Calculabilité, décidabilité, problème de l'arrêt

1. Définition et idée générale

Une **fonction calculable** est une fonction pour laquelle il existe un algorithme qui donne le résultat *pour toute entrée* de son *domaine de définition* (comme en maths, une fonction informatique n'est définie que pour certaines entrées : une fonction travaillant sur les entiers ne devrait *jamais* recevoir une liste en paramètre).

Un **problème décidable** est un problème auquel on peut répondre par oui ou non, et pour lequel il existe un algorithme qui répond *toujours* correctement.

> ⚠️ **Attention !**
>
> Un problème peut être formulé clairement sans pour autant être calculable ou décidable. Le problème de l'arrêt en est l'exemple central.

2. Le problème de l'arrêt

Question fondamentale : existe-t-il un programme `HALT` qui, lorsqu'on lui donne en entrée le code source d'un *autre* programme `P` et des *arguments* `args` *pour cet autre programme*, répond `True` lorsque `P(args)` se termine, ou `False` lorsque `P(args)` ne se termine pas ? C'est la version informatique d'une question simple en apparence, mais qui n'admet pas de solution algorithmique générale.

> 💭 **Remarque : signification précise de « se terminer »** ▼
>
> Cela signifie ici s'arrêter, *quelle qu'en soit la raison* — résultat correct ou non, erreur ou exception. *Seule la boucle infinie* correspond à « ne pas se terminer ». Quand on parle de terminaison, on ne juge pas la *qualité de la terminaison*, mais *seulement son existence*.

Réponse : le problème de l'arrêt est **INDÉCIDABLE**. Il **n**'existe **pas** d'algorithme qui dise, pour tout programme et toute entrée, si ce programme s'arrêtera. Cette limite est importante parce qu'elle montre qu'il existe des questions mathématiques et informatiques auxquelles *aucun* algorithme ne peut répondre à tous les coups.

> ✅ **La logique de la preuve** ▼
>
> La connaître n'est pas nécessairement indispensable.
>
> Supposons qu'il existe un tel programme `HALT`. Cela signifie que :

- si `P(args)` se termine, `HALT(P, args)` répond `True` ;
- sinon, `HALT(P, args)` répond `False`.

On construit alors le programme suivant, nommé `absurde` :

```
def absurde(x):                 # x est une chaîne de caractères représentant un programme
    if HALT(x, x):              # HALT(x, x) renvoie True si x(x) termine, False sinon
        while True: pass        # Lance une boucle infinie -> NE termine PAS
    # Sinon, le programme se termine (implicitement, par retour normal)
```

Ce programme `absurde` prend un argument `x` en entrée (une chaîne de caractères représentant un programme) :

- si `x` termine, alors `absurde(x)` ne termine pas, car une boucle infinie est lancée ;
- si `x` **ne** termine **pas**, alors `absurde(x)` termine.

Examinons alors les deux seules possibilités sur l'exécution `absurde(absurde)` — oui, on passe à `absurde` son *propre code source* en argument (on assimile le programme à son code source) :

- si `absurde(absurde)` se termine, alors `HALT` répond `True`, et `absurde` rentre dans une boucle infinie… donc `absurde(absurde)` ne termine pas : contradiction !
- si `absurde(absurde)` ne se termine pas, alors `HALT` répond `False`… et `absurde(absurde)` se termine : contradiction encore !

Ces deux cas sont exhaustifs (pour toute exécution, soit elle termine, soit elle ne termine pas), et dans les deux cas on aboutit à une contradiction. On a donc une contradiction systématique, ce qui est… absurde, et permet de conclure que le programme `HALT` n'existe pas.

Remarque importante : un point mérite d'être souligné, car il est rarement explicité. Le programme `absurde` n'est pas un programme quelconque — il est conçu *exprès* pour prendre en entrée une chaîne de caractères représentant un programme. C'est une contrainte délibérée — et c'est précisément elle qui rendra la contradiction possible ! Or `HALT` lui-même repose déjà sur cette convention : il reçoit un code source en premier argument. Passer un code source à `absurde` est donc parfaitement admissible — et passer à `absurde` *son propre code source* l'est tout autant. C'est *précisément ce choix de conception* qui rend l'auto-application `absurde(absurde)` licite, et c'est là que la contradiction surgit.

3. Liens avec d'autres notions

- La **récursivité** permet de construire des programmes qui peuvent s'arrêter ou boucler.
- La **complexité** mesure le coût d'un algorithme, mais ne dit pas si un problème est calculable.
- Le **problème de l'arrêt** éclaire la frontière entre ce qu'un algorithme peut et ne peut pas faire.

Rendu de monnaie et sac à dos

1. Définition et idée générale

Le **rendu de monnaie** et le **sac à dos** sont deux problèmes classiques d'**optimisation**. On cherche *toujours* la meilleure solution possible relativement à une contrainte donnée.

💭 **Remarque : point de vigilance ▼**

Ces problèmes peuvent *parfois* être résolus par un algorithme glouton. Pas *toujours*, donc !

⚠️ **Conseil important**

Revoir la notion d'algorithme glouton avant de lire la suite. En peu de mots : un algorithme glouton fait, à chaque étape, le meilleur choix *local*. Ce faisant, il donne généralement une *bonne* solution, mais qui **n'**est **pas** forcément *la **meilleure***. Trouver des solutions *locales* optimales peut donc aboutir à une solution *globale **non optimale***.

2. Problème du rendu de monnaie

Ce problème se pose lorsqu'un client paye en espèces : il donne une somme trop importante, et il faut lui rembourser le trop perçu. On doit donc atteindre une somme précise, avec une contrainte : utiliser un **minimum** de pièces.

💭 **Remarque : des pièces ou des billets ? ▼**

On assimile pièces et billets. On ne fait pas de distinction sur la nature, seulement sur la valeur de chaque « élément ».

✅ **Solution**

Le rendu de monnaie est souvent l'exemple de problème où un algorithme glouton fonctionne bien :

- on choisit la plus grande « pièce » inférieure à la somme à atteindre et on rend autant de fois que possible cette « pièce » ;
- on choisit la deuxième plus grande « pièce » inférieure à la somme *restant à atteindre* et on rend autant de fois que possible cette « pièce » ;
- etc. jusqu'à atteindre la somme voulue.

L'algorithme glouton peut donner un résultat ***optimal** garanti* **lorsque le « système de pièces » est bien choisi :**

- « optimal garanti » signifie ici que le glouton trouve **LA** meilleure solution ;
- la plupart des « systèmes de pièces » du monde conviennent : ils sont dits **canoniques**. Par exemple :
  - centimes : 1, 2, 5, 10, 20, 50 ;
  - pièces et billets : 1, 2, 5, 10, 20, 50, 100, 200, 500.

💭 **Remarque : un contre-exemple historique ▼**

Voir cette page de Wikipedia (https://fr.wikipedia.org/wiki/Probl%C3%A8me_du_rendu_de_monnaie#Exemples) qui évoque le système anglais avant sa réforme en 1971.

Voici le code Python du rendu de monnaie glouton.

```python
def rendu_glouton(montant : int, pieces : list) -> list:
    """
    Prend en paramètre un montant et une liste de pièces (des entiers).
    On considère qu'on dispose d'une quantité illimitée de pièces, ici.
    Renvoie la liste des pièces à rendre pour résoudre le problème.
    """
    resultat = []
    for piece in sorted(pieces, reverse=True):  # Prendre la plus grande pièce en 1er
        while montant >= piece:                 # Tant que c'est possible...
            resultat.append(piece)              # ...ajouter cette plus grande pièce
            montant -= piece                    # Actualiser le montant restant à rendre
    return resultat
```

3. Problème du sac à dos

On note parfois KP ce problème (de l'anglais *Knapsack Problem*). C'est un problème plus délicat à traiter que le rendu de monnaie...

On veut remplir un sac (à dos... ou pas) :

- *sans dépasser* une capacité donnée (en volume, ou en masse, ou ...) ;
- tout en *maximisant* la « valeur » transportée.

⚠️ **Attention aux variantes du sac à dos !**

Il existe au moins deux variantes de ce problème :

- le sac à dos « binaire » (ou « 0/1 ») : on prend un objet en totalité ou on ne le prend pas ;
- le sac à dos fractionnaire : on peut prendre une *portion* d'un objet.

Dans ce problème, on voit très bien la différence entre une stratégie gloutonne et une stratégie fondée sur la

programmation dynamique :

- un choix local se fondant sur le meilleur rapport « valeur/poids » peut échouer ;
- une solution par *programmation dynamique* explore les combinaisons pertinentes.

💭 Remarque : un petit exemple où l'algorithme glouton échoue ? ▼

Considérons un sac (à dos) d'une capacité de 10 kg. Voici la liste des objets emportables (non fractionnables).

Objet	Masse	Valeur	Rapport valeur/masse
A	1 kg	2 €	2
B	5 kg	6 €	1,2
C	5 kg	6 €	1,2

L'approche gloutonne conduit à choisir A (meilleur rapport valeur/masse), puis B, puis C → 1 + 5 + 5 = 11 kg, supérieur à la capacité. Donc la solution gloutonne se limite à prendre A et B, d'une valeur de 8 € pour une masse de 6 kg. Pourtant, prendre B et C donne une masse de 10 kg et une valeur de 12 €, ce qui est meilleur : le glouton a échoué.

💭 Remarque : logique et apport de la programmation dynamique ▼

La programmation dynamique résout le problème en construisant un tableau de résultats intermédiaires.
L'idée : pour *chaque objet* et *chaque capacité possible*, on répond à la question : *« quelle est la valeur maximale atteignable avec les objets vus jusqu'ici, pour **au plus** cette capacité ? »* On remplit ce tableau ligne par ligne (un objet par ligne), colonne par colonne (une capacité différente par colonne). À chaque case, on choisit la meilleure option entre :

- ne pas prendre l'objet courant (on recopie la valeur de la ligne précédente) ;
- prendre l'objet courant (on ajoute sa valeur à la meilleure solution pour la capacité réduite).

À la fin, la case en bas à droite du tableau contient la valeur optimale. On n'a jamais fait de choix irréversible : on a exploré toutes les combinaisons pertinentes sans les énumérer toutes naïvement.

Avec l'exemple précédent, le tableau construit est :

	0	1	2	3	4	5	6	7	8	9	10
ø	0	0	0	0	0	0	0	0	0	0	0
+A	0	2	2	2	2	2	2	2	2	2	2
+B	0	2	2	2	2	6	8	8	8	8	8
+C	0	2	2	2	2	6	8	8	8	8	12

On obtient bien la meilleure solution ici.

✅ **Solution par programmation dynamique** ▼

La programmation dynamique construit un tableau où chaque case permet de répondre à la question *« quelle est la valeur maximale avec ces objets et cette capacité ? »*. On évite ainsi les mauvais choix irréversibles faits par un glouton, au prix d'un coût en mémoire et en temps plus élevé.

4. Approches top-down et bottom-up

Ce paragraphe est hors programme !

La programmation dynamique peut s'implémenter de deux façons :

- ***« Top-down »*** (approche descendante) : on part du problème global et on le décompose récursivement en sous-problèmes, en mémorisant les résultats au fur et à mesure. (mémoïsation). C'est l'approche la plus naturelle à écrire, on la voit en action dans le calcul des termes de la suite de Fibonacci, dans la fiche sur la programmation dynamique.
- ***« Bottom-up »*** (approche ascendante) : on part des sous-problèmes les plus simples et on remplit un tableau de résultats intermédiaires jusqu'à atteindre la solution globale. C'est l'approche utilisée implicitement dans le tableau du sac à dos ci-dessus.

Les deux approches donnent le même résultat. Le *« bottom-up »* est souvent plus efficace en mémoire (pas de pile d'appels récursifs), mais le *« top-down »* est plus proche de la définition mathématique du problème.

5. Liens avec d'autres notions

- Le **glouton** est souvent la première méthode essayée.
- La **programmation dynamique** est essentielle pour le sac à dos « binaire ».
- La **complexité** permet de comprendre pourquoi certaines approches sont préférables.
- La **récursivité** peut servir à formuler la version naïve.

Algorithmes de tri

1. Définition et idée générale

Un algorithme de **tri** réorganise une liste pour mettre ses éléments dans un ordre donné, généralement croissant.

> ⚠️ **Attention**
>
> Un **tri correct** doit **conserver les valeurs inchangées** et **produire un ordre final *cohérent*.**
>
> Cela signifie que le tri ne doit ni *perdre*, ni *modifier*, ni *ajouter* de valeurs : on souhaite uniquement *permuter* les éléments initialement présents dans la liste, de sorte que ceux-ci soient rangés dans un ordre *défini* (croissant ou décroissant).

2. Illustration immédiate

Trier, c'est ordonner. Ainsi, on souhaite trouver un moyen pour que la liste Python `[3, 1, 4, 1, 5]` devienne, après traitement, `[1, 1, 3, 4, 5]`.

La question algorithmique est : comment faire cela efficacement ?

3. Les tris au programme

En première NSI, les tris rencontrés sont le **tri par insertion** (https://fr.wikipedia.org/wiki/Tri_par_insertion) et le **tri par sélection** (https://fr.wikipedia.org/wiki/Tri_par_s%C3%A9lection). Ces tris sont simples à comprendre, mais peu efficaces sur de grandes listes.

> ✅ **Astuce mnémotechnique**
>
> - Le tri par **insertion** est le tri du joueur de cartes : lorsqu'on veut trier une « main de cartes », on observe une carte et on l'**insère** directement à « la bonne place ».
> - Le tri par sélection consiste en :
>
> - sélectionner la plus petite valeur d'un tableau et la mettre à la première place de ce tableau ;

- sélectionner la *deuxième* plus petite valeur du tableau et la mettre à la **deuxième** place de ce tableau (c'est *la* plus petite parmi les valeurs *restant à trier*) ;
- et ainsi de suite.

En terminale, **le tri-fusion** (https://fr.wikipedia.org/wiki/Tri_fusion) est l'exemple important à connaître, car il illustre la notion de récursivité, et le principe « diviser pour régner » (DPR). En outre, il est efficace en raison d'une bonne complexité.

💭 **Remarque : pourquoi des tris inefficaces sont-ils néanmoins utilisés ?** ▼

Le tri par **insertion** présente un intérêt *réel* lorsque les données sont *presque triées* (exemple typique : on veut ranger « au bon endroit » un score dans un tableau des « high scores », par définition déjà trié). Dans cette situation particulière, le tri par insertion peut être très rapide, *plus efficace* que des tris plus rapides en général.

3.1. Tri par insertion

Considérons un tableau d'entiers `tab` de longueur `n`. Pour ranger ces entiers par ordre croissant, la logique du tri par insertion consiste, pour chaque position `i` allant de `1` à `n-1` (incluses), à « faire glisser » `tab[i]` vers la gauche « autant que possible ».

Pour cela, il faut :

- mémoriser la valeur `tab[i]` dans une variable `x` ;
- faire remonter la valeur `tab[i-1]` à la position `i` *si possible* : c'est le cas si `tab[i-1] > tab[i]` ;
- faire remonter la valeur `tab[i-2]` à la position `i-1` *si possible* (critère similaire) ;
- etc. ;
- lorsque le critère de déplacement n'est plus rempli, on a trouvé la position `k` où placer `x`.

Code Python

```
def tri_insertion(tab):
    n = len(tab)
    if n > 1:                               # Cas limite explicite
        for i in range(1, n):
            x = tab[i]                      # Valeur à « faire glisser »
            k = i
            while k > 0 and tab[k - 1] > x: # Important de vérifier k > 0 !
                tab[k] = tab[k - 1]         # Remontée de la valeur précédente
                k = k - 1                   # Passage à la valeur précédente
            tab[k] = x                      # Placer x à sa bonne position
```

Code Python avec affichage

```
def tri_insertion(tab):
    n = len(tab)
    if n > 1:
        for i in range(1, n):
            x = tab[i]
            print(f"On regarde {x} en position {i}")
            k = i
            while k > 0 and tab[k - 1] > x:
                print(f"  ...on remonte {tab[k - 1]}")
                tab[k] = tab[k - 1]
                print(tab)
                k = k - 1
            print(f"   > On positionne {x} en {k}")
            tab[k] = x
            print(tab)
```

✅ **À retenir** ▼

- Pour une liste de taille n, le coût moyen de son tri est fonction de n^2.
- Ce tri, bien que peu performant en moyenne, est très performant lorsque la liste est « presque triée ».

✏️ **Exemple : hall of fame** ▼

Imaginez un jeu qui conserve la liste ordonnée des meilleurs scores. Insérer un nouveau score amène à insérer une nouvelle valeur dans une liste déjà triée. Ajouter ce nouveau score à la fin de cette liste conduit à créer une liste « presque triée ». Dans cette situation, l'efficacité du tri par insertion est *très bonne* : il suffit de faire glisser la nouvelle valeur à sa bonne place. La complexité est alors en $O(n)$.

3.2. Tri par sélection

On considère toujours un tableau d'entiers `tab` de longueur `n`. La logique du tri par sélection consiste à : - chercher entre les positions `0` et `n-1` celle de la plus petite valeur : notons-la `k`. On échange ensuite les valeurs en position `0` et `k` :

```
tab[0], tab[k] = tab[k], tab[0]
```

- chercher entre les positions `1` et `n-1` celle de la (2e) plus petite valeur. On échange ensuite les valeurs en position `1` et `k` ; - et ainsi de suite jusqu'à ce que la position de départ de la recherche vaille `n-2`.

Pour cela, il faut parcourir les positions `0` à `n-1` avec une variable `i` et :

- parcourir les positions `i` à `n-1` avec une variable `j` ;
- trouver la plus petite valeur du tableau entre les positions `i` et `n-1` : notons `k` sa position ;
- échanger `tab[i]` et `tab[k]` (seulement si `i != k`).

Code Python

```python
def tri_selection(tab):
    n = len(tab)
    if n > 1:                               # Cas limite explicite
        for i in range(n - 1):
            m = tab[i]                      # Minimum initialisé
            pos_m = i                       # Position de ce min. initialisée
            for j in range(i, n):           # Recherche du minimum "au-delà"
                if tab[j] < m:
                    m = tab[j]              # "Meilleur" minimum trouvé
                    pos_m = j               # Sa position
            if pos_m != i:
                tab[i], tab[pos_m] = tab[pos_m], tab[i]     # Échange
```

Code Python avec affichage

```python
def tri_selection(tab):
    n = len(tab)
    if n > 1:
        for i in range(n - 1):
            print(tab)
            m = tab[i]
            pos_m = i
            for j in range(i, n):
                if tab[j] < m:
                    m = tab[j]
                    pos_m = j
            print(f"    - Min : {tab[pos_m]} en {pos_m} ∈ [{i};{n-1}]")
            if pos_m != i:
                tab[i], tab[pos_m] = tab[pos_m], tab[i]
                print(f"    - On échange les valeurs aux pos {pos_m} et {i}")
```

3.3. Tri-fusion

Voir la fiche dédiée.

4. Liens avec d'autres notions

- La **récursivité** est au cœur du tri-fusion.
- La méthode **diviser pour régner** décrit parfaitement sa structure.
- La **complexité** du tri fusion est en $O(n \log n)$, celle des tris abordés en classe de première est, en moyenne, en $O(n^2)$.
- Le **logarithme base 2** aide à comprendre cette complexité.

Diviser pour régner et recherche dichotomique

1. Définition et idée générale

La stratégie **diviser pour régner** consiste à découper un problème en sous-problèmes *plus petits*, à résoudre ces sous-problèmes, puis à combiner leurs réponses.

> ⚠️ **Attention !**
>
> Cette méthode fonctionne bien quand les sous-problèmes sont *vraiment* plus simples, et que leur combinaison est *« facile »*.

2. Utilisation

On rencontre cette technique dans :

- la recherche dichotomique ;
- le tri-fusion.

2.1. Exemple simple : la recherche dichotomique

La recherche dichotomique se fait sur une liste *triée* par ordre croissant :

- on « vise au milieu » de la liste ;
- soit on a trouvé la valeur cherchée : c'est gagné !
- soit la valeur cherchée est plus petite que la valeur au milieu : on relance une recherche dichotomique sur la première moitié de la liste ;
- soit la valeur cherchée est plus grande que la valeur au milieu : on relance une recherche dichotomique sur la deuxième moitié de la liste ;
- on réitère le processus jusqu'à ce qu'on ait trouvé la valeur cherchée, ou jusqu'à ce qu'on ne puisse plus « couper en deux » la liste (auquel cas, la valeur cherchée n'est pas présente dans la liste).

Version récursive

```
def recherche_dichotomique(tab, x):
    if tab == []:
        return False
    milieu = len(tab) // 2
    if tab[milieu] == x:
        return True
    if x < tab[milieu]:
        return recherche_dichotomique(tab[:milieu], x)
    return recherche_dichotomique(tab[milieu+1:] , x)
```

💭 Remarque : toujours exclure la position `milieu` !

En effet, si `x` n'est pas à la position `milieu`, on cherche soit dans `tab[:milieu]`, soit dans `tab[milieu+1:]`. Dans les deux cas, la position `milieu` est exclue (dans le *slicing* Python, la valeur finale est *toujours* ***exclue***).

⚠️ Attention au coût ! ▼

Dans le code précédent, le *slicing* `tab[:milieu]` ou `tab[milieu+1:]` créent des **copies** de (portions de) la liste `tab` d'origine. Cela induit un surcoût *important* en temps d'exécution : en effet, si l'on tient compte du coût des copies, la complexité peut atteindre $O(n \log n)$ au lieu de $O(\log n)$ au mieux. Ici, les facilités offertes par Python se payent au prix fort en termes d'efficacité... Pour éviter ces copies, on utilise des indices dans la fonction. Cela donne :

```
def recherche_dichotomique(tab, x, gauche, droite):
    if gauche > droite:
        return False
    milieu = (gauche + droite) // 2
    if tab[milieu] == x:
        return True
    if x < tab[milieu]:
        return recherche_dichotomique(tab, x, gauche, milieu - 1)
    else:
        return recherche_dichotomique(tab, x, milieu + 1, droite)

lst = [10, 20, 30, 50, 70, 90]
recherche_dichotomique(lst, 42, 0, len(lst)-1)     # Passer les bornes initiales, ici
```

Version itérative

```python
def recherche_dichotomique(tab, x):
    deb, fin = 0, len(tab) - 1
    while deb <= fin:
        milieu = (deb + fin) // 2
        if tab[milieu] == x:
            return True
        elif x < tab[milieu]:
            fin = milieu - 1
        else:
            deb = milieu + 1
    return False
```

> ⚠️ **Pièges classiques de ce code**
>
> La recherche dichotomique itérative présente deux pièges.
>
> - Si `tab[milieu]` n'est pas la valeur cherchée, inutile de continuer à garder la position `milieu` dans l'intervalle de recherche. Comme on sait déjà qu'elle ne peut pas convenir, il faut la retirer, avec `milieu - 1` ou `milieu + 1` selon le cas. De cette façon, on évite le risque de retomber sur la « même case » et de bloquer l'algorithme. Dit autrement, on s'assure ainsi que la *condition d'arrêt* de la boucle `while` sera bien atteinte.
> - Le calcul de la position moyenne, en raison de la (nécessaire) division entière (ou *euclidienne*) par `2`, peut amener à ce que `deb` et `fin` aient la *même valeur*. Le point précédent garantit alors que, si la valeur cherchée n'est ni en `deb` (ni en `fin`, qui est égal à `deb`), on aura au tour suivant `deb > fin`, et la boucle s'arrêtera.

2.2. Exemple plus avancé : le tri-fusion

Voir la fiche dédiée.

3. Points d'attention complémentaires

- Diviser ne suffit pas : il faut aussi savoir recombiner.
- Le gain de complexité vient essentiellement de la division répétée en deux parties (à peu près) égales.
- Le tri-fusion est l'exemple classique à connaître.
- Attention aux facilités de Python qui, si elles permettent un code concis et simple, masquent la complexité de certaines opérations.

4. Liens avec d'autres notions

- Le **tri-fusion** est l’exemple central.
- La **récursivité** est très souvent utilisée pour l’implémentation.
- La **complexité** permet d’évaluer le gain.
- La **recherche dichotomique** repose elle aussi sur une division par deux.

Programmation dynamique

1. Définition et idée générale

La programmation dynamique consiste à résoudre un problème en mémorisant les résultats des sous-problèmes déjà calculés afin d'éviter les recalculs inutiles. C'est ce qu'on appelle la **mémoïsation**.

> ✅ **Quand l'employer ?**
>
> La programmation dynamique est surtout utile quand les *mêmes sous-problèmes* reviennent *plusieurs fois*.

> 💭 **Remarque : mémoïsation *vs* mémorisation** ▼
>
> **Mémorisation** est un terme général : c'est le fait de stocker quelque chose en mémoire.
> **Mémoïsation** est un terme technique précis en informatique : c'est une stratégie d'optimisation qui consiste à mettre en cache le résultat d'un appel de fonction selon ses arguments, pour éviter de le recalculer si la fonction est rappelée avec les mêmes arguments. C'est donc une forme particulière de mémorisation.
>
> **En résumé :** toute mémoïsation *est* une mémorisation, mais toute mémorisation **n'**est **pas** une mémoïsation.

2. Applications

La programmation dynamique se distingue de « diviser pour régner » (DPR) :

- en programmation dynamique, les sous-problèmes sont « plus petits », mais ils se *chevauchent* (ou se répètent, se recoupent…) ;
- dans DPR, les sous-problèmes sont « plus petits » **mais** *différents* (…*a priori* : il se *pourrait* qu'ils se recoupent, mais ce serait un *hasard*).

C'est particulièrement visible avec le calcul récursif « naïf » des termes de la suite de Fibonacci (voir ci-dessous).

2.1. Exemple simple : la suite de Fibonacci

La suite de Fibonacci est définie par :

- $F_0 = 0$ et $F_1 = 1$;
- $F_n = F_{n-1} + F_{n-2}$.

Cette définition est *récursive* : on calcule un nouveau terme de la suite en additionnant les deux termes précédents.

> **Version récursive naïve**

```python
def fibo_rec(n):
    print(f"Calcul de F_{n}")
    if n <= 1:
        return n
    else:
        return fibo_rec(n-1) + fibo_rec(n-2)
```

Cette définition récursive est simple à écrire, mais recalcule sans cesse les mêmes valeurs.

✏️ **Exemple : en pratique** ▼

L'instruction `fibo_rec(5)` provoque l'affichage suivant :

```
Calcul de F_5
Calcul de F_4
Calcul de F_3
Calcul de F_2
Calcul de F_1
Calcul de F_0
Calcul de F_1
Calcul de F_2
Calcul de F_1
Calcul de F_0
Calcul de F_3
Calcul de F_2
Calcul de F_1
Calcul de F_0
Calcul de F_1
5
```

Version « programmation dynamique »

```python
def fibo_dyn(n, memo=None):
    if memo is None:
        memo = {}
    if n in memo:
        return memo[n]
    print(f"Calcul de F_{n}")
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fibo_dyn(n - 1, memo) + fibo_dyn(n - 2, memo)
    return memo[n]
```

Cette version utilise une mémoïsation : si `fibo_dyn(n)` a déjà été calculé, on peut réutiliser le résultat sans le recalculer.

✏️ **Exemple : en pratique** ▼

L'instruction `fibo_dyn(5)` provoque l'affichage suivant :

```
Calcul de F_5
Calcul de F_4
Calcul de F_3
Calcul de F_2
Calcul de F_1
Calcul de F_0
5
```

💭 **Remarque : pour aller plus loin (considérations avancées)** ▼

⚠️ **Attention aux arguments par défaut *mutables* !**

En Python, les valeurs par défaut des arguments sont *évaluées* ***une seule fois***, au moment de la définition de la fonction — **pas** à chaque appel. Si cette valeur est un objet *mutable* (liste, dictionnaire…), il est **partagé entre tous les appels** qui n'en fournissent pas explicitement un.

✏️ **Exemple : dans le cas qui nous occupe** ▼

C'est pour cela que l'on trouve `memo=None` dans la définition de `fibo_dyn` : l'initialisation explicite dans le corps est *sûre*.

```python
def fibo_dyn(n, memo=None):
    if memo is None:
        memo = {}   # nouveau dictionnaire à chaque appel "racine"
```

Si on avait écrit :

```python
def fibo_dyn(n, memo={}):  # ⚠️ PIÈGE !
```

…alors *le* ***même*** *dictionnaire serait* ***réutilisé*** *d'un appel à l'autre* .

Cela peut sembler pratique, mais peut provoquer des effets de bord difficiles à déboguer. Ainsi, les résultats d'un premier appel `fibo_dyn(5)` seraient encore dans `memo` lors d'un appel ultérieur `fibo_dyn(10)`.

Dans ce cas précis, ce serait même *bénéfique* — mais c'est un **hasard**, et en général le résultat sera **faux !**

Règle à retenir : ne jamais mettre une liste ou un dictionnaire comme valeur par défaut d'un argument. Utiliser None et initialiser dans le corps de la fonction.

✏️ **Exemple : une situation *vraiment* moisie** ▼

Le code suivant montre une fonction créant *récursivement* la liste des entiers entre `n` et `1` :

```python
def liste_des_entiers(n, resultats=[]):
    if n == 0:
        return resultats
    resultats.append(n)
    return liste_des_entiers(n - 1, resultats)

print(liste_des_entiers(3))  # → [3, 2, 1]  ✅ (premier appel)
print(liste_des_entiers(3))  # → [3, 2, 1, 3, 2, 1]  ⚠️ la liste s'allonge ☠️ !
```

Au deuxième appel, la liste par défaut contient *déjà* `[3, 2, 1]` du premier appel — les nouveaux éléments s'y ajoutent, au lieu de repartir d'une liste vide.

Pour corriger ce code, il suffit de faire :

```
def liste_des_entiers(n, resultats=None):
    if resultats is None:
        resultats = []
    if n == 0:
        return resultats
    resultats.append(n)
    return liste_des_entiers(n - 1, resultats)
```

2.2. Exemple plus avancé

Le problème du sac à dos en version 0/1 se prête très bien à la programmation dynamique (dans cette version « binaire » du problème du sac à dos, « on prend un objet *en totalité*, ou pas du tout » — on *ne* peut *pas* prendre une *fraction* des objets disponibles). En effet, on compare plusieurs choix possibles pour une même capacité restante.

3. Points d'attention complémentaires

- La programmation dynamique évite les recalculs.
- Elle repose sur la mémoïsation (mémorisation des résultats déjà établis).
- Elle sert souvent à obtenir une solution optimale.

> ⚠️ **Différence avec un algorithme glouton**
>
> La programmation dynamique n'est pas la même chose que le glouton : elle ne se contente pas d'un choix *localement optimal* immédiat. Elle explore toutes les combinaisons *possibles* et garantit une solution optimale, là où le glouton peut échouer.

4. Liens avec d'autres notions

- Fibonacci est l'exemple classique à connaître.
- La **récursivité** s'invite volontiers dans le paysage.
- Un algorithme *glouton* peut échouer **là où la programmation dynamique réussit !**
- Le **problème du sac à dos** illustre bien l'intérêt de la méthode (voir la fiche associée).
- La **complexité** est souvent fortement améliorée par rapport à une version naïve.

Tri-fusion

1. Définition et idée générale

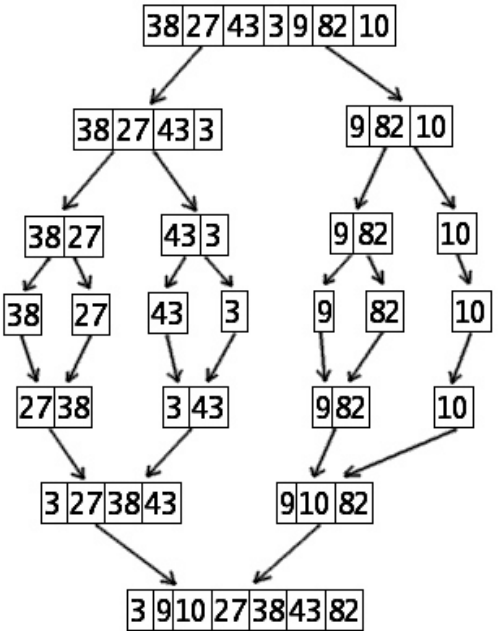
Le **tri-fusion** est un algorithme de tri qui divise une liste en deux, trie les deux moitiés, puis fusionne les résultats. C'est un bon exemple de l'approche **diviser pour régner** (au programme de terminale NSI).

Sa complexité est en $O(n \log n)$: c'est efficace !

2. Illustration

Cela ressemble à une situation où deux personnes décident de collaborer pour classer chacune une moitié d'une pile de feuilles, avant de réunir les deux piles déjà ordonnées.

Voici une illustration provenant de Wikipedia, illustrant le processus avec des nombres :



Cliquez ici pour une visualisation dynamique de la version récursive (https://www.recursionvisualizer.com/?function_definition=def%20fusion%28gauche%2C%20droite%29%3A%20%20%20%20%0A%20%20%20%20resultat%20%3D%20%5B%5D%0A%20%20%20%20%0A%20%20%20%20resultat%20%3D%20resultat%20%2B%20gauche%5Bindex_gauche%3A%5D%20%2B%20droite%5Bindex_droite%3A%5D%0A%

3. Algorithmes

3.1. Tri-fusion récursif

L'algorithme ci-dessous, tiré de Wikipedia, définit le fonctionnement d'une fonction récursive `tri_fusion` (une version itérative est possible). Elle effectue le tri-fusion en faisant appel à la fonction auxiliaire `fusion` qui, elle, ne s'occupe que de la fusion.

```
entrée : un tableau T
sortie : une permutation triée de T
fonction tri_fusion(T[1, …, n])
    si n ≤ 1
        renvoyer T
    sinon
        renvoyer fusion(tri_fusion(T[1, …, n/2]), tri_fusion(T[n/2 + 1, …, n]))
```

💭 **Remarque : précisions diverses ▼**

- Le pseudo-code ci-dessus montre le recours à une logique de *« slicing »*. Il est toujours possible de l'éviter (grâce à une compréhension de liste en Python, par exemple).
- Cela peut avoir de l'importance : en Python, le *slicing* effectue une **copie** de la liste initiale, ce qui a un *coût* (en temps et en mémoire).
- La numérotation des cases du tableau est faite à partir de `1` : en Python, on compte à partir de `0`.
- Lorsqu'on trouve `n/2` dans le pseudo-code, il faut comprendre *« partie **entière** de `n/2` »*.
  En Python, on écrirait `n // 2`.

3.2. Fusion récursive de deux listes

La fonction suivante est également définie récursivement (là encore une version itérative existe). Le symbole `⊕` désigne la *concaténation* de tableaux.

```
entrée : deux tableaux TRIÉS, nommés A et B
sortie : un tableau trié qui contient exactement les éléments des tableaux A et B
fonction fusion(A[1, …, a], B[1, …, b])
    si A est le tableau vide
        renvoyer B
    si B est le tableau vide
        renvoyer A
    si A[1] ≤ B[1]
        renvoyer A[1] ⊕ fusion(A[2, …, a], B)
    sinon
        renvoyer B[1] ⊕ fusion(A, B[2, …, b])
```

💭 **Remarque : concaténation de tableaux en pratique ? ▼**

En pratique, concaténer des listes Python se fait simplement avec l'opérateur `+`. Mais le faire un grand nombre de fois a aussi un *coût*, car cette fusion *crée un **nouvel objet** en mémoire*.

4. Codes Python

4.1. Fusion

La fonction de fusion compare les premières valeurs de deux listes triées et construit progressivement une nouvelle liste triée.

Version récursive

```python
def fusion(A, B):
    if A == [] :
        return B
    if B == [] :
        return A
    if A[0] <= B[0]:
        return [A[0]] + fusion(A[1:], B)
    else:
        return [B[0]] + fusion(B[1:], A)
```

Cette version utilise intensivement le *slicing* et la concaténation de listes Python : cela se paie *cher* en temps d'exécution !

Version itérative

```python
def fusion(gauche, droite):
    len_g, len_d = len(gauche), len(droite)
    resultat = [None] * (len_g + len_d)   # On crée d'emblée UNE liste de la bonne taille, qui
    # sera peuplée par les valeurs de `gauche` et `droite`, dans le bon ordre, puis renvoyée.
    i = j = 0
    while i < len_g and j < len_d:        # Tant qu'il reste des éléments dans les 2 listes
        if gauche[i] <= droite[j]:        # on compare les valeurs en cours : la plus petite
            resultat[i+j] = gauche[i]     # va dans la liste fusionnée, qu'elle vienne de gauche
            i += 1                        # ...
        else:                             # ...
            resultat[i+j] = droite[j]     # ou de la liste de droite.
            j += 1                        # On actualise la position associée, pour poursuivre.
    if j == len_d:                        # Si on a épuisé la liste de droite
        while i < len_g:                  # on complète, avec les valeurs restant dans la liste
            resultat[i+j] = gauche[i]     # de gauche, la liste qui sera renvoyée à la fin.
            i += 1
    else:                                 # Sinon,
        while j < len_d:                  # on fait de même avec ce qui reste
            resultat[i+j] = droite[j]     # dans la liste de droite.
            j += 1
    return resultat
```

On cherche ici à éviter absolument le *slicing* et la concaténation des `list` Python provoquée par le `+`. Le code est moins lisible, l'idée générale est sensiblement moins perceptible… mais le programme en résultat est plus performant.

💭 **Remarque : mais pourquoi ce code est-il plus performant ?** ▼

Python est un langage très expressif : il permet simplement de « faire des choses » qui demanderaient plus d'effort dans un autre langage. Mais il faut parfois aller au-delà de la « magie de Python » pour comprendre ce qui se cache derrière… et les conséquences qui en découlent. Ainsi, le recours au *slicing* (ou à une compréhension de liste) ainsi que l'emploi de la concaténation de listes amènent Python à *créer de **nouveaux** objets* en mémoire : cela a un coût ! De même l'utilisation des méthodes `append()` ou `extend()` entraîne régulièrement des surcoûts. Tout cela finit par dégrader le niveau de performance d'un code !

4.2. Tri-fusion récursif

```python
def tri_fusion(tab):
    if len(tab) <= 1:
        return tab
    milieu = len(tab) // 2
    gauche = tri_fusion(tab[:milieu])
    droite = tri_fusion(tab[milieu:])
    return fusion(gauche, droite)
```

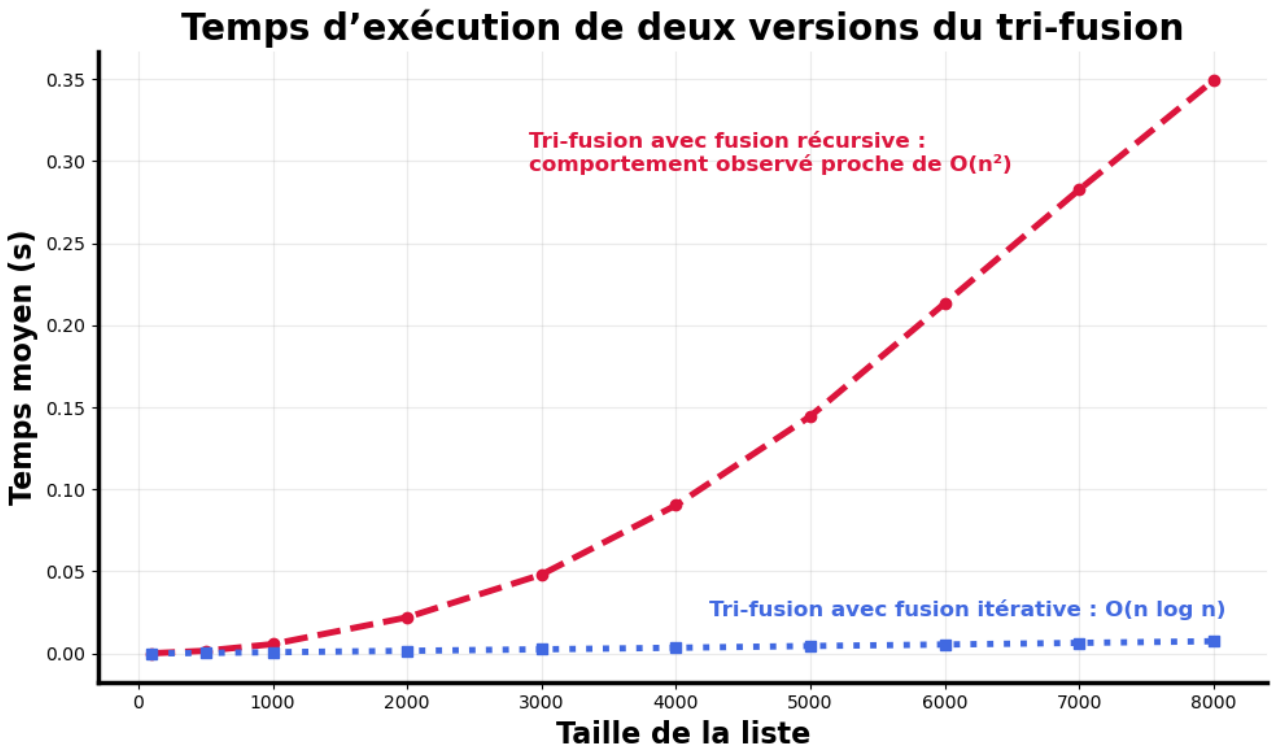
- On voit bien la récursivité du tri-fusion dans ce code, et son lien direct avec la méthode « diviser pour régner ».
- Le cas de base est une liste de taille 0 ou 1 (qui ne nécessite donc aucun traitement).

💭 **Remarque : il y a *slicing* ! ▼**

Certes, et on peut à nouveau le remplacer par des compréhensions de listes.

💭 **Remarque : *quid* des performances ? ▼**

Voici un graphique illustrant la différence entre une implémentation *itérative* ***efficace*** de la partie fusion du tri-fusion, et la *fusion récursive* ***naïve*** de cette même fusion. Le *coût* ***pratique*** de la version *récursive* ***naïve*** ne correspond pas à la complexité *théorique* de l'idée derrière le tri-fusion. C'est bien la version Python récursive avec *slicing* et concaténations qui dégrade *fortement* le comportement observé, au point qu'on observe un comportement proche de $O(n^2)$.



5. Liens avec d'autres notions

- La **récursivité** est au cœur de cet algorithme.
- Ce tri relève de l'approche **diviser pour régner**.
- La **complexité** du tri-fusion est en $O(n \log n)$.
- La **recherche dichotomique** partage l'idée de couper le problème en deux.