Récursivité avancée

I Définition. Exemples.

Définition 1

On adopera la définition naïve suivante : une fonction récursive est une fonction qui s'appelle elle-même.

Remarque 1 : "Qui s'appelle elle-même" est à prendre au sens large : on a déjà vu le cas de la **récursivité croisée**, où la fonction f appelle une fonction g qui appelle la fonction f.

Exemples 1 : Plein d'exemples de fonctions qu'on peut définir de façon récursive. On les code.

Exemple 2 : Un exemple pour lequel vous avez vu une implémentation itérative **mais c'est idiot**, parce que c'est un algorithme fondamentalement récursif : l'exponentiation rapide! On peut commencer par rappeler le principe de l'exponentiation non rapide, qui peut naturellement se coder récursivement aussi.

II Récursivité enveloppée, récursivité terminale, itération

J'ai essayé d'insister sur le fait que Caml était optimisé pour gérer efficacement la récursivité, là où Python est surtout efficace pour l'itération. Je vais maintenant essayer de montrer que l'opposition pertinente n'est pas à faire entre itération et récursivité mais plutôt entre récursivité terminale et récursivité enveloppée.

II.1 Itération

L'itération, c'est la répétition d'une suite d'instructions à l'aide d'une boucle. On dispose de deux type de boucles :

1. les boucles **conditionnelles**, qui sont les boucles ..... ;
2. et les boucles **inconditionnelles**, qui sont les boucles .....

Exemple 3 : On a vu dans le premier chapitre (ou en ITC) plein de fonctions pouvant s'écrire avec des boucles : .....

Notons, et c'est important pour la suite, qu'on pouvait aussi les écrire sans boucle.

II.2 Récursivité enveloppée

La plupart des fonctions récursives qu'on écrit naturellement sont **récursives enveloppées**.

Définition 2

Une fonction est dite **récursive enveloppée** lorsque ses appels récursifs se font *à l'intérieur* d'une fonction :

$$f(x) = \begin{cases} \text{si } x \text{ est un cas de base alors une valeur de base} \\ \text{sinon } g(x, f(h_1(x)), \dots, f(h_k(x))). \end{cases}$$

La fonction g **enveloppe** les appels récursifs (ou l'unique appel récursif si $k = 1$).

Exemples 4 : Reprenons les exemples 1 et 2 et identifions les enveloppes.

L'utilisation de la récursivité enveloppée implique l'utilisation d'une **pile d'exécution**.

Exemple 5 : Qu'est-ce qui se passe quand on lui demande de calculer 3!, en fait ?

Pire : dans le cas d'appels récursifs multiples, on a un **arbre des appels récursifs**. Prenons le cas de F_4.

Cela nous fait deux motifs de vigilance lorsqu'on écrit des fonctions récursives (enveloppées) :

1. La **pile d'exécution** : elle occupe de l'espace mémoire, il faut s'assurer de ne pas en occuper trop, ou d'utiliser un langage qui optimise la gestion de cette pile (OCaml = ♡).
2. L'**arbre des appels récursifs** : si on **duplique** inutilement les appels récursifs alors on va multiplier inutilement les mêmes calculs et produire un algorithme trop lent, et ceci est indépendant du langage utilisé.

II.3 Récursivité terminale

Mais toutes les fonctions récursives que l'on écrit ne sont pas récursives enveloppées !

Exemple 6 : l'exemple typique c'est le calcul du pgcd à l'aide de l'algorithme d'Euclide (version récursive).

Définition 3

Une fonction $f : E \to X$ est dite **récursive terminale** lorsqu'elle est de la forme :

$$f(x) = \begin{cases} \text{si x est un cas de base alors une valeur de base} \\ \text{sinon } f(g(x)) \end{cases}$$

Exemple 7 : Un autre exemple ? .....

Exemple 8 : Écrivons maintenant une fonction **récursive terminale** qui calcule la factorielle.

II.4 En guise de conclusion

En 2014, on m'a posé une bonne question, mais l'important, c'est la réponse :

III Diviser pour régner

On a vu plus haut comment "désenvelopper" la factorielle. On peut le faire aussi pour Fibonacci. Mais pour d'autres algorithmes comme l'exponentiation rapide ou le tri par fusion, c'est beaucoup plus délicat.

On s'intéresse ici à une classe d'algorithmes fondamentalement récursifs enveloppés, mais efficaces, mettant en œuvre une même stratégie appelée "diviser pour régner".

III.1 Définition

Définition 4 : Stratégie "diviser pour régner"

La stratégie "diviser pour régner" consiste à diviser le problème en sous-problèmes de taille $\lfloor \frac{n}{\lambda} \rfloor$, à traiter récursivement ces sous-problèmes, puis à on reconstruit la solution globale à l'aide des solutions partielles.

On sent bien (on le précisera plus bas) qu'une telle stratégie est d'autant plus efficace qu'il y a peu de sous-problèmes, ou que ceux-ci sont de petite taille (*i. e.* λ est grand).

III.2 Exponentiation rapRecherche dichotomique dans un tableau trié

On a codé cet algorithme dans le premier chapitre. On veut trouver la position d'un élément dans une certaine plage d'un tableau trié.

⇝ Quels sont les cas de base ?
⇝ Combien y a-t-il de sous-problèmes ? Quels sont-ils ? Quelle est leur taille ?
⇝ Comment reconstruit-on la solution globale à partir des solutions de sous-problèmes ?

III.3 Exponentiation rapide

On a codé plus haut cet algorithme. On veut calculer x^n.

⇝ Quels sont les cas de base ?
⇝ Combien y a-t-il de sous-problèmes ? Quels sont-ils ? Quelle est leur taille ?
⇝ Comment reconstruit-on la solution globale à partir des solutions de sous-problèmes ?

III.4 Tri par fusion

On doit avoir codé cet algorithme en TD. Sinon, on le fait tout de suite. Il est adapté au tri **d'une liste** (et pas d'un tableau). Il se comporte de trois étapes :

- Diviser la liste en deux moitiés ;
- Trier récursivement chacune des deux moitiés ;
- Fusionner les deux moitiés triées pour reconstituer la liste triée.

Mêmes questions :
⇝ Quels sont les cas de base ?
⇝ Combien y a-t-il de sous-problèmes ? Quels sont-ils ? Quelle est leur taille ?
⇝ Comment reconstruit-on la solution globale à partir des solutions de sous-problèmes ?

IV Terminaison des fonctions récursives

IV.1 Terminaison des fonctions itératives

Ce **n'est pas** l'objet de ce chapitre. Mais un bref rappel : pour montrer qu'une boucle `while` termine, on utilise un **variant de boucle**, c'est-à-dire une expression entière dépendant des variables de la fonction qui décroît strictement à chaque itération de la boucle.

Exemples 9 :

IV.2 Un problème à résoudre

Une fonction récursive n'a aucune raison de toujours terminer. Pourquoi ?

IV.3 Cas des fonctions $\mathbb{N} \to X$

Lemme 1

Si $f : \mathbb{N} \to X$ est une fonction récursive telle que :

- le cas $n = 0$ est un cas de base (la fonction ne s'appelle pas elle-même sur cet argument) ;
- les autres cas se traitent par (un nombre borné d') appels récursifs sur des entiers strictement plus petits ;

Alors f termine.

Démonstration. .....

Remarque 2 : Même si on n'a pas $f : \mathbb{N} \to X$, on peut parfois s'y ramener !
Par exemple, pour montrer qu'une fonction définie sur les *'a listes* termine, il suffit de montrer :

-
- et

Le cas de $\mathbb{N} \to X$ est en fait un cas particulier du cas général.
On verra comment retrouver le cas des *'a listes* comme un autre cas particulier de ce cas général plus bas.

Exemple 10 : L'exponentiation rapide termine.

IV.4 Bon ordre sur un type ou un ensemble

Définition 5

On dit que $(E, \leqslant_E)$ est **bien ordonné**, ou que $\leqslant_E$ **est un ordre bien fondé sur** E lorsque toute partie non vide de E a un plus petit élément.

Exemples 11 :

Théorème 1

Dans un ensemble bien ordonné, toute suite strictement décroissante d'éléments est **finie**.

IV.5 Terminaison des fonctions récursives

Corollaire 1

Si E est bien ordonné et $f : E \to F$ est une fonction récursive telle que :

- les éléments minimaux de E sont des cas de base (f ne s'appelle pas elle-même sur ces argument) ;
- les autres cas se traitent par appels récursifs sur des éléments de E strictement plus petits ;

Alors f termine.

Exemple 12 : Calcul des binomiaux via la formule de Pascal.

Remarque 3 : Des fois, ça se passe mal (on ne peut pas utiliser le théorème précédent, mais les fonctions terminent quand même... ou ne terminent pas... ou on ne sait pas...). C'est la vie. Un exemple pas sympa : la fonction 91 de McCarthy.

V Correction des fonctions récursives

V.1 Correction des fonctions itératives

Ce **n'est pas** l'objet de ce chapitre. Mais un bref rappel : pour montrer qu'une fonction itérative est correcte, on utilise un **invariant de boucle**, c'est-à-dire un énoncé dépendant des variables de la fonction qui reste inchangé à chaque itération de la boucle, qui est vrai avant la première itération de la boucle, et qui donne le résultat demandé à l'issue de la dernière itération de la boucle.

Exemple 13 : Prenons le calcul itératif d'une factorielle.

V.2 Correction des fonctions récursives

Théorème 2

Si E est bien ordonné alors on a $\Big(\forall x \in E,\ \big(\forall y < x,\ P(y)\big) \Rightarrow P(x)\Big) \Longrightarrow \big(\forall x \in E,\ P(x)\big)$.

Remarquons que .....

Corollaire 2

Si E est bien ordonné et $f : E \to F$ est une fonction récursive qui termine, telle que :

- f est correcte sur ses cas de base ;
- sur les autres cas, f est correcte en supposant qu'elle l'est sur ses appels récursifs ;

alors f est correcte.

Exemple 14 : Prenons l'exemple de l'exponentiation rapide.

Exemple 15 : Reprenons l'exemple du calcul des binomiaux via la formule de Pascal.

VI Complexité des fonctions récursives

VI.1 Définitions

Définition 6

La complexité **en temps** d'un algorithme **dans le pire des cas** est le plus grand nombre possible d'opérations élémentaires effectuées pour obtenir le résultat en fonction de la taille de la donnée entrée.

Attention, les notions d'*opération élémentaire* et de *taille des données* dépend du contexte où elle doit être précisée. On dit qu'on a différents *modèles de calcul*.

Remarque 4 : Variantes :

1. **Complexité dans le meilleur des cas** : on remplace "pire" par "meilleur".
2. **Complexité en moyenne** : pour un entier n donné, on fait la moyenne de toutes les complexités possibles pour toutes les entrées de l'algorithme de taille n donnée. On considère équiprobables toutes les données de taille n.
3. **Complexité amortie** : moyenne des complexités obtenues pour une succession de n applications de l'algorithme.
4. **Complexité en espace** (dans le pire des cas) : la plus grande taille possible de la mémoire utilisée pour obtenir le résultat en fonction de la taille de la donnée entrée. Le sens du mot "taille" est précisé par le contexte.

Définition 7

• On dit que $(c_n)_n$ est dominée par $(u_n)_n$ et on note $c_n = O(u_n)$ lorsqu'il existe une constante $K > 0$ telle que $c_n \leqslant K u_n$ APCR.

• On dit que $(c_n)_n$ est de même ordre de grandeur que $(u_n)_n$ et on note $c_n = \Theta(u_n)$ lorsqu'il existe des constantes $k_1, k_2 > 0$ telles que $k_1 u_n \leqslant c_n \leqslant k_2 u_n$ APCR.

En pratique, on essaie d'exprimer une complexité c_n par son ordre de grandeur : $c_n = \Theta(\cdots)$.

Lorsque ce n'est pas possible, on se contente d'une domination : $c_n = O(\cdots)$.

VI.2 Complexités usuelles

On s'intéressera principalement aux algorithmes dont la complexité c_n est :

1. Bornée : .....
2. Logarithmique : .....
3. Linéaire : .....
4. Quasi-linéaire : .....
5. Quadratique : .....
6. Polynomiale : .....
7. Exponentielle : .....

Remarque 5 : $\log_b(n) = \Theta(\lg(n))$.

VI.3 Exemples élémentaires

Exemples 16 : La factorielle, l'exponentiation non rapide. On commence par préciser le modèle de calcul.

Exemple 17 : Le tri par insertion. On commence par préciser le modèle de calcul.

VI.4 Algorithmes de type "diviser pour régner"

Complexité d'un algorithme mettant en œuvre une stratégie "diviser pour régner" :

- coût de la division en sous-problèmes : d(n)
- coût d'un sous-problème : $c_{\lfloor \frac{n}{\lambda} \rfloor}$
- coût de la reconstruction : r(n)

Le coût total est : $c_n = ac_{\lfloor \frac{n}{\lambda} \rfloor} + f(n)$ où a est le nombre de sous-problèmes et $f(n) = d(n) + r(n)$.

Exemples 18 : L'exponentiation rapide, la recherche dichotomique. On commence par préciser le modèle de calcul.

Exemple 19 : Le tri par fusion. On commence par préciser le modèle de calcul.

VI.5 Des algorithmes déraisonnables et comment les améliorer

Exemple 20 : On reprend le calcul du $n^{\text{ième}}$ nombre de Fibonacci. On note c_n la complexité de l'algorithme naïf :

Il faut faire mieux ! Faisons mieux.

Plus généralement : on souhaite calculer le $n^{\text{ième}}$ terme d'une suite définie par récurrence forte. Pour éviter de dupliquer *ad nauseam* des appels récursifs inutiles, on peut **mémoriser** les valeurs successives de la suite dans un tableau, pour s'en servir ensuite afin de calculer les suivantes [1]. On remplace alors la récursivité par de l'itération.

Exemple 21 : Écrire une fonction `catalan : int` $\to$ int telle que `catalan n` calcule le $n^{\text{ième}}$ nombre de Catalan C_n, de complexité en temps en $O(n^2)$ (on compte les sommes et les produits). Sachant : $C_0 = 1$ et $\forall n \geqslant 1,\ C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$.

1. Cette technique s'appelle dans le programme d'informatique de tronc commun le "calcul de bas en haut". On voit aussi en tronc commun une autre méthode appelée "mémoïsation". On en redira un mot.