Logique et approche mathématique de la programmation — notes provisoires

M. Rigo

2019–2020

Un *problème* possède un caractère générique ; par exemple: trier par ordre croissant les éléments d'une liste $(\ell_1, \ldots, \ell_k)$, calculer la puissance n-ième d'un entier a, estimer avec une précision d un zéro d'un polynôme $P(x)$. Une *instance* d'un problème est la donnée concrète à traiter ; par exemple la liste $(8, 1, 2, 3, 9)$ doit être classée par ordre croissant, calculer 23^{17}, trouver un réel x_0 tel que $|P(x_0)| < 0,0001$ avec $P(x) = 5x^7 - 9x + 2$.

On désire écrire un algorithme (et l'implémenter) qui va répondre au problème considéré. Cet algorithme doit prendre en entrée toute instance (valide) du problème et en fournir la solution. Par exemple, un algorithme de tri doit pouvoir être appliqué à une liste quelconque de longueur arbitraire (pour autant que les éléments de la liste puissent être comparés deux à deux).

Remarque 1. Dans ces quelques pages, on ne s'attachera pas à prouver l'exactitude des algorithmes présentés (même si cela devrait être fait). De même, on ne s'attachera que très peu aux questions de complexité (et donc d'efficacité). Le but premier est de développer une pensée algorithmique.

1 Procédures récursives

Un *procédure récursive* est une procédure (une fonction, un sous-programme) qui peut s'appeler elle-même. Tout comme dans la technique de preuves par récurrence, pour être bien définie, une telle procédure doit posséder un cas de base (ne faisant pas appel à la procédure elle-même) et l'étape d'induction revient alors à exprimer la solution du problème à partir de la solution d'un problème plus simple. Ainsi, les appels successifs à la procédure récursive se font avec des instances "de plus en plus simples" pour, *in fine*, arriver à une instance de base.

On utilise typiquement ce type de procédure lorsqu'on peut *réduire* le problème initial à un problème plus simple et de même nature. Ainsi, pour résoudre un problème P sur une instance I de "taille" n, il suffit souvent de savoir résoudre ce problème sur une instance J de taille strictement inférieure à n. En itérant la procédure, on finit par traiter un cas "trivial" pour lequel la réponse est immédiate et on propage la réponse obtenue jusqu'à la solution pour I.

1.1 Quelques exemples simples

Les fonctions décrites ci-dessous existent parfois de façon native dans le langage de programmation. Notre but est ici d'illustrer la récursion.

Exemple 1. La *factorielle* de n est classiquement définie par $n! = n.(n-1)!$ pour tout $n \geq 1$ et $0! = 1$. Si on note f, la fonction $f : \mathbb{N} \to \mathbb{N}, n \mapsto n!$, on a alors

$$f(n) = \begin{cases} 1 & \text{si } n = 0 \\ n.f(n-1) & \text{sinon.} \end{cases}$$

On est donc bien dans le schéma adéquat: on arrive à exprimer le calcul de $f(n)$ à partir du résultat du calcul de $f(n-1)$. En `Python`, cela se traduit comme suit.

```
# renvoie la factorielle du naturel n
def factorial(n):
    if n==0:
        return(1)
    else:
        return(n*factorial(n-1))
```

Que se passe-t-il lorsqu'on appelle la fonction `factorial(3)` ? Lors de son exécution, la fonction va appeler `factorial(2)` qui, à son tour, va appeler `factorial(1)` puis `factorial(0)`. Dans ce dernier cas, il n'y a plus d'appel récursif à effectuer. La fonction `factorial(0)` renvoie la valeur 1. Une fois cette valeur renvoyée, l'appel à `factorial(1)` peut être terminé par le calcul `1*factorial(0)` et donc renvoyer 1. C'est maintenant l'appel à `factorial(2)` qui se conclut par le calcul `1*factorial(2)` pour renvoyer 2 et ainsi de suite.

Pour visualiser cette suite d'appels, nous pouvons modifier le code `Python` comme suit pour afficher, dans l'ordre, la suite d'opérations réalisées lors de l'exécution:

```
# renvoie la factorielle de l'entier n
def factorial(n):
    print("appel factorial(",n,")")
    if n==0:
        print("factorial(",n,") renvoie 1")
        return(1)
    else:
        calcul=n*factorial(n-1)
        print("factorial(",n,") renvoie ",calcul)
        return(calcul)

factorial(4)
```

Voici la sortie console obtenue (comparez-la avec la Figure 1):

```
appel factorial( 4 )
appel factorial( 3 )
appel factorial( 2 )
appel factorial( 1 )
```

```
appel factorial( 0 )
factorial( 0 ) renvoie  1
factorial( 1 ) renvoie  1
factorial( 2 ) renvoie  2
factorial( 3 ) renvoie  6
factorial( 4 ) renvoie  24
```

Remarque 2. Chaque appel dispose de ses propres variables locales. Ainsi, même si de façon générique, la fonction `factorial()` utilise une variable locale `calcul`, chaque appel crée sa propre "copie" de la variable. Il n'y a donc aucune confusion possible au moment de l'exécution. Par exemple l'appel à `factorial(3)` n'a accès qu'à une unique variable locale `calcul` qui sera affectée avec le résultat de `3*factorial(2)`.

De façon imagée, vous pouvez imaginer, comme pour une pile d'assiettes ou de boîtes (cf. Fig. 1) que l'on empilerait une à une, que les appels successifs à la fonction `factorial()` s'empilent. Une fois tous les appels empilés, on trouve au sommet de la pile le cas de base. Celui-ci étant connu, on peut alors commencer à dépiler les assiettes une à une. Par ailleurs, chaque boîte (donc chaque sous-programme) contient les variables locales correspondant à l'appel. Au sein d'une boîte, on n'a pas accès au contenu des autres boîtes.

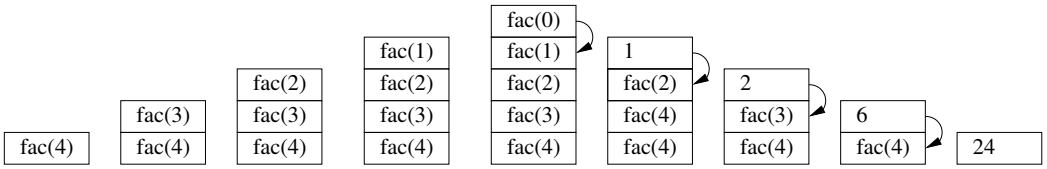

Figure 1: Appels récursifs successifs.

Exemple 2. Le miroir d'un mot (d'une chaîne de caractère) s'obtient en lisant les lettres du mot de droite à gauche. Ainsi, le miroir du mot `Python` est `nohtyP`. Il s'agit d'une fonction définie sur l'ensemble des chaînes de caractères (pour une chaîne `mot`, on l'obtient directement avec `mot[::-1]`). Elle peut être définie, de façon récursive, comme suit.[1] Si m est vide (la chaîne de caractères de longueur 0), alors $f(m)$ est vide. Soient m un mot et a un caractère,

$$f(ma) = a\, f(m).$$

Il est sous-entendu que l'on considère comme opération, la concaténation (en `Python`, la concaténation de deux chaînes est réalisée par l'opérateur `+`). Par exemple, le miroir du mot `Python` est `n` suivi du miroir du mot `Pytho`.

```
# renvoie le miroir d'une chaine
def miroir(mot):
    if len(mot)==0:
        return(mot)
    else:
        return(mot[-1]+miroir(mot[0:-1]))
```

[1] L'idée est d'avoir une définition faisant appel à la fonction elle-même mais sur des instances plus simples. On va donc raisonner sur la longueur du mot et définir le miroir d'un mot de longueur n à partir du miroir d'un mot de longueur $n-1$.

Exemple 3. Etant donnés deux naturels a et n, on veut calculer a^n. Cette fonction est directement disponible avec `a**n`. Nous allons ici fournir trois implémentations. La première n'est pas récursive, les deux autres le sont. La dernière utilise un moins grand nombre de multiplications (et est donc plus efficace).

Par définition, a^n est le produit de n facteurs égaux à a. En partant de 1, il suffit donc de multiplier n fois par a pour obtenir le résultat escompté. Remarquez qu'en procédant de la sorte, on gère aussi le cas d'un exposant nul.

```
# renvoie a exposant n
def exp_iter(a,n):
    temp=1
    # si n==0, on ne rentre pas dans la boucle
    for i in range(0,n):
        temp*=a
    return temp
```

Passons à présent à une première définition récursive basée sur le fait que $a^n = a.a^{n-1}$ si $n \geq 1$. Considérons dès lors la fonction $f : \mathbb{N}^2 \to \mathbb{N}, (a, n) \mapsto a^n$. On a

$$f(a,n) = \begin{cases} 1 & \text{si } n = 0 \\ a.f(a,n-1) & \text{sinon.} \end{cases}$$

Il s'agit bien d'une définition récursive de l'exponentiation codée comme suit. On a ramené le calcul de a^n à celui de a^{n-1}.

```
# renvoie a exposant n
def slow_exp(a,n):
    if n==0:
        return 1
    else:
        return(a*slow_exp(a,n-1))
```

Si comme pour l'exemple de la factorielle, on désire voir le déroulement du programme, on peut réaliser des affichages montrant les appels et les valeurs renvoyées. On a exactement le même comportement que celui décrit à la Fig. 1. En particulier, le calcul a^n requiert n appels à la fonction.

```
# renvoie a exposant n
def slow_exp(a,n):
    print("appel a slow_exp(",a,",",n,")")
    if n==0:
        print("renvoie 1")
        return 1
    else:
        calcul=a*slow_exp(a,n-1)
        print("renvoie",calcul)
        return calcul
```

Considérons la variante suivante (il est évident que $g(a,n) = a^n$). La stratégie consiste ici à tirer parti d'élévations successives au carré.

$$g(a,n) = \begin{cases} 1 & \text{si } n = 0 \\ g(a^2, n/2) & \text{si } n > 0 \text{ est pair} \\ a.g(a^2,(n-1)/2) & \text{si } n \text{ est impair.} \end{cases}$$

```
# renvoie a exposant n
def fast_exp(a,n):
    if n==0:
        return 1
    elif n%2==0:
        return(fast_exp(a**2,n//2))
    else:
        return(a*fast_exp(a**2,(n-1)//2))
```

En utilisant une version modifiée du programme précédent (dans lequel des affichages sont ajoutés), on peut suivre les appels pour `fast_exp(2,13)`

```
appel à fast_exp(2 , 13)
appel à fast_exp(4 , 6)
appel à fast_exp(16 , 3)
appel à fast_exp(256 , 1)
appel à fast_exp(65536 , 0)
fast_exp(65536 , 0) renvoie 1
fast_exp(256 , 1) renvoie 256
fast_exp(16 , 3) renvoie 4096
fast_exp(4 , 6) renvoie 4096
fast_exp(2 , 13) renvoie 8192
```

On peut suivre les appels pour `fast_exp(2,21)`

```
appel à fast_exp(2 , 21)
appel à fast_exp(4 , 10)
appel à fast_exp(16 , 5)
appel à fast_exp(256 , 2)
appel à fast_exp(65536 , 1)
appel à fast_exp(4294967296 , 0)
fast_exp(4294967296 , 0) renvoie 1
fast_exp(65536 , 1) renvoie 65536
fast_exp(256 , 2) renvoie 65536
fast_exp(16 , 5) renvoie 1048576
fast_exp(4 , 10) renvoie 1048576
fast_exp(2 , 21) renvoie 2097152
```

Remarque 3. Puisqu'à chaque étape l'exposant est divisé par 2 (plus exactement, n est remplacé par $\lfloor n/2 \rfloor$), on s'aperçoit que le nombre d'appels est en $\mathcal{O}(\log_2(n))$.

Exemple 4. La suite de Fibonacci $(F_n)_{n\geq 0}$ est définie par récurrence par

$$F_0 = F_1 = 1 \quad \text{et} \quad F_{n+2} = F_{n+1} + F_n, \forall n \geq 0.$$

Une telle définition se prête évidemment très bien pour la définition d'une fonction récursive:

```
# renvoie le n-ieme nombre de Fibonacci
def fibonacci_rec(n):
    print("appel fib(",n,")")
    if n==0:
        return 1
    elif n==1:
        return 1
    else:
        return fibonacci_rec(n-1)+fibonacci_rec(n-2)
```

En termes de performances, il faut cependant être prudent. En effet, chaque appel à la fonction va effectuer deux nouveaux appels (comme on le voit à la dernière ligne du code) et ce, de façon "aveugle" (sans mémoire des appels précédents). Représentons sous forme d'arbre (cf. Fig. 2) les différents appels effectués pour calculer F_4. Chaque appel est symbolisé par une flèche (l'ordre des appels est indiqué à côté de chaque flèche). L'origine de la flèche donne la procédure ayant fait ces appels. Il n'y a plus de nouveaux appels quand on arrive aux cas de base correspondant à F_0 et F_1. Pour qu'une valeur puisse être renvoyée, il faut que tous les "successeurs" aient été évalués.

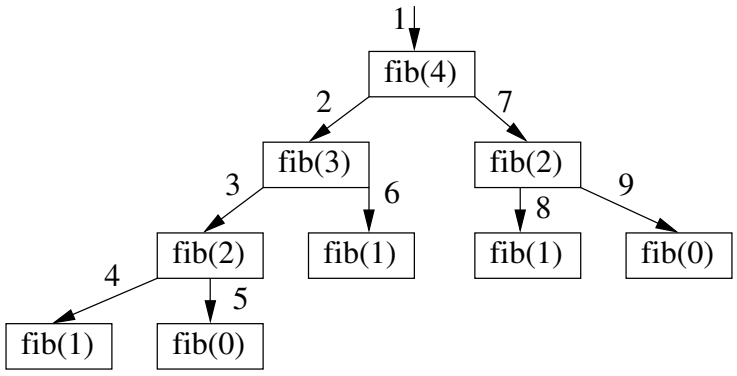

Figure 2: Calcul de F_4.

Comme on le vérifie sur la console, les appels successifs sont:

```
appel à fib( 4 )
appel à fib( 3 )
appel à fib( 2 )
appel à fib( 1 )
appel à fib( 0 )
appel à fib( 1 )
appel à fib( 2 )
appel à fib( 1 )
appel à fib( 0 )
```

Remarque 4. On pourrait en fait démontrer (par récurrence) que le nombre total d'appels à la fonction `fibonacci_rec()` pour calculer F_n est précisément

F_{n+1}. Mais les nombres de Fibonacci ont une croissance exponentielle! A une constante $c > 0$ près, on a

$$F_n \sim c\left(\frac{1+\sqrt{5}}{2}\right)^n$$

c'est-à-dire

$$\lim_{n\to\infty} F_n/((1+\sqrt{5})/2)^n = c.$$

Il ne s'agit donc pas d'une implémentation efficace. Ceci explique dès lors pourquoi l'appel à `fibonacci_rec(40)` prend plus d'une minute sur un processeur Intel i7. Pour mesurer le temps d'exécution, on peut utiliser le module `time`.

```
import time

start = time.time()
print(fibonacci_rec(40))
end = time.time()
print("temps de calcul ", end - start)
```

Si on analyse le problème, on s'aperçoit qu'on calcule à plusieurs reprises les mêmes nombres de Fibonacci. A différents moments, on fait appel à la même fonction mais puisqu'aucune valeur précédemment calculée n'a été stockée, elles sont toutes recalculées à chaque fois que cela s'avère nécessaire. L'idée de la *programmation dynamique* est de conserver en mémoire certaines valeurs précédemment calculées (et ainsi, éviter de les recalculer sans cesse). Il s'agit de trouver un bon compromis entre stockage en mémoire (on ne veut pas stocker trop de valeurs) et appels à la fonction. Pour la suite de Fibonacci, il suffit de conserver les valeurs de F_0 à F_{n-1} pour permettre le calcul de F_n. On va donc stocker n nombres en mémoire au lieu d'appeler une fonction un nombre exponentiel de fois ! Le programme ci-dessous garde en mémoire (dans une liste `fib`) les nombres de Fibonacci jusqu'à avoir atteint F_n.

```
# on utilise une variable globale
fib=[1,1]

# renvoie le n-ieme nombre de Fibonacci
def fibonacci_2(n):
    global fib
    # si la liste est trop courte,
    # calculer un element supplementaire
    if len(fib)<=n:
        fib.append(fib[-1]+fib[-2])
        fibonacci_2(n)
    return fib[n]
```

Remarque 5. A titre de comparaison, l'appel à `fibonacci_rec(40)` prend plus d'une minute sur un processeur Intel i7 et nécessite 331160281 appels récursifs. Alors que `fibonacci_2(40)` s'exécute en moins d'un dix-millième de seconde !

Il y a encore d'autres variantes. On pourrait se passer d'appels récursifs et calculer d'une traite, une liste contenant les n premiers termes de la suite de Fibonacci désirés.

```
# on utilise une variable globale
fib=[1,1]

# renvoie le n-ieme nombre de Fibonacci
def fibonacci_3(n):
    global fib
    if len(fib)<=n:
        for i in range(0,n-len(fib)+1):
            fib.append(fib[-1]+fib[-2])
    return fib[n]
```

Exemple 5. Le "*subset sum problem*": on dispose d'une liste X de nombres naturels et d'un naturel S. On pourrait travailler avec des ensembles, mais ici, on n'interdit pas d'avoir un même élément répété plusieurs fois. La question est de déterminer s'il existe ou non un sous-ensemble de X (plus précisément une sous-liste) dont la somme des éléments vaut S. Ainsi, dans un premier temps, on veut uniquement fournir la réponse Vrai/Faux. Par exemple, l'instance $X = (1, 8, 3, 4, 7, 2, 2)$ et $S = 13$ est positive. Par exemple, $8+3+2 = 13$. Par contre, $X = (1, 8, 3, 4, 7, 2, 2)$ et $S = 100$ est négative.

Remarque 6. Pour une instance (X, S) donnée, on peut toujours répondre à la question. En effet, il suffit de passer en revue les $2^{\#X}$ sous-listes de X, pour chacune d'entre elles calculer la somme des éléments et vérifier si le résultat vaut S. Comme nous allons le voir, passer en revue ces sous-listes peut se faire de façon récursive.

La stratégie (récursive) est de réduire le problème à une instance plus "simple" (donc à une liste de longueur strictement inférieure).

Soient $X = (x_1, \ldots, x_{k-1}, x_k)$ avec $k \geq 1$ et $S \in \mathbb{N}$. Si une solution positive existe soit elle utilise le dernier élément x_k (i.e., la sous-liste recherchée contient x_k) et dans ce cas, on est ramené à trouver une solution pour l'instance formée de $(x_1, \ldots, x_{k-1})$ et $S - x_k$. Soit elle n'utilise pas le dernier élément et dans ce cas, on est ramené à trouver une solution pour l'instance formée de $(x_1, \ldots, x_{k-1})$ et S. Les cas de base à envisager sont

- si $S = 0$, alors la réponse est trivialement positive, prendre la liste vide convient;
- si $S < 0$, alors la réponse est trivialement négative ;
- enfin, si $S > 0$ et $X = \emptyset$, la réponse est négative.

Dans tous les autres cas, on a une liste non vide et une valeur de $S > 0$.

```
# renvoie True/False en fonction
# de l'instance (X,S) du probleme
def subset_sum(X,S):
    print("appel subset_sum(",X,",",S,")")
```

```
# si somme nulle , OK
if S==0:
    return True
# sinon , dans les autres cas triviaux
elif S<0 or X==[]:
    return False
# reste le cas "general"
# on utilise ou non le dernier element
else:
    avec_der=subset_sum(X[:-1],S-X[-1])
    sans_der=subset_sum(X[:-1],S)
    return(avec_der or sans_der)
```

La dernière ligne du code effectue un "ou logique" sur les deux résultats renvoyés à savoir y-a-t-il une solution utilisant le dernier élément de la liste ? puis, y-a-t-il une solution n'utilisant pas le dernier élément de la liste ? Ainsi, la réponse renvoyée sera `False` uniquement lorsque les deux évaluations donnent `False`.

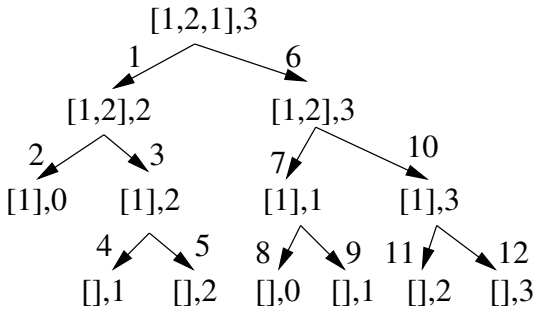

Figure 3: Appels successifs $X = (1, 2, 1)$ et $S = 3$.

La Figure 3 reprend le déroulement du programme à partir de l'instance $X = (1, 2, 1)$ et $S = 3$. On a encore une structure d'arbre car, dans le cas général, il est nécessaire d'effectuer deux appels avec une liste dont la longueur a diminué d'une unité. Dans cette figure, les flèches sont numérotées selon l'ordre des appels. Si on avait permuté les deux dernières instructions et écrit:

```
    sans_der=subset_sum(X[:-1],S)
    avec_der=subset_sum(X[:-1],S-X[-1])
```

la suite d'appels aurait été différente.

Remarque 7. Le programme va tout de même construire l'arbre "complet", c'est-à-dire qu'il va, *in fine*, passer en revue tous les sous-ensembles possibles. En effet, à chaque étape, on demande d'évaluer l'existence d'une solution utilisant ou non le dernier élément de la liste en cours. Si une solution positive existe, il serait intéressant d'arrêter l'exploration. Si, à un moment donné, une instance est évaluée à `True`, il est inutile de poursuivre l'exploration.

On peut dès lors modifier le code comme suit.

```
# renvoie True/False en fonction
# de l'instance (X,S) du probleme
def subset_sum(X,S):
    print("appel subset_sum(",X,",",S,")")
    # si somme nulle, OK
    if S==0:
        return True
    # sinon, dans les autres cas triviaux
    elif S<0 or X==[]:
        return False
    # reste le cas "general"
    # on utilise ou non le dernier element
    else:
        avec_der=subset_sum(X[:-1],S-X[-1])
        # deux lignes de code supplementaire
        if avec_der:
            return True
        sans_der=subset_sum(X[:-1],S)
        return(avec_der or sans_der)
```

L'ajout des deux lignes de code permet de limiter l'exploration. En effet, si une solution est trouvée en utilisant le dernier élément, on renvoie directement `True` et on quitte donc la fonction. La suite du code n'étant plus à évaluer, on ne cherchera pas une éventuelle solution n'utilisant pas le dernier élément. De façon imagée, on coupe la branche droite de l'arbre. En fonction de l'instance considérée, il peut tout de même arriver que l'on doive faire appel à la fonction un nombre exponentiel de fois par rapport à la taille de la liste (e.g., si S est supérieur à la somme des éléments de X, puisque l'instance est négative, on devra parcourir l'arbre dans son entièreté).

Avec cette version modifiée, on pourrait maintenant vouloir afficher la (première) solution trouvée. On applique la même stratégie que ci-dessus. Si une solution est trouvée, inutile d'explorer les autres pistes (un `return` permet de sortir de la procédure sans poursuivre l'évaluation des autres lignes de code). Pour rappel, la méthode `append` modifie une liste mais ne renvoie rien.

```
# renvoie la premiere solution positive
# de l'instance (X,S) du probleme
# renvoie ''impossible'' sinon.
def find_subset_sum(X,S):
    if S==0:
        return([])
    elif S<0 or X==[]:
        return("impossible")
    temp=find_subset_sum(X[:-1],S-X[-1])
    if temp!="impossible":
        temp.append(X[-1])
        return(temp)
    temp=find_subset_sum(X[:-1],S)
    if temp!="impossible":
```

```
            return(temp)
        return("impossible")
```

Si on utilise `find_subset_sum([1,3,2,3],7)`, les appels successifs sont

```
[1, 3, 2, 3] 7
[1, 3, 2] 4
[1, 3] 2
[1] -1
[1] 2
[] 1
[] 2
[1, 3] 4
[1] 1
[] 0
```

et le résultat renvoyé est `[1, 3, 3]`.

1.2 Générer n-uples et permutations

Nous voudrions générer tous les n-uples formés d'éléments de $\{1,\dots,k\}$. Il y a donc k^n éléments à engendrer. Si n était fixé, il suffirait d'imbriquer n boucles `for` (ce qui pourrait être inélégant et fastidieux pour de grandes valeurs de n). Ici, n est un paramètre du programme. Il n'est donc pas possible de prédire sa valeur et donc de rédiger le programme en imbriquant un nombre "variable" de boucles.

Nous pouvons une fois encore nous en sortir en raisonnant de manière récursive. En effet, pour générer un n-uple, il suffit de choisir le premier élément puis de le faire suivre de tous les $(n-1)$-uples possibles. Et donc, à une étape quelconque, si on a déjà généré un préfixe, on le complète par un élément quelconque de l'ensemble pour obtenir un préfixe plus long. On continue jusqu'à obtenir une suite de longueur n. On a donc l'implémentation suivante.

```
# genere les n-uples de {1,...,k}
# avec un prefixe donne

def genere(pref,n,k):
    if len(pref)==n:
        print(pref)
    else:
        for i in range(1,k+1):
            genere(pref+[i],n,k)

genere([],3,2)
```

Ainsi, partant de la suite vide, `genere([],3,2)` va engendrer les 8 triplets formés de 1 et 2.

Il n'y a pas grand chose à modifier pour produire toutes les permutations de $\{1,\dots,n\}$. Si on dispose déjà d'un préfixe formé d'éléments deux à deux distincts, il suffit de compléter ce préfixe par un élément de $\{1,\dots,n\}$ non encore rencontré. Pour ce faire, on peut utiliser la différence ensembliste.

```
# genere les permutations de {1,...,n}
# ayant un prefixe donne

def perm(pref,n):
    if len(pref)==n:
        print(pref)
    else:
        for i in set(range(1,n+1)).difference(set(pref)):
            perm(pref+[i],n)
```

1.3 Quelques procédures classiques de tri

Plusieurs algorithmes de tris sont construits récursivement.

Le Quick sort ou tri rapide.

- Partitionnement : Placer un élément de la liste (appelé *pivot*) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui lui sont supérieurs, soient à sa droite. Autrement dit, si p est le pivot, après partitionnement, la liste doit être de la forme

  $$(x_1, \ldots, x_j, p, x_{j+1}, \ldots, x_n)$$

  avec $x_i \leq p$ si $i \leq j$ et $x_i \geq p$ si $i > j$.
- Pour chacune des deux sous-listes $(x_1, \ldots, x_j)$ et $(x_{j+1}, \ldots, x_n)$, on définit un nouveau pivot et on répète le partitionnement.
- Répéter *récursivement*, jusqu'à ce que l'ensemble des éléments soit trié.

Le pivot étant un élément arbitraire de la liste, nous pouvons par simplicité, considérer qu'il s'agit du dernier. La fonction `partition` travaille sur une portion de liste comprise entre 2 bornes. Un exemple permettra de mieux comprendre ce qui est attendu.

```
l=[8,9,2,3,4,7,1,7,5]
print(partition(l,0,5))
print(l)
```

La fonction `partition(l,0,5)` renvoie `3` et la liste vaut à présent

```
[2, 3, 4, 7, 8, 9, 1, 7, 5]
```

En effet, ayant passé les paramètres 0 et 5, on ne travaille que sur les 6 premiers éléments de la liste. Le sixième élément valant 7, c'est lui qui est pris comme pivot.

$$\underbrace{8, 9, 2, 3, 4, \mathbf{7}}, 1, 7, 5.$$

Les nombres $2, 3, 4$ sont inférieurs à 7. Par contre, $8, 9$ sont supérieurs. On doit donc obtenir une nouvelle liste où 7 occupera la quatrième position (donc l'indice 3 puisqu'on commence à compter à 0). C'est pour cette raison que 3

est renvoyé. Voici une implémentation. On passe en revue les éléments de la liste grâce à la variable `j`. La variable `i` stocke la position du dernier élément inférieur ou égal au pivot rencontré. Quand un nouvel élément, en position `j`, inférieur ou égal au pivot est rencontré, on permute les contenus de `liste[j]` et de `liste[i+1]`. En particulier, la variable `i` doit être incrémentée d'une unité (puisqu'on vient de rencontrer un nouvel élément $\leq$ pivot). On peut dès lors d'abord incrémenter `i` puis permuter les contenus de `liste[j]` et de `liste[i]`. Quand on a passé en revue tous les éléments, il reste à bien positionner le pivot et à renvoyer son indice.

```
# on suppose que le pivot est l'element liste[sup]
# modifie la liste dans
# l'intervalle liste[inf],...,liste[sup]
# renvoie la position du pivot
def partition(liste, inf, sup):
    # indice du dernier elt. <= pivot
    i = inf-1
    pivot = liste[sup]
    for j in range(inf, sup):
        if liste[j] <= pivot:
            i = i+1
            liste[i],liste[j] = liste[j],liste[i]
    liste[i+1], liste[sup] = liste[sup], liste[i+1]
    return (i+1)
```

Si on l'applique cette fois à la liste dans son entièreté

```
l=[8,9,2,3,4,7,1,7,5]
print(partition(l,0,8))
print(l)
```

Le pivot est 5, la liste modifiée est

```
[2, 3, 4, 1, 5, 7, 9, 7, 8]
```

et la valeur renvoyée est 4 (indice où se trouve le pivot 5). La suite est plus simple. Il reste à implémenter le tri proprement dit.

```
def quicksort(liste,inf,sup):
    if inf<sup:
        pivot = partition(liste,inf,sup)
        quicksort(liste, inf, pivot-1)
        quicksort(liste, pivot+1, sup)
```

Le test est important car il permet de gérer le cas où l'une des deux sous-listes restant à trier est vide ou réduite à un élément. Comme d'habitude, regardons le déroulement d'une exécution.

```
def quicksort(liste,inf,sup):
    if inf<sup:
        pi = partition(liste,inf,sup)
        print("position pivot:",pi)
```

```
        print("appel avec",liste, inf, pi-1)
        quicksort(liste, inf, pi-1)
        print("appel avec",liste, pi+1, sup)
        quicksort(liste, pi+1, sup)

l=[8,9,2,3,4,7,1,7,5]
quicksort(l,0,8)
print(l)
```

```
position pivot: 4
appel avec [2, 3, 4, 1, 5, 7, 9, 7, 8] 0 3
position pivot: 0
appel avec [1, 3, 4, 2, 5, 7, 9, 7, 8] 0 -1
appel avec [1, 3, 4, 2, 5, 7, 9, 7, 8] 1 3
position pivot: 1
appel avec [1, 2, 4, 3, 5, 7, 9, 7, 8] 1 0
appel avec [1, 2, 4, 3, 5, 7, 9, 7, 8] 2 3
position pivot: 2
appel avec [1, 2, 3, 4, 5, 7, 9, 7, 8] 2 1
appel avec [1, 2, 3, 4, 5, 7, 9, 7, 8] 3 3
appel avec [1, 2, 3, 4, 5, 7, 9, 7, 8] 5 8
position pivot: 7
appel avec [1, 2, 3, 4, 5, 7, 7, 8, 9] 5 6
position pivot: 6
appel avec [1, 2, 3, 4, 5, 7, 7, 8, 9] 5 5
appel avec [1, 2, 3, 4, 5, 7, 7, 8, 9] 7 6
appel avec [1, 2, 3, 4, 5, 7, 7, 8, 9] 8 8
```

Le Merge sort ou tri fusion.

- Si la liste n'a qu'un élément, elle est triée.
- Sinon, séparer la liste en deux sous-listes de "même" longueur (à une unité près, si la longueur est impaire) ; `G` contient la première moitié des éléments et `D` la seconde moitié.
- Trier *récursivement* les deux parties.
- Fusionner les deux listes triées en une seule liste triée. L'idée (cf. Fig. 4) est d'avoir deux curseurs pour parcourir linéairement les deux sous-listes. A chaque fois, on prend le plus petit des deux éléments comparés et marqués respectivement dans `G` et `D` par un curseur (pour construire la liste triée) et on fait avancer le curseur correspondant.

```
def mergesort(liste):
    if len(liste) >1:
        milieu = len(liste)//2
        G = liste[:milieu]    # on travaille sur des copies
        D = liste[milieu:]    # on a coupe liste en 2
        mergesort(G)
```

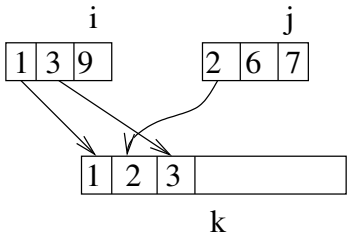

Figure 4: Merge sort.

```
mergesort(D)

# proceder a la fusion de G et D qui sont triees
i = j = k = 0
while i < len(G) and j < len(D):
    if G[i] < D[j]:
        liste[k] = G[i]
        i+=1
    else:
        liste[k] = D[j]
        j+=1
    k+=1

# Quand on sort de la boucle, on n'a pas
# necessairement parcouru tout G ou D,
while i < len(G):
    liste[k] = G[i]
    i+=1
    k+=1
while j < len(D):
    liste[k] = D[j]
    j+=1
    k+=1
```

2 Graphics.py

Extrait de la documentation : "The package graphics.py is a simple object oriented graphics library designed to make it very easy for novice programmers to experiment with computer graphics in an object oriented fashion. It was written by John Zelle for use with the book "Python Programming: An Introduction to Computer Science" (Franklin, Beedle & Associates). The most recent version of the library can obtained at `http://mcsp.wartburg.edu/zelle/python`"

Si votre implémentation ne dispose pas de cette librairie, le plus simple est de la copier dans le même répertoire que le fichier source qui doit l'utiliser. De cette façon, vous aurez accès aux fonctionnalités *ad hoc*. La documentation fait moins de 10 pages. Nous ne présentons ici que quelques fonctionnalités.

Pour ouvrir une fenêtre, par exemple de 1000 pixels sur 800, on peut procéder

comme suit. On travaille typiquement avec les coordonnées du coin inférieur gauche puis du coin supérieur droit.

```
from graphics import *

fenetre = GraphWin(width = 1000, height = 800)
fenetre.setCoords(0, 0, 1000, 800)
```

La libraire dispose du type `Point`. On peut définir ses coordonnées, sa couleur et on dispose d'une méthode pour afficher le point dans une fenêtre définie précédemment. Pour rappel, la couleur d'un pixel est composé de trois composantes primaires rouge, verte et bleue prenant des valeurs comprises entre 0 et 255. La couleur noire correspond à $(0, 0, 0)$ et le blanc à $(255, 255, 255)$. On arrive de cette façon à un total de 2^{24} nuances (plus de 16 millions de couleurs). Les lignes de code suivantes définissent un pixel rouge et l'affichent dans la fenêtre.

```
pt = Point(100,50)
pt.setOutline(color_rgb(255,0,0))
pt.draw(fenetre)
```

Remarquer l'emploi (obligatoire) des majuscules prescrit par la librairie. Pour deux points donnés, on peut définir un segment par ses extrémités et ensuite l'afficher dans une fenêtre. On affiche par exemple un segment vert.

```
line = Line(Point(100,50), Point(200,129))
line.setOutline(color_rgb(0,255,0))
line.draw(fenetre)
```

Un rectangle est défini par son coin inférieur gauche et son coin supérieur droit. On définit ici une couleur de remplissage (puisqu'il s'agit d'une surface et pas d'un trait). On pourrait aussi définir la couleur de sa frontière.

```
un_rectangle = Rectangle(Point(10, 20), Point(92,123))
un_rectangle.setFill(color_rgb(0,155,255))
un_rectangle.draw(fenetre)
```

De la même façon, un disque est défini par son centre (un point) et un rayon.

```
disque = Circle(Point(50,50),10)
disque.setFill(color_rgb(0,155,255))
disque.draw(fenetre)
```

Dans une fenêtre précédemment définie, on peut attendre un clic de l'utilisateur (en fait la fonction renvoie l'endroit cliqué):

```
fenetre.getMouse()
```

3 Le backtracking

Illustrons cette technique avec le problème des n reines : peut-on placer sur un échiquier $n \times n$, n reines de telle façon qu'aucune d'entre elles ne puisse capturer une autre reine ? Pour rappel, selon les règles classiques des échecs, une reine se déplace d'un nombre arbitraire de cases horizontalement, verticalement ou suivant l'une des deux diagonales issues de la case où elle se trouve.

Convenons que k reines placées dans les $k \leq n$ premières colonnes de l'échiquier sont codées par une liste de longueur k donnant les lignes occupées par chacune d'elles. On pourrait pour résoudre le problème des n reines, passer en revue les n^n listes de longueur n et pour chacune tester si la configuration obtenue est valide. Avec 8^8, on est déjà à plus de 16 millions de configurations à tester.

Cette méthode est trop brutale. En effet, certaines configurations ne méritent pas d'être explorées. Par exemple, celles débutant par $[1, 1]$ ou $[1, 2]$ ne donneront aucune solution valide. Il est donc inutile d'essayer de les compléter. L'idée du backtracking est de partir d'une configuration valide (disons constituée des k premières positions) et d'essayer de l'étendre avec une reine supplémentaire. Si plusieurs options sont disponibles, on essaie la première d'entre elles (on suppose qu'on peut ordonner ces options) et on conserve ce choix le plus longtemps possible. Si à une étape donnée, on arrive dans une impasse, on "remonte" à la dernière étape à laquelle un choix a été effectué et on essaie alors le choix suivant.

Voici la définition que l'on trouve sur `Wikipedia`. Le *retour sur trace* (appelé aussi *backtracking* en anglais) est une famille d'algorithmes pour résoudre des problèmes algorithmiques, notamment de satisfaction de contraintes (optimisation ou décision). Ces algorithmes permettent de tester systématiquement l'ensemble des affectations potentielles du problème. Ils consistent à sélectionner une variable du problème, et pour chaque affectation possible de cette variable, à tester récursivement si une solution valide peut-être construite à partir de cette affectation partielle. Si aucune solution n'est trouvée, la méthode abandonne et revient sur les affectations qui auraient été faites précédemment (d'où le nom de retour sur trace).

Dans la Figure 5, on considère le problème des 4 reines. Pour chaque configuration partielle, on indique les extensions valides possibles. Un point rouge montre une voie sans issue: aucune extension n'est valide depuis cette configuration. Autrement dit, on a testé toutes les extensions possibles, comme `[1,3,1]`, `[1,3,2]`, `[1,3,3]`, `[1,3,4]`.

Si on désire passer en revue toutes les solutions possibles, on peut procéder comme suit. Partant de la configuration `[]`, on emprunte la branche la plus à gauche : on descend d'un niveau avec `[1]`, puis on étend cette configuration par `[1,1]`, puis `[1,2]`, pour finalement obtenir une configuration valide `[1,3]`. On essaie encore une fois d'étendre cette configuration. Si cela n'est pas possible, on remonte dans l'arbre jusqu'au dernier embranchement pour lequel une autre option est encore disponible (inexplorée). Ainsi, on remonte en `[1]` pour ensuite explorer `[1,4]`.

On ne détaillera pas ici la fonction `ajoutValide()` qui teste si une nouvelle reine occupe une position valide (la fonction renvoie un booléen).

```
# variable globale
```

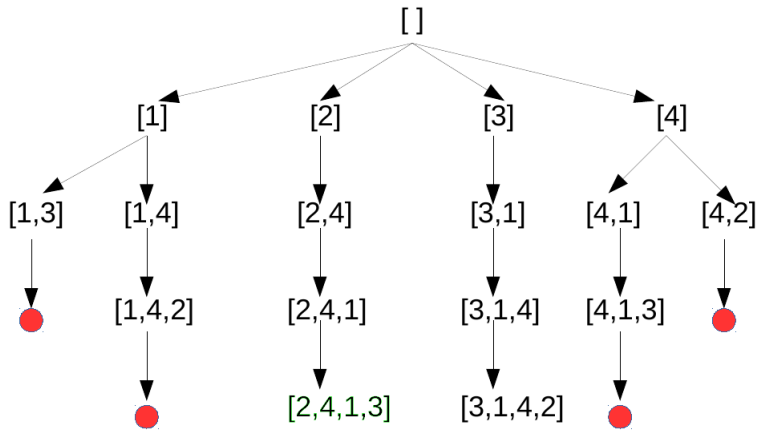

Figure 5: Exploration des configurations.

```
n=8

# partant d'une liste de longueur k,
# renvoie True si on peut completer celle-ci
# par une reine en position new dans la colonne k+1

def ajoutValide(liste,new):
    sortie=True
    for i in range(0,len(liste)):
        if liste[i]==new or liste[i]==new-len(liste)+i
                         or liste[i]==new+len(liste)-i:
            sortie=False
    return sortie

# partant d'une liste, ajouter une reine
# de toutes les facons possibles

def nouvelleReine(liste):
    # si on a une liste de longueur n, afficher la solution
    if len(liste)==n:
        print(liste)
    else:
        # sinon, on essaie d'etendre
        for j in range(1,n+1):
            if ajoutValide(liste,j):
                nouvelleReine(liste+[j]) # recursif

nouvelleReine([])
```

Voici l'évolution de la liste transmise à chaque appel à `nouvelleReine`. Elle illustre parfaitement le parcours de l'arbre de la Figure 5.

```
[]
[1]
[1, 3]
[1, 4]
[1, 4, 2]
[2]
[2, 4]
[2, 4, 1]
[2, 4, 1, 3]
[3]
[3, 1]
[3, 1, 4]
[3, 1, 4, 2]
[4]
[4, 1]
[4, 1, 3]
[4, 2]
```

Chaque nouvel appel à `nouvelleReine()` se fait à partir d'une configuration valide puisqu'un test a été réalisé en amont. On ne voit donc pas apparaître ici l'ensemble des pistes explorées. Pour véritablement afficher l'ensemble des tests réalisés, on peut ajouter un `print()` au niveau de la boucle `for` (on aurait pu aussi l'intégrer à la fonction `ajoutValide()`).

```
...
for j in range(1,n+1):
            print(liste+[j])
            if ajoutValide(liste,j):
                nouvelleReine(liste+[j]) # recursif
```

```
[1]
[1, 1]
[1, 2]
[1, 3]
[1, 3, 1]
[1, 3, 2]
[1, 3, 3]
[1, 3, 4]
[1, 4]
[1, 4, 1]
[1, 4, 2]
[1, 4, 2, 1]
[1, 4, 2, 2]
[1, 4, 2, 3]
[1, 4, 2, 4]
[1, 4, 3]
[1, 4, 4]
[2]
[2, 1]
[2, 2]
[2, 3]
[2, 4]
```

```
[2, 4, 1]
[2, 4, 1, 1]
[2, 4, 1, 2]
[2, 4, 1, 3]
[2, 4, 1, 4]
[2, 4, 2]
[2, 4, 3]
[2, 4, 4]
[3]
[3, 1]
[3, 1, 1]
[3, 1, 2]
[3, 1, 3]
[3, 1, 4]
[3, 1, 4, 1]
[3, 1, 4, 2]
[3, 1, 4, 3]
[3, 1, 4, 4]
[3, 2]
[3, 3]
[3, 4]
[4]
[4, 1]
[4, 1, 1]
[4, 1, 2]
[4, 1, 3]
[4, 1, 3, 1]
[4, 1, 3, 2]
[4, 1, 3, 3]
[4, 1, 3, 4]
[4, 1, 4]
[4, 2]
[4, 2, 1]
[4, 2, 2]
[4, 2, 3]
[4, 2, 4]
[4, 3]
[4, 4]
```

Plutôt que d'explorer l'arbre à la recherche de toutes les solutions, on pourrait se contenter de la première solution trouvée. On peut procéder de la sorte en définissant d'abord une fonction auxiliaire qui donne la configuration suivante à tester. Par exemple, après s'être rendu compte que [1,3] menait à une impasse, on doit passer à [1,4]. Dans le problème des n reines, si le dernier élément de la liste vaut n, la configuration suivante est donnée par la configuration qui suit la liste privée de sa dernière colonne.

```
# [...,i] -> [...,i+1] si i<n
# [X,n] -> le successeur de [X]
# par exemple, [1,3,n] -> [1,4]
def suivant(liste):
    if liste==[]:
        return [1]
```

```
    elif liste[-1]<n:
        return liste[:-1]+[liste[-1]+1]
    else:
        return suivant(liste[:-1])
```

Remarque 8. Pour tester si une configuration `conf` $= (x_1, \ldots, x_k, x_{k+1})$ est valide, sachant que la configuration $(x_1, \ldots, x_k)$ l'était, on peut tirer profit de la fonction `ajoutValide()` définie précédemment. En effet, il suffit d'appliquer

```
ajoutValide(conf[:-1],conf[-1])
```

qui teste si x_{k+1} peut étendre de façon admissible la configuration $(x_1, \ldots, x_k)$.

On peut à présent passer à la recherche d'une solution. Tant qu'on ne dispose pas d'une configuration de longueur n, on poursuit la recherche. Quand on a une configuration de la bonne longueur, il faut encore tester si cette dernière est valide (d'où la présence de la seconde condition).

```
def premiereSolution():
    conf=[1]
    while len(conf)<n or
                not ajoutValide(conf[:-1],conf[-1]):
        # si on a une confuration valide,
        # alors on descend d'un niveau dans l'arbre
        # et on ajoute une composante
        if ajoutValide(conf[:-1],conf[-1]):
            conf=conf+[1]
        else:
            conf=suivant(conf)
    return(conf)

print(premiereSolution())
```

Avec $n = 5$, voici l'évolution de la variable locale `conf`.

```
[1]
[1, 1]
[1, 2]
[1, 3]
[1, 3, 1]
[1, 3, 2]
[1, 3, 3]
[1, 3, 4]
[1, 3, 5]
[1, 3, 5, 1]
[1, 3, 5, 2]
[1, 3, 5, 2, 1]
[1, 3, 5, 2, 2]
[1, 3, 5, 2, 3]
[1, 3, 5, 2, 4]
```