails lors du prochain TP. Nous allons étudier ici un tri classique, bien connu des joueurs de cartes, permettant de trier une liste (ou un tableau) `L` : le tri par insertion.

Nous allons générer une nouvelle liste triée `Lt` contenant les mêmes éléments que `L` sans modifier la liste initiale[2] de la façon suivante :

1. On initialise `Lt` avec la liste vide.
2. On parcourt les éléments de `L` un par un et on les insère correctement dans `Lt` de la façon suivante :
   2.1) on détermine l'indice `p` auquel il faut insérer l'élément dans `Lt`,
   2.2) on « coupe » `Lt` à l'indice `p` et on insère l'élément entre les deux tranches.
3. On renvoie `Lt`.

2. On parle de tri non en place ou encore d'absence d'effet de bord

L'algorithme est simple, à condition de savoir déterminer où insérer l'élément. C'est l'objet des deux prochains exercices, le dernier étant dédié au codage du tri lui-même.

3.1. Recherche de l'indice d'insertion

On donne dans le fichier `TP_Dicho.py` (exercice 6) le code de la fonction `positionN` qui renvoie la position d'insertion par une méthode naïve ; comme d'habitude, cette fonction réalise en moyenne de l'ordre de n comparaisons.
On peut améliorer cela via un algorithme dichotomique codée impérativement et assez proche de ce que l'on a fait dans l'exercice précédent :

- On initialise deux variables d et f qui représentent la tranche de `L` dans laquelle chercher la position d'insertion de `e`.
- Tant que la tranche n'est pas vide, on note m l'indice de l'élément central de la tranche : `m = (d+f)//2`, deux cas sont alors possibles :
  1. si `L[m] >= e`, alors on met à jour d et f pour chercher la position d'insertion de `e` dans la tranche (d, m) de `L`,
  2. si `L[m] < e`, alors on met à jour d et f pour chercher la position d'insertion de `e` dans la tranche $(m+1, f)$ de `L`.
- À l'issue de la boucle, d et f ont la même valeur. Cette valeur commune étant l'indice auquel insérer l'élément.

Encore une fois, cette version dichotomique ne réalisera en moyenne que de l'ordre de $\log_2(n)$ comparaisons. On trouvera ci-dessous quelques exemples de recherche d'un élément `e` dans la liste `L = [1, 2, 2, 2, 3, 4, 5]`, i désignant le nombre de passages dans la boucle `while` :

	$e=0$			$e=2$			$e=3$			$e=6$		
i	d	m	f	d	m	f	d	m	f	d	m	f
0	0	3	7	0	3	7	0	3	7	0	3	7
1	0	1	3	0	1	3	4	5	7	4	5	7
2	0	0	1	0	0	1	4	4	5	6	6	7
3	0	0	0	1	1	1	4	4	4	7	7	7

Exercice 6 **Recherche d'indice d'insertion dichotomique** [Sol 6] Complétez le code de la fonction `positionD` qui réalise l'algorithme précédent et testez-la.

3.2. Le tri par insertion

Exercice 7 **Tri par insertion** [Sol 7]

1. Écrire la fonction `triInsN(L:list)->list` qui réalise le tri par insertion de la liste `L` avec recherche de position d'insertion naïve ainsi que la fonction `triInsD(L:list)->list` qui réalise le tri par insertion de la liste `L` avec recherche de position d'insertion dichotomique.
2. Justifier brièvement que la première fonction fait en moyenne de l'ordre de n^2 comparaisons tandis que la seconde n'en fait que de l'ordre de $n\log_2(n)$.
   Exécutez le code, complet, des fonctions `genlist` et `illustre` et observez ce que renvoie `illustre(5000)`. **Attention :** la fonction `illustre` ne fonctionne que si les fonctions de la question précédente ont été correctement codées.

4. RECHERCHE DE LA TRANCHE DE SOMME MAXIMALE

Attaquons-nous à un problème plus difficile : soit `L` une liste non vide de réels ou d'entiers relatifs de longueur n. On appelle « meilleure » tranche de `L` la tranche (d,f) de `L` de somme maximale. Par exemple, la meilleure tranche de la liste `Test = [-2,1,-2,3,1,-3,2,-1,4,-1]` est la tranche $(3,9)$: `L[3:9] = [3,1,-3,2,-1,4]` et sa somme, *i.e.* la somme de ses éléments, vaut 6.

On cherche à coder une fonction `m_tranche(L:list)` qui renvoie le triplet (s,d,f) où (d,f) caractérise la meilleure tranche de `L` et où s est sa somme.

La première idée, naïve, serait de parcourir à l'aide de deux boucles `for` imbriquées toutes les valeurs possibles de d et de f pour sélectionner la meilleure tranche. Cela conduirait à un nombre de calculs de l'ordre de n^2 (voir de n^3 si on s'y prend mal).

Une approche dichotomique est possible, elle est sans surprise bien plus efficace[3] car elle conduit à un nombre de calculs de l'ordre de $n\log_2(n)$: notons `m = n//2` l'indice de l'élément central de `L`, la meilleure tranche de `L` se situe :

1. soit dans la moitié gauche de `L`, c'est-à-dire `L[:m]`,
2. soit dans la moitié droite de `L`, c'est-à-dire `L[m:]`,
3. soit est à cheval sur m, c'est-à-dire qu'elle contient les éléments d'indices $m-1$ et m de `L`.

Pour coder la fonction `m_tranche` par cette approche on va :

3. On peut toutefois faire encore mieux : il est possible de trouver la meilleure tranche avec un nombre de calculs de l'ordre n avec des méthodes de programmation dynamique.

1. Coder une fonction non récursive `m_tranche_mil(L,d,m,f)` qui renvoie le triplet `(s,deb,fin)` où `(deb,fin)` caractérise la meilleure tranche de `L` contenue dans `L[d:f]` et à cheval sur m, de somme s.
   Pour ce, on pourra remarquer qu'il suffit de concaténer la meilleure tranche finissant à $m-1$ avec la meilleure commençant à m.
2. Coder une fonction récursive auxiliaire `m_tranche_aux(L, d, f)` qui renvoie le triplet `(s,deb,fin)` où `(deb,fin)` caractérise la meilleure tranche de `L` contenue dans `L[d:f]`, de somme s. Pour ce, on pourra déterminer :
   2.1) la meilleure tranche « centrale » grâce à la fonction précédente,
   2.2) la meilleure tranche « de la moitié gauche » grâce à un appel récursif,
   2.3) la meilleure tranche « de la moitié droite » grâce à un appel récursif.
   Puis sélectionner la meilleure des trois.
   Notons que le cas terminal est celui d'une liste à un unique élément car les moitiés gauche et droite d'une liste à deux éléments ou plus contiennent au moins un élément.
3. Enfin coder la fonction `m_tranche(L:list)` qui consiste uniquement à un appel à la fonction précédente correctement initialisée.

Exercice 8 **Recherche de la meilleure tranche** [Sol 8] Complétez les codes des trois fonctions détaillées ci-dessus et testez votre fonction `m_tranche` sur la liste `Test`.

Exercice 9 **Une application** [Sol 9] Soit une liste `L` des valeurs boursières journalières d'une action, sa valeur au jour 0 étant sa valeur d'introduction sur le marché. Par exemple on donne la liste :

```
Cours = [100,113,110,85,105,102,86,63,81,101,94,106,101,79,94,90,97],
```

l'action est introduite sur le marché au jour 0 à une valeur de 100 euros, elle en vaut 113 à l'issue du jour 1, puis 110 à l'issue du jour 2 *etc.*

Les questions posées sont : à l'issue de quels jours faut-il acheter puis vendre l'action pour réaliser le profit maximal ? Et que vaut ce profit ? Par exemple dans le cas de la liste `Cours`, il faut acheter l'action à l'issue du jour 7 (63 euros) puis la revendre à l'issue du jour 11 (106 euros) pour réaliser le profit maximal (43 euros).

Codez une fonction permettant de répondre à ces questions.

À retenir

1. Algorithmes dichotomiques classiques : savoir coder l'exponentiation rapide et la recherche dichotomique d'un élément dans une liste triée, récursivement et impérativement.

2. Complexité : travailler par dichotomie permet de passer d'une complexité linéaire à une complexité logarithmique. C'est un gain très notable en temps d'exécution en consommation d'énergie.

SOLUTIONS DES EXERCICES

Solution 1

1.
```python
def expo_rapide_rec(x:float,n:int)->float:
    '''Renvoie la valeur de x^n (récursivement, méthode dichotomique)'''
    if n == 0:
        return 1
    else:
        if n%2 == 0:
            return expo_rapide_rec(x*x,n//2)
        else:
            return x*expo_rapide_rec(x*x,n//2)
```

2.
```python
def approx_dicho(f,a:float,b:float,e:float)->list:
    '''Renvoie un encadrement à e près de l'unique sol. de f(x)=0 sur [a;b]'''
    if b-a <= e:
        return [a,b]
    else:
        m = (a+b)/2
        if f(a)*f(m) < 0:
            return approx_dicho(f,a,m,e)
        else:
            return approx_dicho(f,m,b,e)
def g(x):
    return x**5+x-1
```

```
>>> approx_dicho(g,0,1,0.0001)
[0.75482177734375, 0.7548828125]
```

Solution 2

```python
def devine_dicho_aux(p:int, a:int, b:int, t:int)->int:
    '''Renvoie le nb de tentatives nécessaires pour trouver p par dicho. entre a et b, t étant le nombre de tentatives déjà effectuées'''
    m = (a+b)//2
    if m == p:
        return t+1
    else:
        if p < m:
            return devine_dicho_aux(p,a,m-1,t+1)
        else:
            return devine_dicho_aux(p,m+1,b,t+1)

def devine_dicho_rec(p:int, n:int)->int:
    '''Renvoie le nb de tentatives nécessaires pour trouver p par dicho entre 0 et n'''
    return devine_dicho_aux(p, 0, n, 0)
```

Solution 3

1.
```python
def approx(f,a:float,b:float,e:float)->list:
    '''Renvoie un encadrement à e près de l'unique sol. de f(x)=0 sur [a;b]'''
    i,s = a,b
    while s - i > e:
        m = (i+s)/2
        if f(i)*f(m) < 0:
            i, s = i, m
        else:
            i,s = m,s
    return [i,s]
```

Si on teste à présent avec la fonction g définie dans un précédent exercice :

```
>>> approx(g, 0, 1, 0.0001)
[0.75482177734375, 0.7548828125]
>>> approx_dicho(g, 0, 1, 0.0001)
[0.75482177734375, 0.7548828125]
```

2.
```python
def expo_rapide(x:float,n:int)->float:
    '''Renvoie la valeur de x^n'''
    N, X, R = n, x, 1
    while N > 0:
        if N%2 == 1:
            R = R*X
        N, X = N//2, X*X
```

```
    return R
```

Solution 4

```
def present_dicho_rec(e,L:list)->bool:
    '''Renvoie True si e appartient à la liste L supposée \
    ↪ triée par ordre
    croissant, False sinon'''
    n = len(L)
    if n == 0:
        return False
    else:
        m = n//2
        if L[m] == e:
            return True
        elif L[m] > e:
            return present_dicho_rec(e, L[:m])
        else:
            return present_dicho_rec(e, L[m+1:])
```

```
>>> L = [1, 2, 3, 4, 2, 1]
>>> present_dicho_rec(0, L)
False
>>> present_dicho_rec(2, L)
True
```

Solution 5

```
def present_dicho(e,L:list)->bool:
    '''Renvoie True si e appartient à la liste L supposée \
    ↪ triée par ordre
    croissant, False sinon. Codée impérativement'''
    n = len(L)
    d,f = 0,n-1
    while d <= f:
        m = (d+f)//2
        if L[m] == e:
            return True
        elif L[m] > e:
            d,f = d,m-1
        else:
            d,f = m+1,f
    return False
```

```
>>> L = [1, 2, 3, 4, 2, 1]
>>> present_dicho(0, L)
False
>>> present_dicho(2, L)
True
```

Solution 6

```
def positionD(e,L:list)->int:
    '''Renvoie la position à laquelle il faut insérer e dans \
    ↪ la liste L'''
    n = len(L)
    d, f = 0, n
    while d < f:
        m = (d+f)//2
        if e <= L[m]:
            d, f = d, m
        else:
            d, f = m+1, f
    return d
```

```
>>> L = [1, 2, 2, 2, 3, 4, 5]
>>> positionD(3, L)
4
>>> positionN(3, L)
4
>>> positionD(7, L)
7
>>> positionN(7, L)
7
```

Pour observer et comprendre l'évolution des variables `d, f`, on peut utiliser ce programme.

```python
def positionD(e,L:list)->int:
    '''Renvoie la position à laquelle il faut insérer e dans \
    ↪ la liste L'''
    n = len(L)
    d, f = 0, n
    print("d=", d, "f=", f)
    while d < f:
        m = (d+f)//2
        if e <= L[m]:
            d, f = d, m
        else:
            d, f = m+1, f
        print("d=", d, "f=", f)
    return d
```

```
>>> L = [1, 2, 2, 2, 3, 4, 5]
>>> positionD(0, L)
d= 0 f= 7
d= 0 f= 3
d= 0 f= 1
d= 0 f= 0
0
>>> positionD(2, L)
d= 0 f= 7
d= 0 f= 3
d= 0 f= 1
d= 1 f= 1
1
>>> positionD(3, L)
d= 0 f= 7
d= 4 f= 7
d= 4 f= 5
d= 4 f= 4
4
>>> positionD(6, L)
d= 0 f= 7
d= 4 f= 7
d= 6 f= 7
d= 7 f= 7
7
```

Solution 7

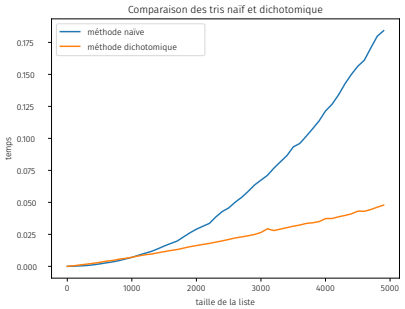
1.

```python
def triInsN(L:list)->list:
    '''Tri par insertion naïve de la liste L'''
    Lt,n = [], len(L)          # on initialise Lt avec la \
    ↪ liste vide
    for i in range(n):      # on parcourt les éléments de L
        e = L[i]
        p = positionN(e,Lt)   # on repère où les insérer \
        ↪ dans Lt
        Lt = Lt[:p]+[e]+Lt[p:]# on l'insère dans Lt
    return Lt

def triInsD(L:list)->list:
    '''Tri par insertion dichotomique de la liste L'''
    Lt,n = [],len(L)          # on initialise Lt avec la \
    ↪ liste vide
    for i in range(n):      # on parcourt les éléments de L
        e = L[i]
        p = positionD(e,Lt)   # on repère où les insérer \
        ↪ dans Lt
        Lt.insert(p, e) # on l'insère dans Lt avec méthode \
        ↪ insert cette fois (pour varier un peu)
    return Lt
```

2. La première fonction fait n recherches naïves dans la liste `L_t` qui a une taille au plus n. Ces n recherches faisant en moyenne de l'ordre de n comparaisons, le nombre total de comparaisons est bien de l'ordre de n^2.
La deuxième fonction fait n recherches dichotomiques dans la liste `L_t` qui a une taille au plus n. Ces n recherches faisant en moyenne de l'ordre de $\log_2(n)$ comparaisons, le nombre total de comparaisons est bien de l'ordre de $n\log_2(n)$.

```
illustre3(5000)
```



Solution 8

```python
def m_tranche_mil(L:list,d:int,m:int,f:int):
    '''Renvoie le triplet (s,deb,fin) où (deb,fin) caractérise la meilleure tranche de L contenue dans L[d:f] et à cheval sur m et où s est sa somme'''

    #Recherche de la meilleure tranche de L[d:f] débutant à l'indice m
    maxD, fin = L[m], m+1
    S = L[m]
    for i in range(m+1, f):
        S = S+L[i]
        if S > maxD:
            maxD = S
            fin = i+1

    #Recherche de la meilleure tranche de L[d:f] finissant à l'indice m-1
    maxG, deb = L[m-1],m-1
    S = L[m-1]
    for i in range(2,m-d+1):
        S = S+L[m-i]
        if S > maxG:
            maxG = S
            deb = m-i

    #Concaténation des deux tranches
    s = maxG+maxD
    return s,deb,fin

def m_tranche_aux(L:list,d:int,f:int):
    '''Renvoie le triplet (s,deb,fin) où (deb,fin) caractérise la meilleure tranche de L contenue dans L[d:f] et où s est sa somme'''
    if f-d == 1:
        return L[d], d, f
    else:
        m = (f+d)//2                #m est l'indice central de L[d:f]
        Tg = m_tranche_aux(L,d,m)   #meilleure tranche de L[d:f] à gauche de m(exclus)
        Tc = m_tranche_mil(L,d,m,f) #meilleure tranche de L[d:f] à cheval sur m
        Td = m_tranche_aux(L,m,f)  #meilleure tranche de L[d:f] à droite de m(inclus)
        Lt = [Tg, Tc, Td]    #liste de ces trois meilleures tranches
        # sélection de la meilleure des trois
        res = Lt[0]
        for i in range(1, 3):
            if Lt[i] > res:
                res = Lt[i]
        return res

def m_tranche(L:list):
    '''Renvoie le triplet (s,deb,fin) où (deb,fin) caractérise la meilleure tranche
    de L et où s est sa somme'''
    return m_tranche_aux(L, 0, len(L))
```

Solution 9 Pour répondre à la question, il faut chercher la meilleure tranche non pas de la liste `L` de départ, mais de celle `V` de ses variations journalières : `V[i] = L[i]-L[i-1]`, puis penser à décaler les indices de la tranche.

```python
def var_jour(L:list)->list:
    '''Renvoie la liste des variations journalières de L'''
    V,n = [0],len(L)
    for i in range(1,n):
        V.append(L[i]-L[i-1])
    return V

def m_profit(L:list):
    '''Renvoie le triplet (p,a,v) où p est le profit maximal, a le jour d'achat
    et v le jour de vente'''
    s,d,f = m_tranche(var_jour(L))
    return (s,d-1,f-1)
```