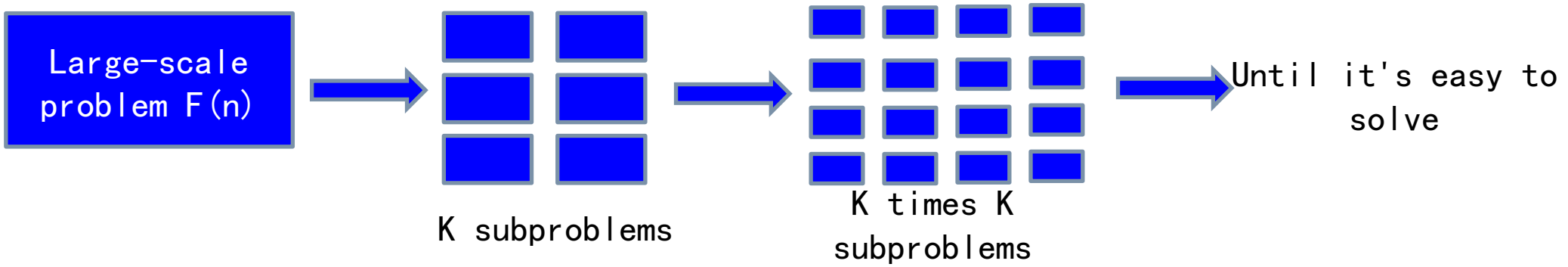
Recursion and divide-and-conquer strategies

Learning points:

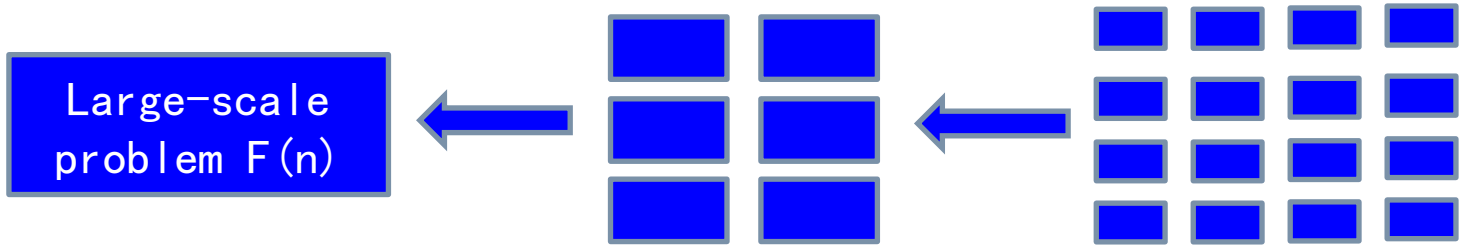
- Understand the concept of recursion and grasp the divide and conquer strategy for designing effective algorithms.
- Learn divide and conquer strategy design skills by examples.

The time for computer to solve a problem is proportional to the size of the problem. The smaller the problem, the shorter the computational time and the easier it is to deal with. It is difficult to solve large-scale problems directly. In order to effectively solve the target problem, the main idea is to divide a large problem that is difficult to solve directly into some smaller problems of the same size, so that each can be broken up, divide and conquer.

- The large-scale problem is divided into K subproblems and solved separately. If the size of the subproblem is still not small enough, then it is divided into K subproblems, and so on recursively, until the size of the problem is small enough, it is easy to find the solution.



- The solution of the small scale problem is merged into the solution of a larger scale problem, and the solution of the original problem is gradually found.



2.1 Recursion -- concept

Algorithms that directly or indirectly call themselves are called recursive algorithms. A function defined in terms of itself is called a recursive function. Divide-and-conquer subproblems are often smaller patterns of the original problem, which facilitates the use of recursive techniques. Divide and conquer and recursion are like twin brothers, which are often used in algorithm design at the same time, resulting in many efficient algorithms.

2.1 Recursion - Example

(1) Factorial: The factorial of a positive integer is the product of all positive integers less than or equal to that number, 0 factorial equals to 1. The factorial of the natural number n is written as n!

Definition of non-recursive mode:

$$n! = 1 \times 2 \times 3 \times \cdots n$$

Definition of recursive mode:

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

The initial value

The recursive definition

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

Note: Each recursive function must have a non-recursively defined initial value, otherwise it cannot be evaluated recursively.

2.1 Recursion - Example

（ 2 ） Fibonacci sequence: infinite sequence 1,1,2,3,5,8,13,21,34,55... , called the Fibonacci sequence. When n>1, the *n*th term of the sequence is the sum of its first two terms.

Definition of non-recursive mode:

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

Thinking: How to write recursively?

2.1 Recursion - Example

（ 2 ） Fibonacci sequence: infinite sequence 1,1,2,3,5,8,13,21,34,55... , called the Fibonacci sequence. When n>1, the *n*th term of the sequence is the sum of its first two terms.

Definition of recursive mode:

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

```
int fibonacci(int n)
{
  if (n<=1) return 1;
  return fibonacci(n-1)+fibonacci(n-2);
}
```

2.1 Recursion - Example

(3) Permutation problem - full permutation

Design a recursive algorithm to generate n elements {r1, r2,... ,rn}.

a) Two numbers 4 and 5, all arranged: {4, 5} and {5, 4}; Two numbers, 3 and 5, in full order: {3, 5} and {5, 3}; Two numbers 3 and 4, all arranged: {3, 4} and {4, 3}

b) Three numbers 3, 4 and 5, in full order: {3, 4, 5}, {3, 5, 4} and {4, 3, 5}, {4, 5, 3} and {5, 4, 3}, {5, 3, 4}

What is the relationship between a) and b)?

2.1 Recursion - Example

(3) Permutation problem - full permutation

Set R = {r1, r2,... rn} is the n elements to be permuted, Ri=R-{ri}. The full permutation of the elements in a set X is denoted perm(X).

(ri)perm(X) denotes the permutation obtained by prefixing each permutation of the full perm(X).

2.1 Recursion - Example

(3) Permutation problem - full permutation

The full permutation of R can be generically defined as follows.

When n=1, perm(R)=(r), where r is the unique element in the set R;

When n>1, perm(R) is given by (r1)perm(R1), (r2) perm(R2),... , (rn)perm(Rn).

2.1 Recursion - Example

(3) Permutation problem - full permutation

When n=1, perm(R)=(r), where r is the unique element in the set R;

When n>1, perm(R) is given by (r1)perm(R1), (r2) perm(R2),... , (rn)perm(Rn).

Understanding: In practice, all the numbers in the set are swapped with the first number, so always dealing with the next n − 1 numbers

perm(R) = (r1)perm(R1)+(r2)perm(R2)+…+(rn)perm(Rn)

Denoted by r1, r2... .rn is the combination of all permutations starting with (prefix).

All permutations of 1,2,3

- Traversing elements, swapping each element with the first, {1}{23}, {2}{13}, {3}{12}. A total of three groups
- For each group, repeat until the remaining one element, except for the first element. For example, {23} is traversed and swapped with the first element in it to actually get {23} and {32}.
- End up with 6 kinds of combinations.

All permutations of 1,2,3,4

- Traversing elements, swapping each element with the first, in four groups: {1}{234}, {2}{134}, {3}{124}, and {4}{123}
- For each group, repeat until the remaining one element, except for the first element. As in the first group, except for the first element, {234} is left to traverse, and the problem becomes the full permutation of {234}. The second to fourth groups were similar.

2.1 Recursion - Example

(4) Integer partition problem

The positive integer n expressed as the sum of a series of positive integers: n = n1 + n2 +... + nk, where n1≥n2≥... ≥nk≥1, k≥1. This representation of a positive integer n is called a partition of positive integers n. Find the number of different partitions of the positive integer n.

2.1 Recursion – Example

（4）Integer partition problem

The positive integer 6 can be divided in the following 11 different ways:

6;
5+1;
4+2, 4+1+1;
3+3, 3+2+1, 3+1+1+1;
2+2+2, 2+2+1+1, 2+1+1+1+1;
1+1+1+1+1+1

q(n, m) is used to represent the partition number of positive integers. Where, m represents the maximum value of n1, n1<=m, and the following recursive relation of q (n, m) can be established.

1. When m=1, that is, when the largest addend n1 is at most 1, there is only one combination of any positive integer. So n = 1+1+1+... + 1.

e.g.:

q(6,1)=1;

6=1+1+1...+1

2.1 Recursion – Example

（4）Integer partition problem

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & n<m\\ 1+q(n,n-1) & n=m\\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

```
int q(int n, int m)
{
    if ((n < 1) || (m < 1)) return 0;
    if ((n == 1) || (m == 1)) return 1;
    if (n < m) return q(n,n);
    if (n == m) return q(n,m-1)+1;
    return q(n,m-1)+q(n-m,m);
}
```

2. When m>n, q(n, m) = q(n, n),

And obviously, when the positive integer n =6, q(6, m) = q(6, 6) .

3. q (n, n) = 1 + q (n, n-1); The partition of a positive integer n consists of the partition n1=n and the partition n1≤n-1.

2.1 Recursion – Example

(4) Integer partition problem

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & n<m\\ 1+q(n,n-1) & n=m\\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

```
int q(int n, int m)
{
    if ((n < 1) || (m < 1)) return 0;
    if ((n == 1) || (m == 1)) return 1;
    if (n < m) return q(n,n);
    if (n == m) return q(n,m-1)+1;
    return q(n,m-1)+q(n-m,m);
}
```

e.g.:
q(6,6):
6;
5+1;
4+2，4+1+1;
3+3，3+2+1，3+1+1+1;
2+2+2，2+2+1+1，2+1+1+1+1;
1+1+1+1+1+1
q(6,5):
5+1;
4+2，4+1+1;
3+3，3+2+1，3+1+1+1;
2+2+2，2+2+1+1，2+1+1+1+1;
1+1+1+1+1+1。
So, q(6,6)=q(6,5)+1

2.1 Recursion – Example

(4) Integer partition problem

$$q(n,m)=\begin{cases}1 & n=1,m=1\\ q(n,n) & n<m\\ 1+q(n,n-1) & n=m\\ q(n,m-1)+q(n-m,m) & n>m>1\end{cases}$$

```
int q(int n, int m)
{
    if ((n < 1) || (m < 1)) return 0;
    if ((n == 1) || (m == 1)) return 1;
    if (n < m) return q(n,n);
    if (n == m) return q(n,m-1)+1;
    return q(n,m-1)+q(n-m,m);
}
```

4. q(n, m)=q(n,m-1)+q(n-m, m),n>m>1.

The partition of m consists of the largest addend m-1 and the partition of m.

e.g.:

q(6,2) = q(6,1)+q(4,2)

q(6,2):

2+2+2，2+2+1+1，2+1+1+1+1；1+1+1+1+1+1。

q(6,1):

1+1+1+1+1+1

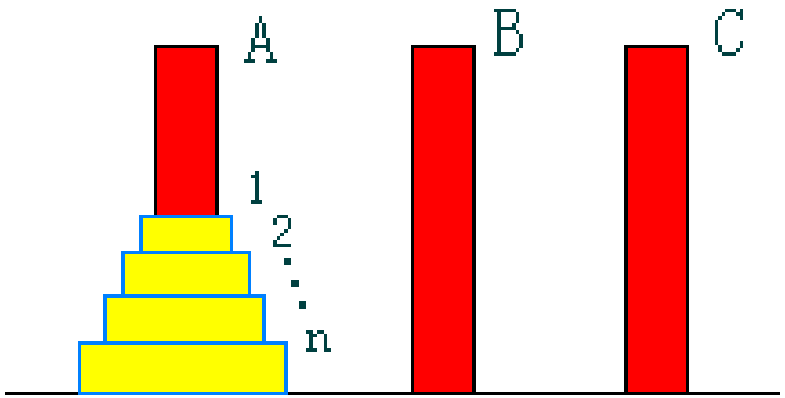
q(4,2):

2+2，2+1+1，1+1+1+1；

2.1 Recursion - Example

(5) Hanoi Tower Questions

Let A, B, and C be three towers. You start with a stack of n disks on tower A, and the disks are stacked from top to bottom, from smallest to largest. The disks are numbered 1,2,... n. It is now requested that the stack of disks from tower A be moved to Tower B and still be stacked in the same order.



2.1 Recursion - Example

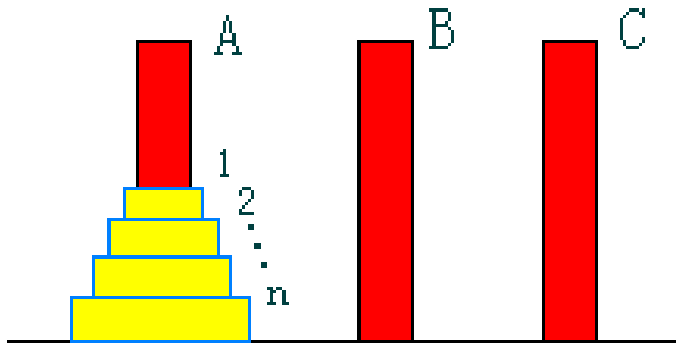
(5) Hanoi Tower Questions

The following movement rules should be observed when moving the disk:

Rule 1: Only move one disk at a time.

Rule 2: It is not allowed at any time to push a larger disk on top of a smaller disk;

Rule 3: The disk can be moved to any tower in A, B, C under the premise of satisfying movement rules 1 and 2.

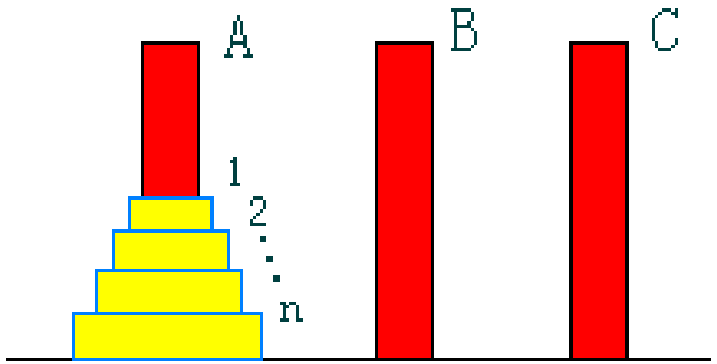


2.1 Recursion - Example

(5) Hanoi Tower Questions

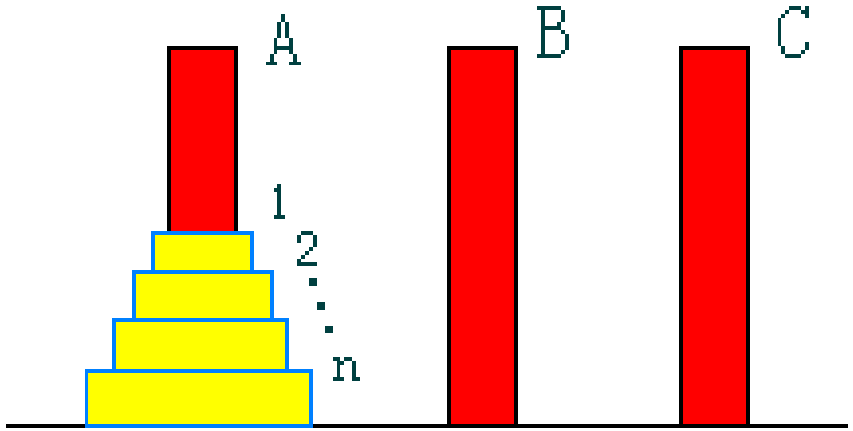
Recursion idea:

1. When n=1, just move the disk from A to tower B;
2. When n>1, using tower C as an aid, try to place n − 1 of the smaller disks on C, then the largest remaining disks on B, and finally n − 1 of the smaller disks on B.



2.1 Recursion – Example

(5) Hanoi Tower Questions




```
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);
     }
  }
```

Put n-1 of the smaller disks on C.
Put the largest remaining disk on B
Put n-1 of the smaller disks on B

Recursive summary

Advantages: clear structure, readable, and easy to use mathematical induction to prove the correctness of the algorithm, so it is very convenient for the design of the algorithm, debugging program.

Disadvantages: The recursive algorithm is less efficient and consumes more computing time and storage space than the non-recursive algorithm. The bigger the problem, the less efficient it is.

Solution: Eliminate recursive calls in recursive algorithms and convert them into non-recursive algorithms.

Recursive summary

- A user-defined stack is used to simulate the recursive call stack of the system. It is recursive in nature. According to the specific program, the recursive call stack is simplified to reduce the operation of the stack and compress the stack space.
- Recursion is used to implement recursive functions;
- Some recursions can be transformed into tail recursions (tail recursions are evaluated from the end and the corresponding result is calculated each time. That is, function calls appear at the tail of the caller's function, and since they are tails, there is no need to save any local variables).

2.2 Divide and conquer method

The problems that can be solved by divide and conquer generally have the following characteristics:

- The problem can be easily solved if it is scaled down to a certain extent;
- The problem can be decomposed into several smaller identical problems, that is, the problem has the optimal substructure property

2.2 Divide and conquer method

- The solutions of the subproblems decomposed by this problem can be merged into the solution of this problem (whether the divide-and-conquer method can be used completely depends on whether the problem has this feature or not. If the first two features are possessed, but the third feature is not, greedy algorithm or dynamic programming can be considered).
- The subproblems of this problem are independent of each other, namely between the subproblems do not include the public subproblems (this characteristic relates to the efficiency of divide-and-conquer. If the subproblems are not independent, divide and conquer does a lot of unnecessary work, solves the common subproblem repeatedly. In this case, although divide and conquer can also be used, dynamic programming is generally better.)

2.2 Divide and conquer method

Basic steps:

```
divide-and-conquer(P)
  {
    if ( | P | <= n0) adhoc(P);   //Tackle small scale problems directly
    divide P into smaller subinstances P1,P2,...,Pk；//Decomposition problem
    for (i=1,i<=k,i++)
      yi=divide-and-conquer(Pi);  //Recursively solving each subproblem
    return merge(y1,...,yk);   //The solution of each subproblem is merged into the solution of the
                                                                    original problem
  }
```

2.2 Divide and conquer method

From a lot of practice, people have found that when designing algorithms using divide and conquer, it is best to make the subproblems roughly the same size. It is effective to divide a problem into K subproblems of equal size. This practice of making the subproblems roughly equal in size comes from the idea of **balancing the subproblems**, which is almost always better than balancing the subproblems of

2.2 Divide and conquer method

Time complexity analysis:

- Suppose we have a problem of size *n*, and the time function is *T(n)*.
- Divided into *k* problems, the size of each problem is *n/m*, then the time function of each sub-problem is *T(n/m)*.
- The smallest subproblem is *n*=1, and it takes 1 unit of time to solve a problem of size 1
- Merging the solution of the subproblem into the solution of the original problem requires $f(n) = n^d$ time units

$$T(n) = \begin{cases} O(1) & n = 1 \\ kT(n/m) + f(n) & n > 1 \end{cases}$$

2.2 Divide and conquer method

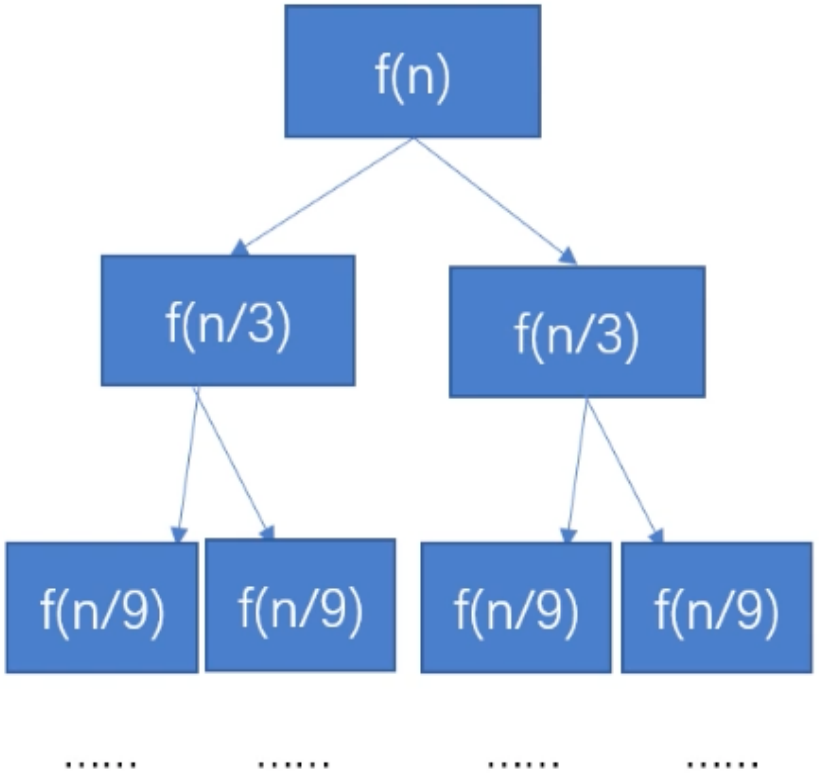
Recursion tree

Recursion tree is an image representation in the **iterative process**, which is often used to solve the recursive equation, and its solution representation is more concise and clear than the general iteration.

Iteration: The process of repeatedly performing a series of operations to obtain the following quantities in turn. Each result of this process is obtained by applying the same procedure to the previous result.

2.2 Divide and conquer method

$$T(n)=2*T(n/3)+ f(n)$$



2.2 Divide and conquer method

The main theorem:

In the recursive formula $T(n)=kT(n/m)+f(n)$, $f(n)=n^d, d\geq 0$

$$T(n)\in\begin{cases}O(n^d) & k<m^d\\ O(n^d\log_m n) & k=m^d\\ O(n^{\log_m k}) & k>m^d\end{cases}$$