Chapter 3 Simple Recursion

Yonghui Wu

School of Computer Science, Fudan University

yhwu@fudan.edu.cn

Wechat: 13817360465

Chapter 3 Simple Recursion

- 3.1 Calculation of Recursive Functions
- 3.2 Solving Problems by Recursive Algorithms
- 3.3 Sloving Recursive datum

- The programming technique of a program calling itself is called ***recursion***.
- There are two kinds of recursions:
  - direct recursion
  - indirect recursion

- A stack is used to implement a recursive process.
  - When a program invokes itself, it stores the point of return and pushes local variables of the current layer into a stack;
  - When it backtracks, it returns to the point of the current layer and pops local variables of the current layer from the stack.
- If a recursive process can't reach the end or its recursion times are too more, it will cause stack overflow.

A recursive process:

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

If n is an even number, then $f(n)$ can't reach f(1), and the program will run out of limit.

3.1 Calculation of Recursive Functions

For example, the recursive definition of factorial function $n!$ is as follow:

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

Based on the recursive definition, we can use recursive function $fac(n)$ to solve it.

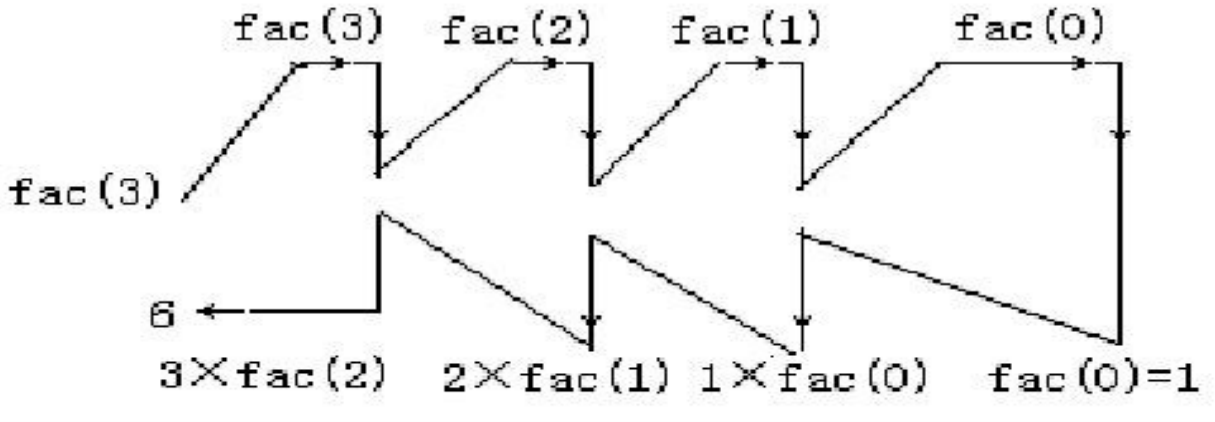
```
int   fac(int n);

{ if (n==0) return 1;                    // end condition of recursion

    if (n>=1) return n*fac(n-1);         // recursion

}
```

The recursive process of *fac*(3) is showed in Figure:



In the program, *fac*(0)=1 is called the end condition of recursion. The process *fac*(3)→*fac*(2)→*fac*(1)→*fac*(0) is a recursive process; and the process *fac*(0)→*fac*(1)→*fac*(2)→*fac*(3) is a back substitution process (*fac*(0)=1 is back substitution to *fac*(1), *fac*(1) is back substitution to *fac*(2), ……, until we calculate *fac*(3)=6).

Hints for recursion

- A recursive process can be directly implemented by a program based on the definition of recursive functions.
- For a complex problem, if it can be decomposed into several relative simple subproblems, these subproblems' solutions are same or similar, and the original problem can be solved as long as these subproblems are solved, then it is a recursive solution.
- When the subproblems can be solved directly, the decomposition terminates. Such subproblems are called the end conditions of recursion. If the recursive function can't reach the end conditions, then the program will fail due to stack overflow.

3.2 Solving Problems by Recursive Algorithms

- If the initial states and the goal states are showed in the problem description, and rules and constraints for the expended states are the same, then a recursive algorithm can be used to find a solution from the initial states to the goal states.

- How to represent a state with value parameters or local variables in decomposition should be noticed to recover its original state in backtracking.
- If storage of these parameters is large (such as array) and initial values should be passed by the main program, in order to avoid memory overflow, these variables must be set as global variables, and before backtracking it is required to restore its original value before recursion.
- The end conditions of recursion should be determined.
- The search area and constraints should be determined. That is, in what situation the end conditions can’t be reached, and in what area search is done, and the recursive process can continue.

3.2.1 Red and Black

- **Source: ACM Japan 2004 Domestic**
- **IDs for Online Judge: POJ 1979**

- There is a rectangular room, covered with square tiles. Each tile is colored either red or black. A man is standing on a black tile. From a tile, he can move to one of four adjacent tiles. But he can't move on red tiles, he can move only on black tiles. Write a program to count the number of black tiles which he can reach by repeating the moves described above.

- **Input**
- The input consists of multiple data sets. A data set starts with a line containing two positive integers *W* and *H*; *W* and *H* are the numbers of tiles in the x- and y- directions, respectively. *W* and *H* are not more than 20.
- There are H more lines in the data set, each of which includes W characters. Each character represents the color of a tile as follows.
- '.' - a black tile
- '#' - a red tile
- '@' - a man on a black tile(appears exactly once in a data set)
- The end of the input is indicated by a line consisting of two zeros.
- **Output**
- For each data set, your program should output a line which contains the number of tiles he can reach from the initial tile (including itself).

Analysis

- Recursive method (backtracking) can be used to count the number of black tiles that the man can reach.

- Suppose
- *n* and *m* are the numbers of tiles in the x-direction and y-direction, respectively;
- *ans* is the number of black tiles which the man reach, and its initial value is 0;
- *map* is to represent the rectangular room covered with square tiles, where *map*[*i*][*j*] is a character to represent the tile whose positions in the x-direction and y-direction are *i* and *j* respectively;
- *visited* is visited marks for the man, where *visited*[*i*][*j*]==true means the man has reached the tile whose position is (*i*, *j*).

- The recursive function is *search*(i, j), where
  - State: The man's current position is (i, j).
  - The position before the recursion is the position of initial tile.

- The recursive function is *search*(*i*, *j*), where
  - End condition of recursion:
  - If the current position is out of the rectangular room (*i*<0||*i*>= *n*||*j*<0||*j*>=*m*) , or can't pass (*map*[*i*][*j*] == '#'), or the position has been visited (*visited*[*i*][*j*]==true), then backtrack;
  - Otherwise, for the current position (*i*, *j*), *visited*[*i*][*j*]=true, the number of tiles that the man has reached increases 1 (++*ans*), and then continue recursion.

- The recursive function is *search*(*i*, *j*), where
  - Search area: for the current position (*i*, *j*), recursively search the four neighbor positions (*search*(*i*-1, *j*); search(*i*+1, *j*); search(*i*, *j*-1); search(*i*, *j*+1);).

3.3 Solving Recursive datum

- When we construct a mathematical model for a problem, sometimes we find its data structure is in a recursive form.

3.3.1 Symmetric Order

- **Source: ACM Mid-Central USA 2004**
- **IDs for Online Judge: POJ 2013, ZOJ 2172**

- In your job at Albatross Circus Management (yes, it's run by a bunch of clowns), you have just finished writing a program whose output is a list of names in nondescending order by length (so that each name is at least as long as the one preceding it). However, your boss does not like the way the output looks, and instead wants the output to appear more symmetric, with the shorter strings at the top and bottom and the longer strings in the middle. His rule is that each pair of names belongs on opposite ends of the list, and the first name in the pair is always in the top part of the list. In the first example set below, Bo and Pat are the first pair, Jean and Kevin the second pair, etc.

- **Input**
- The input consists of one or more sets of strings, followed by a final line containing only the value 0. Each set starts with a line containing an integer, *n*, which is the number of strings in the set, followed by *n* strings, one per line, sorted in nondescending order by length. None of the strings contain spaces. There is at least one and no more than 15 strings per set. Each string is at most 25 characters long.
- **Output**
- For each input set print "SET *n*" on a line, where *n* starts at 1, followed by the output set as shown in the sample output.

Analysis

- The list of names in nondescending order by length is *s*[1]…*s*[*n*]. The format of output is symmetric, with the shorter strings at the top and bottom, and the longer strings in the middle. Lengths of names in the upper half part are ascending, and lengths of names in the lower half part are descending. There are two solutions to the problem: non-recursive method and recursive method.

[1] Non-recursive Method

- The input consists of one or more sets of strings sorted in nondescending order by length. The output is symmetric, with the shorter strings at the top and bottom and the longer strings in the middle. So the upper half of output is as follow:
  - *s*[1]
  - *s*[3]
  - *s*[5]
  - ……
  - *s*[*n*], if *n* is an odd number; or *s*[*n*-1], if *n* is an even number.
- That is, for statement " for (int *i*=1; *i*<=*n*; *i*+=2) cout<<*s*[*i*]<<endl;" will implement the upper half of output.
- The lower half of output is as follow:
  - *s*[*n*-(*n*%2)]
  - *s*[*n*-(*n*%2)-2]
  - *s*[*n*-(*n*%2)-4]
  - ……
  - *s*[2]
- That is, for statement " for (int *i*= *n*-(*n*%2); *i*>1; *i*-=2) cout<<*s*[*i*]<<endl;" will implement the lower half of output.

[2] Recursive Method

Firstly the program outputs $s[1]s[3]\ldots s[n]$, if n is odd; or the program outputs $s[1]s[3]\ldots s[n-1]$; if n is even. Then the program outputs $s[n]s[n-2]\ldots s[2]$, if n is even; or $s[n-1]s[n-3]\ldots s[2]$, if n is odd.

n strings are divided into $\left\lceil \frac{n}{2} \right\rceil$ groups, and each group contains two adjacent strings $s[1]s[2]$, $s[3]s[4]$, …, and so on.

If $s[k]s[k+1]$ $(1 \le k < n)$ is in a group, firstly input the first parameter $s[k]$ and output it, then input the second parameter $s[k+1]$ and push it into a stack. After the last group is inputted, elements in the stack are popped and outputted. It can be implemented by recursive function *print*(n).

- The process for *print*(*n*) is as follow:
  - Input and output the first string of the current group *s*[*k*]; *n*--;
  - If *n*>0, then input the second string of the current group *s*[*k*+1] and push it into a stack; *n*--. If *n*>0, then recursively call *print*(*n*) until *n*==0.
  - In backtrack, *s*[*n*-1]*s*[*n*-3]…s[2], if *n* is even; *s*[*n*]*s*[*n*-2]… s[2], if *n* is odd; are popped from the stack and outputted.