Introduction à l'algorithmique et la complexité (et un peu de CAML)

Prouvons que nos algorithmes sont corrects

Nicolas Nisse

Université Côte d'Azur, Inria, CNRS, I3S, France

Cours dispensés en MPSI (option Info) au CIV, depuis 2011-

http://www-sop.inria.fr/members/Nicolas.Nisse/lectures/prepa/

Outline

Prouver qu'un Algorithme est Correct

Un algorithme est correct si la méthode fait ce qu'exige la spécification

plus précisément, si pour toute donnée d'entrée vérifiant la précondition, la méthode

1. se termine et
2. renvoie une donnée de sortie vérifiant la postcondition.

Il y a donc **DEUX** choses à vérifier pour **prouver** qu'un algorithme est correct.

Prouver qu'un algorithme est correct demande donc DEUX étapes

Terminaison : pour un algorithme récursif, ou si il y a des boucles, la preuve est **par récurrence**, en montrant qu'il existe un convergent, i.e. une quantité qui diminue à chaque itération, vivant dans un ensemble ordonné bien fondé (où il n'existe pas de suites infinies strictement décroissantes)

Correction : de même, la preuve est **par récurrence**, en montrant qu'il existe un invariant de boucle. Intuitivement, il s'agit : d'un ensemble de propriétés qui relient les variables du programme, invariant avant, pendant et après la boucle. L'itération change la valeur des variables mais pas les relations qui les lient.

Dans ce chapitre, nous donnons de nombreux exemples (à connaître) pour illustrer les principes informellement décrits ci-dessus.

Terminaison, ordre bien fondé

Si un algorithme ne contient que des instructions séquentielles (e.g., algo. non récursif sans *For* ou *While*), il termine forcément (possiblement avec une erreur si il y a un problème de syntaxe).

L'algorithme (stupid_Algo) de droite ne termine pas !
Comprenez vous pourquoi ?

("i" augmente infiniment)

```
#let stupid_algo n =
        let i = ref (n+1) in
        while !i>n do
                i:= !i +1;
        done;
        "termine";;
stupid_algo : int -> string = <fun>
```

Pour prouver que l'algorithme termine, il faut trouver une quantité qui diminue strictement. Cela n'est pas suffisant !

Dans l'algo. stupid_Algo2 de droite, $x = (1/i)_{n \leq i}$ décroît strictement (converge vers 0) mais n'atteint pas 0 ! L'algo. ne termine pas :(

```
#let stupid_algo2 n =
        let i = ref n in
        let x = ref (1. /. float_of_int(n)) in
        while not(!x == 0.) do
                i := !i +1;
                x := 1. /. float_of_int(!i);
        done;
        "termine";;
stupid_algo2 : int -> string = <fun>
```

Il faut trouver un convergent dans un ordre bien fondé

Une relation d'ordre R sur un ensemble E non vide est bien fondée si il n'existe pas de suite infinie strictement décroissante.

Typiquement, on cherche un convergent dans $\mathbb{N}$ ou dans $\mathbb{N}^c$ ordonné par l'ordre lexicographique.

Terminaison, ordre bien fondé

Si un algorithme ne contient que des instructions séquentielles (e.g., algo. non récursif sans *For* ou *While*), il termine forcément (possiblement avec une erreur si il y a un problème de syntaxe).

L'algorithme (stupid_Algo) de droite ne termine pas !
Comprenez vous pourquoi ?

("i" augmente infiniment)

```
#let stupid_algo n =
        let i = ref (n+1) in
        while !i>n do
                i:= !i +1;
        done;
        "termine";;
stupid_algo : int -> string = <fun>
```

Pour prouver que l'algorithme termine, il faut trouver une quantité qui diminue strictement. Cela n'est pas suffisant !

Dans l'algo. stupid_Algo2 de droite, $x = (1/i)_{n \leq i}$ décroît strictement (converge vers 0) mais n'atteint pas 0 ! L'algo. ne termine pas :(

```
#let stupid_algo2 n =
        let i = ref n in
        let x = ref (1. /. float_of_int(n)) in
        while not(!x == 0.) do
                i := !i +1;
                x := 1. /. float_of_int(!i);
        done;
        "termine";;
stupid_algo2 : int -> string = <fun>
```

Il faut trouver un convergent dans un ordre bien fondé

Une relation d'ordre R sur un ensemble E non vide est bien fondée si il n'existe pas de suite infinie strictement décroissante.

Typiquement, on cherche un convergent dans $\mathbb{N}$ ou dans $\mathbb{N}^c$ ordonné par l'ordre lexicographique.

Terminaison, ordre bien fondé

Si un algorithme ne contient que des instructions séquentielles (e.g., algo. non récursif sans *For* ou *While*), il termine forcément (possiblement avec une erreur si il y a un problème de syntaxe).

L'algorithme (stupid_Algo) de droite ne termine pas !
Comprenez vous pourquoi ?

("i" augmente infiniment)

```
#let stupid_algo n =
        let i = ref (n+1) in
        while !i>n do
                i:= !i +1;
        done;
        "termine";;
stupid_algo : int -> string = <fun>
```

Pour prouver que l'algorithme termine, il faut trouver une quantité qui diminue strictement. Cela n'est pas suffisant !

Dans l'algo. stupid_Algo2 de droite, $x = (1/i)_{n \leq i}$ décroît strictement (converge vers 0) mais n'atteint pas 0 ! L'algo. ne termine pas :(

```
#let stupid_algo2 n =
        let i = ref n in
        let x = ref (1. /. float_of_int(n)) in
        while not(!x == 0.) do
                i := !i +1;
                x := 1. /. float_of_int(!i);
        done;
        "termine";;
stupid_algo2 : int -> string = <fun>
```

Il faut trouver un convergent dans un ordre bien fondé

Une relation d'ordre R sur un ensemble E non vide est bien fondée si il n'existe pas de suite infinie strictement décroissante.

Typiquement, on cherche un convergent dans $\mathbb{N}$ ou dans $\mathbb{N}^c$ ordonné par l'ordre lexicographique.

Terminaison (et Retour sur la suite de Syracuse)

En général (en tout cas, pour ce cours), ce n'est pas bien difficile de prouver qu'un programme termine (mais il fait penser à le faire !). Informellement, il faut procéder par "blocks d'instructions":

opération élémentaire: (e.g., opération arithmétique, comparaison, affectation) : termine

If XXX then YYY else ZZZ: si vous avez déjà montré que *XXX*, *YYY* et *ZZZ* terminent, alors la conditionnelle termine également.

For $i = 1$ to n do XXX: si *XXX* termine et que n est fini, alors cette boucle devrait terminer. Mais attention, il faut vérifier que *XXX* ne fait pas "n'importe quoi" avec le compteur i. Par exemple, si *XXX* décrémente i de 1 à chaque itération, alors la boucle va réaliser l'itération $i = 1$ indéfiniment.

While $i < n$ do XXX: vérifier d'abord que *XXX* termine, puis trouver un convergent (ici, il faut montrer que i va dépasser n).

Cependant, ce n'est pas toujours aussi facile :(
Par exemple, pour l'algorithme de la suite de Syracuse (slide 27 du cours précédent), on ne sait pas !!. Ça signifie qu'aucun convergent n'a été identifié, et que, pour toutes les entrées qui ont été essayées, l'algorithme a terminé (bouclé sur $(1, 4, 2, 1, \ldots)$).

Outline

Somme des n premiers entiers (algorithme itératif)

Nous avons déjà vu l'algorithme suivant (slide 26, Châpitre précédent). Prouvons qu'il est correct.

```
>       Caml Light version 0.80

#let sommeIteratif n =
        let somme_courante = ref 0 in
        for i = 1 to n do
                somme_courante := !somme_courante + i
        done;
        !somme_courante ;;
sommeIteratif : int -> int = <fun>
#
```

Terminaison : L'unique boucle ("for") réalise n itérations ($i = 1$ à n) et chaque itération est une somme ("$+i$"). Il y a donc un nombre fini d'itérations qui sont chacune finie (le *convergent* "$n - i$" diminue strictement jusqu'à 0). Donc l'algorithme termine.

Correction : Déterminer l'invariant de boucle est la partie difficile. Ici, l'*invariant* est "après la k^{me} itération ($i = k$), *somme_courante* $= \sum_{j=0}^{k} j$".

Ensuite, il faut le prouver (généralement par récurrence). Ici, récurrence sur k : après la 0^{me} itération (juste avant $i = 1$), *somme_courante* $= 0 = \sum_{j=0}^{0} j$ (**OK**) ; si après la k^{me} itération, *somme_courante* $= \sum_{j=0}^{k} j$, alors après la $(k+1)^{me}$ itération, *somme_courante* $= (\sum_{j=0}^{k} j) + k + 1 = \sum_{j=0}^{k+1} j$ (**OK**).

Pour finir, il faut montrer qu'après la dernière itération (ici la n^{me}), on obtient bien le résultat désiré. Ici : *somme_courante* $= \sum_{j=0}^{n} j$, c'est bien la somme des n premiers entiers.

Exponentiation, calcul de n^c (algorithme itératif)

Étant donnés deux entiers n et c, calculons n^c et prouvons que l'algorithme est correct. Notons que n et c sont supposés être des entiers (pré-conditions), si ce n'est pas le cas, on ne répond de rien...

```
>       Caml Light version 0.80

#let exponentiation n c =
        let produit_courant = ref 1 in
        for i = 1 to c do
                produit_courant := !produit_courant * n
        done;
        !produit_courant ;;
exponentiation : int -> int -> int = <fun>
#
```

Terminaison : L'unique boucle ("for") réalise n itérations ($i = 1$ à c) et chaque itération est un simple produit ("$*n$"). Il y a donc un nombre fini d'itérations qui sont chacune finie. Donc l'algorithme termine.

Correction : L'invariant de boucle est "après la k^{me} itération ($i = k$), $produit\_courant = n^k$".

Preuve par récurrence sur k : après la 0^{me} itération (juste avant $i = 1$), $produit\_courant = 1 = n^0$; si après la k^{me} itération, $produit\_courant = n^k$, alors après la $(k+1)^{me}$ itération, $produit\_courant = n^k * n = n^{k+1}$.

Après la dernière itération (la c^{me}), $produit\_courant = n^c$, c'est bien la résultat attendu.

Recherche de maximum dans un tableau (**vect**)

Étant donné un tableau (ici un vecteur en Caml), on cherche la valeur maximum présente dans le tableau et une position de cette valeur dans le tableau en entrée.

```
>       Caml Light version 0.80

#let maximumTab tab =
        let n = vect_length tab in
        let courant_max = ref tab.(0) in
        let courant_pos = ref 0 in
        for i = 1 to (n-1) do
                if tab.(i)> !courant_max
                then
                        begin
                        courant_max := tab.(i);
                        courant_pos := i
                        end
                else ();
        done ;
        (!courant_max,!courant_pos);;
maximumTab : 'a vect -> 'a * int = <fun>
#
```

Terminaison : L'unique boucle ("for") réalise $O(n)$ itérations ($i = 1$ à $n-1$) et chaque itération est une simple boucle conditionnelle. Il y a donc un nombre fini d'itérations qui sont chacune finie.

Correction : L'invariant de boucle est "après la k^{me} itération, *courant_max* contient la valeur maximum présente dans les k+1 premières cases de *tab*, et *courant_pos* contient l'indice d'une case ($\leq k$) de *tab* avec cette valeur". Preuve par récurrence sur k : laissée au lecteur.

Après la dernière itération, on a le résultat voulu (preuve laissée au lecteur).

Logarithme en base b, calcul de $\lceil \log_b n \rceil$

Soient 2 entiers n et $b > 1$, calculons $\lceil \log_b n \rceil$, "le plus petit entier k tel que $b^k \geq n$".

```
>       Caml Light version 0.80

#let logarithme n b =
        let log_courant = ref 1 in
        let puiss_courant = ref b in
        while !puiss_courant < n do
                puiss_courant := !puiss_courant * b ;
                log_courant :=  !log_courant + 1
        done;
        !log_courant ;;
logarithme : int -> int -> int = <fun>
#
```

Correction : ici, on prouve l'invariant de boucle avant la terminaison. L'invariant de boucle est "avant la k^{me} itération, $log\_courant = k$, $puiss\_courant = b^k$ et (si la k^{me} itération est réalisée) $b^{k-1} < n$".

Terminaison : Comme $log\_courant = k$ augmente strictement, alors, d'après l'invariant, $puiss\_courant = b^k$ aussi (car $b > 1$). Donc, les valeurs prises par $n - puiss\_courant$ sont une suite d'entiers strictement décroissante et minorée par 0 (par la condition du "while") (ordre bien fondé). Donc l'algorithme termine.

Évaluation de polynôme

1re difficulté : comprendre qu'il FAUT DÉFINIR une façon de représenter un polynôme (ici en CAML). 2de difficulté : savoir COMMENT représenter un polynôme.
Par ex., $P[X] = \sum_{i<n} a_i X^i$ est représenté par un tableau (**vect**) $[|a_0, \cdots, a_{n-1}|]$

Voici donc un algorithme qui prend en entrée un tableau $[|a_0, \cdots, a_{n-1}|]$ (représentant le polynôme $P[X] = \sum_{0 \leq i<n} a_i X^i$) et un entier y, et retourne $P(y) = \sum_{0 \leq i<n} a_i y^i$.

```
>       Caml Light version 0.80

#let evalPolynom tab y =
        let n = vect_length tab in
        let resultat_courant = ref 0 in
        for i = 0 to n-1 do
                let puiss = ref 1 in
                for j = 0 to i do
                        puiss := !puiss * y
                done ;
                resultat_courant := !resultat_courant + (tab.(i) * !puiss)
        done;
        !resultat_courant ;;
evalPolynom : int vect -> int -> int = <fun>
#
```

calcul de "y^i"

ajout de "a_i * y^i"

Terminaison : Double boucles imbriquées : le boucle "externe" (i) réalise n itérations, et chaque itération est une boucle qui réalise $j = O(n)$ itérations, chaque itŕation étant une simple opération (finie), soit au total $n * O(n) * O(1) = O(n^2)$ opérations : l'algorithme termine.

Correction : L'invariant de boucle est "après l'itération $j = \ell \leq i$ (boucle "interne") de l'itération $i = k$ de la boucle "externe", $resultat\_courant = \sum_{i=0}^{k-1} a_i y^i$ et $puiss = y^\ell$ (preuve par réc. laissée au lecteur).

Tri (par **selection**) des éléments d'un tableau

Étant donné un tableau $T = [t_0, \cdots, t_{n-1}]$ d'éléments d'un ensemble ordonné (disons d'entiers), on veut modifier la position des éléments dans T tel, qu'à la fin, les éléments de T soient ordonnés (e.g., du plus petit au plus grand) (tri sur place).

Algorithme par Sélection: Initialement, vous tenez toutes vos cartes (non triées) dans votre main droite. À chaque étape, vous prenez (selectionnez) la carte la plus petite qui se trouve dans votre main droite, et la placez "à droite" dans votre main gauche. À la fin, vos cartes sont triées par ordre croissant dans votre main gauche !!

```
#let Echange t i j =
        let tmp = t.(i) in
        t.(i) <- t.(j) ;
        t.(j) <- tmp ;;
Echange : 'a vect -> int -> int -> unit = <fun>
#let triSelection t =
        let n = vect_length t in
        for i = 0 to n-2 do
                let min_courant = ref t.(i) in
                let index_courant = ref i in
                for j=(i+1) to (n-1) do
                        if t.(j)< !min_courant then
                                begin
                                min_courant := t.(j);
                                index_courant := j;
                                end
                done;
                Echange t i !index_courant;
        done;
        t;;
triSelection : 'a vect -> 'a vect = <fun>
#let t = [| 6;8;2;9;4;7;3|];;
t : int vect = [|6; 8; 2; 9; 4; 7; 3|]
#triSelection t;;
- : int vect = [|2; 3; 4; 6; 7; 8; 9|]
```

Rappel: fonction "Echange" (slide 17, cours précédent).

Terminaison : Double boucles imbriquées.

Correction : L'invariant de boucle est "après l'itération $i = k$, les $k+1$ plus petits éléments de t ont été ordonnés aux $k+1$ premières positions de t" (preuve par réc. laissée au lecteur).

Algorithme d'Euclide

Rappelons que $pgcd(x; y) = pgcd(y; x \mod y)$ où $pgcd(x; y)$ est le plus grand diviseur commun de x et y, et $x \mod y$ est le reste de la division euclidienne de x par y.

Que fait l'algorithme suivant ?

```
#let AlgoEuclide a b =
        let x = ref a in
        let y = ref b in
        while !y > 0 do
                let tmps = !y in
                y := (!x) mod (!y);
                x := tmps;
        done;
        !x;;
AlgoEuclide : int -> int -> int = <fun>
```

Algorithme d'Euclide

Rappelons que $pgcd(x;y) = pgcd(y;x \mod y)$ où $pgcd(x;y)$ est le plus grand diviseur commun de x et y, et $x \mod y$ est le reste de la division euclidienne de x par y.

Que fait l'algorithme suivant ?

```
#let AlgoEuclide a b =
        let x = ref a in
        let y = ref b in
        while !y > 0 do
                let tmps = !y in
                y := (!x) mod (!y);
                x := tmps;
        done;
        !x;;
AlgoEuclide : int -> int -> int = <fun>
```

Terminaison : $(x;y)$ décroît strictement dans l'ordre lexicographique. En particulier, y est un entier qui décroît strictement. Comme $\mathbb{N}$ est bien fondé, y atteint 0

Correction : Par induction sur le nombre d'itération de la boucle "tant que" : l'invariant de boucle est $pgcd(a;b) = pgcd(x;y)$. Lorsque $y = 0$, l'algorithme renvoie $x = pgcd(x;0) = pgcd(x;y) = pgcd(a;b)$.

Outline

Somme des n premiers entiers (algorithme récursif)

Nous avons déjà vu l'algorithme suivant (slide 29, Châpitre précédent). Prouvons qu'il est correct.

```
>       Caml Light version 0.80

#let rec sommeRecursif n =
        match n with
        | 0 -> 0;
        | n -> n + sommeRecursif (n-1);;
sommeRecursif : int -> int = <fun>
#sommeRecursif 18;;
- : int = 171
```

Un algorithme récursif se prouve généralement (tout le temps) par récurrence !!!

Terminaison : Prouvons par récurrence sur n que l'algorithme termine. Pour $n = 0$, c'est évident. Supposons que l'algorithme termine pour $n-1$. Alors, appliqué à n, l'algorithme exécute $sommeRecursif(n-1)$ (donc termine d'après l'hypothèse de récurrence) puis fait une addition. Donc, il termine.

Correction : Par récurrence sur n, prouvons que $u_n = sommeRecursif(n) = n(n+1)/2$ ("equiv." à un invariant de boucle). Trivialement vrai pour $n = 0$ puisque $u_0 = sommeRecursif(0) = 0$. De plus, pour $n > 0$, $u_n = sommeRecursif(n) = n + sommeRecursif(n-1) = n + u_{n-1} = n + n(n-1)/2$ (par récurrence), et donc $sommeRecursif(n) = n(n+1)/2$.

Exponentiation, calcul de n^c (algorithme récursif)

Étant donnés deux entiers n et c, calculons n^c et prouvons que l'algorithme est correct.

```
>       Caml Light version 0.80

#let rec exponentiationRec n c =
        match c with
        | 0 -> 1
        | c -> n * (exponentiationRec n (c-1));;
exponentiationRec : int -> int -> int = <fun>
#exponentiationRec 10 5;;
- : int = 100000
```

Terminaison : Prouvons par récurrence sur c que l'algorithme termine. Pour $c = 0$, c'est évident. Supposons que l'algorithme termine pour $c-1$. Alors, appliqué à c, l'algorithme exécute $\mathit{exponentiationRec}(n, c-1)$ (donc termine d'après l'hypothèse de récurrence) puis fait une multiplication. Donc, il termine.

Correction : Par récurrence sur c, prouvons que $u_c = \mathit{exponentiationRec}(n, c) = n^c$. Trivialement vrai pour $c = 0$ puisque $u_0 = \mathit{exponentiationRec}(n, 0) = 1$. De plus, pour $c > 0$, $u_c = \mathit{exponentiationRec}(n, c) = n * \mathit{exponentiationRec}(n, c-1) = n * n^{c-1}$ (par récurrence), et donc $\mathit{exponentiationRec}(n, c) = n^c$.