Performances des boucles sous R

Tutoriel Tanagra

Introduction

J'assure mon cours de "Programmation R" en Master à cette époque de l'année. Lorsque j'aborde la question des boucles, je dis traditionnellement que ce n'est pas une bonne idée, le temps d'exécution étant souvent prohibitif. Je conseille aux étudiants de modifier leur code de manière à exploiter les aptitudes des fonctions de la famille des apply. Nous réalisons ensuite une série d'exercices pour voir comment ce type d'adaptation pourrait se mettre en place.

J'ai dû relativiser cette affirmation dans la période récente, parce que l'interpréteur R a énormément progressé avec les versions 3.4.0 puis 3.5.0. Depuis la 3.4.0 notamment, les boucles sont maintenant compilées à la volée (*just-in-time*), améliorant considérablement les temps de traitement sans que l'on ait à modifier en quoique ce soit notre code.

J'avais par le passé étudié les outils et astuces pour programmer efficacement sous R. Dans ce tutoriel, nous nous pencherons plus en détail sur la question des boucles en comparant les performances de la structure `for()` avec une solution passant par un `sapply()`. Je prendrai comme prétexte la programmation de l'algorithme de tri par sélection pour illustrer mon propos.

V.1 - Programmation naïve

Le tri par sélection revient à chercher les minimums successifs dans le vecteur de données. L'algorithme est de complexité quadratique, peu efficace à vrai dire. Mais ce n'est pas vraiment un problème dans notre contexte. Nous nous intéressons aux comportements des différents types de boucles sous R.

Voici une première implémentation, traduction directe du pseudo-code que l'on retrouve sur Wikipédia. La fonction prend en entrée un vecteur, et renvoie en sortie le vecteur des valeurs triées.

```
#fonction de tri par sélection
#recherche des min successifs
tri_for <- function(values){
  #nombre de valeurs
  n <- length(values)
  #boucle pour chaque min
```

```
  for (i in 1:(n-1)){
    #recherche de min. dans la portion
    #du vecteur concerné
    j_min <- i
    for (j in (i+1):n){
      if (values[j] < values[j_min]){
        j_min <- j
      }
    }
    #échange si nécessaire
    if (j_min != i){
      tmp <- values[j_min]
      values[j_min] <- values[i]
      values[i] <- tmp
    }
  }
 #renvoyer le vecteur trié
 return(values)
}
```

Nous avons deux boucles imbriquées :

- La première se charge de rapporter le i-ème minimum à chaque étape i (for i).
- La seconde recherche le minimum dans le sous-vecteur de taille (n-i) (for j).

Pour évaluer le comportement de notre fonction, nous générons un vecteur de valeurs aléatoires, que nous trions avec la fonction native de R [`sort()`] pour vérifier le bon fonctionnement de nos implémentations.

```
#générer un vecteur de valeurs aléatoires
set.seed(1)
valeurs <- runif(30000)

#tri de référence
print(system.time(ref <- sort(valeurs)))

##    user  system elapsed
##       0       0       0
```

On remarquera au passage la rapidité de la fonction `sort()`.

Nous pouvons lancer maintenant notre fonction sur le même jeu de données.

```
#trier avec des boucles for imbriquées
system.time(res_for <- tri_for(valeurs))

##    user  system elapsed
##   24.79    0.00   24.80
```

Le temps de calcul est tout autre. Mais bon, encore une fois, l'enjeu ici est de regarder la tenue des différentes solutions que nous pourrons apporter pour réaliser cette fameuse double boucle.

Une petite vérification quand-même pour s'assurer que notre implémentation est correcte.

```
#vérification du vecteur trié
print(identical(ref,res_for))
```

```
## [1] TRUE
```

Oui, notre fonction renvoie un résultat qui matche avec celui de `sort()`.

V.2 - Boucle externe avec sapply()

Il n'y a pas si longtemps encore, deux boucles imbriquées aurait été suicidaire sous R. La solution passait par l'utilisation des fonctions de la famille des `apply()`.

Dans notre cas, nous remplaçons la boucle externe (for i) par un appel de la fonction `sapply()`.

Cela nous oblige à réécrire la recherche du minimum à l'intérieur de chaque sous-vecteurs (la boucle j) sous la forme d'une fonction de rappel (fonction callback) "**`id_min`**", elle manipule en variable globale le vecteur "**`values`**". D'où l'utilisation du signe d'affectation spécifique "**`<<-`**" pour que les attributions effectuées sur le vecteur soit bien répercutée au niveau de la fonction englobante **`tri_sapply()`**.

```
#même tri, mais avec les sapply
tri_sapply <- function(values){
  #nombre de valeurs
  n <- length(values)

  #fonction interne de
  #recherche de min. et échange
  id_min <- function(i){
    j_min <- i
    for (j in (i+1):n){
      if (values[j] < values[j_min]){
        j_min <- j
      }
    }
    #échange si nécessaire
    if (j_min != i){
      tmp <- values[j_min]
      #attention -- manip. de variable globale
      #signe d'affectation spécifique avec <<
      values[j_min] <<- values[i]
      values[i] <<- tmp
    }
  }
```

```
  #appliquer la fonction
  sapply(1:(n-1),id_min)
  #renvoyer le résultat
  return(values)
}
```

Voyons ce que donne cette nouvelle implémentation sur notre vecteur des valeurs à trier.

```
#trier avec un sapply
print(system.time(res_sapply <- tri_sapply(valeurs)))

##    user  system elapsed
##   49.80    0.00   49.89

print(identical(ref,res_sapply))

## [1] TRUE
```

Déterminant il y a quelques temps encore, la modification s'avère contreproductive ici. Les appels successifs de la fonction callback force l'interpréteur à des séries de manipulations qui s'avèrent pénalisantes visiblement. Le temps de calcul est doublé.

Et auparavant ? R 3.3.3

Est-ce que R a vraiment progressé est la première question qui m'est venue à l'esprit à l'issue de cette première expérimentation. J'ai téléchargé et installé la **version 3.3.3** de R et j'ai relancé le code source ci-dessus :

- Avec la boucle externe for (**tri_for**), le temps de calcul est de 272 secondes (temps utilisateur).
- Pour la solution sapply() (**tri_sapply**), il est de 271 secondes. **Le coût des appels répétés à la fonction callback par sapply() est compensé par la rapidité de celle-ci dans l'émulation de la boucle**. Le temps global reste néanmoins dégradé parce que la fonction callback "id_min" demeure basée sur une boucle for.

Plus de doute maintenant : notre version **R 3.6.1** de ce tutoriel a considérablement progressé par rapport à la **R 3.3.3** ; la compilation à la volée a nettement amélioré le comportement des boucles dans R ; les astuces de programmation, qui étaient couramment exploitées sous R il y a peu encore (émuler une boucle avec un sapply par ex.), se révèlent moins déterminantes, dans ce contexte-ci tout du moins (soyons prudents, je vois déjà les fondus de R me tomber dessus).

V.3 - Utilisation de which.min()

Fort de cet enseignement, nous revenons à notre première solution. Pour la recherche du minimum dans le sous-vecteur de taille (n-i), nous substituons la fonction native `which.min()` à la boucle interne (for j). Cette portion du code étant appelée à de très nombreuses reprises [(n-1) fois], toute amélioration même minime peut engendrer des gains significatifs.

```
#tri_for avec le which
tri_for_which <- function(values){
  #nombre de valeurs
  n <- length(values)
  #boucle pour chaque min
  for (i in 1:(n-1)){
    #recherche de min. dans la portion
    #du vecteur concerné
    j_min <- which.min(values[i:n]) + (i-1)
    #échange si nécessaire
    if (j_min != i){
      tmp <- values[j_min]
      values[j_min] <- values[i]
      values[i] <- tmp
    }
  }
  #renvoyer le vecteur trié
  return(values)
}
```

Nous l'appliquons à notre vecteur de valeurs.

```
#trier for avec which
system.time(res_for_which <- tri_for_which(valeurs))

##    user  system elapsed
##    2.33    0.00    2.33

print(identical(ref,res_for_which))

## [1] TRUE
```

Nous avons divisé par 10 le temps d'exécution ! Comme quoi, tant que cela est possible, nous avons toujours intérêt à utiliser les fonctions natives de R qui, elles, sont compilées.

Remarque : J'ai aussi testé par ailleurs l'utilisation de `which.min()` dans la solution basée sur la boucle externe `sapply()`. Le temps de calcul est resté décevant par rapport à la boucle for i.

V.4 - Dernière optimisation

On peut toujours améliorer son code. Dans cette dernière tentative, je crée un vecteur à part pour recueillir les minimums successifs, il n'y a plus d'échanges à effectuer à l'intérieur du vecteur "**values**", ce qui me permet dans la foulée de supprimer la condition d'affectation "if (j_min != i). Est-ce que ce sera concluant ?

```
#tri_for en grattant un peu plus
tri_for_optimized <- function(values){
  #nombre de valeurs
  n <- length(values)
  #initialiser un vecteur de résultat
  res <- rep(0,n)
  #boucle pour chaque min
  for (i in 1:n){
    #recherche de min. dans la portion
    #du vecteur concerné
    j_min <- which.min(values)
    #recopier la valeur min.
    res[i] <- values[j_min]
    #supprimer cette valeur du vecteur initial
    values <- values[-j_min]
  }
  #renvoyer le vecteur trié
  return(res)
}
```

Sur notre vecteur de valeurs…

```
#trier avec tentative optimisation
system.time(res_for_optimized <- tri_for_optimized(valeurs))

##    user  system elapsed
##    2.03    0.00    2.03

print(identical(ref,res_for_optimized))

## [1] TRUE
```

… le gain n'est pas évident (un peu quand-même, 30 s sur 2mn 33 s, mais pas autant que je l'espérais) par rapport à la solution précédente. Les solutions me paraissaient intuitives pourtant.

Bien sûr, il est toujours possible d'essayer de faire mieux. Mais je sais par expérience que ce type d'exercice tourne souvent très vite à l'acharnement, et qu'il peut se faire au détriment de la lisibilité du code, un des critères importants de la programmation. J'ai décidé de m'arrêter là en ce qui me concerne.

Conclusion

"Sous R, pas de boucles !!!" est une affirmation péremptoire qu'il faut à présent savoir réviser. Elles ne sont plus aussi rédhibitoires que naguère dans les versions récentes de R (**R 3.6.1** pour ce tutoriel).

Bon, il reste que l'interpréteur ne peut pas réfléchir à notre place. Il nous appartient en tant que programmeur de coder astucieusement en tenant compte de ses spécificités. Voilà une conclusion qui me plait bien j'avoue.

Références

R. Rakotomalala, "Programmation R - Supports de cours".

Tutoriel Tanagra, "Programmer efficacement sous R", février 2019.