Récursivité

Laurent Signac – CC-BY-SA – 26-09-22 1358 d4649e2cf8b8ce940c17

> J'imagine qu'il est tentant, si le seul outil dont vous disposez est un marteau, de tout considérer comme un clou. – *Abraham Maslow*

La récursivité est le principe qui permet de définir quelque chose en utilisant dans la définition l'objet qu'on est en train de définir. Il existe des structures informatiques récursives (les piles, les arbres…) et des fonctions (ou procédures) récursives. Nous parlerons ici des fonctions et procédures récursives, qui sont donc des fonctions, ou procédures, qui s'appellent elles-mêmes.

De nombreux problèmes informatiques se prêtent à un traitement récursif, donnant ainsi lieu à des solutions souvent très élégantes. Toutefois, tout programme récursif peut être écrit de manière itérative. On choisira donc l'approche qui convient le mieux au problème posé. En particulier si le problème a déjà une structure récursive ou si le problème peut être décomposé en sous-problèmes plus petits de même nature, on choisira généralement une approche récursive.

Dans ce qui suit, l'objectif est de proposer des solutions récursives même si, dans certains cas, une résolution itérative aurait pu être préférable (on le précisera dans ce cas).

1 Fonction puissance

On désire implémenter une fonction qui prend x (flottant) et n (entier positif) en paramètres et renvoie x^n.

Les propriétés suivantes (avec p entier) vont nous permettre de trouver une écriture récursive de la fonction.

- $x^{2p} = (x \times x)^p$
- $x^{2p+1} = x \times x^{2p}$

Dans ces deux cas, qui recouvrent tous les cas possibles, on transforme l'élévation à la puissance n par une élévation à une puissance plus petite. On a donc décomposé le problème en problèmes plus petits de même nature.

Les propriétés précédentes suffisent à définir correctement les puissances entières à condition qu'on n'omette pas d'ajouter :

- $x^0 = 1$

De ce qui précède, on déduit la fonction suivante (ici en pseudo-code, vous êtes invités à l'écrire en Python et à tester) :

```
fonction puissance(x: flottant, n: entier) -> flottant
    si n est égal à 0
        retourner 1
    si n est pair
        retourner puissance(x * x, n / 2)
    sinon
        retourner x * puissance(x, n - 1)
```

Notons dès à présent les éléments qu'on retrouve dans une fonction récursive :

- l'objet de taille n est défini à partir d'un objet *de taille plus petite*
- les *petits objets* ne sont pas définis de manière récursive

De manière plus générale, on doit s'assurer que les différents appels récursifs finiront par un appel qui ne le sera pas (sans quoi on parle de récursivité infinie). Très souvent, lors de ces appels, un paramètre diminue, mais ce n'est pas toujours le cas.

La preuve de la fonction qui précède consiste à remarquer que dans la définition, les appels sont corrects (`x * x`, `n / 2`, `n - 1` sont bien des paramètres corrects pour la fonction et la valeur de retour est bien toujours

un entier), que la fonction se termine, et que dans ce cas, le résultat est correct. En effet, l'exposant diminue nécessairement, de 1 s'il est impair ou de moitié s'il est pair. On atteint donc **nécessairement** la valeur 0 (ce n'est pas si évident, on pourrait avoir n qui diminue, mais passe de 1 à -1, par exemple…). La fin de la preuve consiste à réutiliser les propriété mathématiques utilisées pour concevoir la fonction. On suppose que le résultat est correct jusqu'à un certain n (il l'est pour $n = 0$), et on montre qu'il l'est alors pour $n+1$, quelle que soit la parité de n.

2 Suite de Fibonacci

La suite de Fibonacci est définie ainsi :

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \, si \, n > 1 \end{cases}$$

La fonction qui calcule F_n en fonction de n est très facile à programmer récursivement. Voici le pseudo-code et le code Python :

```
fonction fibo(n: entier) -> entier
    si n vaut 0, retourner 0
    si n vaut 1, retourner 1
    retourner fibo(n - 1) + fibo(n - 2)
```

```python
def fibo(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibo(n - 1) + fibo(n - 2)
```

Ici aussi, la fonction est correctement définie (on ajoute 2 entiers qu'on renvoie, et `n - 1` comme `n - 2` sont des entiers), et se termine (`n` finit par être inférieur ou égal à 1). Dans le cas où le calcul se termine, on montre par récurrence que le calcul est correct en utilisant directement la définition de la suite.

Mais attention aux performances. Les deux appels récursifs rendent l'exécution impraticable pour des valeurs de n de quelques dizaines. Les arbres des appels pour `fibo(5)` et `fibo(7)` sont représentés fig. 1 et fig. 2. Utiliser une solution récursive, quoi qu'élégante, est ici inefficace.

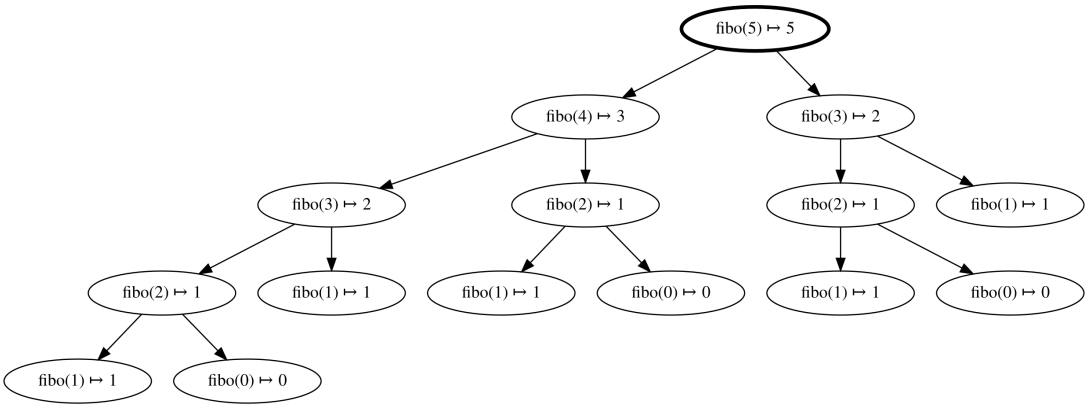

Figure 1: Arbre des appels de `fibo(5)`

3 Conception et preuve d'un algorithme récursif

Forts des deux exemples précédents, nous pouvons en extraire une méthode de conception d'un algorithme récursif, qui contient la plupart du temps les trois étapes suivantes :

1. **Spécification précise de la fonction/procédure** à écrire et choix parfois non évident (voir plus loin les tours de Hanoï) de ses paramètres. Ces derniers contiennent presque toujours un élément qui indique la *taille* du problème

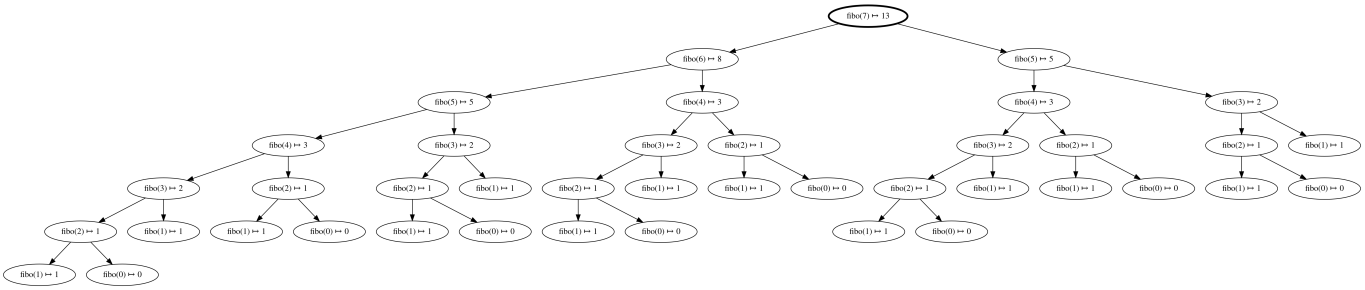

Figure 2: Arbre des appels de `fibo(7)`

2. **Résolution des certains cas simples**, qui correspondront aux conditions d'arrêt de la récursivité (ces cas sont souvent ceux qui correspondent aux problèmes de petite taille).
3. **Décomposition du cas général**, en faisant appel à une ou plusieurs instances de plus petite taille (le plus souvent) du même problème.

La preuve d'un algorithme récursif consistera généralement à :

1. Vérifier que la fonction/procédure est **bien définie** (en particulier que les types de paramètres et de retours sont cohérents)
2. Prouver la terminaison du calcul
3. Prouver qu'en cas de terminaison, le résultat est correct

4 Exemples

4.1 Fonctions mystérieuses

Voici quelques fonctions récursives. On peut vérifier qu'elles sont bien définies, et prévoir le résultat de leur exécution.

4.1.1 `myst1(n)`

Voici une première fonction, donnée en pseudo-code :

```
fonction myst1(n: entier) -> chaîne de caractères
  si n > 0:
    renvoyer la concaténation de n, myst1(n-1) et n
  sinon
    renvoyer une chaîne vide
```

Et voici l'équivalent en Python (notez l'ajout de l'appel à `str` pour lever l'ambiguïté sur la concaténation) :

```python
def myst1(n: int) -> str:
  if n > 0:
      return str(n) + myst1(n - 1) + str(n)
  else:
      return ""
```

Que vaut `myst1(3)` (vous devez trouver sans tester et exécuter la fonction bien sûr) ?

Commençons par vérifier qu'elle est bien définie. Les annotations nous indiquent que le paramètre est entier et que la fonction renvoie une chaîne :

- l'appel à `myst1(n - 1)` est correct car si `n` est entier, `n - 1` l'est aussi.
- selon le résultat du test, la fonction renvoie une chaîne vide (qui est bien une chaîne), ou bien la concaténation (`+`) des trois élément suivants, qui est encore une chaîne :
  - `str(n)` qui est bien une chaîne,
  - `myst(n - 1)` qui en est une par hypothèse,

  - et `str(n)`
- le paramètre `n` entier diminue dans le cas récursif (il passe à `n - 1`), et deviendra donc forcément inférieur ou égal à 0, qui correspond à l'appel non récursif.

Toutes ces considérations permettent de conclure que la fonction est correctement définie et s'arrête.

⚙ Pour déterminer ce que vaut `myst1(3)`, on peut dresser un tableau des appels. Sur la ligne `n=0`, on inscrit ce que renvoie `myst1(0)`. Sur la ligne `n=1`, on inscrit ce que renvoie `myst1(1)`, qui est déduit de la ligne précédente etc.

n	myst1(n)
0	
1	
2	
3	

4.1.2 `myst2(n)`

Voici une seconde fonction, donnée en pseudo-code. Que vaut `myst2(4)` ?

```
fonction myst2(n: entier) -> chaîne de caractères
  si n > 0:
    renvoyer la concaténation de myst2(n-1), n et myst2(n-1)
  sinon
    renvoyer une chaîne vide
```

📝 Les mêmes arguments et le même type de tableau permettent de conclure sur ce que renvoie `myst2(4)`. À vous de remplir le tableau.

n	myst2(n)
0	
1	
2	
3	
4	

4.2 Tours de Hanoï

> Le mandarin N. Claus (de Siam),[1] nous raconte qu'il a vu, dans ses voyages pour la publication des écrits de l'illustre Fer-Fer-Tam-Tam, dans le grand temple de Bénarès, au-dessous du dôme qui marque le centre du monde, trois aiguilles de diamant, plantées dans une dalle d'airain, hautes d'une coudée et grosses comme le corps d'une abeille. Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet ; c'est la tour sacrée de Brahma. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille de diamant sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes !

Le jeu des tours de Hanoï comporte trois piquets surmontés de n disques (voir fig. 3). L'objectif est de passer les disques du piquet de départ (le A sur l'image) au piquet d'arrivée (le B) en suivant les règles :

- on ne peut déplacer qu'un disque à la fois
- on ne peut saisir qu'un disque situé en sommet de pile
- on ne doit jamais placer un disque sur un disque plus petit

[1] du collège Li-Sou-Stian

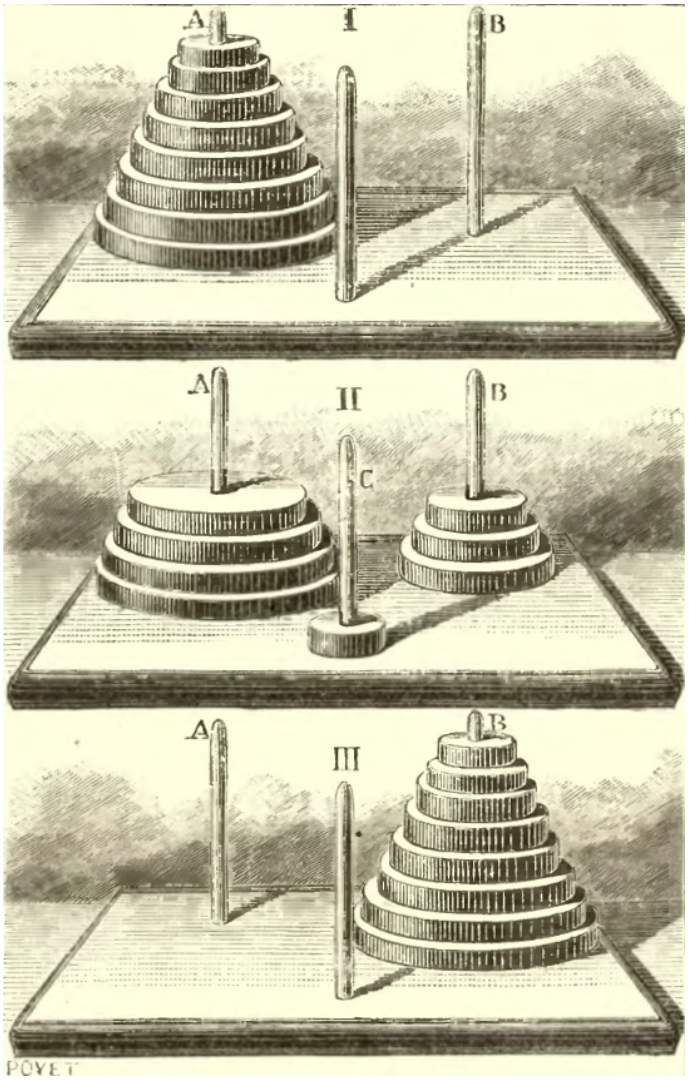

Figure 3: Jeu des tours de Hanoï

Si vous voulez réfléchir au problème sans indice supplémentaire, c'est maintenant qu'il faut le faire. Sinon, continuez à lire.

Ici aussi, l'objectif est de se ramener à un problème similaire plus petit : pour déplacer une tour de taille n, utiliser le fait qu'on sait déplacer une tour de taille $n-1$ (ça aurait pu être $n-2$ ou $n/2$). On peut imaginer deux stratégies, dont une seule est correcte, représentées figure 4.

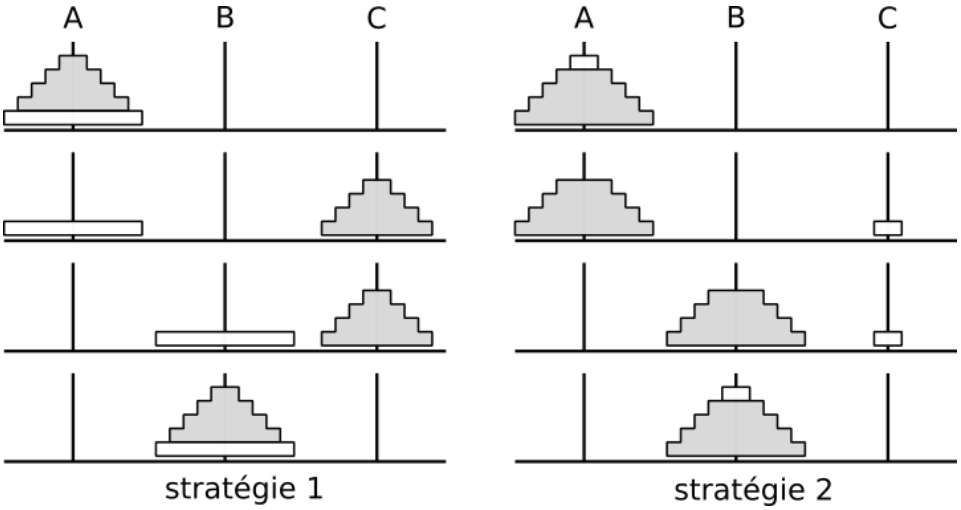

Figure 4: Deux stratégies (une seule est correcte) pour le jeu des tours de Hanoï

Comprendre la signification des deux stratégies, pourquoi elles sont récursives, et surtout pourquoi une seule des deux est correcte est un excellent exercice.

⚙ Essayez de déterminer en quoi les stratégies proposées sont récursives, et pourquoi une seule des deux est correcte (et laquelle).

Si tout ceci est compris, vous serez en mesure d'écrire une procédure récursive, qui, si on lui indique le nombre de disques, est capable de donner la liste des déplacements permettant de résoudre le casse-tête, par exemple pour passer la tour du piquet 1 au piquet 3. Pour 4 disques :

```
1 → 2
1 → 3
2 → 3
1 → 2
3 → 1
```

```
3 → 2
1 → 2
1 → 3
2 → 3
2 → 1
```

```
3 → 1
2 → 3
1 → 2
1 → 3
2 → 3
```

Il est important de se remémorer les trois règles de conception déjà données, et d'essayer de les appliquer *à la lettre* sur cet exemple :

1. spécification précise de la fonction/procédure, contenant la *taille* du problème
2. résolution de certains cas simples
3. décomposition du cas général

⚙ La solution à ce problème est très courte (l'algorithme comme le code Python tiennent en moins de 6 lignes). Vous êtes invités à trouver vous-même cette solution :

```
.
.
.
.
.
.
.
```

4.3 Problème du voleur

Attention, ce problème peut heurter la sensibilité des plus jeunes. Il y est question de stratégie de cambriolage.

Un voleur décide de visiter les maisons d'une rue. Suite à une phase de repérage, il sait quel bénéfice (dans une unité arbitraire) il pourra tirer en visitant chacune des maisons. Il sait aussi que pour éviter de se faire prendre, il devra absolument éviter de visiter deux maisons voisines (voir fig. 5).

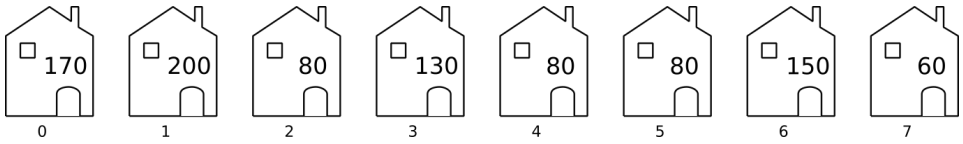

Figure 5: Le dilemme du voleur

L'objectif est de sélectionner les maisons à dévaliser qui maximisent le gain. Parce qu'il peut y avoir plusieurs solutions, on peut aussi ajouter des contraintes comme : minimiser le nombre de maisons visitées, ou minimiser la distance à parcourir.

Dans le cas qui précède, une solution optimale consiste à dévaliser les maisons 1, 3 et 6, pour un gain de 480.