3
Récursivité

1. La **récursivité** n'est pas une théorie mais un **modèle de conception** d'algorithme (*design pattern*). On n'exposera donc que des considérations générales, qui serviront de guides, que nous illustrerons par quelques exemples qui complètent les exemples présentés dans le chapitre *Algorithmes fondamentaux*.

2. Comme la récursivité en algorithmique est avant tout une question de style, il y a ceux qui aiment :

To iterate is human, to recurse divine.
L. Peter Deutsch

Et ceux qui n'aiment pas :
I don't believe in recursion as the basis of all programming. This is a fundamental belief of certain computer scientists [...] But to me, seeing recursion as the basis of everything else is just a nice theoretical approach to fundamental mathematics [...], not a day-to-day tool.
For practical purposes, Python-style lists [...], and sequences in general, are much more useful to start exploring the wonderful world of programming than recursion. They are some of the most important tools for experienced Python programmers, too.

Guido van Rossum (créateur de Python)

I
Principes

3. ▲ Fonctions récursives
*On dit qu'une fonction est **récursive** lorsque l'écriture de cette fonction fait appel à la fonction elle-même.*

4. Programmation récursive
Le **style récursif** de programmation consiste donc à *répéter une opération* au moyen d'une fonction qui s'appelle elle-même.
Ce style s'oppose au **style itératif** qui utilise une boucle conditionnelle (*while*) ou non (*for*) pour répéter un même type de calcul.

5. Le problème de la terminaison
Répéter une opération est la raison d'être d'un algorithme. Mais il faut aussi savoir s'arrêter.
5.1 On sait à l'avance quand une boucle *for* va s'arrêter.
5.2 Pour une boucle *while*, on connaît une condition qui décide de l'arrêt de la boucle et il n'est pas toujours simple de démontrer que cette condition sera vérifiée.
Dans le meilleur des cas, on sait qu'une boucle conditionnelle s'arrêtera et on sait estimer le nombre d'opérations effectuées.
Parfois (et ce n'est pas toujours une erreur de programmation), une boucle conditionnelle est infinie et ne s'arrête pas sans une intervention de l'utilisateur.
5.3 Puisqu'une fonction récursive s'appelle elle-même, un algorithme récursif résout un problème donné en résolvant un ou plusieurs problèmes *de même nature*.
Pour qu'un tel algorithme ne nous emporte pas dans un calcul infini, il faut s'assurer que chaque sous-problème résolu nous rapproche vraiment du terme du calcul et en particulier quels sont les cas exceptionnels qui vont décider de la fin du calcul.

I.1 Méthode

6. L'exécution d'une fonction récursive ressemble à la descente d'un escalier :
– Passer d'un problème à un sous-problème consiste à descendre d'une marche ;
– On parvient au bas de l'escalier lorsque le ou les cas exceptionnels, dits **cas de base**, apparaissent et marquent la fin du calcul.

7. How to
Recursion strategy: first test for one or two base cases that are so simple, the answer can be returned immediately.
Otherwise, make a recursive a call for a smaller case (that is, a case which is a step towards the base case).
Assume that the recursive call works correctly, and fix up what it returns to make the answer.
`https://codingbat.com/java/Recursion-1`

Autrement dit : inspirez-vous des exemples simples et lorsque l'occasion se présente, tâchez de les imiter.

8. Exemples simples
On a présenté plusieurs exemples d'algorithmes récursifs dans le chapitre *Algorithmes fondamentaux*.
– Maximum d'une liste
– Présence d'un élément dans une liste
– Position d'un élément dans une liste
– Calcul du pgcd par l'algorithme d'Euclide
– Décomposition d'un entier en base 2
– Exponentiation rapide
– Recherche dichotomique dans une liste triée
– Recherche d'un motif dans une chaîne de caractères.
On recommande d'étudier ces exemples avant d'aller plus loin, et en particulier de comparer ces algorithmes avec leurs analogues impératifs.

Exemple détaillé : Tri sélection

9. La *description* qu'on va donner du **tri sélection** est typique d'un algorithme récursif.
Sa *mise en œuvre* sera elle aussi typique : la résolution du problème posé fera apparaître des sous-problèmes qui seront résolus de proche en proche (approche *top down*).

10. Le *tri sélection* consiste à repérer le plus grand élément d'une liste `L` à le placer en fin de liste et à recommencer avec la sous-liste `L[:-1]`.
10.1 Cas de base
Comme on s'en doute, une liste réduite à un seul élément est déjà triée. Le cas de base est donc le cas où la longueur de la liste est égale à 1.
10.2 Appel récursif
Le code est alors simple à écrire.

```
def trier(L):
  if len(L)==1:                # Cas de base :
    return L                   # la liste est déjà triée
  else:
    placer_max(L)
    return trier(L[:-1])+[L[-1]]  # Appel récursif
```

Il reste à écrire la fonction `placer_max(L)` qui modifie la liste `L` en plaçant la plus grande valeur en queue de liste.

11. Position du plus grand élément
On va parcourir la liste `L` en comparant à chaque étape deux éléments consécutifs : on les permute pour que le plus grand des deux soit placé après le plus petit.
11.1 Lors du parcours, on va rencontrer le plus grand élément de la liste. Dès lors, ce plus grand élément sera systématiquement comparé aux éléments qui le suivent et, de permutation en permutation, ce plus grand élément finira à la dernière place de la liste.
11.2 C'est simple en style impératif.

```
def placer_max(L):
  n = len(L)
  for i in range(n-1):
    tranposer(L, i)
```

11.3 C'est moins évident à formuler récursivement et l'algorithme récursif n'est pas plus efficace que l'algorithme impératif. Pour ces raisons, l'algorithme impératif est préférable !
Il n'y a rien à faire si la longueur n de la liste `L` est égale à 1 (*cas de base*).
Sinon, il faut localiser le maximum de la liste parmi les n éléments et c'est cette opération qui peut être formulée récursivement.

```
def placer_max(L):
  n = len(L)
  if n>1:
    placer_max_rec(0, n, L)
```

L'exécution de `placer_max_rec(i, n, L)` consiste à localiser récursivement le maximum parmi les $(n-i)$ derniers éléments de la liste `L`.

```
def placer_max_rec(i, n, L):
  #                  Cas de base : ne rien faire
  if i+1<n:        # Appel récursif
    transposer(L, i)
    placer_max_rec(i+1, n, L)
```

11.4 Transposition
Quel que soit le style de programmation retenu, l'opération élémentaire consiste à ranger par ordre croissant deux éléments consécutifs d'une liste.

```
def transposer(L, i):
  if L[i+1]<L[i]:
    L[i], L[i+1] = L[i+1], L[i]
```

On utilise ici le fait qu'une liste peut être modifiée par une fonction (type *mutable*).

I.2 Avantages de la récursivité

12. Il est fréquent qu'une suite $(u_n)_{n\in\mathbb{N}}$ soit définie par une relation de récurrence simple et qu'on ne sache pas exprimer u_n directement en fonction de n.
De même, l'écriture d'une fonction récursive peut être *plus simple* que l'écriture de la même fonction en style impératif (revoir à ce propos les exemples présentés dans le chapitre *Algorithmes fondamentaux*).
L'exemple des Tours de Hanoï **[25]** montre que l'écriture d'un algorithme récursif peut être *beaucoup plus simple* que celle d'un algorithme impératif.

13. Il est facile de prouver la correction et la terminaison d'un algorithme récursif correct et qui termine, puisqu'il suffit d'exploiter la relation de récurrence sur laquelle on a fondé cet algorithme.

14. Certains problèmes sont naturellement récursifs :
– l'évaluation d'une fonction définie par une relation de récurrence (qu'on pourra traduire directement par une fonction récursive) ;
– une opération qui porte sur une structure de données récursive unidimensionnelle (liste, pile...) ou bidimensionnelle (arbre...).
On aura avantage à coder ces problèmes au moyen d'une fonction récursive — il peut arriver qu'on ne puisse pas faire autrement !

15. La *programmation fonctionnelle* ignore la notion de variable (et donc d'affectation) et se contente d'évaluer des fonctions. Dans un tel cadre, toute itération repose sur une fonction récursive.

I.3 Limites de la récursivité

16. Pour un algorithme récursif, il n'est pas toujours facile de comprendre les opérations effectuées, leur nombre ou l'ordre dans lequel elles sont effectuées.
Dans certains cas, le fonctionnement d'un algorithme récursif revêt un aspect magique : ça marche et si on comprend pourquoi (la relation de récurrence explique à elle seule le fonctionnement), on ne voit pas toujours clairement pourquoi.

17. L'approche *top down* appliquée plus haut pour le Tri sélection ([**10**], [**11**]) est séduisante, mais elle demande patience et rigueur pour être mise en œuvre.
Son principal défaut est qu'on ne peut pas tester le code avant d'avoir tout codé !

Efficacité

18. Quand une fonction peut être programmée récursivement et itérativement, les deux styles sont équivalents en temps de calcul : la récursivité ne permet pas de diminuer le nombre de calculs à effectuer.
Il est même possible qu'un algorithme récursif, *mal mis en œuvre*, soit d'une efficacité lamentable comme on va le voir sur les deux exemples qui suivent.

19. Coefficients binomiaux
19.1 On peut utiliser la formule du triangle de Pascal pour calculer les coefficients binomiaux :

$$\forall\, 0 \leqslant k \leqslant n, \quad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}.$$

```
def cb(n, k):
  if (k>n):
    return 0
  elif (k==0) or (k==n):
    return 1
  else:
    return cb(n-1, k-1)+cb(n-1,k)
```

Cet algorithme est catastrophique : les seules opérations effectuées sont des additions (de 0 ou de 1) et il faut effectuer $\mathcal{O}(2^n)$ opérations pour calculer $\binom{n}{k}$!
19.2 Si on veut calculer récursivement les coefficients binomiaux avec une efficacité raisonnable, c'est la formule de récurrence qu'il faut changer ! Avec la relation

$$\forall\, 1 \leqslant k < n, \quad \binom{n}{k} = \frac{n-k}{k} \times \binom{n}{k-1}$$

on obtient un algorithme beaucoup plus efficace puisqu'il suffit de $\mathcal{O}(n)$ multiplications pour calculer $\binom{n}{k}$.

```
def coeff_binomial(n, k):
  if (k==0):
    return 1
  else:
    return coeff_binomial(n, k-1)*(n-k)//k
```

(On *sait* que l'entier $(n-k)\binom{n}{k-1}$ est divisible par k, c'est pourquoi la division euclidienne donne ici le résultat exact.)

20. Nombres de Fibonacci

20.1 Si on calcule les nombres de Fibonacci en exploitant naïvement la relation de récurrence qui les définit :

$$\forall\, n \in \mathbb{N}, \quad F_{n+2} = F_{n+1} + F_n$$

on aboutit aussi à un algorithme catastrophique.

```
def F(n):
  if (n<2):
    return 1
  else:
    return F(n)+F(n-1)
```

Ici encore, la seule opération effectuée est l'addition de 1 et on doit donc effectuer F_n additions pour calculer F_n !

20.2 Pour écrire un algorithme récursif raisonnablement efficace, il faut modifier la relation de récurrence :

$$\forall\, n \in \mathbb{N}, \quad \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix}.$$

20.3 Dans ce cas, on effectue $\mathcal{O}(n)$ appels récursifs pour calculer F_n, mais l'unique opération effectuée à chaque appel récursif est assez coûteuse : il s'agit d'un produit matriciel.
On aura donc intérêt à utiliser l'exponentiation rapide, qui permet de calculer F_n en effectuant seulement $\mathcal{O}(\lg n)$ produits matriciels.

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Profondeur maximale de récursion

21. Somme des éléments d'une liste

On souhaite ici calculer la somme des éléments d'une liste de nombres.

$$L = (x_k)_{0 \leqslant k < n}$$

21.1 Version impérative

L'algorithme suivant calcule la somme des éléments d'une liste `L`.

```
def somme(L):
  S = 0     # S_0 = 0
  for x in L:
    S += x  # S_k = S_{k-1} + x_k
  return S
```

Cet algorithme affecte à la variable `S` les valeurs suivantes :

$$\begin{aligned}
&\text{(Initialisation)} && 0 \\
&\text{(Itération)} && 0 + x_0 \\
& && x_0 + x_1 \\
& && x_0 + x_1 + x_2 \\
& && \vdots \\
& && x_0 + x_1 + \cdots + x_{n-2} + x_{n-1}
\end{aligned}$$

et retourne la dernière valeur calculée.

21.2 Version récursive

L'algorithme suivant calcule la somme des éléments d'une liste de manière récursive.

```
def S(L):
  if len(L)==0: # Cas de base : liste vide
    return 0
  else:         # Appel récursif
    return L[0]+S(L[1:])
```

21.3 Le *cas de base* est celui de la liste vide : par convention, la somme des éléments de cette liste est égale à 0.
Dans le cas général, la somme de la liste `L`$=(x_0, \ldots, x_{n-1})$ est égale à la somme de x_0 et de la somme de la sous-liste `L[1:]`$=(x_1, \ldots, x_{n-1})$. Il est donc bien clair que cette fonction calcule la somme des éléments de la liste `L`.

21.4 En pratique, les calculs effectués sont les suivants.
Dans un premier temps, on descend l'escalier en mettant les calculs en attente.

$$\begin{aligned}
&x_0 + \texttt{S(L[1:])} \\
&x_0 + (x_1 + \texttt{S(L[2:])}) \\
&\cdots \\
&x_0 + (x_1 + (x_2 + (\cdots + (x_{n-1} + \texttt{S([])}))))
\end{aligned}$$

Dans un second temps, on effectue les calculs en remontant l'escalier.

$$\begin{aligned}
&x_0 + (x_1 + (x_2 + (\cdots + (x_{n-1} + (0)))))\\
&\cdots \\
&x_0 + (x_1 + (x_2 + \cdots + x_{n-1})) \\
&x_0 + (x_1 + x_2 + \cdots + x_{n-1}) \\
&x_0 + x_1 + x_2 + \cdots + x_{n-1}
\end{aligned}$$

22. Comme on vient de le voir, un algorithme itératif exécute un certain nombre d'opérations les unes à la suite des autres : chacun son tour !

22.1 Au contraire, un algorithme récursif descend l'escalier marche après marche, entame un nouveau calcul à chaque marche et ne commence vraiment à calculer qu'une fois arrivé en bas de l'escalier.
Les calculs entamés constituent une pile et chaque appel récursif place un nouvel élément au sommet de la pile.

22.2 Pour ne pas saturer la mémoire de l'ordinateur, Python limite le nombre d'appels récursifs à 1 000.
Concrètement, si on applique la fonction `S` à une liste dont la longueur est strictement supérieure à 1 000, on reçoit le message d'erreur suivant.

```
RuntimeError : maximum recursion depth exceeded
```

II

Exemples classiques

II.1 Résolution de l'équation de Bezout

23. Si le pgcd de deux entiers a et b est égal à d, alors il existe deux entiers u_0 et v_0 qui vérifient l'**équation de Bézout** :

$$au_0 + bv_0 = d.$$

23.1 Cas de base
Si $b = 0$, alors $d = a$ est un pgcd de a et b et le couple $(u_0, v_0) = (1, 0)$ est une solution de l'équation de Bézout.
23.2 Appel récursif
Si $b \neq 0$, alors on effectue la division euclidienne de a par b :

$$a = qb + r.$$

D'après l'algorithme d'Euclide, le pgcd de a et b est aussi égal au pgcd de b et r. Il existe donc deux entiers u_1 et v_1 tels que

$$bu_1 + rv_1 = d.$$

On en déduit que

$$d = bu_1 + (a - qb)v_1 = av_1 + b(u_1 - qv_1)$$

et en particulier que le couple

$$(u_0, v_0) = (v_1, u_1 - qv_1)$$

est une solution de l'équation de Bézout.
23.3 Après cette analyse du problème, le codage récursif est simplissime.

```
def Bezout(a, b):
  if (b==0):  # Cas de base
    return (1, 0)
  else:       # Appel récursif
    q, r = a//b, a%b
    u, v = Bezout(b, r)  # u1, v1
    return (v, u-q*v)    # u0, v0
```

23.4 On peut utiliser cet algorithme en calculant à la main, mais c'est plutôt malcommode ! Mieux vaut utiliser l'algorithme de Blankinship qui résout l'équation de Bézout tout en calculant le pgcd.

II.2 Énumération du groupe symétrique

24. Le groupe symétrique $\mathfrak{S}_n$ est l'ensemble des permutations de

$$E_n = \{1, 2, \ldots, n\}.$$

Comment obtenir la liste complète des $n!$ éléments de $\mathfrak{S}_n$?
24.1 Une permutation $\sigma \in \mathfrak{S}_n$ est une bijection $s : E_n \to E_n$. Elle peut donc être identifiée à la liste

$$\big(\sigma(1), \ldots, \sigma(n)\big).$$

24.2 Cas de base
Si $n = 1$, il n'y a qu'une seule permutation : l'identité et cette permutation est représentée par la liste (1).
24.3 Appel récursif
Soit $n \geqslant 2$ et supposons connue une énumération

$$S = (s_k)_{0 \leqslant k < N}$$

du groupe symétrique $\mathfrak{S}_{n-1}$ (avec $N = (n-1)!$).

À chaque permutation $s_k \in \mathfrak{S}_{n-1}$ représentée par la liste

$$(\sigma_1, \ldots, \sigma_{n-1}),$$

on peut associer les listes

$$\begin{aligned}
&(n, \sigma_1, \sigma_2, \ldots, \sigma_{n-2}, \sigma_{n-1})\\
&(\sigma_1, n, \sigma_2, \ldots, \sigma_{n-2}, \sigma_{n-1})\\
&(\sigma_1, \sigma_2, n, \ldots, \sigma_{n-2}, \sigma_{n-1})\\
&\ldots\\
&(\sigma_1, \sigma_2, \ldots, \sigma_{n-2}, n, \sigma_{n-1})\\
&(\sigma_1, \sigma_2, \ldots, \sigma_{n-2}, \sigma_{n-1}, n)
\end{aligned}$$

qui représentent n permutations appartenant à $\mathfrak{S}_n$.
On se convainc assez facilement qu'on obtient ainsi toutes les permutations de $\mathfrak{S}_n$.
24.4 Le code s'en déduit facilement.

```
def permutations(n):
  if n==1:
    return [[1]]
  else:
    S = permutations(n-1)
    Sn = []
    for s in S:
      for i in range(n):
        Sn.append(s[:i]+[n]+s[i:])
    return Sn
```

Il n'est pas très difficile de transformer cet algorithme récursif en algorithme impératif et on se rend facilement compte que le code récursif est plus clair.

II.3 Les Tours de Hanoï

25. Les Tours de Hanoï sont un casse-tête inventé à la fin du XIX^e siècle par le mathématicien Édouard Lucas.

Cliché : `www.lespetitscopeaux.fr`

Il s'agit de déplacer un par un n disques de diamètres différents d'une tour à une autre, tout en conservant leur ordre décroissant (les plus grands disques sous les plus petits).
26. On dispose de trois tours, indexées par 0, 1 et 2.

```
Tour = [ "A", "B", "C" ]
```

26.1 Le jeu consiste à déplacer une pile de n disques de la tour 0 jusqu'à la tour 2.

```
def jouer(n):
    deplacer_pile(n, 0, 2)
```

26.2 Cas de base
Pour déplacer une pile de 0 disque, il n'y a rien à faire.
La fonction récursive `deplacer_pile(h, a, b)` ne déclenche donc un appel récursif que si la hauteur h est au moins égale à 1.

26.3 Appel récursif
Pour déplacer une pile de h disques de la tour a vers la tour b, on procède en trois temps :
– on déplace une pile de $(h-1)$ disques de la tour initiale a vers une tour intermédiaire c ;
– on déplace la pièce restante de la tour a vers la tour b ;
– on déplace la pile de $(h-1)$ disques de la tour intermédiaire c vers la tour finale b.

```
def deplacer_pile(h, a, b):
  c = tour_intermediaire(a, b)
  if h>0:
    deplacer_pile(h-1, a, c)
    deplacer_piece(a, b)
    deplacer_pile(h-1, c, b)
```

26.4 Comme $0+1+2=3$, si les indices a et b désignent deux tours, l'indice de la troisième tour peut se calculer facilement.

```
def tour_intermediaire(a, b):
  return 3-(a+b)
```

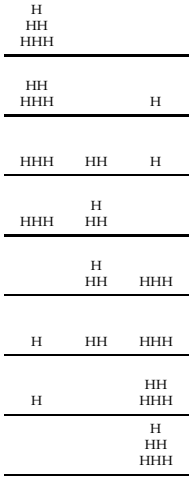
26.5 Pour le déplacement d'une seule pièce, on se contente d'afficher un message.

```
def deplacer_piece(a, b):
  s = "Déplacer de {} vers {}".format(Tour[a], Tour[b])
  print(s)
```

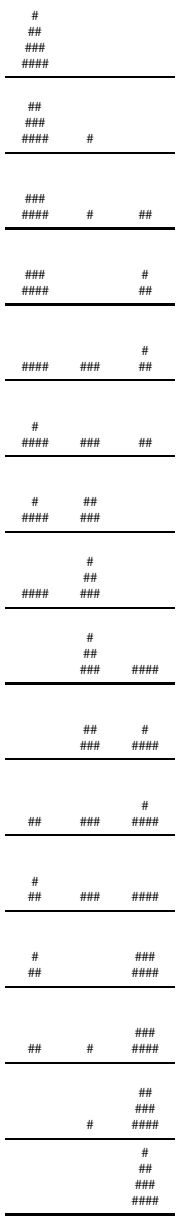
27. Résolution avec $n=3$ disques.

27.1 L'exécution du code précédent donne la liste des mouvements à effectuer.
– Déplacer de la tour A vers la tour C
– Déplacer de la tour A vers la tour B
– Déplacer de la tour C vers la tour B
– Déplacer de la tour A vers la tour C
– Déplacer de la tour B vers la tour A
– Déplacer de la tour B vers la tour C
– Déplacer de la tour A vers la tour C

27.2 Un code sensiblement plus élaboré (fichier `Hanoi.py` disponible en ligne) permet de représenter graphiquement la solution.



28. Résolution avec $n=4$



29. Il est facile de deviner sur ces deux exemples qu'un code impératif serait considérablement plus compliqué à écrire !

III

Diviser pour régner

30. La méthode **Diviser pour régner** (*divide and conquer*), qui met en œuvre des algorithmes récursifs, est à l'origine d'algorithmes particulièrement efficaces (**tri rapide**, **tri fusion**...).

31. Complexités
On note n, le nombre de données sur lesquelles porte un algorithme.
31.1 ▲ *La complexité de l'algorithme est* ***linéaire*** *lorsqu'elle est* $\mathcal{O}(n)$ *ou* ***quasi-linéaire*** *lorsqu'elle est* $\mathcal{O}(n \ln n)$.
31.2 ▲ *La complexité est* ***quadratique*** *lorsqu'elle est* $\mathcal{O}(n^2)$.
31.3 Subdivisons l'ensemble des données : au lieu d'un seul jeu de n données, on dispose de d jeux de n/d données.
Le nombre total d'opérations est alors estimé par

$$d \times \frac{n}{d} = n \quad \text{ou par} \quad d \times \frac{n}{d} \ln \frac{n}{d} \sim n \ln n$$

dans le cas d'une complexité linéaire ou quasi-linéaire : la subdivision des données est alors sans intérêt particulier.
En revanche, dans le cas d'un algorithme quadratique, le nombre total d'opérations est estimé par

$$d \times \left(\frac{n}{d}\right)^2 = \frac{n^2}{d}$$

et on constate ici que la subdivision abaisse la complexité du traitement des données.

32. Principe de la méthode
La méthode **Diviser pour régner** tire partie de la remarque précédente et s'applique en trois temps.
32.1 Divide
Les données de petite taille sont traitées par un algorithme quelconque (correct et fiable). Si cet algorithme n'est pas performant (complexité quadratique ou pire), le fait de l'appliquer sur un petit volume de données n'est pas gênant.
Au-dessus d'un certain *seuil*, les données à traiter sont scindées en sous-ensembles.
32.2 Recur
Chaque sous-ensemble est traité récursivement, de manière à n'appliquer l'algorithme que sur des ensembles de données inférieurs au seuil choisi.
32.3 Conquer
On achève ensuite le traitement en rassemblant de manière cohérente les divers sous-ensembles traités.
32.4 Dans cette méthode, la récursivité permet de coder simplement des algorithmes efficaces sans être directement responsable de l'efficacité de ces algorithmes.

33. Cas des algorithmes de tri
33.1 Les algorithmes de tri simples (tri sélection, tri insertion, tri bulle...) sont tous quadratiques et sont donc des exemples de choix pour appliquer la méthode *diviser pour régner*.
33.2 Il est intéressant à ce sujet de revenir sur les estimations du [**31.3**].
En effet, si on applique un algorithme de tri simple sur une liste dont la taille est inférieure à un certain seuil fixé (ce qui sera le cas pour le tri rapide et pour le tri fusion), l'algorithme de tri s'exécutera à coût constant (complexité temporelle en $\mathcal{O}(1)$).
33.3 La complexité de l'algorithme résultant du principe *Diviser pour régner* dépendra donc seulement
– de l'algorithme de division des données en sous-ensembles de données ;
– du nombre de sous-ensembles ainsi constitués
– et de l'algorithme de synthèse des données triées pour obtenir le résultat final.

III.1 Tri rapide (quicksort)

34. L'algorithme de **tri rapide** est un des plus efficaces connus à ce jour. Il est utilisé par divers langages (Python, Java...). Inventé à la fin des années 1950 par C.A.R. Hoare, il a été perfectionné au fil des années.

35. Principe général
Le tri rapide illustre la méthode Diviser pour régner.
35.1 Divide
Une liste vide ou réduit à un seul élément est déjà triée !
Une liste L de longueur $n \geqslant 2$ est divisée en trois sous-listes en fonction d'une **valeur pivot** p :
– une liste L_- dont les éléments sont strictement inférieurs à la valeur pivot ;
– une liste $L_=$ dont les éléments sont tous égaux à la valeur pivot (il est probable que cette liste soit beaucoup plus petite que les deux autres) ;
– une liste L_+ dont les éléments sont tous strictement supérieurs à la valeur pivot.
35.2 Recur
La liste $L_=$ est triée par construction, puisque tous ses éléments sont égaux.
On applique l'algorithme aux sous-listes L_- et L_+.
35.3 Conquer
Il reste à concaténer les listes triées L_-, $L_=$ et L_+ (dans cet ordre) pour obtenir une version triée de la liste initiale.

36. Mise en œuvre simplifiée
36.1 Le début du code est sans mystère.

```
def trier(T):
  if (len(T)<2):  # Cas de base
    return T
  else:           # Appel récursif
    I, E, S = partition(T)
    return trier(I)+E+trier(S)
```

36.2 On remarquera une première simplification dans ce code : l'appel à la fonction `partition` duplique les données (chaque valeur de la liste T réapparaît dans une des listes I, E ou S).
Non seulement, cette simplification donne un algorithme de complexité spatiale importante (complexité en $\mathcal{O}(n)$), mais elle n'est pas nécessaire puisque cette étape peut être réalisée *en place* (complexité en $\mathcal{O}(1)$), en permutant les valeurs de T (et donc sans créer de liste auxiliaire).
36.3 Nous proposons également une version très simplifiée de la fonction `partition`. Alors que le choix du pivot est crucial pour l'efficacité de l'algorithme, nous faisons au plus simple : le pivot est la première valeur de la liste.

```
def partition(T):
  pivot = T[0]
  I, E, S = [], [], []
  for x in T:
    if (x<pivot):
      I.append(x)
    elif (x>pivot):
      S.append(x)
    else:
      E.append(x)
  return I, E, S
```

Les trois listes auxiliaires sont nommées I (pour *valeurs Inférieures*), E (pour *valeurs Égales*) et S (pour *valeurs Supérieures*).
36.4 Avec ce code très simplifié, la complexité spatiale est élevée [**36.2**] et la complexité temporelle n'est pas maîtrisée : pour des listes d'un certain type, elle peut devenir en $\mathcal{O}(n^2)$ (comparable aux algorithmes de tri les plus banals).
Des codes plus élaborés s'exécutent avec une complexité temporelle proche de la complexité moyenne, c'est-à-dire en $\mathcal{O}(n \ln n)$.

III.2 Tri fusion (merge sort)

37. Le **tri fusion** est une autre application de la méthode *Diviser pour régner* et donne un algorithme de tri dont la complexité spatiale est aussi en $\mathcal{O}(n\,\ell n\,n)$, quoique cet algorithme soit un peu moins rapide que le tri rapide en pratique.

38. Principe général
On distingue deux *cas de base*.
– Une liste vide ou réduite à un seul élément est déjà triée.
– Une liste de deux éléments est triée en une comparaison (et au plus une transposition).

38.1 Divide
Une liste de longueur $n \geqslant 3$ est divisée en deux sous-listes de taille moitié.

$$T_1 = (T_k)_{0\leqslant k<p} \qquad T_2 = (T_k)_{p\leqslant k<n}$$

avec $n = 2p$ ou $n = 2p+1$.

38.2 Recur
On applique le tri fusion aux deux sous-listes T_1 et T_2.

38.3 Conquer
On réunit les sous-listes T_1 et T_2 *triées* en une seule liste *triée*.

Mise en œuvre simplifiée

39. Comme pour le tri rapide, nous allons présenter une mise en œuvre simplifiée de l'algorithme. Ici encore, la simplification est rendue possible par la duplication des données : la complexité spatiale est au service de la simplicité du code.

40. Fonction principale
On peut obtenir un code très proche de la description qu'on vient de donner.

```
def merge_sort(T):
  n = len(T)
  #                  Cas de base : n ⩽ 2
  if n<2:
    return T
  elif n==2:
    if T[0]>T[1]:
      T[1], T[0] = T[0], T[1]
    return T
  else:              # Appels récursifs : n ⩾ 3
    p = n//2
    debut = merge_sort(T[:p])   # Tri de T1
    fin = merge_sort(T[p:])     # Tri de T2
    # Fusion des deux moitiés triées
    return fusion(debut, fin)
```

Lors de la création des listes `debut` et `fin`, les données sont dupliquées.

41. Fonction auxiliaire
La fonction `fusion` prend en argument deux listes triées par ordre croissant et retourne une liste triée par ordre croissant déduite de ces deux listes. Une fois encore, l'écriture récursive va nous donner un code vraiment simple.

41.1 Cas de base
Si l'une des listes est vide, il n'y a rien de particulier à faire : on retourne directement l'autre liste.

41.2 Appel récursif
Si aucune des deux listes n'est vide, on compare leurs minimums respectifs. On extrait la plus petite de ces deux valeurs (qui est le minimum global) à l'aide de `pop` et on fusionne ce qui reste.

41.3 Ici encore, on s'attache à écrire un code aussi proche que possible de la description de l'algorithme, ce qu'on arrive à faire en maniant plusieurs listes.

```
def fusion(T1, T2):
  # Cas de base
  if len(T1)==0:
    return T2
  elif len(T2)==0:
    return T1
  else:  # Appel récursif
    if T1[0]<T2[0]:
      a = T1.pop(0)
    else:
      a = T2.pop(0)
    return [a]+fusion(T1, T2)
```

42. On a obtenu, comme pour le tri rapide, un code simple au prix d'une complexité spatiale élevée. Ceux qu'un tel choix gêne pourront méditer la réflexion suivante.

> *There are two ways of constructing a software design: One way is to make it so simple that there are* obviously *no deficiencies, and the other way is to make it so complicated that there are no* obvious *deficiencies. The first method is far more difficult.*
>
> C.A.R. Hoare

Notre sponsor pour le cours sur la récursivité