Fonctions d'ordre supérieur, approfondissements en récursivité

16 octobre 2023

Plan

Administratif

Retours sur la semaine dernière

Ordre supérieur

Administratif

Programme du Devoir Surveillé

Le programme pour le DS est de tout depuis le début de l'année :

- Filtrage, Récursivité, Typage ;
- Listes, calcul sur les listes ;
- Programmation Impérative ;
- Tableaux, calcul sur les tableaux ;
- Complexité.

Gestion du temps en DS

Les épreuves sont généralement trop **longues** pour être terminées. Il ne faut pas hésiter à sauter des questions, quitte à laisser de la place ou commencer une nouvelle copie double.
Les épreuves sont généralement **difficiles** et vous évaluent sur des choses que *vous ne savez à priori pas*. Il ne faut pas paniquer.

Mise en page, code

- Encadrer ou souligner les résultats, les réponses dans la mesure du possible ;
- Vous pouvez utiliser une couleur ou une indentation différente pour le code ;
- Ajouter une phrase introductive avant votre code qui explique brièvement votre code (*« On utilise une fonction auxiliaire munie d'un accumulateur qui ... », « J'introduis une référence qui enregistre ... », ...*) ;
- Ajouter des commentaires spécifiques dans votre code pour certaines parties qui ne sont pas évidente, en particulier si vous avez l'impression que votre code est bricolé.

TP de cette semaine

Nous allons travailler en TP sur les TP des semaines passées. Pensez à prendre vos feuilles de TP si possible, mais je vous renverrai une version mise à jour.

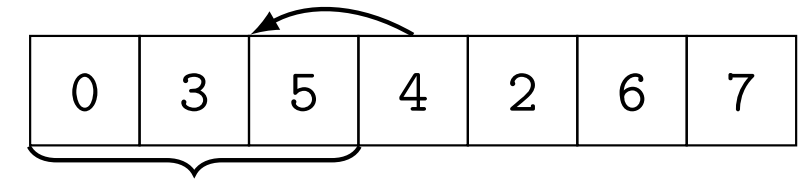
Durant les vacances

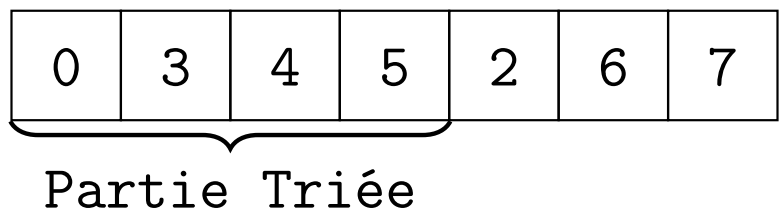
Un DM pour le 20 novembre sur la représentation des entiers.
Vous pouvez avancer sur des exercices des TP que nous n'avons pas traités, je suis disponible par e-mail pour répondre à vos questions.
Si vous avez accumulé du retard, ou si la programmation n'est pas évidentes, vous pouvez :

- Faire des fiches (sur les syntaxe ocaml, sur la récursivité, sur les différences entre liste/chaînes/tableau...) ;
- Refaire des exos des TP, s'il vous manque des corrections, vous pouvez chercher sur le net, ou m'envoyer un e-mail ;
- Regarder le programme de NSI, ou chercher d'autres ressources sur internet.

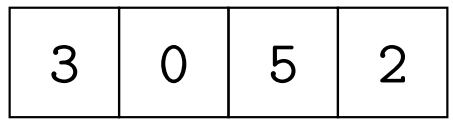
Retours sur la semaine dernière

Présentation du tri par insertion

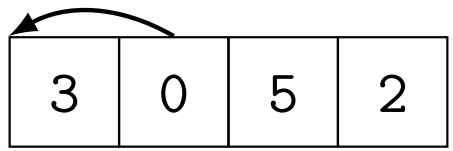




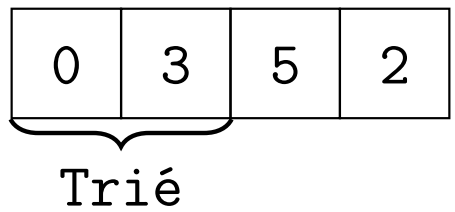
Exemple tri par insertion



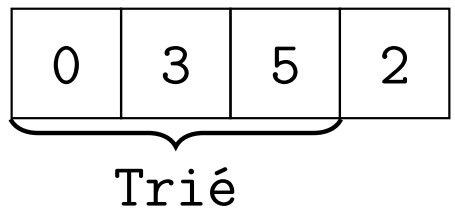
Exemple tri par insertion



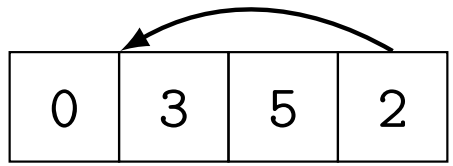
Exemple tri par insertion



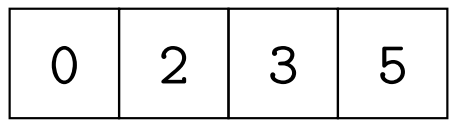
Exemple tri par insertion



Exemple tri par insertion



Exemple tri par insertion



Implémentation du tri par insertion

```
let inserer t k =
    for i = 0 to k - 1 do
        if t.(i)> t.(k) then
        (let a = t.(k) in
        t.(k) <- t.(i) ;
        t.(i) <- a)
    done
```

Implémentation du tri par insertion

```
let inserer t k =
    for i = 0 to k - 1 do
        if t.(i)> t.(k) then
        (let a = t.(k) in
        t.(k) <- t.(i) ;
        t.(i) <- a)
    done
```

```
let tri_insertion t =
            let n = Array.length t in
            for i = 0 to n-1 do
                inserer t i
            done
```

Implémentation du tri par insertion

```
let inserer t k =
    for i = 0 to k - 1 do
        if t.(i)> t.(k) then
        (let a = t.(k) in
        t.(k) <- t.(i) ;
        t.(i) <- a)
    done
```

```
let tri_insertion t =
            let n = Array.length t in
            for i = 0 to n-1 do
                inserer t i
            done
```

La complexité est quadratique (on parcours le tableau de 0 à 0, puis de 0 à 1, puis de 0 à 2, ..., puis de 0 à n), c'est-à-dire en $O(n^2)$.

Tri fusion : unir

```
let unir t i j k =
    let temporaire = Array.make (k - i + 1) 0 in
    for l = 0 to k-i do
        temporaire.(l)<-t.(i+l)
    done ;
    let a = ref i in
    let b = ref (j+1) in
    for l = 0 to k - i do
        if (!a>j) || (!b<=k &&t.(!a)>t.(!b))
        then (temporaire.(l) <- t.(!b) ; b:= !b + 1)
        else (temporaire.(l) <- t.(!a) ; a:= !a + 1)
    done ;
    for l = 0 to k - i do
        t.(l + i) <- temporaire.(l)
    done
```

Tri fusion

```
let tri_fusion t =
            let n = Array.length t in
            let rec aux i k =
                if i = k then ()
                else
                let j = (k+i)/2 in
                aux i j ; aux (j+1) k ; unir t i j k
            in aux 0 (n-1)
```

Ordre supérieur

Portée

Définition 1 : Portée

La **portée** d'un identifiant est l'étendue au sein de laquelle cet identifiant est liée.

Par identifiant, on entend le plus souvent nom de variable, ou nom de fonction.
La portée est donc constituée des endroits dans le programme où la variable est définie.

Portée des variables

C'est la variable de plus petite portée qui est prioritaire en cas d'ambiguïté.

```
1 let a = 0 in
2     print_int a ; print_newline ();
3     let a = 1 in
4         print_int a ; print_newline ()
```

Portée des variables

C'est la variable de plus petite portée qui est prioritaire en cas d'ambiguïté.

```
let a = 0 in
    print_int a ; print_newline ();
    let a = 1 in
        print_int a ; print_newline ()
```

```
0
1
```

Fonction anonyme

Certaines fonctions prennent en argument d'autres fonctions.

```
let f g (x,y) = (g x, g y)
```

```
let g x = x + 1 in
f g (1, 2)
```

Fonction anonyme

Certaines fonctions prennent en argument d'autres fonctions.

```
let f g (x,y) = (g x, g y)
```

```
let g x = x + 1 in
f g (1, 2)
```

Le mot clef fun permet d'introduire des fonction anonymes :

```
f (fun x -> x + 1) (1, 2)
```

Cela est similaire au mot-clé lambda en python.

Quelques remarques sur le mot-clef fun

On peut utiliser les fonction anonyme pour permettre d'ajouter une annotation de type.

```
let ajouter : int -> int -> int =
    fun x y -> x + y
```

Quelques remarques sur le mot-clef fun

On peut utiliser les fonction anonyme pour permettre d'ajouter une annotation de type.

```
let ajouter : int -> int -> int =
    fun x y -> x + y
```

Le mot-clé rec se met toujours au même endroit :

```
let rec fact =
    fun n -> match n with
    | 0 -> 0
    | _ -> n * fact (n - 1)
```

Quelques remarques sur le mot-clef fun

On peut utiliser les fonction anonyme pour permettre d'ajouter une annotation de type.

```
let ajouter : int -> int -> int =
    fun x y -> x + y
```

Le mot-clé rec se met toujours au même endroit :

```
let rec fact =
    fun n -> match n with
    | 0 -> 0
    | _ -> n * fact (n - 1)
```

En règle générale, on peut se passer du mot-clé fun.

Fonctions d'ordre supérieur

Définition 2 : Fonction d'ordre supérieur

Une fonction d'ordre supérieur est une fonction qui vérifie au moins l'une des propriétés suivantes :

- elle prend en argument une fonction ;
- elle renvoie une fonction.

Fonction en argument

```
let rec appliquer f l = match l with
| [] -> []
| p::q -> f p::appliquer f q
```

Fonction en argument

```
let rec appliquer f l = match l with
| [] -> []
| p::q -> f p::appliquer f q
```

```
appliquer (fun x -> x * 2) [1; 2; 3; 4; 5]
```

```
[2; 4; 6; 8; 10]
```

Fonction en résultat

```
1 let f x = fun y -> x + y
```

Que fait la fonction `f` ?

Fonction en résultat

```
1 let f x = fun y -> x + y
```

Que fait la fonction `f` ? Quel est sa signature ?

Fonction en résultat

```
1 let f x = fun y -> x + y
```

Que fait la fonction `f` ? Quel est sa signature ?

```
int -> (int -> int)
```

Fonction en résultat

```
1 let f x = fun y -> x + y
```

Que fait la fonction `f` ? Quel est sa signature ?

```
int -> (int -> int)
```

Sans les parenthèses, cela nous donne :

```
int -> int -> int
```

Applications partielle en OCaml

```
let rec puissance x n = match n with
|0 -> 1
|n -> x * puissance x (n-1)
```

```
int -> int -> int
```

En OCaml, on peut décider de n'appliquer qu'une partie des arguments (tout en gardant l'ordre) pour obtenir une fonction des autres arguments.

```
let puissance_de_deux = puissance 2
```

```
int -> int
```

```
puissance_de_deux 3
```

Équivalent

```
let f x y z = ....
```

```
let f =
    fun x ->
        fun y ->
            fun z ->...
```

Curryfication

Définition 3 : Curryfication

La **Curryfication** est le processus qui transforme une fonction à plusieurs arguments en une fonction de son premier argument

```
let f (x, y) = x + y
```

```
int * int -> int
```

```
let f x y = x + y
```

```
int -> int -> int
```

Quand une fonction est sous cette forme, on parle de forme *curryfiée*.

Exercice

Proposer une fonction `curryfier` de signature `('a * 'b -> 'c) -> a -> b -> c` telle que `( curryfier f) x y` soit égal à `f (x, y)` pour tout `f`, `x` et `y`.

Exercice

Proposer une fonction `curryfier` de signature `('a * 'b -> 'c) -> a -> b -> c` telle que `( curryfier f) x y` soit égal à `f (x, y)` pour tout `f`, `x` et `y`.

```
1 let curryfier f = fun x y -> f (x, y)
```

Exercice

Proposer une fonction `curryfier` de signature `('a * 'b -> 'c) -> a -> b -> c` telle que `( curryfier f) x y` soit égal à `f (x, y)` pour tout `f`, `x` et `y`.

```
let curryfier f = fun x y -> f (x, y)
```

```
let curryfier f x y = f (x, y)
```

Exercice

Proposer une fonction `curryfier` de signature `('a * 'b -> 'c) -> a -> b -> c` telle que `( curryfier f) x y` soit égal à `f (x, y)` pour tout `f`, `x` et `y`.

```
let curryfier f = fun x y -> f (x, y)
```

```
let curryfier f x y = f (x, y)
```

Proposer une fonction de `decurryfier` de signature `('a -> 'b -> 'c) -> a * b -> c` telle que `( decurryfier f) (x, y)` soit égal à `f x y` pour tout `f`, `x` et `y`.

Exercice

Proposer une fonction curryfier de signature ('a * 'b -> 'c) -> a -> b -> c telle que (curryfier f) x y soit égal à f (x, y) pour tout f, x et y.

```
let curryfier f = fun x y -> f (x, y)
```

```
let curryfier f x y = f (x, y)
```

Proposer une fonction de decurryfier de signature ('a -> 'b -> 'c) -> a * b -> c telle que (decurryfier f) (x, y) soit égal à f x y pour tout f, x et y.

```
let decurryfier f = fun (x, y) -> f x y
```

Exercice

Proposer une fonction `curryfier` de signature `('a * 'b -> 'c) -> a -> b -> c` telle que `( curryfier f) x y` soit égal à `f (x, y)` pour tout `f`, `x` et `y`.

```
let curryfier f = fun x y -> f (x, y)
```

```
let curryfier f x y = f (x, y)
```

Proposer une fonction de `decurryfier` de signature `('a -> 'b -> 'c) -> a * b -> c` telle que `( decurryfier f) (x, y)` soit égal à `f x y` pour tout `f`, `x` et `y`.

```
let decurryfier f = fun (x, y) -> f x y
```

```
let decurryfier f (x, y) = f x y
```

Appel récursif

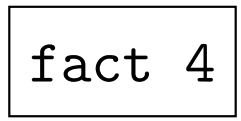
Définition 4 : Appel récursif

Un **appel récursif** est un appel de fonction effectué par la fonction elle-même.

Les appels récursifs sont mémorisés sur une pile qui permet de se souvenir des appels qui ont été faits pour le moment.

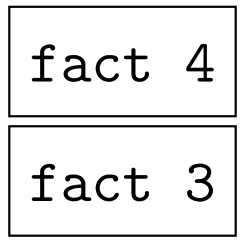
Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



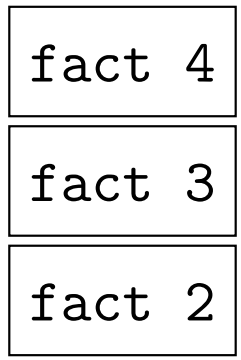
Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



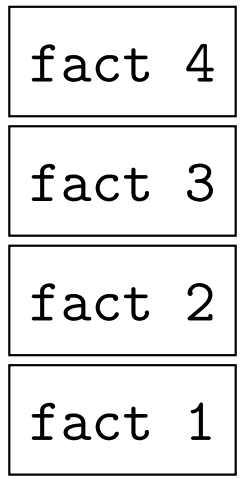
Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



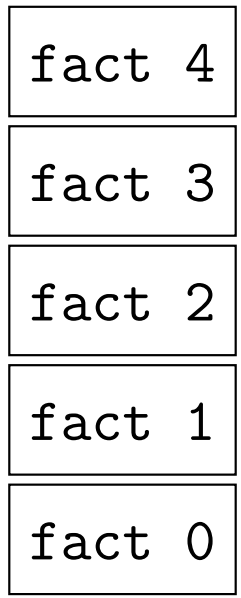
Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



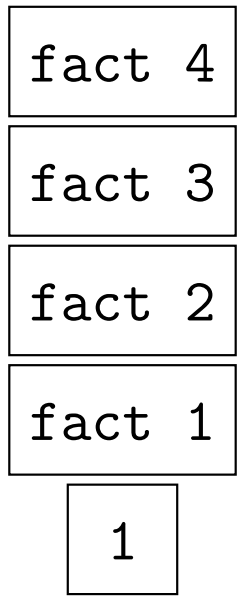
Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



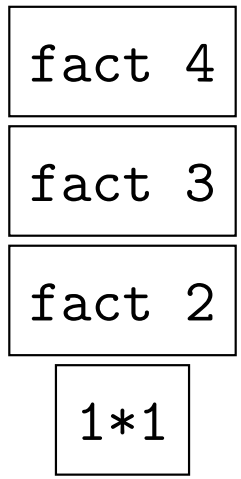
Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```



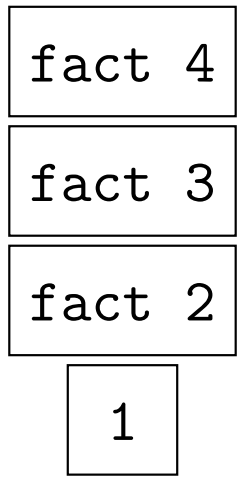
Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```



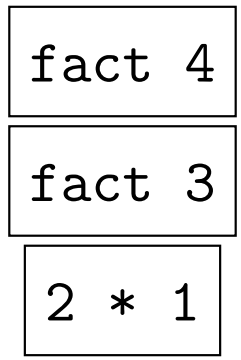
Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```



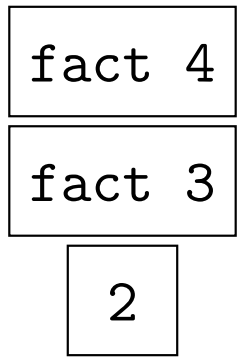
Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```



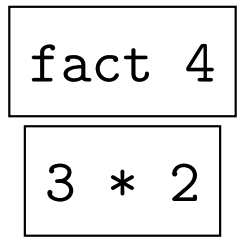
Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```



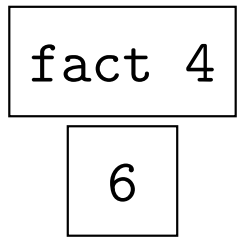
Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



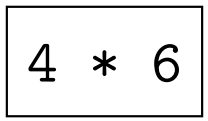
Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```



Appels récursifs

```
1 let rec fact n = match n with
2 | 0 -> 1
3 | _ -> n * fact (n - 1)
```



Appels récursifs

```
let rec fact n = match n with
| 0 -> 1
| _ -> n * fact (n - 1)
```

24

Récursivité terminale

Définition 5

Une fonction récursive est dite **récursive terminale** si la dernière chose réalisée lors de l'appel est léventuel 'appel récursif lui-même.

Le résultat du dernier appel récursif est celui du premier appel.

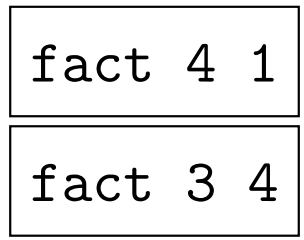
Exemple

```
1 let rec fact n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

```
fact 4 1
```

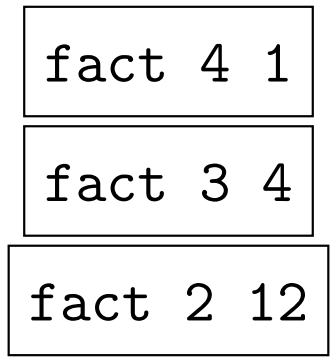
Exemple

```
1 let rec fact n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```



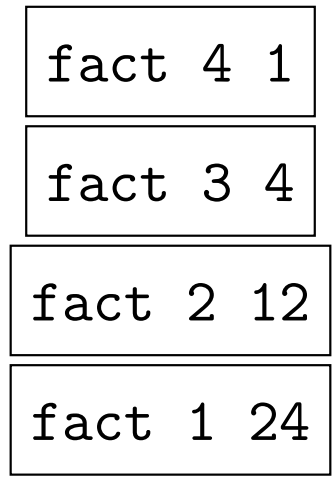
Exemple

```
let rec fact n acc = match n with
                   | 0 -> acc
                   | _ -> fact (n - 1) (acc * n)
```



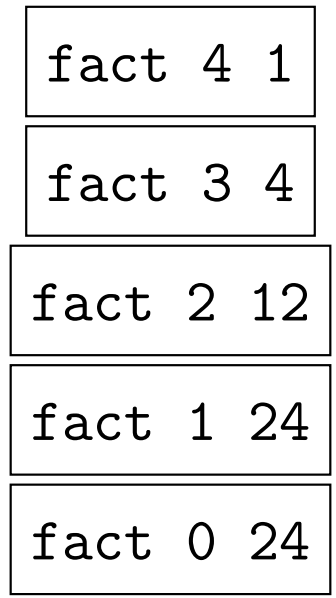
Exemple

```
let rec fact n acc = match n with
                   | 0 -> acc
                   | _ -> fact (n - 1) (acc * n)
```



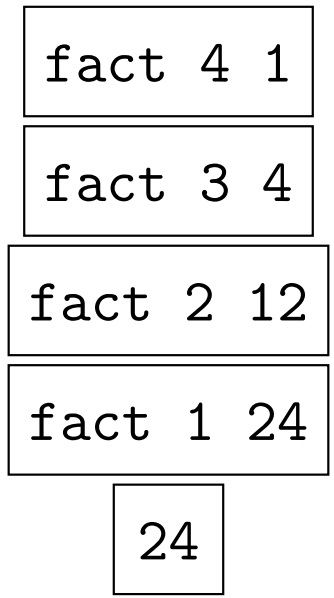
Exemple

```
let rec fact n acc = match n with
                   | 0 -> acc
                   | _ -> fact (n - 1) (acc * n)
```



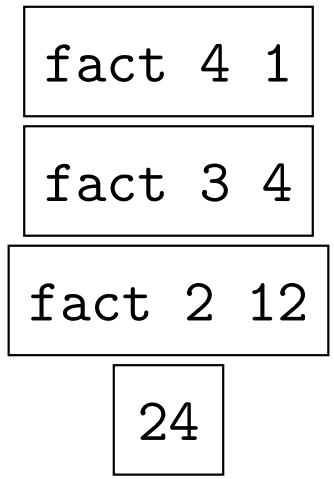
Exemple

```
let rec fact n acc = match n with
                   | 0 -> acc
                   | _ -> fact (n - 1) (acc * n)
```



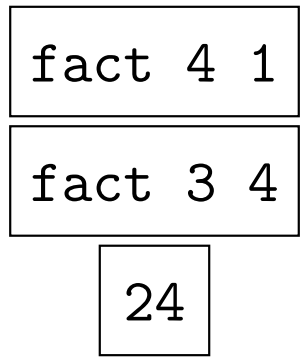
Exemple

```
let rec fact n acc = match n with
                   | 0 -> acc
                   | _ -> fact (n - 1) (acc * n)
```



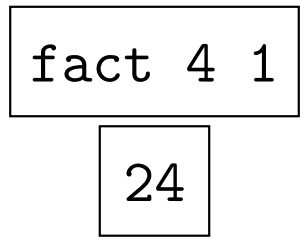
Exemple

```
1 let rec fact n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```



Exemple

```
1 let rec fact n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```



Exemple

```
1 let rec fact n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

24

Optimisation

```
1 let rec f n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

```
fact 4 1
```

Optimisation

```
1 let rec f n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

```
fact 3 4
```

Optimisation

```
1 let rec f n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

```
fact 2 12
```

Optimisation

```
1 let rec f n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

```
fact 1 24
```

Optimisation

```
1 let rec f n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

```
fact 0 24
```

Optimisation

```
1 let rec f n acc = match n with
2               | 0 -> acc
3               | _ -> fact (n - 1) (acc * n)
```

24

Différents types d'erreurs en fonction de la récursivité

```
1 let rec f x = 1 + f x
```

Différents types d'erreurs en fonction de la récursivité

```
1 let rec f x = 1 + f x
```

Dépassement de pile

Différents types d'erreurs en fonction de la récursivité

```
1 let rec f x = 1 + f x
```

Dépassement de pile

```
1 let rec f x = f x
```

Différents types d'erreurs en fonction de la récursivité

```
1 let rec f x = 1 + f x
```

Dépassement de pile

```
1 let rec f x = f x
```

Boucle infinie.