Analysis and Design of Algorithms

Chapter 4: Recursive Algorithm

School of Software Engineering © Yanling Xu

Recursive Algorithm

Recursion :

a procedure or subroutine, whose implementation references itself

Example 1: Recursive evaluation of n !

Iterative Definition

$$n! = 1 \times 2 \times 3 \times \ldots \times (n - 1) \times n$$

Recursive definition

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n > 0 \end{cases}$$

initial condition

recurrence relation

Recursive Algorithm

- ***Example 1: Recursive evaluation of n ! (‘cont)***

```
Algorithm F(n)
 if n=0
     return 1                          //base case
 else
     return F (n -1) * n               //general case
```

Recursive Algorithm

Example 1: Recursive evaluation of n ! ('cont)

input size: n

basic operation: multiplication

Times of Basic operation for F(n)

$C(0)= 0$

initial condition

$C(n)= C(n-1)+1$

recurrence relation

to find the initial condition, to see when the call stop in the pseudocode

to solve recurrences,

method of backward substitutions

$$
\begin{aligned}
C(n) &= C(n-1)+1 \\
&= [C(n-2)+1]+1 = C(n-2)+2 \\
&= [C(n-3)+1]+2 = C(n-3)+3 \\
&\ldots\ldots \\
&= [C(n-n)+1]+n-1 = n
\end{aligned}
$$

can be proved by mathematical induction

Recursive Algorithm

Example 2: Fibonacci numbers

$$F(n)=\begin{cases} 1 & n=0 \\ 1 & n=1 \\ F(n-1)+F(n-2) & n>1 \end{cases}$$

Iterative Definition

$$F(n)=\frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n-\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$C(n) \in \Theta(\phi^n)$

Recursive Algorithm

Example 3: Ackerman Function

$$
\begin{cases}
A(1,0) = 2 & \\
A(0,m) = 1 & m \geq 0 \\
A(n,0) = n+2 & n \geq 2 \\
A(n,m) = A(A(n-1,m),m-1) & n,m \geq 1
\end{cases}
$$

- ***E.g.***

$$A(0,y) = y+1$$
$$A(1,y) = y+2$$
$$A(2,y) = 2y+3$$
$$A(3,y) = 2^{y+3} - 3$$
$$A(4,y) = \frac{2^{2^{\cdot^{\cdot^{2}}}}}{y+3} - 3$$

- ***power towers***

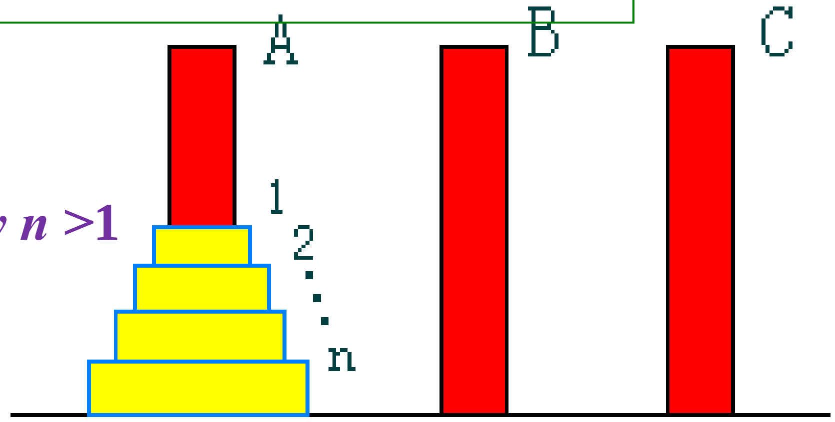
- ***Its value grows rapidly, even for small inputs. For example A(4,2) is an integer of 19,729 decimal digits***

- ***No Iterative Definition***

Recursive Algorithm

Example 4: The Tower of Hanoi Puzzle

```
void hanoi(int n, int a, int b, int c)
  {
     if (n > 0)
     {
       hanoi(n-1, a, c, b);
       move(a,b);
       hanoi(n-1, c, b, a);
     }
  }
```



$C(1) = 1$

$C(n) = 2C(n-1) + 1 = 2^n - 1$ *for every* $n > 1$

$C(n) \in \Theta(2^n)$

Recursive Algorithm

- ***Example 4: The Tower of Hanoi Puzzle ('cont)***
  - Recurrence Relations

input size: the number of disks, n

basic operation: moving one disk

total number of moving : C(n)

$$C(1) = 1$$

$$C(n) = 2C(n-1) + 1 = 2^n - 1 \quad \textit{for every } n > 1$$

$$C(n) \in \Theta(2^n)$$

$C(n) = 2C(n-1) + 1$

$= 2(2C(n-2)+1)+1 = 2^2C(n-2)+2+1 = \ldots$

$= 2^iC(n-i)+2^{i-1}+2^{i-2} + \ldots + 2+1 = \ldots$

$= 2^{n-1}C(1)+2^{n-2}+2^n-3+\ldots+2+1$

$= 2^{n-1}+2^{n-2}+2^{n-3}+\ldots+2+1$ 等比数列

$= (1-q^n)/(1-q) = (2^n-1)/(2-1) \quad = 2^n-1$

Recursive Algorithm

Example 5: permutation problem 排列问题

- Recursive algorithm to make all permutations for list $\{r_1,r_2,\ldots,r_n\}$。

  - $R=\{r_1,r_2,\ldots,r_n\}$， $R_i=R-\{r_i\}$。
  - perm(R): all permutation for all elements in R 。
  - (r_i)perm(X): to add a prefix in front of each permutation in perm(X)
  - perm(R)：

  If n=1， perm(R)=(r)， r is the only element in set R；

  If n>1， perm(R) consists of (r_1)perm(R_1)， (r_2)perm(R_2)， …， (r_n)perm(R_n) .

Recursive Algorithm

Example 5: permutation problem (‘cont)

```
template <class Type>
void Perm(Type list[], int k, int m)
{ // create all permutation of list [k: m ] with prefix list[0,k-1]
    if (k == m) {//only one element to be done
        for (int i = 0; i <= m; i++)
            putchar(list[i]);
         putchar(‘\n’);
    }
    else // several permutations for list[k: m ] are created by recursive
        for (i=k; i <= m; i++) {
            Swap (list[k], list[i]);
            Perm (list, k+1, m);
                                                //    Swap (list [k], list [i]);
        }
}
inline void Swap(Type & a,  Type & b)
{
    Type temp = a;     a = b;     b = temp; }
```

Recursive Algorithm

Important Recurrence Type

- *Decrease-by-one recurrences*
  - ***A decrease-by-one algorithm solves a problem by exploiting a relationship between a given instance of size n and a smaller size $n - 1$.***
  - ***Example: $n!$***
  - ***The recurrence equation for investigating the time efficiency of such algorithms typically has the form***

    $T(n) = T(n\text{-}1) + f(n)$
- *Decrease-by-a-constant-factor recurrences*
  - ***A decrease-by-a-constant algorithm solves a problem by dividing its given instance of size n into several smaller instances of size n/b, solving each of them recursively, and then, if necessary, combining the solutions to the smaller instances into a solution to the given instance.***
  - ***Example: binary search.***
  - ***The recurrence equation for investigating the time efficiency of such algorithms typically has the form***

    $T(n) = aT(n/b) + f(n)$

Recursive Algorithm

- *Decrease-by-one recurrences*
  - ***One (constant) operation reduces problem size by one.***

    $T(n) = T(n\text{-}1) + c \qquad\qquad T(1) = d$

    Solution: $T(n) = (n\text{-}1)c + d$ *linear*

  - ***A pass through input reduces problem size by one.***

    $T(n) = T(n\text{-}1) + c\ n \qquad\qquad T(1) = d$

    Solution: $T(n) = [n(n+1)/2 - 1]\ c + d$ *quadratic*

    $T(n) = T(n-1)+cn = T(n-2)+c(n-1)+cn = \ldots$

    $= T(1)+c[2+\ldots+(n-1)+n]$

Recursive Algorithm

- *Decrease-by-a-constant-factor recurrences*
  – The Master Theorem

$T(n) = aT(n/b) + f(n)$, where $f(n) \in \Theta(n^k)$, $k>=0$

1. $a < b^k$ $\quad T(n) \in \Theta(n^k)$
2. $a = b^k$ $\quad T(n) \in \Theta(n^k \log n)$
3. $a > b^k$ $\quad T(n) \in \Theta(n^{\log_b a})$

- ***Example:***
  - $T(n) = T(n/2) + 1$ $\quad \Theta(\log n)$
  - $T(n) = 2T(n/2) + n$ $\quad \Theta(n \log n)$
  - $T(n) = 3\,T(n/2) + n$ $\quad \Theta(n^{\log_2 3})$