Chapitre 4 : Récursivité et Introduction à OCaml

18 octobre 2023

1 Récursivité

On considère la fonction `fact` suivante, qui est une implémentation de la fonction mathématique factorielle :

```
int fact(int n)
    {
    int r = 1;
    for (int i = 1; i <= n; i++){
        r = r*i;
    }
    return r;
    }
```

La façon dont on fait le calcul permettant d'obtenir la valeur de $n!$ ressemble à la façon dont on définit $n!$ comme un produit :

$$n! = \prod_{i=1}^{n} i$$

En mathématiques, on peut également définir la factorielle de cette façon :

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

On observe que la valeur de $n!$ est définie en fonction de celle de $(n-1)!$ quand n est différent de 0 : on a besoin de la fonction factorielle à l'intérieur de sa propre définition.

Définition

Une fonction **récursive** est une fonction qui fait apel à elle-même dans sa propre définition.

En C, il est simplement possible de définir des fonctions de façon récursive en les appelant à l'intérieur du corps de la fonction.

Exemple. La fonction factorielle peut donc être définie récursivement en C de cette manière :

```
int fact(int n){
    if (n == 0){
        return 1;
    }
    else{
        return n*fact(n-1);
    }
}
```

Exemple. On a vu l'exponentiation naïve pour calculer a^n, en initialisant une variable à 1, en la multipliant par a à l'intérieur d'une boucle faisant n tours de boucles et en renvoyant sa valeur en sortie de boucle.

Une façon de définir récursivement le calcul de puissance est la suivante :

$$\begin{cases} a^n = 1 & \text{si } n = 0 \\ a^n = a \times a^{n-1} & \text{sinon} \end{cases}$$

On peut donc, en C, définir l'exponentiation naïve récursivement :

```
int exp_naive(int a, int n){
    if (n == 0){
        return 1;
    }
    else{
        return a*exp_naive(a,n-1);
    }
}
```

Exemple. On a également vu l'exponentiation rapide dans le cas itératif pour calculer a^n avec un ordre de grandeur inférieur pour le temps de calcul. On utilisait pour ça la propriété suivante : $a^n = a^{2q+r} = a^r(a^2)^q$, où $r = 0$ ou 1 est le reste dans la division euclidienne de n par 2, et $q = \lfloor \frac{n}{2} \rfloor$ est le quotient dans la division euclidienne de n par 2. En particulier, on peut faire un calcul de $(a^2)^q$ à l'aide de cette propriété, et continuer jusqu'à avoir un quotient nul.

On peut réécrire cette propriété, d'une façon à définir récursivement l'exponentiation rapide :

$$\begin{cases} a^n = 1 & \text{si } n = 0 \\ a^n = (a^{\lfloor \frac{n}{2} \rfloor})^2 & \text{si } n \neq 0 \text{ et } n \text{ est pair} \\ a^n = a \times (a^{\lfloor \frac{n}{2} \rfloor})^2 & \text{si } n \neq 0 \text{ et } n \text{ est impair} \end{cases}$$

Cela permet de définir l'exponentiation rapide récursivement :

```
int exp_rapide(int a, int n){
    assert (a>0);
    if (n == 0){
        return 1;
    }
    else{
        int b = exp_rapide(a,n/2);
        if (n%2 == 0){
            return b*b;
        }
        else {
            return a*b*b;
        }
    }
}
```

1.1 Arbre d'appels

Dans les différentes fonctions ci-dessus, la définition est telle que l'on a deux situations possibles lorsqu'on appelle la fonction sur une valeur donnée :

— La valeur d'entrée produit un **appel récursif**
— La valeur d'entrée est telle qu'aucun appel récursif n'est fait dans le calcul : c'est un **cas terminal** (exemple : 0 pour la factorielle et les deux exponentiations)

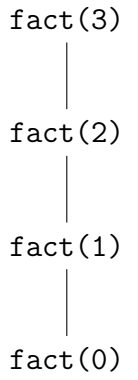
Lorsque l'on appelle une fonction récursive sur une valeur qui n'est pas un cas terminal, celle-ci va produire un appel récursif de la fonction sur une nouvelle valeur ; si cette nouvelle valeur ne correspond pas à un cas terminal, un nouvel appel est effectué, et ainsi de suite jusqu'à ce que tous les appels récursifs arrivent sur des cas terminaux.

Exemple. Lors de l'appel de `fact(3)`, on a un appel à `fact(2)`, qui lui-même entraîne un appel récursif à `fact(1)`, qui entraîne un appel à `fact(0)`.

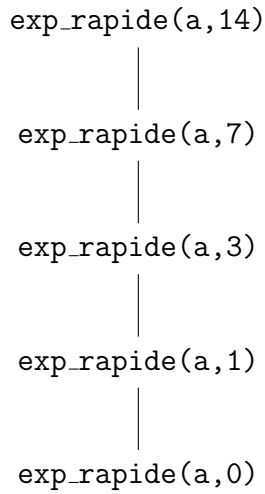
`fact(0) = 1` correspond à un cas terminal : `fact(1)` peut alors être calculé en fonction de `fact(0)`, `fact(2)` en fonction de `fact(1)` et `fact(3)` en fonction de `fact(2)`.

la liste des appels qui sont produits à partir de l'appel d'une fonction sur une valeur donnée peut être représentée par un arbre : un noeud au sommet de l'arbre correspond à l'appel initial, et chacun des appels récursifs à l'intérieur de l'appel initial va constituer une branche qui donnera elle-même lieu à un sous-arbre. Une telle structure arborescente est nommée **arbre d'appel**.

Exemple. arbre d'appel pour `fact(3)` :



Exemple. Arbre d'appel pour `exp_rapide(a,14)` :



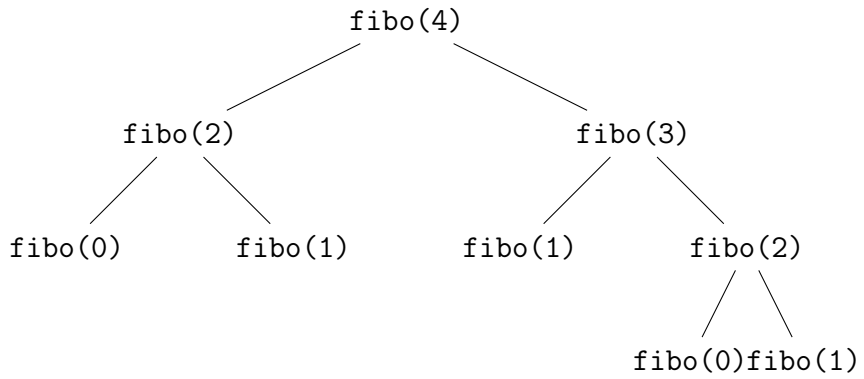
Les exemples précédents montrent des structures arborescentes avec une unique branche, car la fonction est appelée une seule fois. Il est évidemment possible d'avoir des arbres avec

plusieurs branches, lorsqu'il y a plusieurs appels récursifs à l'intérieur de la définition.

Exemple. Une version récursive naïve du calcul du n-ième terme de la suite de Fibonacci peut être obtenue ainsi :

```
int fibo(int n){
    if (n == 0){
        return 0;
    }
    else if (n==1){
        return 1;
        }
        else {
            return fibo(n-1) + fibo(n-2);
        }
    }
}
```

L'arbre d'appels pour `fibo(4)` est le suivant :



On avait vu que l'on pouvait programmer de façon itérative une fonction pour renvoyer le n-ième terme de la suite de Fibonacci, en complexité linéaire. On remarque facilement que l'algorithme ci-dessus, en revanche, ne sera pas très efficace, pour deux raisons :

— pour un appel sur la valeur n, on a à chaque fois un appel sur les valeurs $n-1$ et $n-2$: chaque niveau de l'arbre est rempli jusqu'au $\lfloor n/2 \rfloor$-ième, et un niveau k de l'arbre contient 2^k appels quand il est rempli : le nombre d'appels croît exponentiellement en fonction de n, et par conséquent la complexité également.

— Il y a également un problème de complexité spatiale : chacun des noeuds de l'arbre existe en mémoire tant que le calcul des valeurs qu'il appelle n'est pas fait. On aura donc un nombre de valeurs stockées temporairement qui sera exponentiel en fonction de la taille de l'entrée.

Ce phénomène se produit car des appels récursifs sont effectués plusieurs fois. On verra qu'il existe des façons d'éviter ces appels récursifs superflus (mémoïzation).

1.2 Récursif vs Itératif

On a vu qu'il était possible, pour plusieurs problèmes algorithmiques, de programmer des algorithmes correspondant à leur spécification en utilisant des techniques récursives ou itératives.

Il n'y a rien qui permette d'affirmer clairement que l'une des techniques est meilleure que l'autre. On sait que pour tout algorithme itératif, il existe une version récursive équivalente, et vice-versa. L'un des facteurs principaux dans la décision de recourir à une écriture récursive ou itérative : on verra qu'OCaml favorise l'utilisation de la récursion, à l'inverse de Python, par exemple.

Chaque méthode a ses avantages et ses inconvénients. Il peut être plus difficile de trouver le schéma de définition par récurrence d'une fonction. En revanche, celui-ci peut parfois permettre d'exprimer et donc de programmer de façon très simple certains problèmes.

Exemple. On considère le jeu des tours de Hanoi : on a trois tiges fixes, une tour de n disques empilés par taille décroissante sur l'une des tiges. On souhaite déplacer intégralement la tour sur une tige finale en déplaçant les disques selon ces règles :

- On ne déplace que les disques au sommet des tiges ;
- On déplace un disque à la fois ;
- On ne place jamais un disque sur un autre disque de taille inférieure.

On peut imaginer le problème dont la spécification est la suivante :

- **Entrée :** un entier naturel non-nul n
- **Sortie :** la liste des déplacements de disques à effectuer pour déplacer une tour de Hanoi de taille n de la tige gauche à la tige droite.

On peut définir récursivement, de façon très simple, cette liste de mouvement en fonction de n :

- Si $n = 1$: on déplace le seul disque de la tige gauche à la tige droite.
- Si $n > 1$:
  - On déplace la sous-tour de taille $n - 1$ de la gauche au milieu, avec la tige droite comme intermédiaire ;
  - On déplace le disque de base de gauche à droite ;
  - On déplace la sous-tour de taille $n - 1$ du milieu vers la droite, avec la tige gauche comme intermédiaire.

2 Introduction à OCaml

2.1 OCaml

OCaml est une implémentation du langage de programmation CAML. CAML est un langage de programmation conçu et maintenu essentiellement par des chercheurs de l'Inria, développé à partir des années 1980.

OCaml est un langage **fonctionnel**, bien qu'il permette l'utilisation de certains traits de programmation impérative. La programmation fonctionnelle est un paradigme de programmation dans lequel les calculs sont considérés comme des évaluations de fonctions. Cela influe en particulier sur la pratique de programmation, par exemple en privilégiant - ou en contraignant à - l'utilisation de la récursivité plutôt que des méthodes itératives.

OCaml est un langage qui peut aussi bien être compilé (comme C) qu'interprété (comme Python). On enregistre un programme en OCaml dans un fichier .ml. Lorsque l'on a un `fichier .ml`, on peut, en écrivant les lignes suivantes dans un terminal :

- Le compiler vers du bytecode avec :
  `ocamlc fichier.ml -o nomdefichier` (`nomdefichier.exe` sous Windows),
  puis l'exécuter avec `ocamlrun nomdefichier` (.exe)
- Le compiler vers du code natif avec :
  `ocamlopt fichier.ml -o nomdefichier` (`.exe`),
  puis l'exécuter avec `./nomdefichier` (`.exe`)
- Interpréter le contenu du fichier avec `ocaml fichier.ml`

On peut également simplement lancer un toplevel (ce qui s'apparente à une interface en lignes de commandes) avec la commande `ocaml` seule dans le Terminal ; on peut alors écrire les lignes de programme ocaml qui seront directement interprétées dans le Terminal.

2.2 Syntaxe de OCaml

cf TP 03.

3 Types définis récursivement

De même que l'on peut définir des fonctions récursivement, il est possible de définir des types récursivement. De la même façon qu'il faut, pour les fonctions, avoir des cas terminaux pour lesquels il n'y a pas d'appels récursifs et des cas pour lesquels on fait des appels récursifs, pour définir un type récursivement, il faut des cas de base ne dépendant pas de variables du type en cours de définition et des cas qui en dépendent.

En OCaml, ce sont les énumérations qui permettent de définir des types récursifs, à l'aide des constructeurs prenant un argument qui peut être du type en cours de définition.

On peut citer plusieurs exemples de types qui l'on pourra définir récursivement :

3.1 Listes

Les listes peuvent être définies récursivement de la façon suivante. Une liste d'éléments est :

— Soit la liste vide

— Soit la donnée de la tête de liste (élément) et de la suite de la liste (liste d'éléments)

Le type liste construit de cette façn est prédéfini en OCaml.

3.2 Arbres

Les arbres, qui sont composés de Noeuds, peuvent être définies récursivement de la façon suivante. Un arbre est :

— Soit le Noeud vide ;

— Soit la donnée d'un sommet (Noeud) et d'un ensemble de sous-arbres (lsite d'arbres)

3.3 Formules de Logique Propositionnelle

Les formules de logique propositionnelle peuvent être définies récursivement de la façon suivante. Une formule est :

— Soit une variable propositionnelle (cas de base) ;

— Soit la conjonction $\wedge$ de deux formules de logique propositionnelle ;

— Soit la conjonction $\vee$ de deux formules de logique propositionnelle ;

— Soit la négation $\neg$ d'une formule de logique propositionnelle.

(On peut éventuellement rajouter l'implication $\Rightarrow$ ou l'équivalence $\Leftrightarrow$ entre deux formules de logique propositionnelle, que l'on peut également construire avec les opérations ci-dessus)

4 Récursion Mutuelle

De la même manière que l'on peut re-définir plusieurs variables en fonction de leurs valeurs à un instant donné, on peut définir récursivement deux fonctions en faisant appel à l'une dans la définition de l'autre et vice-versa. On parle de **récursion mutuelle** (ou récursivité croisée).

Exemple. On peut définir mathématiquement les nombres pairs et impairs de la façon suivante. Soit n un entier naturel :

— n est pair si $n = 0$ ou $n - 1$ est impair

— n est impair si $n \neq 0$ et $n - 1$ est pair.

Pour pouvoir définir la parité de cette façon en OCaml, il faut une écriture où les définitions des deux fonctions se trouvent à l'intérieur d'un même `let`. On peut utiliser la syntaxe suivante :

```
let rec even n = (n = 0) || odd (n-1)
and odd n = (n <> 0) && even (n-1)
```

5 Pile d'Appels

Lors de l'exécution d'un programme, on occupe de l'espace mémoire (RAM). Celui-ci peut être vu comme un ensemble de cases mémoires avec des adresses. Une partie de cet espace est occupé par le code du programme, une autre par les donnée statiques ; un autre partie, le tas, est utilisée pour les données dynamiques (structures de données créées), et enfin, la **pile** utilise de l'espace de stockage pour organiser la mémoire pour les appels de fonctions, et le stockage de paramètres et de variables locales. L'espace occupé par la pile évolue au fur et à mesure de l'exécution du programme.

lors d'un appel de fonction, par exemple, l'appel initial occupera un espace dédié, appelé *contexte*, contenant les arguments de l'appel, l'espace mémoire pour stocker les variables locales, et la valeur de retour. Lorsqu'une fonction f appelle une fonction g, l'appel de cette fonction g va lui-même entraîner l'occupation d'un nouveau contexte.

On a un empilement des espaces occupés par chaque appel de fonction sur une **pile d'appels.** Lorsque l'appel à une fonction est terminé, l'espace occupé par celle-ci est dépilé : le fait d'avoir l'adresse de retour permet d'utiliser la valeur de retour calculée dans le calcul de la fonction qui a produit l'appel.

Dans le cas d'une fonction récursive, une succession d'appels récursifs vont être produits. Considérons par exemple la fonction `somme` ci-dessous :

```
let rec somme n =
    if n = 0 then 0
    else n + somme (n-1)
;;
```

Voici la façon dont évolue la pile d'appels sur un appel de `somme 3` :

- `somme 3 = 3 + somme 2` : on stocke les informations concernant l'opération réalisée dans le contexte, et l'adresse de retour pour somme(2)
- `somme 2 = 2 + somme 1` : on stocke les informations concernant l'opération réalisée dans le contexte, et l'adresse de retour pour somme(2)
- `somme 1 = 1 + somme 0` : on stocke les informations concernant l'opération réalisée dans le contexte, et l'adresse de retour pour somme(0)
- `somme 0 = 0` : la valeur de retour est 0
- le calcul de somme(1) = 1 + 0 = 1 est effectué, l'espace occupé est libéré et sa valeur de retour est récupérée dans le calcul de somme(2)
- le calcul de somme(2) = 2 + 1 = 3 est effectué, l'espace occupé est libéré et sa valeur de retour est récupérée dans le calcul de somme(3)
- le calcul de somme(3) = 3 + 3 = 6 est effectué, l'espace occupé est libéré et sa valeur de retour est renvoyée.

On observe qu'on a n appels récursifs qui vont entraîner à chaque fois un empilement sur la pile d'appels, jusqu'à arriver sur un cas terminal (somme(0)) qui va permettre un par un d'effectuer les calculs des différents appels récursifs et de "remonter à la surface".

Pour de grandes valeurs de n, on a un risque d'occupation importante de l'espace en mémoire : si on teste somme 100000, un message d'erreur est renvoyé, indiquant un **débordement de pile** (**Stack Overflow** en anglais).

Une façon d'éviter le débordement de la pile d'appels est d'utiliser la **récursivité terminale** :

Définition

Dans le corps d'une fonction f, un appel à une fonction g est **terminal** lorsqu'il s'agit de la dernière opération effectuée par la fonction f, et que celle-ci renvoie simplement le résultat de l'appel à g.

Une fonction récursive est sous forme récursive terminale lorsque l'appel à elle-même dans son corps est la dernière opération effectuée.

L'intérêt de la forme récursive terminale est que, étant donné que le résultat renvoyé par un appel de fonction est le même que celui renvoyé par l'appel de fonction à l'intérieur de son corps, on n'a pas besoin d'empiler les appels. En effet, on peut libérer l'espace de l'appel initial au moment de l'appel de la fonction à l'intérieur du corps. Pour que cela se fasse, il faut que le compilateur soit capable de détecter le caractère récursif terminal d'une fonction (ce qui est le cas en OCaml).

Prenons un contre-exemple : dans la fonction somme, le dernier calcul effectué par l'appel de somme n est l'addition, entre n et somme (n-1). Il faut conserver les informations concernant ce calcul (opération d'addition, variable n) pendant l'appel somme (n-1), et ainsi de suite, créant un empilement dans la pile d'appels, ce qui n'est pas le cas avec une fonction récursive terminale.

Exemple. On peut écrire une version récursive terminale de somme à l'aide d'un accumulateur :

```
let rec somme_acc n acc =
    if n = 0 then acc
    else somme_acc (n - 1) (acc + n)
```

On obtient la valeur attendue pour somme n et prenant 0 comme valeur de l'accumulateur en entrée : `somme_acc 0 n`. Pour éviter d'avoir à rentrer un accumulateur, on peut définir somme en utilisant `somme_acc` comme fonction auxiliaire :

```
let somme n =
    let rec somme_acc n acc =
        if n = 0 then acc
        else somme_acc (n - 1) (acc + n)
    in somme_acc n 0
;;
```

6 Terminaison et Correction

De la même façon que dans le cas itératif, on peut souhaiter s'assurer qu'une fonction termine, c'est-à-dire qu'elle effectue un nombre fini d'opérations, et qu'elle est correcte, c'est-à-dire que la valeur renvoyée pour une entrée donnée correspond bien à la valeur attendue selon la spécification de la fonction.

Dans le cas itératif, c'était le présence de boucles qui rendait nécessaire l'emploi de techniques spécifiques (variant de boucle, invariant de boucle) pour prouver la terminaison et la correction de fonctions. Dans le cas récursifs, c'est la présence d'appels récursifs qui va obliger à utiliser des techniques de démonstrations comparables.

6.1 Terminaison

Dans le cas itératif, en présence d'une boucle `while`, on utilisait un variant de boucle, qui était positif à l'intérieur de la boucle, entier et strictement décroissant pour établir qu'il y allait avoir un nombre fini d'itérations de la boucle.

Reprenons la fonction `exp_rapide` :

```
let rec exp_rapide a n =
    if n = 0 then 1
    else let b = exp_rapide a (n/2) in
    if n mod 2 = 0 then b*b
    else a*b*b ;;
```

On remarque que de façon similaire, l'appel récursif se fait sur une valeur du deuxième argument égale à `n/2`, qui est strictement inférieur à `n`, la valeur du deuxième argument dans l'appel initial à la fonction, et on observe que le dernier appel récursif est fait lorsque le deuxième argument vaut 0.

On utilisera donc, comme dans le cas itératif, le **technique du variant** où le variant sera une quantité dépendant des arguments, qui sera :

- Positive ;
- Entière ;
- Décroissante au fur et à mesure des appels récursifs successifs

Le variant ne sera pas forcément un des arguments, mais peut être une valeur qui dépend de ceux-ci. Par exemple, dans la fonction `appartient` dans le TP, le cas de base est la liste vide `[]` et les cas récursifs sont ceux où la liste est de la forme `s::t`, et la fonction `appartient` est appelée sur `t`. On peut dans ce cas-là utiliser comme variant la longueur de la liste sur laquelle est appelée la fonction, qui est entière, positive et décroît à chaque appel récursif.

Dans certains cas, il peut être difficile de se ramener à une valeur entière ; dans un autre chapitre, la notion d'ordre bien fondé sera abordée pour avoir des outils de preuve de terminaison dans ces cas particuliers.

6.2 Correction

Dans le cas itératif, pour prouver la correction d'un algorithme, on utilisait un invariant de boucle, qui était une propriété dépendant des valeurs des variables dans la boucle, qui était

vraie pour les valeurs des variables à la fin de chaque itération de la boucle : en particulier, la propriété était vraie pour les valeurs des variables en sortie de la boucle.

Pour prouver que la propriété était vraie à la fin de chaque itération, on utilisait une méthode proche de la récurrence : on prouvait qu'elle était vraie au début de la boucle (initialisation) et que si elle était vraie pour les valeurs des variables à la fin d'une itération, elle l'était également pour les valeurs à la fin de l'itération suivante (conservation).

Dans les cas récursifs, on se retrouvera dans des situations où les valeurs des variables ne sont pas modifiées, comme c'était le cas dans les cas itératifs. On aura des fonctions appelées sur des éléments de taille donnée, et la définition des fonctions sur ces éléments se fera sur leur définition sur des éléments de taille inférieure.

On pourra donc utiliser directement un raisonnement par récurrence sur les valeurs/tailles des entrées.

Théorème (Principe de Récurrence Simple)

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si :

— $P(0)$ est valide, et

— pour tout entier naturel n, la validité de $P(n)$ implique la validité de $P(n+1)$

Alors $P(n)$ est valide pour tout entier naturel n.

Exemple. Preuve de la fonction `exp_naive` définie récursivement :

On note $P(n)$ la propriété "pour tout entier naturel a, `exp_naive a n` renvoie la valeur a^n."

Initialisation : $P(0)$ est valide : pour $n = 0$, $a^n = 1$ est renvoyé par `exp_naive a n`.

Hérédité : Supposons que $P(n)$ est valide et montrons que $P(n+1)$ est valide. `exp_naive a (n+1)` renvoie la valeur `a * exp_naive a n`. Or, `exp_naive a n` renvoie a^n. Donc `exp_naive a (n+1)` renvoie $a \times a^n = a^{n+1}$.

On a donc `exp_naive a n` qui renvoie la valeur a^n pour tout entier naturel n et pour tout entier naturel a.

Avec la récurrence simple, on peut simplement prouver la validité d'une propriété en $i+1$ à partir de la validité en i afin de prouver la validité pour tout entier naturel. En revanche, on peut se retrouver dans une situation où la validité en $i+1$ n'est pas la conséquence de la validité en i, mais d'un autre entier, comme dans le cas de l'exponentiation rapide. On peut dans ce cas utiliser, au lieu de la récurrence simple, la récurrence forte :

Théorème (Principe de Récurrence Simple)

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si :

— Pour tout n , la validité de tous les $P(k)$ pour tout les entiers naturels $k < n$ implique la validité de $P(n)$

Alors $P(n)$ est valide pour tout entier naturel n.

Exemple. Preuve de la fonction `exp_rapide` définie récursivement :

On note $P(n)$ la propriété "pour tout entier naturel a, `exp_rapide a n` renvoie la valeur a^n."

Supposons que $P(k)$ est valide pour tout $k < n$ et montrons que $P(n)$ est alors valide.

— Si $n = 0$, la propriété est vraie.

— Si $n \neq 0$ et n est pair, alors `exp_rapide a n` renvoie la valeur `(exp_rapide a n/2) * (exp_rapide a n/2)`, c'est-à-dire $a^{\frac{n}{2}} \times a^{\frac{n}{2}}$ (car $k = \dfrac{n}{2} < n$), qui vaut a^n.

— Si $n \neq 0$ et n est impair, alors `exp_rapide a n` renvoie la valeur `a * (exp_rapide a n/2) * (exp_rapide a n/2)`, c'est-à-dire $a \times a^{\frac{n-1}{2}} \times a^{\frac{n-1}{2}}$ (car $k = \dfrac{n-1}{2} < n$), ce qui vaut a^n.

On a donc montré que pour tout entier naturel n, pour tout entier naturel a, `exp_rapide a n` renvoie la valeur a^n.

6.3 Complexité

Dans le cas récursif, on peut trouver une expression récursive de la complexité en fonction de la valeur ou de la taille n de l'entrée.

Par exemple, pour la fonction factorielle, définie précédemment, on a :

— Pour $n = 0$, $C(0) = 0$

— Pour $n > 0$, on fait une multiplication de n par `fact (n-1)` : on a donc une opération (multiplication) qui s'ajoute à toutes les opérations de `fact (n-1)`. Donc $C(n) = 1 + C(n-1)$.

On pose donc $u_n = C(n)$: (u_n) est une suite arithmétique de raison 1 et de premier terme 0, donc $C(n) = u_n = n$.

Remarque. On pourra également utiliser une suite (u_n) pour représenter autre chose que $C(n)$ lorsque cela permettra de trouver une expression dont on pourra trouver facilement le terme général.

Par exemple pour l'exponentiation rapide, on a $C(0) = 0$, $C(2k) = 1 + C(k)$ et $C(2k+1) = 2 + C(k)$. Si on considère les puissances de 2, on a : $C(2^{k+1}) = 1 + C(2^k)$. La suite définie par $u_k = C(2^k)$ est donc une suite arithmétique de raison 1 et de premier terme 2, donc $C(2^k) = 2 + k$. On pourra ensuite, pour un entier n quelconque, considérer k tel que $2^k \leq n \leq 2^{k+1}$.