INFO2009-Introduction à l'informatique

Aide à la réussite - La récursivité

Bertrand Alexis

Université de Liège

2024

Rappel

Récursivité

Une fonction est récursive si elle s'appelle elle-même. Des fonctions sont récursives si plusieurs fonctions s'appellent mutuellement.

Attention

Il est possible de provoquer des récursions infinies, à l'instar des boucles infinies, c'est un morceau de code duquel on ne sort jamais. Le nombre de récusions est cependant limité par la taille de la pile de la machine.

Éviter la récursion infinie

- Donner des arguments modifiés avant chaque appel (comme pour les boucles)
- Avoir un cas de base qui ne rappelle pas la fonction.

Complexité

Tout comme pour des fonctions itératives, la complexité en temps et en espace peut être calculée.

Complexité en temps

Pour calculer la complexité en temps d'une fonction récursive, on doit s'intéresser au nombre d'appels qu'un appel à la fonction génère.
On doit donc faire attention :

- aux arguments
- au cas de base
- à l'endroit où le code doit reprendre son exécution après qu'un appel soit terminé

Complexité en espace

La complexité en espace, au lieu de regarder le temps que prend une fonction, va regarder l'espace mémoire nécessaire pour son appel.
Pour calculer la complexité en espace d'une fonction récursive, on doit regarder le nombre de variables créées ainsi que le nombre de contextes d'exécution mis sur la pile d'exécution.
On doit donc faire attention :

- à la complexité en temps (qui correspond au nombre de contextes)
- aux variables créées et invoquées

Règle simplifiée

- On utilise une case mémoire lorsqu'une variable est créée
- On utilise n cases mémoire lorsqu'un tableau de taille n est créé
- On utilise une case mémoire lorsqu'on ajoute un appel récursif à la pile d'exécution

Exercices

1. 1.1 Calculer les complexités en temps et en espace de la fonction suivante, en expliquant votre raisonnement.

```
unsigned f(unsigned n){
    if (n<=1)
        return n;
    return f(n/2) + f(n%2);
}
```

1.2 Écrire une fonction qui calcule exactement la même opération mais sans effectuer d'appel récursif.

Exercices

2. 2.1 Décrire, le plus simplement possible, l'opération effectuée par la fonction C suivante.

```
unsigned f(unsigned v){
    int n;
    if (!v)
        return 0;
    n = v%10 == 9 ? 1 : 0;
    return f(v/10) + n;
}
```

2.2 Quelle est la complexité en temps et en espace de cette fonction ?

2.3 Écrire une fonction qui calcule exactement la même opération mais sans effectuer d'appel récursif.

Exercices

3. *Le triangle de Pascal* est une construction mathématique constituée d'une séquence de lignes de longueur croissante : pour $i = 0, 1, 2, \ldots$, la ligne d'indice i contient $i + 1$ nombres entiers $C_i^0, C_i^1, \ldots, C_i^i$, définis de la façon suivante :
   - Le premier et le dernier élément de chaque ligne sont égaux à 1 : $\forall i \geq 0 : C_i^0 = C_i^i = 1$.
   - Si les lignes sont centrées les unes sur les autres, alors chaque élément autre que le premier et le dernier est égal à la somme des deux éléments de la ligne précédente qui sont situés ommédiatement à da gauche et à sa droite : $\forall 0 < j < i : C_i^j = C_{i-1}^{j-1} + C_{i-1}^j$.

Exercices

3. (suite)
   - 3.1 Écrire une fonction *pascal* prenant en argument l'indice i d'une ligne du triangle de Pascal, et retournant un pointeur vers un vecteur nouvellement alloué contenant les éléments de cette ligne. Réaliser une version itérative et une version récursive de cette fonction.
   - 3.2 Calculer la complexité en temps et en espace des deux versions de la fonction et comparer-les.

Exercices

4. 4.1 Décrire le plus simplement possible ce que calcule la fonction C suivante.

```
unsigned f(char *s, char c){
    unsigned r;
    if (!*s)
        return 0;
    r = f(s+1,c);
    if (*s == c)
        ++r;
    return r;
}
```

4.2 Quelle est la complexité en temps et en espace de cette fonction ?

4.3 Écrire une fonction réalisant exactement la même opération mais sans effectuer d'appel récursif.