数据结构与算法

人工智能与大数据学院

本节课主要内容

- 第3章 栈和队列

① 栈的应用
② 队列的基本概念、顺序存储结构、链式存储结构
③ 递归
④ 汉诺塔问题

三、栈和队列——栈的应用

设计算法实现将十进制转换为二进制。（运用栈的相关知识，需要实现栈的初始化、入栈、出栈和核心算法）

```
#define maxsize 100
  typedef struct
  {
      int data[maxsize];
      int top;
  }sqtacktp;
```

三、栈和队列——栈的应用

设计算法实现将十进制转换为二进制。
（运用栈的相关知识，需要实现栈的初始化、入栈、出栈和核心算法）

```
#define maxsize 100
  typedef struct
  {
      int data[maxsize];
      int top;
  }sqtacktp;
```

```
while( n > 0){
        tmp = n%2;
        n = n/2;
        push(s, tmp);
}
while(! stack_empty(s)){
        tmp = pop(s);
        printf("%d ", tmp);
}
```

三、栈和队列——栈的应用

假设在一个算术表达式中可以包含三种括号:圆括号“(”和“)”，中括号“[”和“]” 和花括号“{”和“}”，并且这三种括号可以按任意的次序嵌套使用。使用栈的相关知识设计算法用来检验输入的算术表达式中使用括号的合法性。顺序栈的类型描述及基本函数定义如下:

```
#define maxlen 100 typedef struct{
    char data[maxlen];
    int top;
}SeqStack;
void initstack(SeqStack *s); bool empty(SeqStack *s);
void push(SeqStack *s, int x); char pop(SeqStack *s);
```

三、栈和队列——队列

1、队列的定义

栈

a. 操作受限的特殊线性表
b. 后进先出
c. 所有的操作只能在线性表的一端进行
d. 进行操作的一端为栈顶，固定端称为栈底
e. 栈顶用一个“栈顶指针”指示

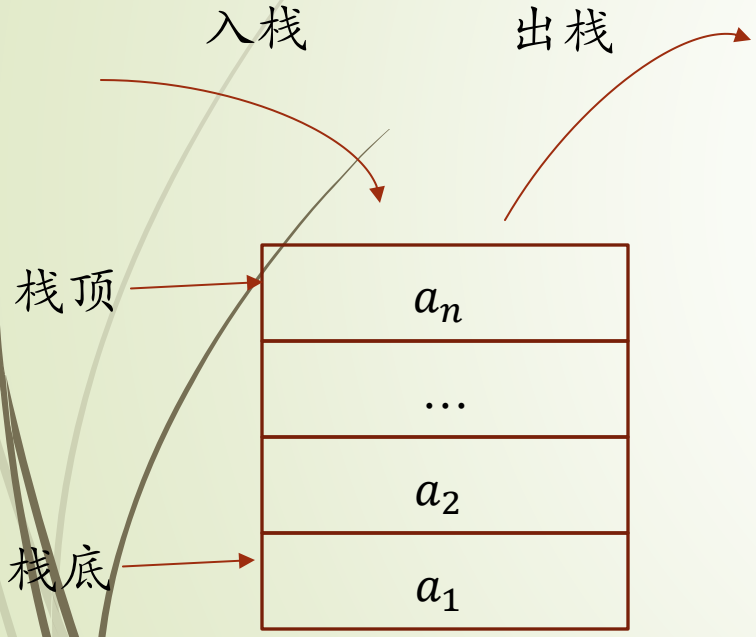
队列

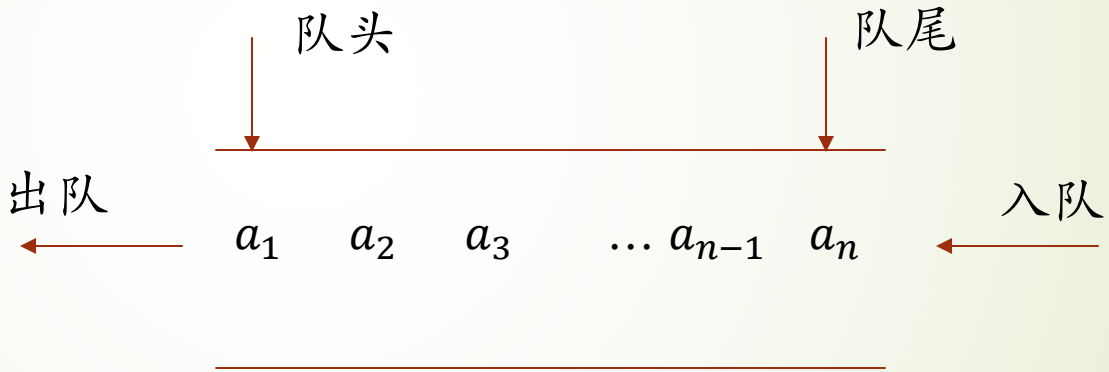
a. 操作受限的特殊线性表
b. 先进先出
c. 插入操作在线性表的一端进行，删除操作在表的另一端进行
d. 允许插入的一端为队尾，允许删除的一端为队头
e. 队尾和队头分别用队尾指针和队头指针指示

1、队列的定义

栈：数据元素$a_1, a_2, \cdots, a_n$依次入栈，出栈的顺序是a_n，$\cdots$，a_2，a_1

队列：数据元素$a_1, a_2, \cdots, a_n$依次入队，出队的顺序不变，依然是$a_1, a_2, \cdots, a_n$





2、队列的基本操作

基本操作	函数名称	操作结果
初始化	Status Queue_Init(QueuePtr q)	若成功，返回success，构造一个q所指向的空队列，否则返回fatal
销毁	Void Queue_Destory(QueuePtr q)	释放q所占空间，队列q不再存在
清空	Void Queue_Clear(QueuePtr q)	清空队列中所有元素，队列q变空
判空	Bool Queue_Empty(QueuePtr q)	若队列q为空，返回true，否则返回false
判满	Bool Queue_Full(QueuePtr q)	若队列q为满，返回true，否则返回false
入队	Status Queue_Push(QueuePtr q, QueueEntry item)	若队列q不满，将item添加到队尾，返回true，否则返回overflow
出队	Status Queue_Pop(QueuePtr q, QueueEntry *item)	若队列q不空，将队头数据元素放入item，并删除该数据元素，返回true，否则返回underflow
取队头元素	Status Queue_Front(QueuePtr q, QueueEntry *item)	若队列q不空，将队头数据元素放入item，返回true，否则返回underflow

3、队列的顺序存储

顺序存储的队列称为顺序队列。
和顺序表一样，顺序队列需要用一个向量空间来存储当前队列中的数据元素。由于队头和队尾随时变化，因此，顺序队列除了数据区外，至少还需要设置队头、队尾两个指针。

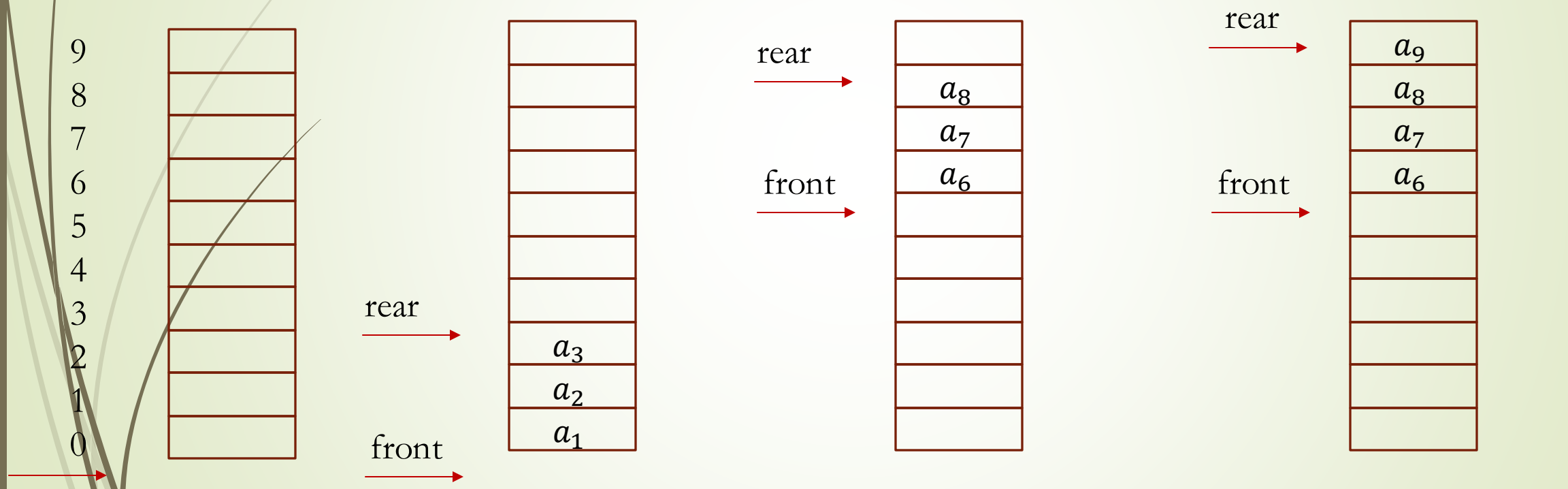
```
typedef struct{
    int elem[maxsize];
    int front, rear;
}squeuetp;
```

3、队列的顺序存储

初始化时队头指针和队尾指针均为-1，即

q->front = q->rear = -1;

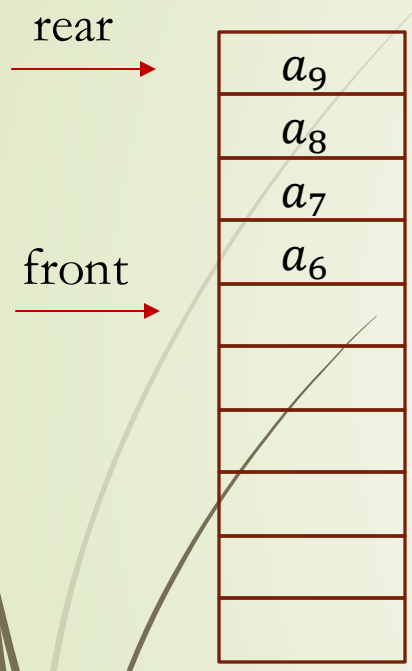
每当插入新的队列元素时，队尾指针增1；每当删除队头元素时，队头指针增1；



front = rear = -1　　front = -1; rear = 2　　front = 5; rear = 8　　front = 5; rear = 9

（a）空队列　　（b）3个元素入队　　（c）若干入队和出队操作后情况　**(d) 假溢出现象**

3、队列的顺序存储



front = 5; rear = 9

和顺序栈类似，顺序队列也有上溢和下溢现象。

随着入队出队的进行，会使整个队列整体向后移动。如果队尾指针已经移到了最后，此时再有元素入队就会出现溢出，而事实上此时队列并不一定真的“满员”，可能还有空闲存储空间，这种现象称为“假溢出”。

3、队列的顺序存储

假溢出现象的发生是由于对指针的操作只增不减造成的。
假溢出的现象不能通过增加存储空间来解决！

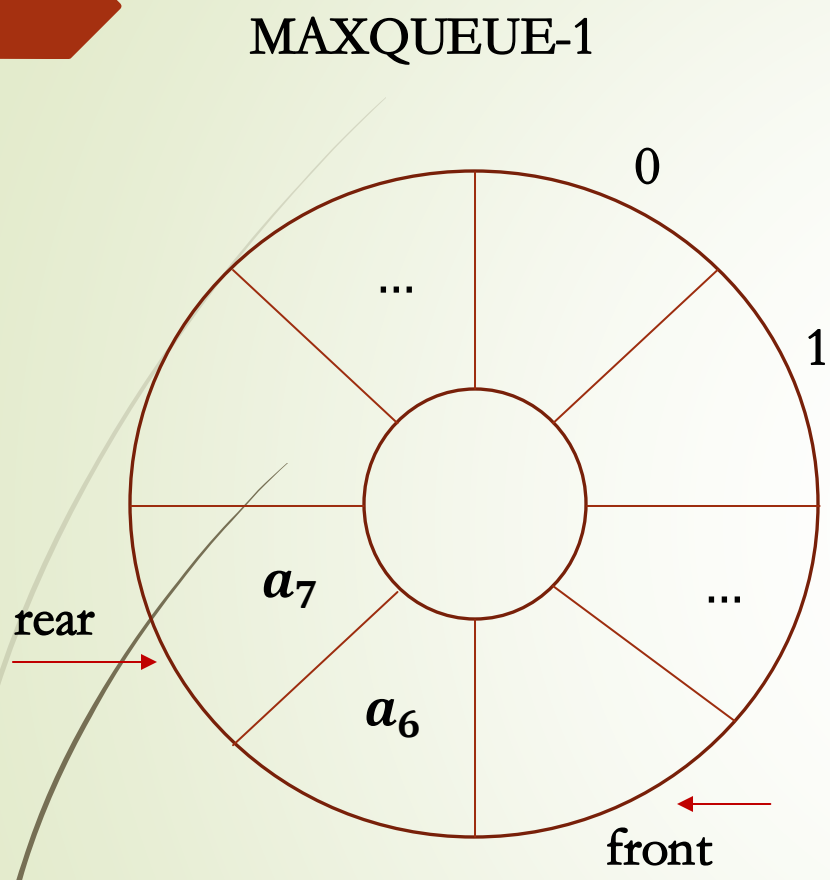
解决假溢出的方法：

固定队头指针永远指向数据区开始位置，如果数据元素出队，则将队列中所有数据元素前移，同时修改队尾指针。
优点：实现简单
缺点：造成大量数据元素移动，引入较大的时间开销

队头和队尾指针都可以移动，只有造成假溢出时，才将队列中所有数据元素依次前移到存储空间前面，同时修改队头和队尾指针。
缺点：带来数据元素移动的时间开销

将队列的数据存储区看成首尾相接的循环结构，头尾指针的关系不变，称其为“**循环队列**”

3、队列的顺序存储



假设为队列分配的存储空间大小为MAXQUEUE，在C语言中，头尾指针的下标范围是0～MAXQUEUE-1，若增加队头或队尾指针，可以利用取模运算实现。例如：

front = (front + 1)%MAXQUEUE;
rear = (rear + 1)% MAXQUEUE;

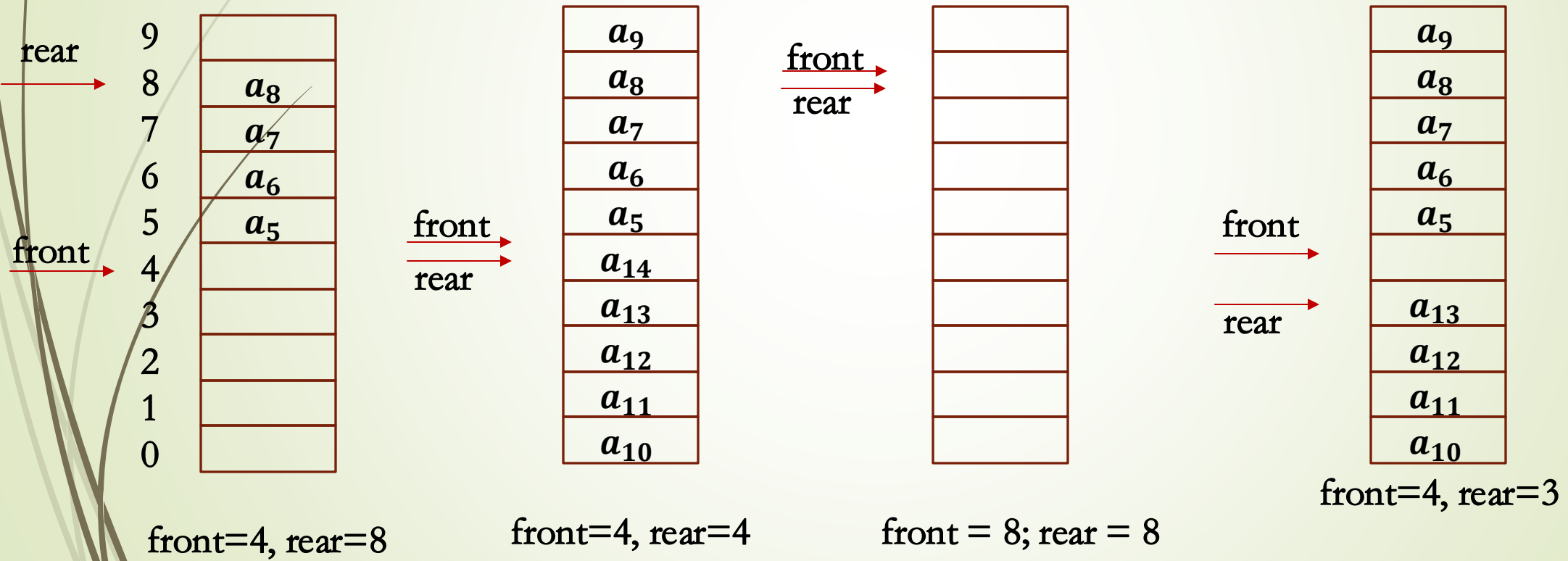
当front或rear为MAXQUEUE-1时，上述两个公式计算的结果为0。这样就使得指针自动由后面转到前面，形成循环的效果

3、队列的顺序存储

图（a）中有4个元素，随着a_9~ a_{14}相继入队，队列中共有10个元素，已占满所有存储空间，此时front和rear相等，如图（b）所示。在队满的情况下有front==rear。

如果在图（a）的基础上，a_5～a_8相继出队，此时队列为空，如图（c）所示，即在队空的情况，同样也有front==rear。

队满和队空的条件相同，仅凭front==rear无法区分循环队列是空还是满！



front=4, rear=8

front=4, rear=4

front = 8; rear = 8

front=4, rear=3

（a）4个元素

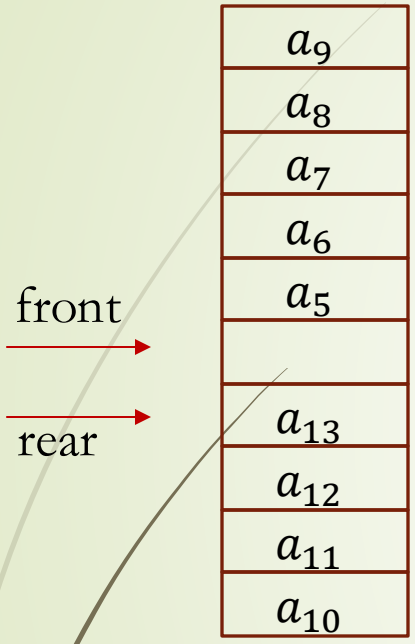
（b）队满

（c）若干入队和出队操作后情况

（d）队满

3、队列的顺序存储

问题：如何判断循环队列的空和满呢?



front=4, rear=3

（d）队满

方法一：

附设一个存储队列中元素个数的变量，如count，当count=0时为队空，当count等于MAXQUEUE时为队满。

方法二：

为队列另设一个标志，用来区分队列是空还是满。

方法三：

少用一个数据元素空间，当数组只剩下一个单元时就认为是队满，此时队尾指针只差一步追上队头指针。即(rear+1)%MAXQUEUE==front。如图(d)所示

4、队列的链式存储

1、链队列的定义

链式存储结构的队列称为链队列

实际上是一个同时带有头指针和尾指针的单链表，头指针指向队头结点，尾指针指向队尾结点。

4、队列的链式存储

1、顺序队列（循环队列）和链队列的定义

循环队列

```
typedef struct{
    int elem[maxsize];
    int front, rear;
}cqueuetp;
```

链队列

```
typedef struct nodetype{
    int data;
    struct nodetype *next;
}nodetype;
```

```
typedef struct{
    nodetype *front;
    nodetype *rear;
}lqueuetp;
```

4、队列的链式存储

2、链队列的基本操作

- **链队列初始化**

```
void initqueue(lqueuetp *lq){
    lq->front = (nodetype*)malloc(sizeof(nodetype));
    lq->front->next = NULL;
    lq->rear = lq->front;
}
```

```
int main(){
    lqueuetp *lq;
    lq = (lqueuetp *)malloc(sizeof(lqueuetp));
    initqueue(lq);
    return 0;
}
```

2、链队列的基本操作

- **链队列判队空**

```
int queueempty(lqueuetp *lq){
    if(lq->front == lq->rear)
        return 1;
    return 0;
}

int main(){
    lqueuetp *lq;
    int tmp = 0;
    lq = (lqueuetp *)malloc(sizeof(lqueuetp));
    initqueue(lq);
    tmp = queueempty(lq);
    return 0;
}
```

2、链队列的基本操作

- **求链队长度**

```
int size(lqueuetp *lq){
    int len = 0;
    nodetype *p = lq->front->next;
    while (p) {
        len++;
        p = p->next;
    }
    return len;
}
```

2、链队列的基本操作

- **链队列入队**

```
void enqueue(lqueuetp *lq, int x){
    nodetype *s;
    s = (nodetype *)malloc(sizeof(nodetype));
    s->data = x;
    s->next = NULL;
    lq->rear->next = s;
    lq->rear = s;
}
```

2、链队列的基本操作

- **链队列出队**

```
int delqueue(lqueuetp *lq){
    int x;
    nodetype *p;
    if(lq->front == lq->rear)
        return NULL;
    else{
        p=lq->front->next;
        lq->front->next = p->next;
        if(p->next == NULL)
            lq->rear = lq->front;
        x=p->data;
        free(p);
        return x;
    }
}
```

2、链队列的基本操作

- **读链队列队头元素**

```
int size(lqueuetp *lq){
    int len = 0;
    nodetype *p = lq->front->next;
    while (p) {
        len++;
        p = p->next;
    }
    return len;
}
```

● 递归的概念

定义：递归是直接或间接调用本身的一种方法。

基本思想：递归就是有去有回（有递推和回归）。它将一个问题划分成一个活多个规模更小的子问题，然后用同样的方法解决这些规模更小的问题，这些问题不断从大到小，从远及近的过程中，会有一个终点，一个临界点，一个到了那个点就不能再往更小，更远的地方走下去的点，这个点叫做**递归出口**。然后从那个点开始，原路返回到原点，就求得了最初想知道的值。

递归的概念

举例：假如你在超市收银台排队结账，你想知道自己现在排第几个。但是排队的人比较多，你懒得去数，于是你向前面的人问道：“你好，请问你排第几个？”，因为如果你知道你前面的人的序号，自己只需要加上1就可以了。但是呢，前面的人也不清楚自己排第几，于是他又问他前面的那个人，依次类推，直到问到了排在最前面的那个人，那个人有点无奈的告诉向他问问题的人：“我排第一”。于是大家都知道自己排第几了。这就是递归。

● 递归的概念

【例】求解2的n次幂，即$f(n)=2^n$

分析：

当n=1时，$f(1)=2^1=2$

当n>1时，可以把原问题$f(n)$分解为$f(n)=2^n=2\times 2^{n-1}=2\times f(n-1)$。

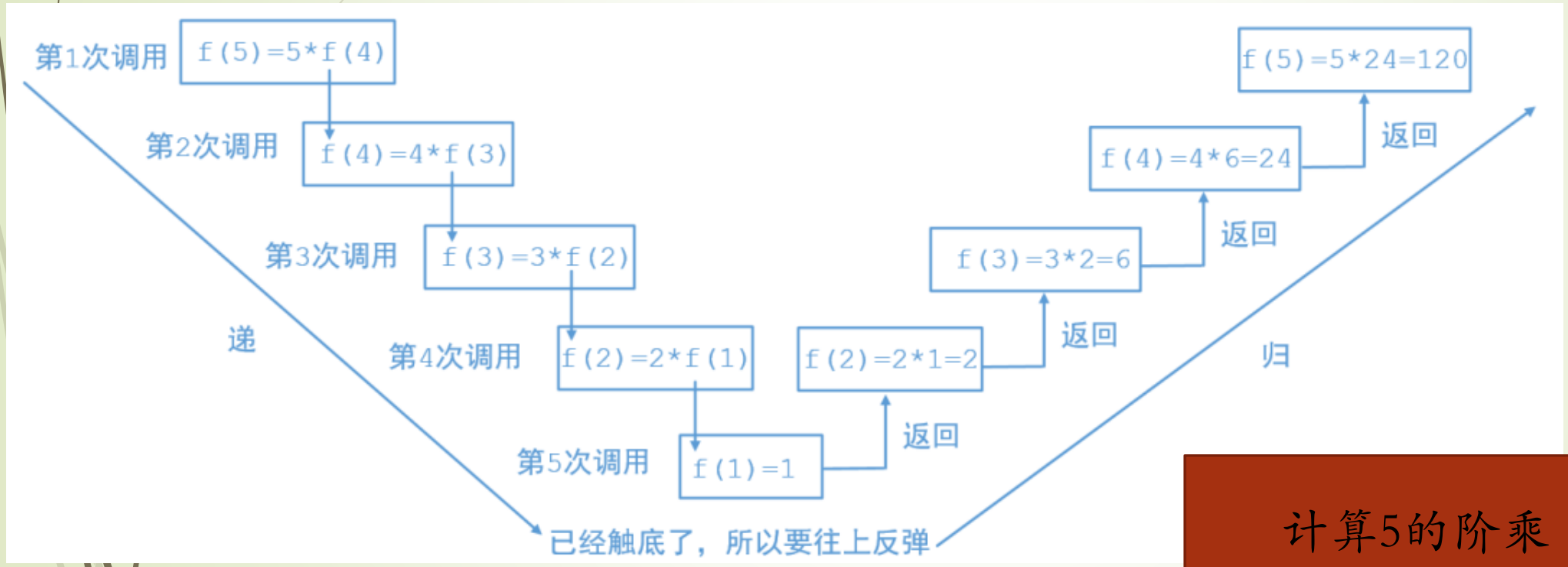
$f(n-1)$与$f(n)$是同一类型的问题，可以用递归方程表示为：

$$f(n)=\begin{cases}2, & n=1;\\ 2\times f(n-1), & n>1。\end{cases}$$

根据这个方案，可以设计出如下递归算法：

```
int compute2n(int n)
{
    if(n=1) return 2;
    if(n>1) return 2*compute2n(n-1);
}
```

假设n等于5，计算5！的递归调用可以用下面这个图描述



计算5的阶乘

递归与迭代的区别

递归：程序调用自身的编程技巧称为递归，是函数自己调用自己。虽然递归是一种非常优美的编程技术，但是，它需要更多的存储空间和时间。而且，由于递归会引起一系列的函数调用，同时还有可能会有一系列的重复计算，因此递归算法的执行效率相对较低。

迭代：利用变量的原值推算出变量的一个新值称为迭代。如果递归是自己调用自己的话，迭代就是函数A不停地调用函数B。每一次对过程的重复称为一次“迭代”。而每次迭代得到的结果会作为下一次迭代的初始值。

● 递归与迭代的区别

递归：

```
int compute2n(int n)
{
    if(n=1) return 2;
    if(n>1) return 2*compute2n(n-1);
}
```

迭代：

```
int compute2n(int n)
{
    int total = 1;
    for(i=1; i<n+1; i++)
    {
        total = total *2;
    }
}
```

两者对比，迭代算法的效率通常更高，在实际求解的过程中更加常用。

● 递归与栈的关系

递归函数的执行过程具有如下三个特点:

① 函数名相同;
② 不断地自调用;
③ 最后被调用的函数要最先被返回。

栈是一种执行“先进后出”或者“后进先出”算法的数据结构。
它是计算机中最常用的一种线性数据结构，比如函数的调用在计算机中就是用系统栈实现的。

递归函数的执行过程与栈的执行规则有相似的“先进后出”

汉诺塔问题

问题描述：

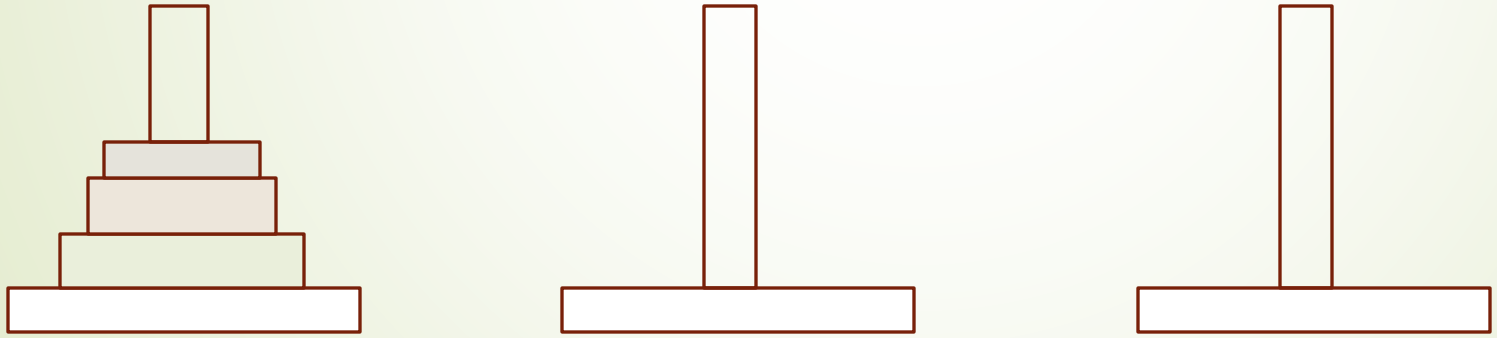
3根金刚石柱子，在其中1根柱子上按照从下往上的顺序从大到小地摆着64片黄金圆盘。要把这些黄金圆盘从下面开始按大小顺序重新把放到另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在3根柱子之间一次只能移动一个圆盘。当圆盘个数很少的时候，任务很容易完成。当圆盘个数超过5个时，情况就变得很复杂，任务很难完成。

汉诺塔问题

问题分析:

以三个圆盘为例，设有A、B、C三根柱子，其中A柱子上有三个从下往上按从大到小的顺序叠放的三个圆盘。

现要求按照汉诺塔问题的规则将三个圆盘移动到C柱子上，且叠放顺序不变。



汉诺塔问题

解决步骤:

1、将A柱子上的第一个圆盘移动到C柱子上;

2、将A柱子上的第二个圆盘移动到B柱子上;

3、将移动到C柱子上的第一个小圆盘叠放到B柱子上的圆盘上;

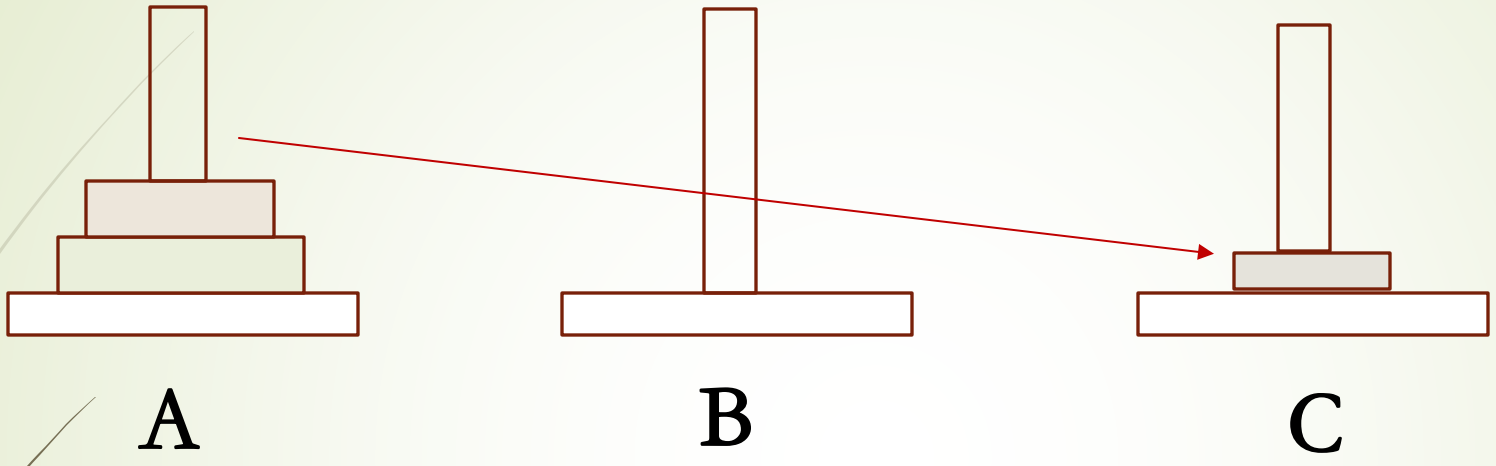
4、再将A柱子上最大的圆盘移动到C柱子上;

5、将B柱子上的最小圆盘放到A柱子上;

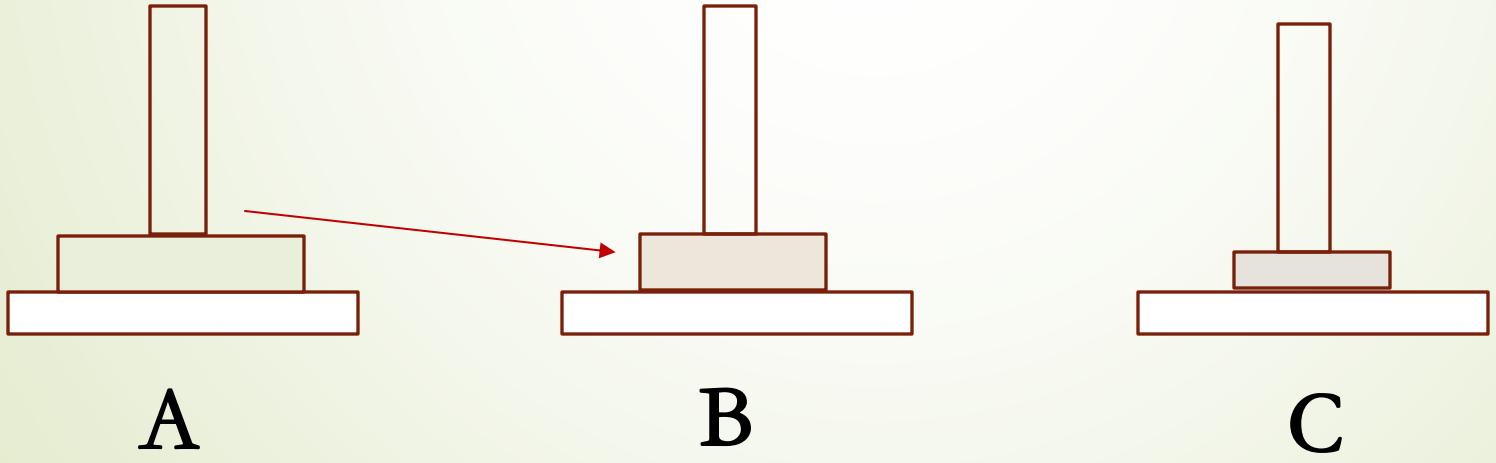
6、将B柱子上的圆盘叠放到C柱子上;

7、将A柱子上最小的圆盘叠放到C柱子上

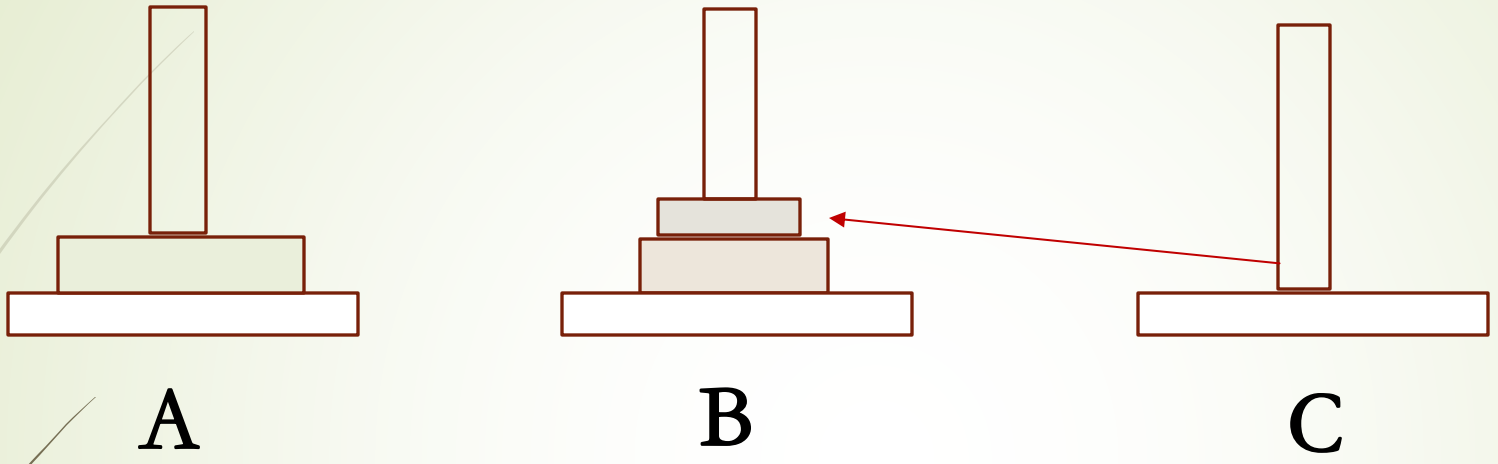
step 1



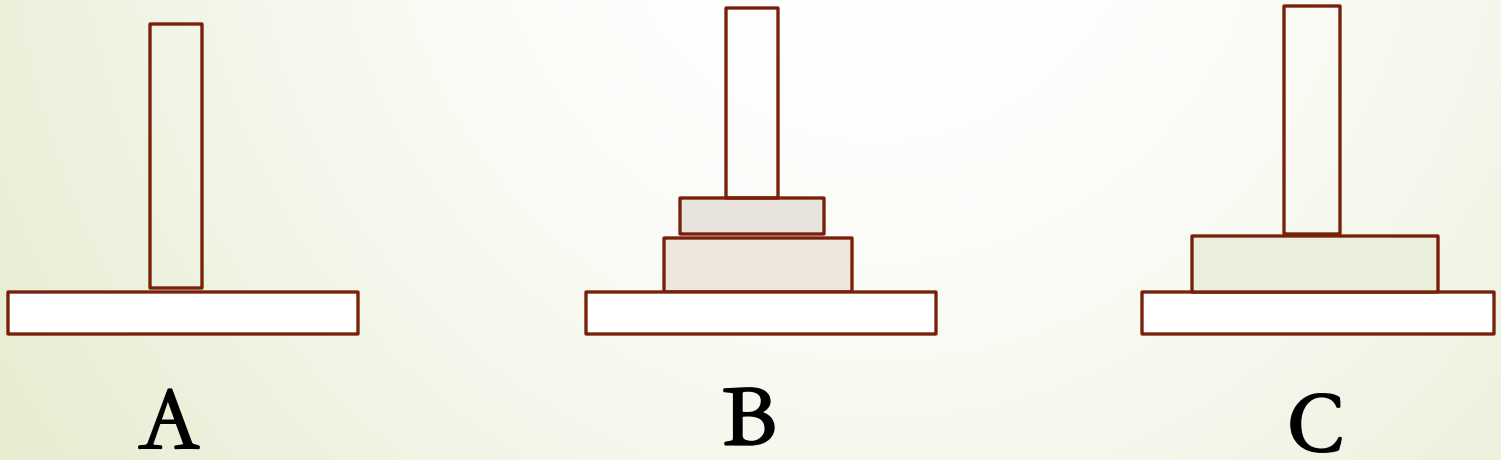
step 2



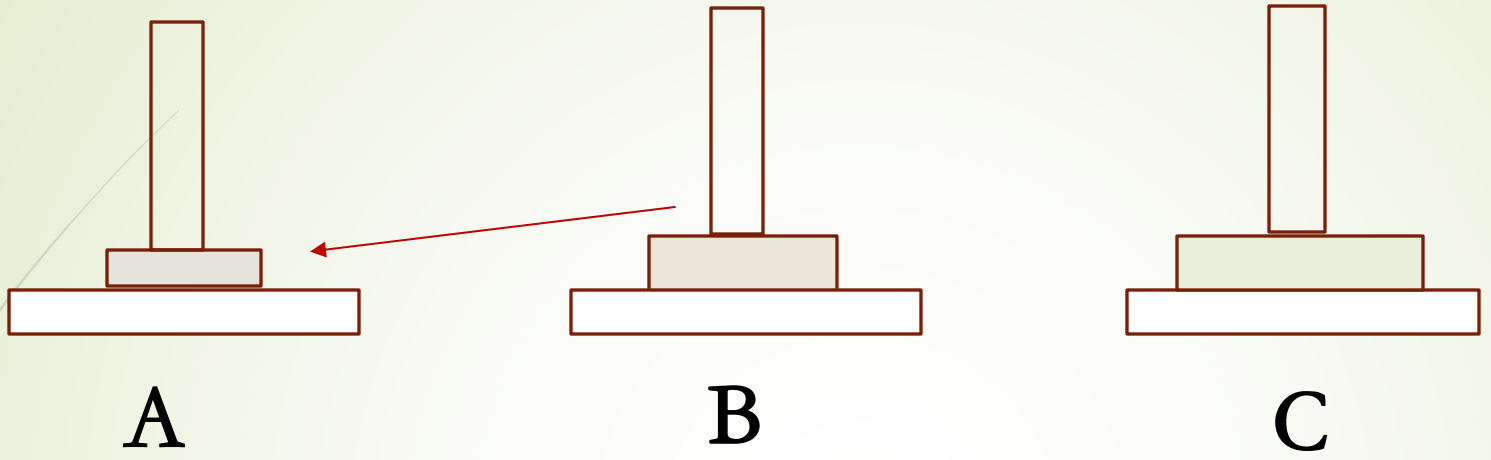
step 3



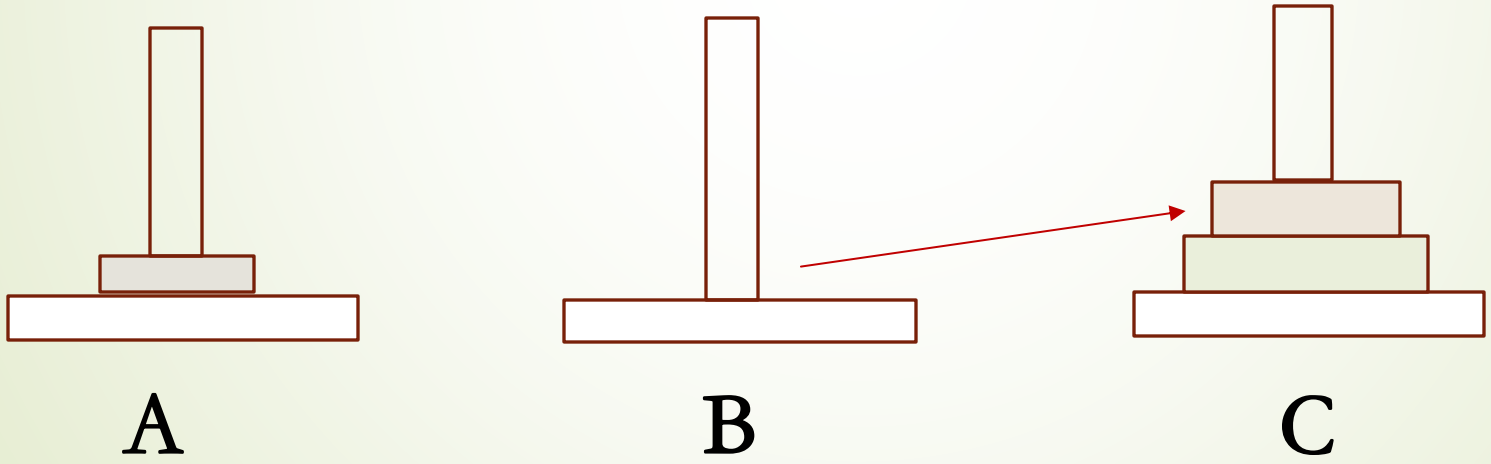
step 4



step 5



step 6



问题总结：

若有n个圆盘，先把A柱子上的n-1个圆盘移动到B柱子上，再把A柱子上剩下的最后一个最大圆盘移动到C柱子上，由于该圆盘是最大的，所以在以后的搬动过程中，它保持不动，然后再将B柱子上的n-1个圆盘借助A柱子移动到C柱子上。这样移动n个圆盘的问题就可以对应为移动n-1个圆盘的问题，且move(n)=2move(n-1)+1，显然，这是一个递归公式，因此可以用递归算法来求解汉诺塔问题。

```c
int Hanoi(int n, char *a, char *b, char *c){
    static int count=0;
    if(n < 1) return -1;
    if(n == 1){
        count += 1;
        printf("%c-->%c\n", a[0], c[0]);
    }else{
        Hanoi(n-1, a, c, b);
        printf("%c-->%c\n", a[0], c[0]);
        count += 1;
        Hanoi(n-1, b, a, c);
    }
    return count;
}

int main(){
    char *a = "A", *b = "B", *c = "C";
    int n=0;
    scanf("%d", &n);
    printf("the number of circle n=%d\n", n);
    int count = Hanoi(n, a, b, c);
    printf("the number of move :%d\n", count);
    return 0;
}
```

```
3
the number of circle n=3
A-->C
A-->B
C-->B
A-->C
B-->A
B-->C
A-->C
the number of move :7
Program ended with exit code: 0
```