РЕКУРСИЯ. РЕКУРСИВНИ ФУНКЦИИ И АЛГОРИТМИ

ВТУ, 13.11.2008 г.

СЪЩНОСТ НА РЕКУРСИЯТА

За да се разбере същността на рекурсията, трябва да се разбере каква е връзката между дефинициите и алгоритмите, които ползват тези дефиниции при реализация на програмите.

Нека дефинираме множество от n естествени числа:

1. Изброяване – $\{1, 2, \ldots, n\}$;
2. Словесно – това е множество от първите n естествени числа;
3. Друг метод – множество от цели числа, започващи с 1 и получаващи се чрез добавяне на 1 към предходния елемент.

Първите две дефиниции са окончателни, а третата не само описва множеството, но и дава алгоритъм за получаване на k-тия елемент на множеството. Такъв алгоритъм може да служи сам по себе си за описание на генерирания резултат, ако инструкциите в него подлежат на формално описание.

СЪЩНОСТ НА РЕКУРСИЯТА

Задачите, които трябва да реши подобен алгоритъм, са:

- Получаване следващия елемент на множеството;
- Проверка за принадлежност;
- Проверка дали са получени всички елементи;

Тази трета дефиниция е рекурсивна, защото тя се дефинира по отношение на самата себе си. Това определение поражда циклично действие, в което се прилага една и съща дефиниция към различни данни. Излизането от цикъла става в момента, в който множеството вече съдържа необходимия брой елементи. Това условие наричаме ***условие за край***.

С помощта на рекурсивните дефиниции е възможно само чрез няколко операции да се дефинира неограничено множество от елементи.

РЕКУРСИВНИ ДЕФИНИЦИИ

- ✓ **Прости типове данни в езика** $C++$: int $(-2^{15} \div 2^{15}-1)$, $double$, $char$ $(0 \div 255)$;

- ✓ **Директории (папки) в ОС Windows** : една директория може да съдържа една или повече директории, докато се достигне до директория, съдържаща само файлове;

- ✓ **Изчисляване на** $n!$: може да се използва формулата $n! = 1*2*3*\ldots*n$, но най-точната формула е $n! = n*(n-1)!, 0! = 1$

Рекурсивна функция:

```
int fact(int n) {
    if(n == 0)
        return 1;
    else
        return n*fact(n - 1);
}
```

Нерекурсивна функция:

```
int fact(int n) {
    int nfact = 1;
    for(int i = 2; i <= n; i++)
        nfact *= i;
    return nfact;
}
```

РЕКУРСИВНИ ФУНКЦИИ

На пръв поглед изглежда странно как една функция може да извика сама себе си и да изпълни повторно собствения си код. Компилаторът трансформира програмния код в обектен, а след това линкерът го преобразува в изпълним модул, като включва достатъчно памет за съхранение на статични и глобални променливи. Обемът на тази памет е определен по време на компилиране. ОП съдържа и динамична област (стек), която се използва, разпределя и освобождава по време на изпълнение на програмата и служи за съхраняване на динамични променливи и активиращи записи. Активиращите записи се създават от ОС при извикване на функция в програмния стек. И обратно - при излизане от функция съответния запис се унищожава и освобождава паметта.

Два вида рекурсивни функции:

- Пряка – когато в тялото на функцията има инструкция с нейното име;
- Косвена – когато в тялото ѝ съществува функция, която я извиква

ПРИМЕРИ ЗА РЕКУРСИЯ

Пряка рекурсия:

int function1(int n) {

$\quad\ldots\ldots\ldots$

$\quad$function1(m); // $m \neq n$

$\quad\ldots\ldots\ldots$

}

Непряка рекурсия:

int fun1(int n) {

$\quad\ldots\ldots\ldots$

$\quad$fun2(m);

$\quad\ldots\ldots\ldots$

}

int fun2(int a) {

$\quad\ldots\ldots\ldots$

$\quad$fun1(b);

$\quad\ldots\ldots\ldots$

}

ПРИМЕРИ ЗА РЕКУРСИЯ

Задача: Дадени са две цели числа a, b, да се намери НОД(a, b)

Алгоритъм на Евклид:

1. Ако $a = b$ – край;
2. Ако $a > b$ – стъпка 1 с числа b и $a\%b$;
3. Ако $a < b$ – стъпка 1 с числа a и $b\%a$;

```
int Euclid(int a, int b) {
    if(a == b) return a;
    else
        if(a > b)
            Euclid(b, a%b);
        else
            Euclid(a, b%a);
}
```

ПРИМЕРИ ЗА РЕКУРСИЯ

– Изчисляване на $n!$ – рекурсивното изпълнение има сложност $\Theta(n)$ и не е по-ефективно от нерекурсивното.

– Изчисляване на x^n – три рекурсивни варианта:

Вариант 1:

```
int PowerN(int x, int n) {
    if(n == 1)
        return x;
    else
        return x*PowerN(x, n - 1);
}
```

Сложността е $\Theta(n)$ и тази реализация е по-неефективна от нерекурсивната, заради допълнителното заемане на памет.

ПРИМЕРИ ЗА РЕКУРСИЯ

Вариант 2:

int PowerN(int x, int n) {
    if($n == 1$) return x;
    else
        if(x%2) return $x*$PowerN($x, n-1$);
        else return PowerN($x, n/2$)*PowerN($x, n/2$);
}

Сложност – $O(\log^2 n)$

Вариант 3:

int PowerN(int x, int n) {
    if($n == 1$) return x;
    else
        if(x%2) return $x*$PowerN($x, n-1$);
        else return PowerN($x * x, n/2$);
}

Сложност – $O(\log n)$

ЧИСЛА НА ФИБОНАЧИ

Числа на Фибоначи: $a_0 = 0, a_1 = 1, a_{n+1} = a_n + a_{n-1}$
Получената редица е: $0, 1, 1, 2, 3, 5, 8, 13, 34, 55, \ldots\ldots$

Известно е, че $\lim_{n\to\infty} a_n \approx (\frac{1+\sqrt{5}}{2})^n$

Задача: Дадено е цяло число n, да се изчисли a_n

Нерекурсивна реализация:

```
int Fibon(int n) {
    int a, b, c;
    a = 0; b = 1;
    for(int i = 2; i <= n; i++) {
        c = b + a; a = b; b = c;
    }
    return c;
}
```

Сложност – $O(n)$

НЕЕФЕКТИВНА РЕКУРСИЯ

Рекурсивна реализация:

int FibonR(int n) {
    if($n == 0 \mid\mid n == 1$)
        return n;
    else
        return FibonR($n-1$) + FibonR($n-2$);
}

Сложност – $O(??)$

РЕШЕНИЕ НА ПРОБЛЕМА

Този проблем може да се реши чрез използването на техника, наречена „меморизация“. Задачата се разбива на подзадачи (не непременно независими) и решението на всяка подзадача се запазва в специална таблица. Когато решаваме дадена подзадача, първо проверяваме в таблицата и ако вече е решена, вземаме решението директно от таблицата.

Рекурсивна реализация с меморизация:

```
int M[200];
int FibonM(int n) {
    if(n == 0 || n == 1) M[n] = n;
    else
        if(M[n] == 0) M[n] = FibonM(n − 1) + FibonM(n − 2);
}
```

Сложност – $O(n)$