Data Structure and Algorithms

Chapter 5 Recursion

Dr. Zhiqiang Liu

School of Software and Microelectronics, Northwest Polytechnical University

Table of Contents

Next Section

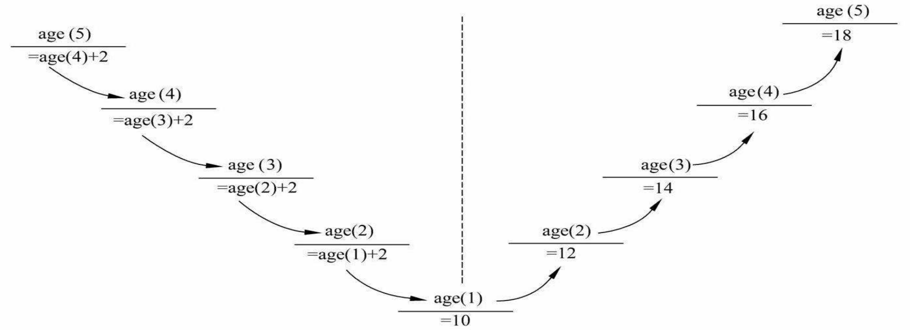
Next Subsection

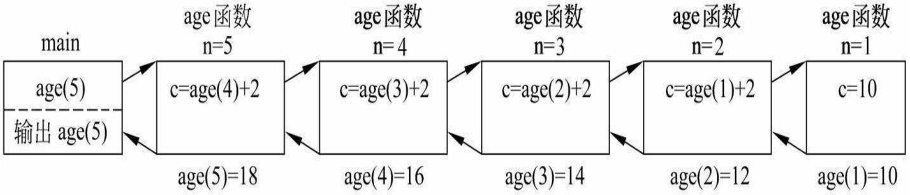
5.1.1 Example

- To calculate the result of age(5)

$$age(n) = \begin{cases} 10, & \text{n=1} \\ age(n-1)+2, & \text{n>1} \end{cases}$$

5.1.1 Example


age (5)
=age(4)+2
age (4)
=age(3)+2
age (3)
=age(2)+2
age(2)
=age(1)+2
age(1)
=10
age(2)
=12
age(3)
=14
age(4)
=16
age (5)
=18


main
age(5)
输出 age(5)
age函数
n=5
c=age(4)+2
age(5)=18
age函数
n=4
c=age(3)+2
age(4)=16
age函数
n=3
c=age(2)+2
age(3)=14
age函数
n=2
c=age(1)+2
age(2)=12
age函数
n=1
c=10
age(1)=10

Next Subsection

5.1.2 Definition of recursion

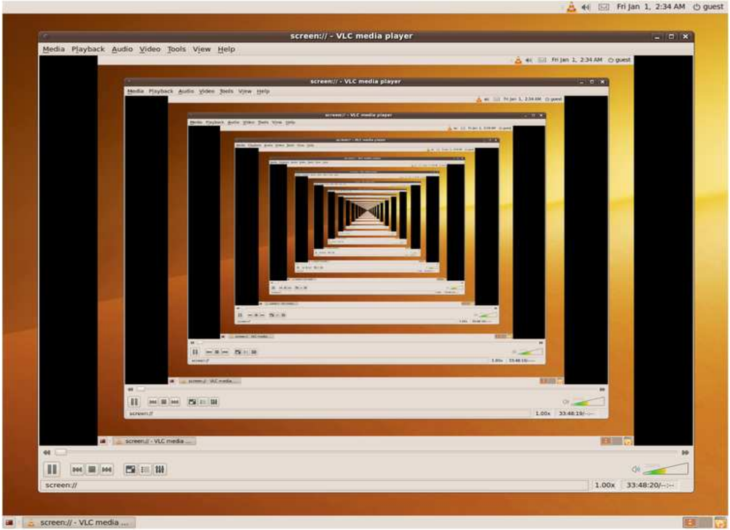
- A recursive function is a function that calls itself.
- It is an important problem-solving tool in computer science and mathematics.
- It is used in programming languages to define language syntax and in data structures to develop searching and sorting algorithms for list and tree structures

5.1.2 Definition of recursion

Recursive image of a clock's face

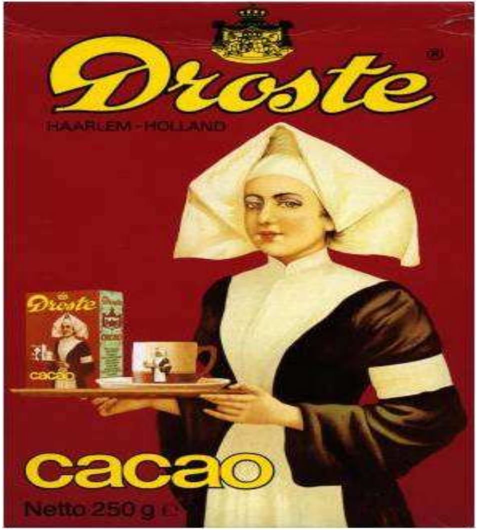
5.1.2 Definition of recursion

Screenshot Recursion



5.1.2 Definition of recursion

How about this picture?



5.1.2 Definition of recursion

Types of recursion

- Recursive definition
  - Factorial function

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

```
// Recursive algorithm for Factorial
long Factorial(long n)
{
 if(n==0) return 1;
 else return n*Factorial(n-1);
}
```

5.1.2 Definition of recursion

Rucursion

- Three typically recursions
  - Recursive definition
  - Recursive structures
  - Recursive solutions for the specific problems

Types of recursion

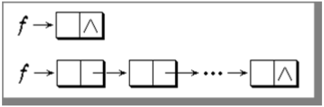
- Recursive definition
  - Fibonacci numbers

$$Fib(n)! = \begin{cases} n, & \mathrm{n} = 0{,}1 \\ Fib(n-1) + Fib(n-2), & \mathrm{n} > 1 \end{cases}$$

```
// Recursive algorithm for Fibonacci array
long Fib(long n)
{
 if(n <= 1)
    return n;
 else
    return Fib(n-1)+Fib(n-2);
}
```

Types of recursion

- Recursive structures
  - Linked list

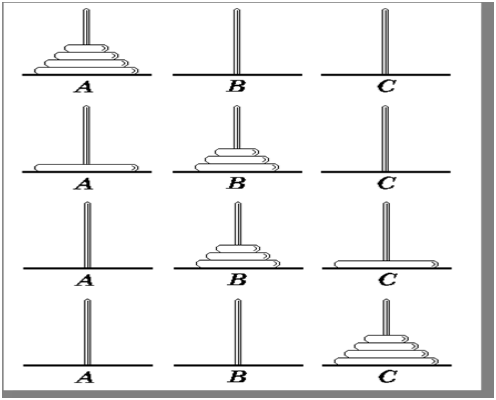


//Traverse linked list and print out the last node

```
template <class Type>
void FindLastData( ListNode<Type> *f ){
    if(f->link == NULL)
        cout << f->data << endl;
    else FindLastData(f->link);
}
```

Types of recursion

- Recursive problems
  - The tower of Hanoi
  - Rules
    1) Move only one disk at a time.
    2) No larger disk can be on top of a smaller disk.



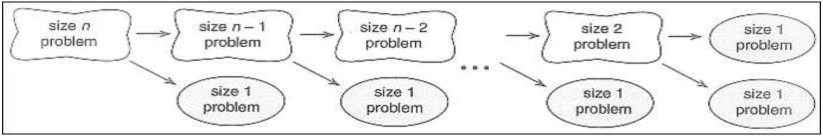
Next Section

5.2 Principles of Recursion

- Problems that can be solved using recursion have the following characteristics:
  - One or more simple cases of the problem have a direct and easy answer – also called base cases.Example: 0! = 1.
  - The other cases can be re-defined in terms of a similar but smaller problem - recursive cases. Example: n! = n (n-1)!
  - By applying this re-definition process, each time the recursive cases will move closer and eventually reach the base case. Example:

$$n! \rightarrow (n-1)! \rightarrow (n-2)! \rightarrow \ldots 1!, 0!. \tag{2.1}$$

- The strategy in recursive solutions is called divide-and-conquer. The idea is to keep reducing the problem size until it reduces to the simple case which has an obvious solution.



5.2 Principles of Recursion

- Designing recursive algorithm
  - Outline your algorithm.
    Combine the stopping rule and the key step, using an if statement to select between them.
  - Find the key step.
    Begin by asking yourself , “How can this problem be divided into parts?” or “How will the key step in the middle be done?”
  - Find a stopping rule.
    This stopping rule is usually the small, special case that is trivial or easy to handle without recursion.
  - Check termination.
    Verify that the recursion will always terminate. Be sure that your algorithm correctly handles extreme cases.
  - Draw a recursion tree.
    The height of the tree is closely related to the amount of memory that the program will require, and the total size of the tree reflects the number of times the key step will be done.

Next Section

5.3 Recursion vs Iteration

- Recursion

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

$$Fib(n)! = \begin{cases} n, & \text{n = 0,1} \\ Fib(n-1)+Fib(n-2), & \text{n > 1} \end{cases}$$

- Iteration

n!=1*2*3*...*(n-1)*n

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2.$$

5.3 Recursion vs Iteration

Calculating factorials

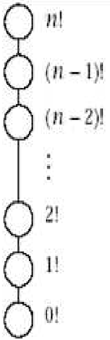
- Recursive algorithm

```
int factorial(int n)
/*    factorial: recursive version
Pre:  n is a nonnegative integer.
Post: Return the value of the factorial of n.
*/
{
    if (n == 0)   return 1;
    else return  n * factorial(n - 1);
}
```

5.3 Recursion vs Iteration

Example of 5!

$$
\begin{aligned}
\text{factorial}(5) &= 5 * \text{factorial}(4) \\
&= 5 * (4 * \text{factorial}(3)) \\
&= 5 * (4 * (3 * \text{factorial}(2))) \\
&= 5 * (4 * (3 * (2 * \text{factorial}(1)))) \\
&= 5 * (4 * (3 * (2 * (1 * \text{factorial}(0))))) \\
&= 5 * (4 * (3 * (2 * (1 * 1)))) \\
&= 5 * (4 * (3 * (2 * 1))) \\
&= 5 * (4 * (3 * 6)) \\
&= 5 * (4 * 6) \\
&= 5 * 24 \\
&= 120.
\end{aligned}
$$



5.3 Recursion vs Iteration

Calculating factorials

- Non-recursive algorithm

```
int factorial(int n)
/*    factorial: iterative version
Pre:  n is a nonnegative integer.
Post: Return the value of the factorial of n.
*/
{   int count, product = 1;
    for (count = 1; count <= n; count++)
        product *= count;
    return product;
}
```

5.3 Recursion vs Iteration

Calculating fibonacci numbers with Recursive function

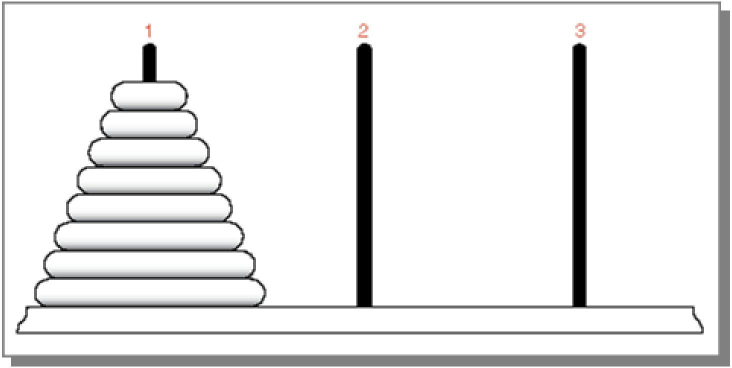
```
int fibonacci(int n)
/*    fibonacci: recursive version
Pre:  The parameter n is a nonnegative integer.
Post: The function returns the nth Fibonacci number. */
{  if (n <= 0)  return 0;
   else if (n == 1)  return 1;
   else   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Calculating fibonacci numbers with Non-recursive function

```
int fibonacci(int n)
{
   int last_but_one;   //  Fibonacci number, F_i-2
   int last_value;     //  Fibonacci number, F_i-1
   int current;        //  current Fibonacci number F_i
   if (n <= 0) return 0;
   else if (n == 1) return 1;
   else {
      last_but_one = 0;
      last_value = 1;
      for (int i = 2; i <= n; i++) {
         current = last_but_one + last_value;
         last_but_one = last_value;
         last_value = current;
      }
      return current;
   }
}
```

Next Section

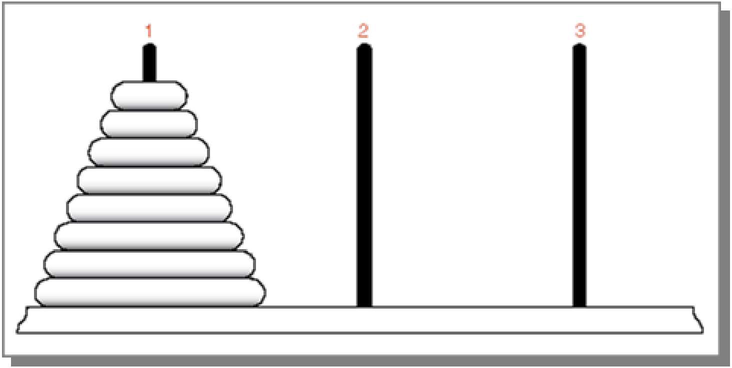
5.4 The tower of Hanoi



- Rules
  - Move only one disk at a time.
  - No larger disk can be on top of a smaller disk.

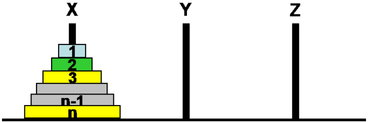
5.4 The tower of Hanoi

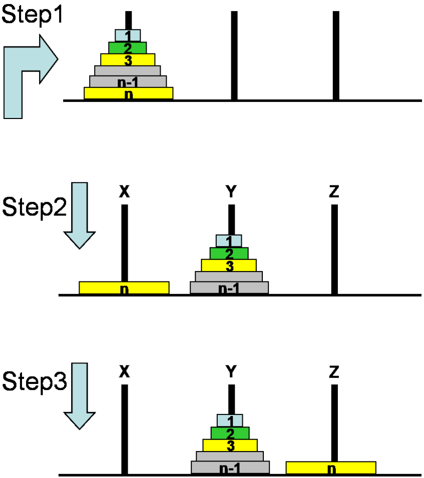
- The initial idea is to concentrate our attention not on the first step (which must be to move the top disk), but rather on the hardest step: moving the bottom disk.



5.4 The tower of Hanoi

Divide & conquer





5.4 The tower of Hanoi

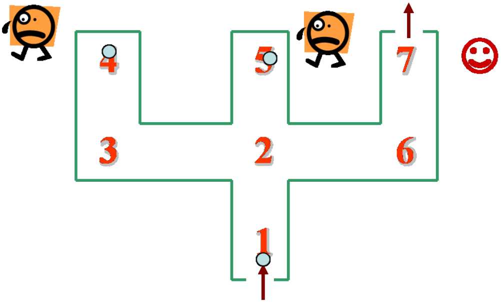
Hanoi recursive function

```
void move(int count, int start, int finish, int temp)
{
    if (count > 0) {
      move(count - 1, start, temp, finish);
      cout <<''Move disk'' << count <<
      ''from'' << start << ''to'' <<
      finish << ''.'' << endl;
      move(count - 1, temp, finish, start);
   }
}
```

Next Section

5.5 The Maze

- Problem
  - A maze is a set of intersections. The traveler enters from one direction and departs along one of three paths: to the left, straight ahead or to the right. A path is identified by the number of the next intersection.



5.5 The Maze

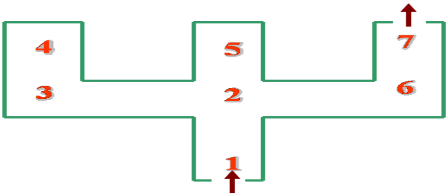
Backtracking

- Solution: backtracking
  - In an effort to obtain a final solution to the maze, we create a series of partial solutions step by step that appear to be consistent with the requirements of the final solution. If we take a step or make a decision that is inconsistent with a final solution, we backtrack one or more steps to the last consistent partial solution.

5.5 The Maze

Intersections	Action	Result
1 (Entrance)	straight	enter 2
2	to left	enter 3
3	to right	enter 4
4 (Dead-end)	backtrack	enter 3
3 (Dead-end)	backtrack	enter 2
2	straight	enter 5
5 (Dead-end)	backtrack	enter 2
2	to right	enter 6
6	to left	enter 7(end)

Mini Maze



Intersections structure

```
struct Intersection {
     int left;
     int forward;
     int right;
}
```

Data for mini Maze

6		
0	2	0
3	5	6
0	0	4
0	0	0
0	0	0
7	0	0
7		

Data for mini Maze

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
class Maze {
private:
    int MazeSize;
    int EXIT;
    Intersection *intsec;
public:
    Maze (char *filename);
    int TraverseMaze(int CurrentPos);
}
```

Maze class implementation

```
Maze :: Maze ( char *filename ){
// build maze by reading intersections and the exit
//intersection number from filename
    ifstream fin;
    fin.open ( filename, ios::in | ios::nocreate );
    if ( !fin ) {
cout<<"The maze data file"<<filename<<"cannot be opened"
   <<endl;
        exit (1);}
    fin >> MazeSize;    //the number of intersections
    intsec = new Intersection[MazeSize+1];
    for ( int i = 1; i <= MazeSize; i++ )
       fin>>intsec[i].left>>intsec[i].forward>>intsec[i
           ].right;
    fin >> EXIT; //the number of the exit intersection
    fin.close ( );
}
```

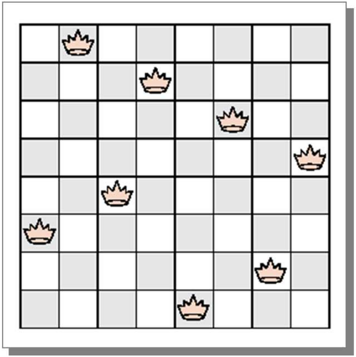
Solution of the maze using backtracking

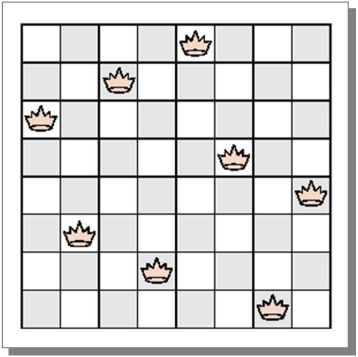
```
int Maze::TraverseMaze ( int CurrentPos ) {
   if ( CurrentPos > 0 ) {  //from the intersection 1
       if ( CurrentPos == EXIT ) { //stopping condition
          cout << CurrentPos << " ";  return 1;
       } else  //attemp to go left
       if (TraverseMaze(intsec[CurrentPos].left ))
          { cout << CurrentPos << “”   ;  return 1; }
       else //left leads to dead end. Try going straight
       if (TraverseMaze(intsec[CurrentPos].forward))
          { cout << CurrentPos << “”   ;  return 1; }
       else //left, straight lead to dead end. Try going right
       if (TraverseMaze(intsec[CurrentPos].right))
          { cout << CurrentPos << “”   ;  return 1;  }
    }
    return 0;   // at a dead end
}
```

Next Section

5.6 Eight Queens Puzzle

- Problem



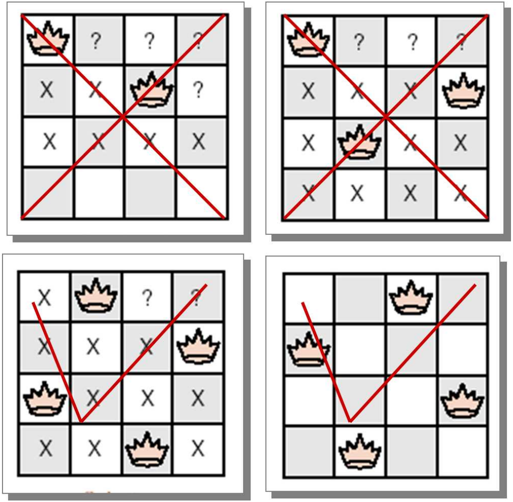


5.6 Eight Queens Puzzle

Example

Four queens

Dead end
Solution



5.6 Eight Queens Puzzle

Program outline

```
solve_from (Queens configuration)
  if Queens configuration already contains eight queens
     print configuration
  else
     for every chessboard square p that is unguarded by configuration {
          add a queen on square p to configuration;
          solve_from(configuration);
          remove the queen from square p of configuration;
      }
```

Main Program

```
int main()
{  int board_size;
   print_information();
   cout << “What is the size of the board? ” << flush;
   cin  >> board_size;
   if (board_size < 0 || board_size > max_board)
      cout << “The number must be between 0 and ” <<
             max_board << endl;
   else {
      //   Initialize empty configuration.
      Queens configuration(board_size);
      //  Find all solutions extending configuration.
      solve_from(configuration);
   }
}
```

5.6 Eight Queens Puzzle

Methods specification

bool Queens :: unguarded(**int** col) **const**;

Post: Returns **true** or **false** according as the square in the first unoccupied row (row count) and column col is not guarded by any queen.

void Queens :: insert(**int** col);

Pre: The square in the first unoccupied row (row count) and column col is not guarded by any queen.

Post: A queen has been inserted into the square at row count and column col; count has been incremented by 1.

5.6 Eight Queens Puzzle

Methods specification

void Queens :: remove(**int** col);

Pre: There is a queen in the square in row count − 1 and column col.

Post: The above queen has been removed; count has been decremented by 1.

bool Queens :: is_solved( ) **const**;

Post: The function returns **true** if the number of queens already placed equals board_size; otherwise, it returns **false**.

The Backtracking Function

```
void solve_from(Queens &configuration)
// Pre:  The Queens configuration represents a partially
// completed arrangement of non-attacking queens on a chessboard.
// Post: All n-queens solutions that extend the given
// configuration are printed. The configuration is restored
// to its initial state. Uses: The class Queens and the
// function solve_from, recursively.
{
   if (configuration.is_solved()) configuration.print();
   else
      for (int col = 0; col < configuration.board_size; col++)
         if (configuration.unguarded(col)) {
            configuration.insert(col);  // Recursively continue to add queens.
            solve_from(configuration);
            configuration.remove(col);
         }
```

```
}
```

The First Data Structure

```
const int max_board = 30;
class Queens {
public:
   Queens(int size);
   bool is_solved() const;
   void print() const;
   bool unguarded(int col) const;
   void insert(int col);
   void remove(int col);
   int  board_size; //   dimension of board = maximum number of queens
private:
   int  count;            //   current number of queens = first unoccupied row
   bool queen_square[max_board][max_board];
};
```

Methods implementation

```
Queens::Queens(int size)
/*
Post: The Queens object is set up as an empty
    configuration on a chessboard with size squares in
    each row and column.
*/
{
   board_size = size;
   count = 0;
   for (int row = 0; row < board_size; row++)
      for (int col = 0; col < board_size; col++)
         queen_square[row][col] = false;
}
void Queens::insert(int col)
// Pre:  The square in the first unoccupied row (row
    count) and column col
//          is not guarded by any queen.
// Post: A queen has been inserted into the square at
    row count and
//          column col; count has been incremented by 1.
```

```
{
   queen_square[count++][col] = true;
}
void Queens::remove(int col)
// Pre:  The square in the current row (row count) and column col is
//           guarded by the other queen.
// Post: The queen has been removed from the square at row count and
//           column col; count has been decremented by 1.
{
   queen_square[count--][col] = false;
}
bool Queens::unguarded(int col) const
// Post: Returns true or false according as the square in the first unoccupied row (row count) and column col is not guarded by any queen.
{
   int i;
   bool ok = true; //   turns false if we find a queen in column or diagonal

```

```
    for (i = 0; ok && i < count; i++)
        ok = !queen_square[i][col];                    // Check upper part of column
    for (i = 1; ok && count - i >= 0 && col - i >= 0; i++)
        ok = !queen_square[count - i][col - i];  // Check upper-left diagonal
    for (i = 1; ok && count - i >= 0 && col + i < board_size; i++)
        ok = !queen_square[count - i][col + i];  // Check upper-right diagonal

    return ok;
}
```

Next Section

5.7 Summary and QUIZ

5.7 Summary and QUIZ

1、执行下列程序后，i的值为（ ）。

```
int f(int x) { return (x>0) ? x*f(x-1): 2; }
main(){
    int i = f(f(1));
}
```

2、设商店有数量充足的10元、5元、2元和1元的零币，如果售货员找零时要求零币数量越少越好，设计计算零币数量的选币算法。

```
int changes(int iAmount){    }
```

3、Homework Title:(背包问题) 假设有一个能装入总体积为T的背包和n件体积分别为w1 , w2 , . . . , wn 的物品，能否从n件物品中挑选若干件恰好装满背包，即使w1 +w2 + . . . + wm=T，要求找出所有满足上述条件的解。例如：当T=10，各件物品的体积1, 8, 4, 3, 5, 2时，可找到下列4组解：(1,4,3,2);(1,4,5);(8,2);(3,5,2)

5.7 Summary and QUIZ

(1) 分析背包问题，得到数学模型

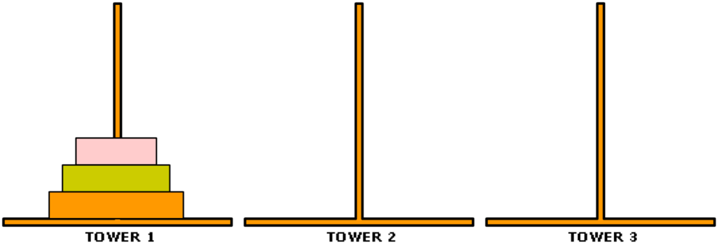
$$Knap(t,n)! = \begin{cases} 1 & \text{t=0} \\ 0 & \text{t<0} \\ 0 & \text{t>0;n<1} \\ knap(t,n-1)\text{或}knap(t,n-1) & \text{t>0,n>=1} \end{cases}$$

(2) 设计算法：递归算法

(3) 程序设计：

5.7 Summary and QUIZ

Tower of Hanoi



5.7 Summary and QUIZ

Procedure of function calling
(函数调用的过程)

Befor calling(调用前):

(1) Pass the parameters and return address to the sub-function (将所有的实参、返回地址传递给被调用函数保存)
(2) Allocate memory for local variables (为被调用函数的局部变量分配存储区)
(3) Jump to the entrance of the sub-function(控制转移到被调用函数入口)

After calling(调用前):

(1) Save the results(保存被调用函数的计算结果)
(2) Release the allocated memory in subfunction(释放被调用函数的数据区)
(3) Return to upper level function via return address(依照被调用函数保存的返回地址将控制转移到调用函数)

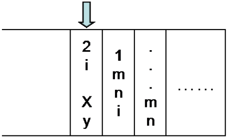
5.7 Summary and QUIZ

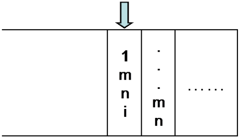
Rule: 多个函数嵌套调用时，按照“后调用先返回”的原则进行。Last calling, first return. Just like last in, first out.

```
int main()
{
    int m,n;
    ...
    first(m,n);
1: ...
...
}
```

```
int first(int s, int t)
{
    int   i;
    ...
    second(i);
2: ...
...
}
```

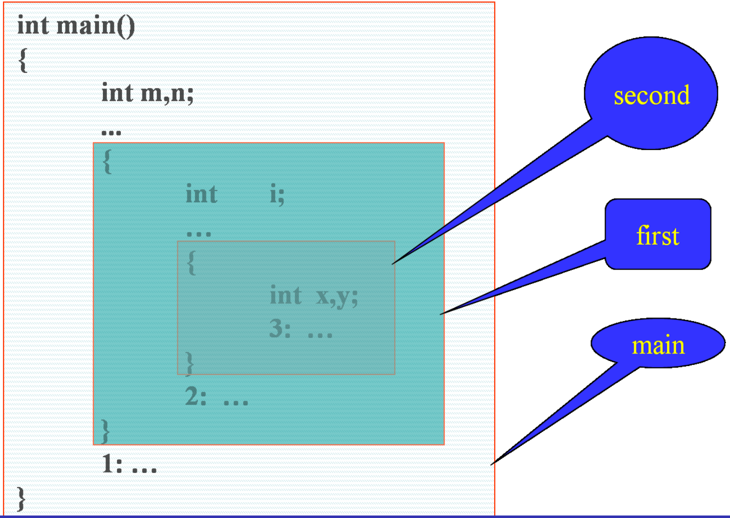
```
int second(int d)
{
    int x,y;
3:    ...
...
}
```





5.7 Summary and QUIZ

Example of function calling



Examples

- Main program and subprograms

3

```
void
main()
{
 ...
 call A(...);
 call D(...);
 ...
 return;
}
```

```
function
A(...)
{
 ...
 call B(...);
```

```
 call C(...);
 ...
 return;
}
```

```
function
B(...)
{
 ...
 ...
 ...
 ...
 ...
 ...
 ...
 return;
}
```

2

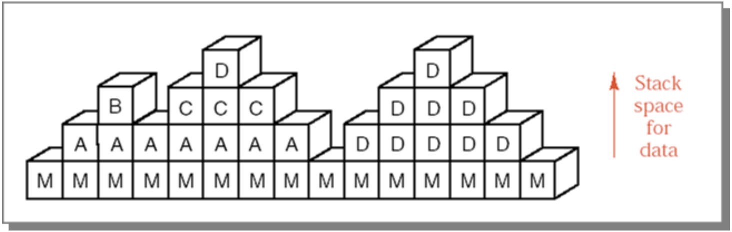
```
function
```

```
C(...)
{
 ...
 ...
 call D(case1);
 ...
 return;
}
```

```
function
D(...)
{
 ...
 ...
 call D(...);
 ...
 return;
}
```

5.7 Summary and QUIZ

Stack frames for subprograms



5.7 Summary and QUIZ

Tree of subprogram calls