Itérations (while)

Algo & Prog avec R

A. Malapert, B. Martin, M. Pelleau, et J.-P. Roy

4 octobre 2021

Université Côte d'Azur, CNRS, I3S, France
`firstname.lastname@univ-cotedazur.fr`

Problème → Algorithme → Programme

- Partons d'un problème.
- Soit à faire calculer la somme des entiers de $[1, n]$, avec n entier ≥ 1.

$$S = 1 + 2 + 3 + \ldots + n - 1 + n$$

- Cette somme S dépendant de n, intéressons-nous à la fonction $S(n)$.

$$S(n) = 1 + 2 + 3 + \ldots + n - 1 + n = \sum_{i=1}^{n} i$$

- Il s'agit de construire un algorithme de calcul de $S(n)$: une méthode mécanique permettant d'arriver au résultat.
- Une fois un tel algorithme obtenu, il s'agira de coder la fonction S dans un langage de programmation, ici R.
- Pour produire au final un programme exécutable sur machine.

Calculs répétitifs

Je veux calculer $S(1000)$
Faire le calcul à la main est facile, long et prodigieusement ennuyeux :
$1 + 2 + 3 + 4 + 5 + 6 + 7 + \ldots + 1000$. Ouf.

Je ne veux pas FAIRE le calcul
je veux le FAIRE FAIRE par un ordinateur (computer).
Il me faut donc écrire un programme court qui explique à la machine par quelles étapes passer. La taille du programme ne dépendra pas de n, mais le temps de calcul probablement.

Trois méthodes classiques pour construire un algorithme

- Le **calcul direct**.
- La **récurrence**.
- La **boucle**.

Les calculs répétitifs : CALCUL DIRECT !

Rarement possible, demande souvent un peu d'astuce, comme celle de Gauss vers ses 14 ans :

$$\begin{array}{rcccccccccc} S(n) = & 1 & + & 2 & + & 3 & + \ldots + & n-1 & + & n \\ S(n) = & n & + & n-1 & + & n-2 & + \ldots + & 2 & + & 1 \\ \hline 2 \times S(n) = & n+1 & + & n+1 & + & n+1 & + \ldots + & n+1 & + & n+1 \end{array}$$

En R

```
> S <- function(n) {return(n*(n+1)/2)}
> cat('S(1000)=', S(1000), '\n')
S(1000)= 500500
```

- La taille du programme (longueur de son texte) ne dépend pas de `n`, qui ne sera fourni qu'à l'exécution.
- Le temps d'exécution du calcul de `S(1000)` est immédiat. Il coûte 3 opérations.

Les calculs répétitifs : RÉCURRENCE

Appréciée des matheux, elle permet souvent d'attaquer des problèmes difficiles en . . . supposant le problème résolu !

Il s'agit de réduire le problème $S(n)$ à un problème $S(k)$ avec $k < n$. On prend souvent $k = n - 1$ ou $n \div 2$. On remarque ici que pour $n > 0$:

$$S(n) = \underline{(1 + 2 + 3 + \ldots + n - 1)} + n = S(n-1) + n$$

Si l'on sait calculer $S(n-1)$, on sait donc calculer $S(n)$. Or on sait calculer $S(1) = 1$, d'où un calcul de proche en proche :

$$S(2) = S(1) + 2 = 3 \qquad S(3) = S(2) + 3 = 6 \qquad S(4) = S(3) + 4 = 10$$

Récurrence (math) $\rightarrow$ récursivité (info)
Une fonction récursive s'appelle elle-même.

```
S <- function(n) {
  if(n <= 0) return(0)
  else return( S(n-1) + n )
}
```

Les calculs répétitifs : RÉCURRENCE

Récurrence (math) → récursivité (info)
Une fonction récursive s'appelle elle-même.

```
S <- function(n) {
  if(n <= 0) return(0)
  else return( S(n-1) + n )
}
```

- La taille du programme ne dépend toujours pas de `n`, qui ne sera fourni qu'à l'exécution.
- Le temps d'exécution du calcul de `S(1000)` est linéaire.
  Il coûte n opérations.

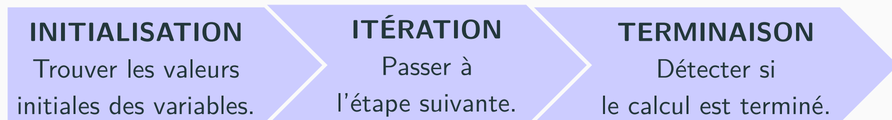
R n'encourage pas la récurrence et ne l'optimise pas !

```
> S(1000)
Erreur : évaluations trop profondément imbriquées : ré
    cursion infinie / ...
```

Plus populaire chez les programmeurs, elle tâche de présenter le calcul comme une succession d'étapes identiques portant sur des variables qui changent de valeur à chaque étape.

L'important est la situation et non l'action.

- ~~Qu'allons-nous faire ?~~
- ~~Comment procéder ?~~
- Où en sommes-nous ?
- Quelle est la situation générale ?
  - J'ai commencé à calculer $S(n)$. En plein milieu du calcul, où en suis-je ?
  - Par exemple, j'ai déjà calculé $1 + 2 + 3 + \ldots + \texttt{i}$. Introduisons une variable `acc` représentant cette accumulation. Nous gérons donc deux variables `i` et `acc`.



Construire une itération

Une fois la situation générale trouvée, voici les trois temps de la construction d'une itération :

Passer à l'étape suivante (ITÉRATION)
Étant en possession de $\texttt{acc} = 1 + 2 + \cdots + \texttt{i}$, on voudra obtenir la valeur de $1 + 2 + \ldots + \texttt{i} + (\texttt{i} + 1)$. Il suffira donc d'ajouter $\texttt{i} + 1$ à *acc* et d'ajouter 1 à `i`.

Détecter si le calcul est terminé (TERMINAISON)
On aura terminé lorsque `i` sera égal à n puisqu'alors `acc` vaudra $S(n)$.

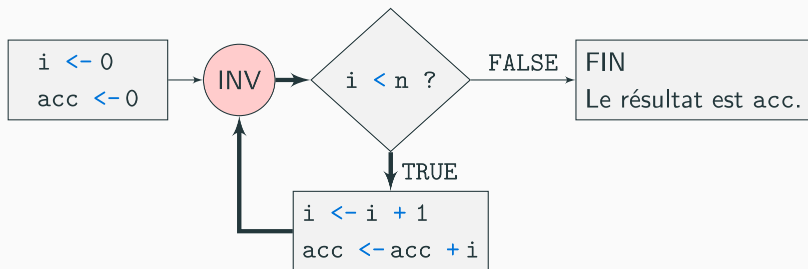
Trouver les valeurs initiales des variables (INITIALISATION)
Au début du calcul, je peux prendre $\texttt{i} = 1$ et $\texttt{acc} = 1$.

Je pourrais aussi prendre $\texttt{i} = 0$ et $\texttt{acc} = 0$ ...

Le tout, c'est que ma situation générale $\texttt{acc} = 1 + 2 + \ldots + \texttt{i}$ soit vraie en début de boucle.

Si l'une de ces étapes s'avère trop difficile, il faudra envisager de trouver une autre situation générale

Visualiser une boucle



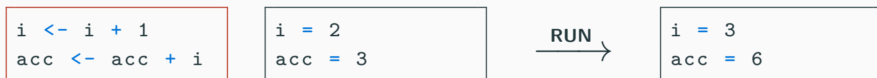
À chaque entrée dans la boucle, au point INV, la situation générale $\texttt{acc} = 1 + 2 + \cdots + \texttt{i}$ est vérifiée !

```
S <- function(n) {
  i <- 0
  acc <- 0
  while(i < n) {
    i <- i + 1
    acc <- acc + i
  }
  return(acc)
}
```

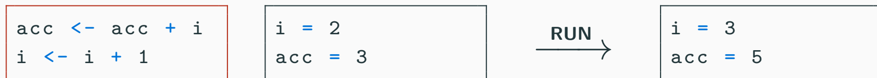
On boucle tant que $\texttt{i} < n$. Une des manières d'exprimer une boucle consiste à utiliser le mot-clé `while` qui signifie "tant que".

Commutation d'instructions

MISE EN GARDE : deux instructions ne commutent pas en général.
Les deux suites d'instructions ci-dessous ne sont PAS équivalentes.



Cette suite d'instruction serait fausse.



Il s'agit d'un point noir des boucles dans les langages impératifs.

Deux instructions peuvent commuter si elles sont indépendantes.
Or ci-dessus, l'affectation de `i` modifie la valeur de `i` dans l'affectation de `acc`.

Problème important à l'heure des processeurs à plusieurs cœurs.
Si chaque cœur modifie des variables ...

Mise au point ou debuggage

Il arrive aux meilleurs programmeurs de produire des programmes incorrects.

Comment puis-je m'aider à détecter une erreur ?
Réponse : en espionnant la boucle . . .

La méthode classique du `printf`
On fait afficher les variables de boucle (les variables qui varient dans la boucle), ici `acc` et i. Observe-t-on une anomalie ?

```
while (i < n) {
  cat('i =', i,'acc =', acc, '\n')
  ...
```

La méthode plus avancée du `browser`
La fonction `browser` interrompt l'exécution de votre programme et permet l'inspection de l'environnement d'où a été appelée `browser`.

```
while (i < n) {
  browser()
  ...
```

Comment savoir si un nombre est premier ?

Les nombres premiers 2, 3, 5, 7, 11... jouent un rôle important dans les codes secrets (cartes bancaires, mots de passe, etc).

Un entier n est toujours divisible par 1 et n, qui sont ses diviseurs triviaux. Par exemple 12 est divisible par 1 et 12 (et par d'autres) ...

Un entier n est premier s'il est ≥ 2 et si ses seuls diviseurs sont les diviseurs triviaux. Par exemple 13 est premier, mais pas 12.

Test de primalité (algorithme naïf)
Pour savoir si un nombre $n \geq 2$ est premier, il suffit donc d'examiner les nombres entiers d de $[2, n-1]$ à la recherche d'un diviseur de n.

- Si l'on en trouve un, on s'arrête au premier avec le résultat `FALSE`.
- Sinon, le résultat sera `TRUE`.

Quelle est la situation générale ?
J'en suis au nombre d que je n'ai pas encore examiné, et je n'ai toujours pas trouvé de diviseur.

Implémentation du test de primalité

```
EstPremier <- function(n) {
  if(n < 2) return(FALSE) # 0 et 1 ne sont pas premiers
  d <- 2 #le premier diviseur non trivial
  while( d < n) {
    if( n %% d == 0) return(FALSE) # Échappement
    d <- d + 1
  }
  return(TRUE)
}
```

```
> cat('2001 est premier:', EstPremier(2001), '\n')
2001 est premier: FALSE
> cat('2003 est premier:', EstPremier(2003), '\n')
2003 est premier: TRUE
```

Quel est le nombre d'opérations de notre test de primalité ?

```
> EstPremier(2**31 -1)
```

Ctrl-D ou Ctrl-C (Keyboard interrupt)

Les boucles infinies et l'instruction break

Boucle `while`

Il est essentiel de s'assurer que la boucle termine, donc que le `<test>` finit par prendre la valeur `FALSE` . Sinon le programme entre dans une boucle infinie et . . . on perd la main, il plante !

```
while (<test>) {
<instruction>
<instruction>
...
}
```

Boucle `repeat`

Pourtant certains programmeurs optent pour un style de boucle infinie dans lequel la décision d'échappement est prise parmi les instructions du corps de la boucle avec l'instruction `break`.

```
repeat {
  <instruction>
  <instruction>
  ...
}
```

```
x <- 1
while(x <= 5) {
  x <- x +1
}
```

```
x <- 1
repeat {
  if(x > 5) break
  x <- x +1
}
```

Boucle `repeat`

L'instruction break provoque une sortie brutale de la boucle, mais le programme continue son exécution après la boucle !

```
repeat {
  <instr>
  if (x > 5) break
  <instr>
}
# reprise ici
```

Quel intérêt ? Il se trouve que certaines boucles de ce type vont avoir le test de sortie en milieu ou en fin de boucle. Il pourrait même y avoir plusieurs tests de sortie . . .

```
x <- 1
repeat {
  if(x %% 11 == 0) break
  if(x %% 17 == 0) break
  x <- 2 * x +1
  print(x)
}
```

RUN →

```
[1] 3
[1] 7
[1] 15
[1] 31
[1] 63
[1] 127
[1] 255
```

Cette boucle est donc la plus générale mais elle suppose que l'on garantisse bien sa terminaison, sinon GARE !

Questions?

Retrouvez ce cours sur le site web

www.i3s.unice.fr/~malapert/R

Calcul de la somme harmonique

Somme harmonique

$$S(n) = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n-1} + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

Oresme (1360) savait déjà que $S(n)$ tend vers $+\infty$ lorsque $n \to +\infty$, les matheux disent que la série harmonique diverge.

Récurrence

```
S <- function(n) {
  if(n <=0) return(0)
  else return(S(n-1) + 1/n)
}
```

Vectorisation (plus tard)

```
S <- function(n) {
  if(n <=0) return(0)
  else return(sum(1 / (1:n))
}
```

Itération

```
S <- function(n) {
  if(n <=0) return(0)
  acc <- 1
  i <- 1
  while(i < n) {
    i <- i + 1
    acc <- acc + 1/i
  }
  return(acc)
}
```

la série harmonique diverge lentement . . .

Combien faut-il de termes dans la somme pour que $S(n) \geq 8$?

Avec la fonction `S`

```
> n <- 1
> while(S(n) < 8) {n <- n + 1}
> n
[1] 1674
```

Avec un code dédié

```
> acc <- 1;
> n <- 1;
> while(acc < 8) {
+ n <- n + 1
+ acc <- acc + 1/n
+ }
> n
[1] 1674
```

Quelle est l'approche la plus sûre ? La plus efficace ?

Et pour que $S(n) \geq 16$?

```
> while(acc < 16) {
+ n <- n + 1
+ acc <- acc +1/n
+ }
> n
[1] 4989191
```