数据结构与算法

计算机学院

朱晨阳 副教授

zhuchenyang07@nudt.edu.cn

后缀表达式

- 后缀表达式把运算符放在它的操作数之后。
- 2*(3+4) - 8/2的后缀表达式为234 + *82/ - 。
- 是否需要操作数栈？ YES
- 是否需要运算符栈？ No
- 为什么？

后缀表达式

- 后缀表达式把运算符放在它的操作数之后。
- 2*(3+4) - 8/2的后缀表达式为234 + *82/ - 。
- 是否需要操作数栈？ YES
- 是否需要运算符栈？ No
- 后缀表达式里：
  - (1)没有括号
  - (2)操作数出现的次序与中缀表达式相同
  - (3)运算符出现的次序是计算次序，按优先级从高到底排列。

后缀表达式的处理

- 计算：扫描后缀表达式，若析取一个操作数，则立即进栈；若析取一个运算符，则将操作数栈的栈顶和次栈顶连续出栈，使出栈的两个操作数执行运算符规定的运算，并将运算结果进栈。后缀表达式扫描结束时，操作数栈的栈顶即为表达式的值。
- 请你给出栈的变化过程：234+*82/-

示例

- 初始
  - 表达式：234+*82/-
  - 操作数栈：{ }
- 第一步
  - 表达式：2;34+*82/-
  - 操作数栈：{2 }
- 第二步
  - 表达式：23;4+*82/-
  - 操作数栈：{2,3 }
- 第三步
  - 表达式：234;+*82/-
  - 操作数栈：{2,3,4 }
- 第四步
  - 表达式：234+;*82/-
  - 操作数栈：{2,7 }
- 第五步
  - 表达式：234+*;82/-
  - 操作数栈：{14 }

示例

- 第六步
  - 表达式：234+*8;2/-
  - 操作数栈：{14,8 }
- 第七步
  - 表达式：234+*82;/-
  - 操作数栈：{14,8,2 }
- 第八步
  - 表达式：234+*82/;-
  - 操作数栈：{14,4 }
- 第九步
  - 表达式：234+*82/-;
  - 操作数栈：{10 }
- 计算结束

中缀表达式转换为后缀表达式

- 在实际的表达式计算中，往往将中缀表达式转换为后缀表达式，再对后缀表达式进行处理。
- 扫描中缀表达式并转换为后缀表达式
  - 是否需要操作数栈？为什么？ No
  - 是否需要运算符栈？为什么？YES
- 操作数出现的次序与中缀表达式相同；使用一个运算符栈来存放“(”和暂时还不能确定计算次序的运算符

中缀表达式转换为后缀表达式

- 当扫描到操作数时直接进入后缀式；
- 扫描到运算符准备进栈时：
  - 当前运算符优先级应高于栈顶运算符优先级，否则通过出栈来满足，出栈的运算符加入后缀表达式。

中缀表达式转换为后缀表达式

- 初始
  - 中缀：2*(3+4)-8/2
  - 后缀：
  - 运算符栈：{ }
- 第一步
  - 中缀：2;*(3+4)-8/2
  - 后缀：2
  - 运算符栈：{ }
- 第二步
  - 中缀：2*;(3+4)-8/2
  - 后缀：2
  - 运算符栈：{* }
- 第三步
  - 中缀：2*(;3+4)-8/2
  - 后缀：2
  - 运算符栈：{*,(}

中缀表达式转换为后缀表达式

- 第四步
  - 中缀：2*(3;+4)-8/2
  - 后缀：23
  - 运算符栈：{*,(}
- 第五步
  - 中缀：2*(3+;4)-8/2
  - 后缀：23
  - 运算符栈：{*,(,+}
- 第六步
  - 中缀：2*(3+4;)-8/2
  - 后缀：234
  - 运算符栈：{*,(,+ }
- 第七步
  - 中缀：2*(3+4);-8/2
  - 后缀：234+
  - 运算符栈：{* }

中缀表达式转换为后缀表达式

- 第八步
  - 中缀：2*(3+4)-;8/2
  - 后缀：234+*
  - 运算符栈：{- }
- 第九步
  - 中缀：2*(3+4)-8;/2
  - 后缀：234+*8
  - 运算符栈：{- }
- 第十步
  - 中缀：2*(3+4)-8/;2
  - 后缀：234+*8
  - 运算符栈：{-,/ }
- 第十一步
  - 中缀：2*(3+4)-8/2;
  - 后缀：234+*82
  - 运算符栈：{-,/ }

中缀表达式转换为后缀表达式

- 第十二步
  - 中缀：2*(3+4)-8/2;
  - 后缀：234+*82/
  - 运算符栈：{- }
- 第十三步
  - 中缀：2*(3+4)-8/2;
  - 后缀：234+*82/-
  - 运算符栈：{ }
  - 结束

后缀表达式

- 后缀表达式把运算符放在它的操作数之后。
- 2*(3+4) - 8/2的后缀表达式为234 + *82/ - 。
- 是否需要操作数栈？ YES
- 是否需要运算符栈？ No
- 后缀表达式里：
  - (1)没有括号
  - (2)操作数出现的次序与中缀表达式相同
  - (3)运算符出现的次序是计算次序，按优先级从高到底排列。

中缀表达式转换为后缀表达式

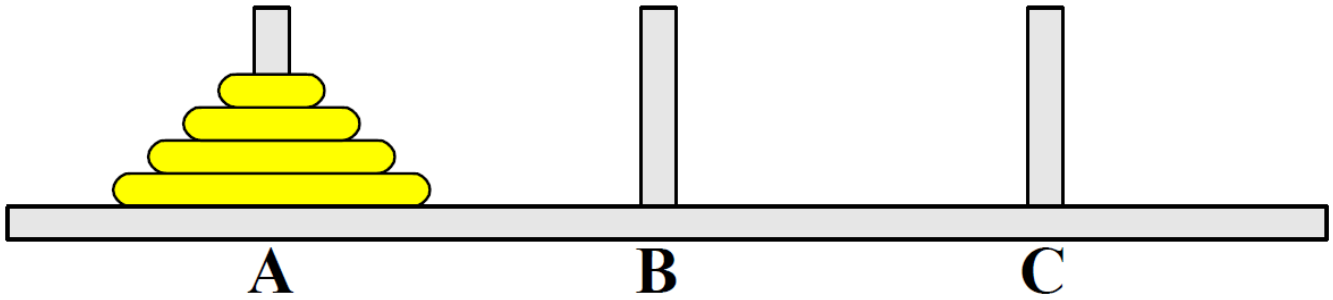
- 在实际的表达式计算中，往往将中缀表达式转换为后缀表达式，再对后缀表达式进行处理。
- 扫描中缀表达式并转换为后缀表达式
  - 是否需要操作数栈？为什么？ No
  - 是否需要运算符栈？为什么？YES
- 操作数出现的次序与中缀表达式相同；使用一个运算符栈来存放“(”和暂时还不能确定计算次序的运算符

实现?

- 算法伪代码是怎样的?

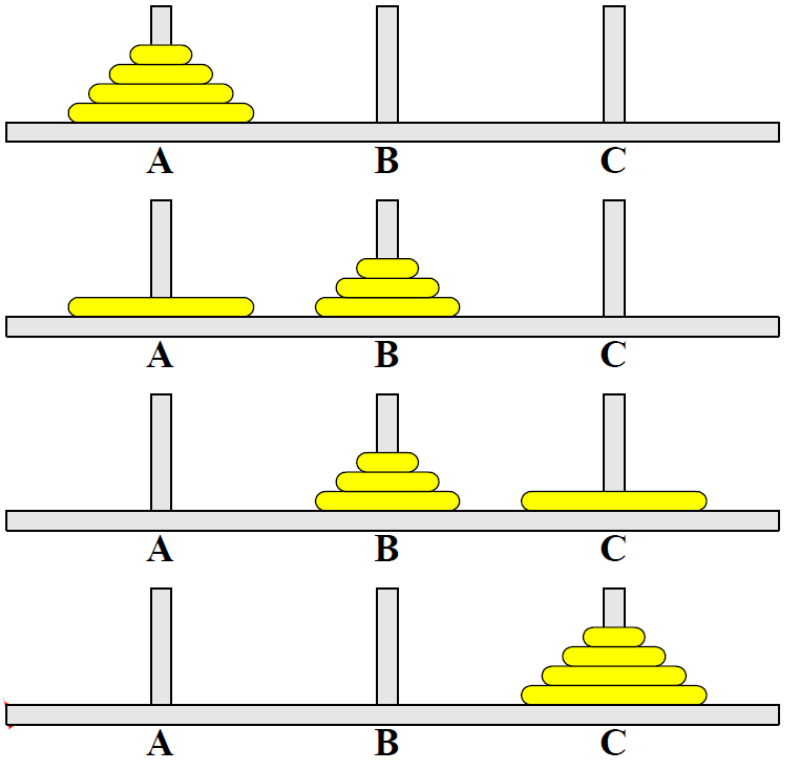
汉诺塔问题

- 汉诺塔问题： A、B、C三根柱子，A柱子上放有n个盘子，每个盘子都比下面的盘子小，要把A柱子上的盘子移到C柱子上，一次只能移动一个盘子，移动过程中大盘不能放在小盘的上面。



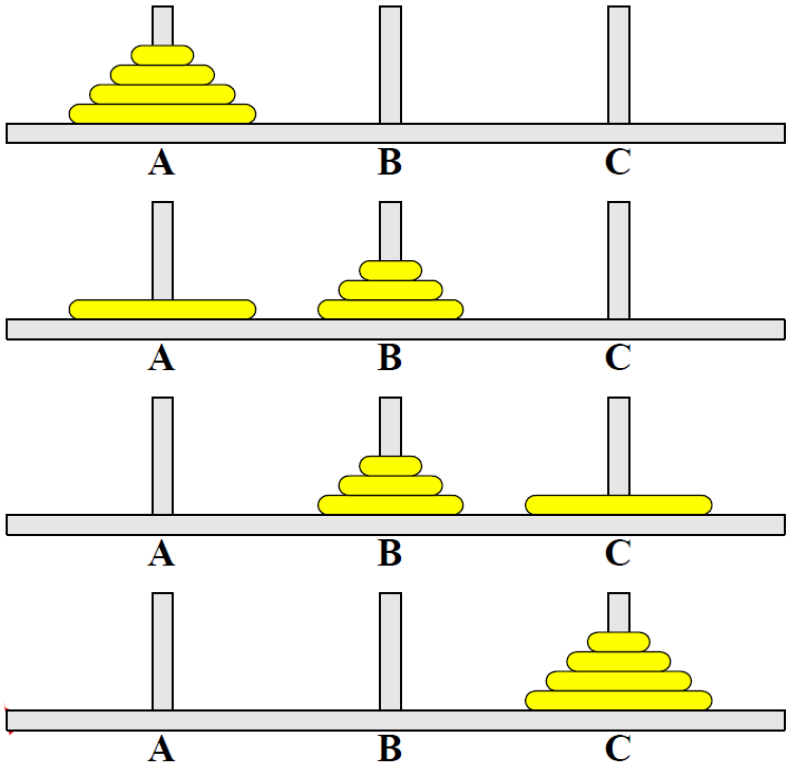
汉诺塔问题

- 如何分解该问题?



汉诺塔问题

1. 先将n-1个盘子从A柱子移到B柱子
2. 然后把最大的盘子移到C柱子
3. 再把n-1个盘子从B柱子移到C柱子。

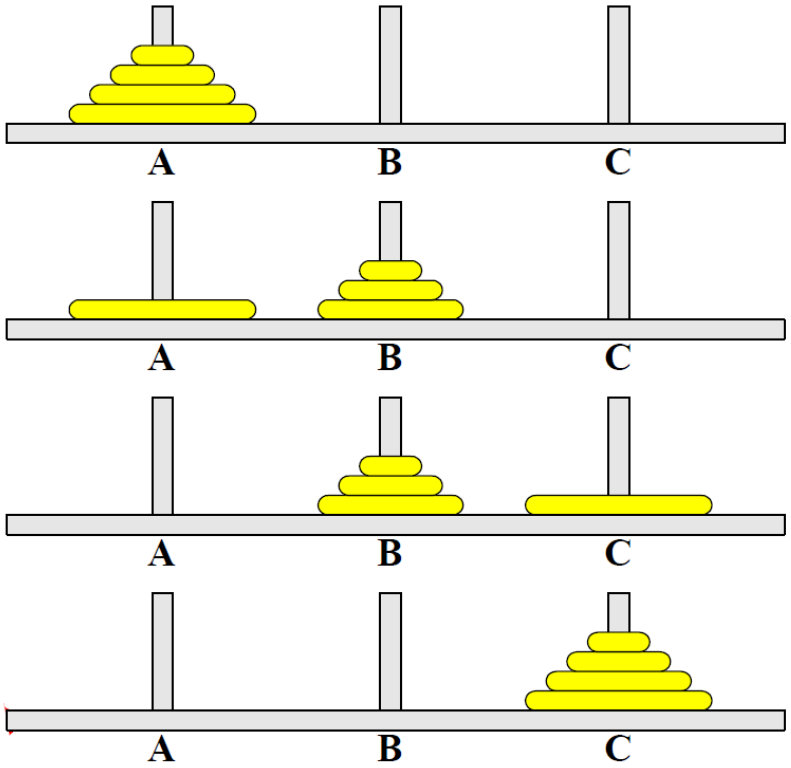


汉诺塔问题

- 为什么这样分解可以递归?
  - (1)和(3)都是n-1的汉诺塔问题
  - (2)是可以直接求解的最小子问题

汉诺塔问题

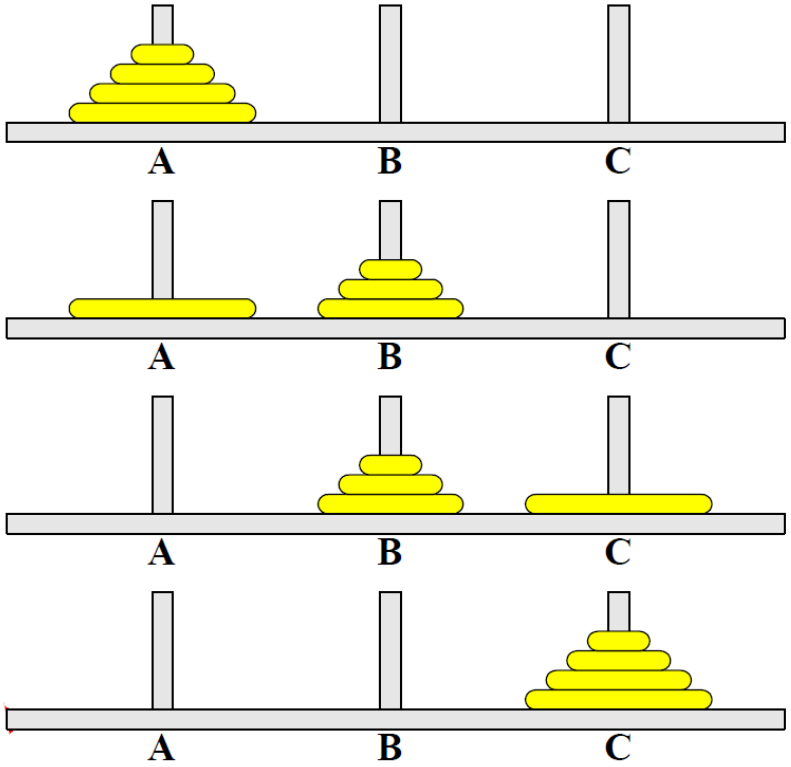
1. 先将n-1个盘子从A柱子移到B柱子
2. 然后把最大的盘子移到C柱子
3. 再把n-1个盘子从B柱子移到C柱子。



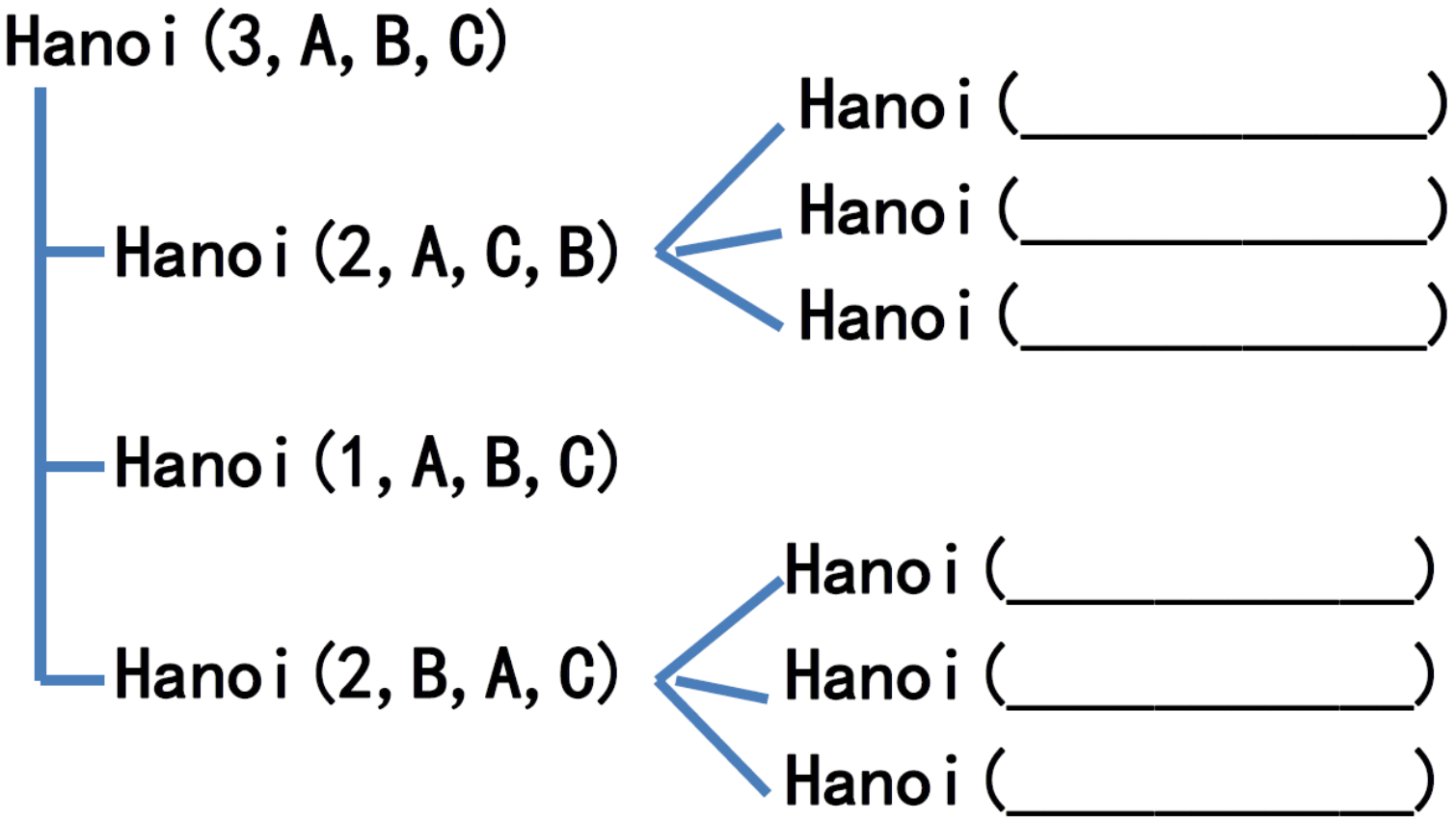
汉诺塔问题

Hanoi(n, A, B, C)

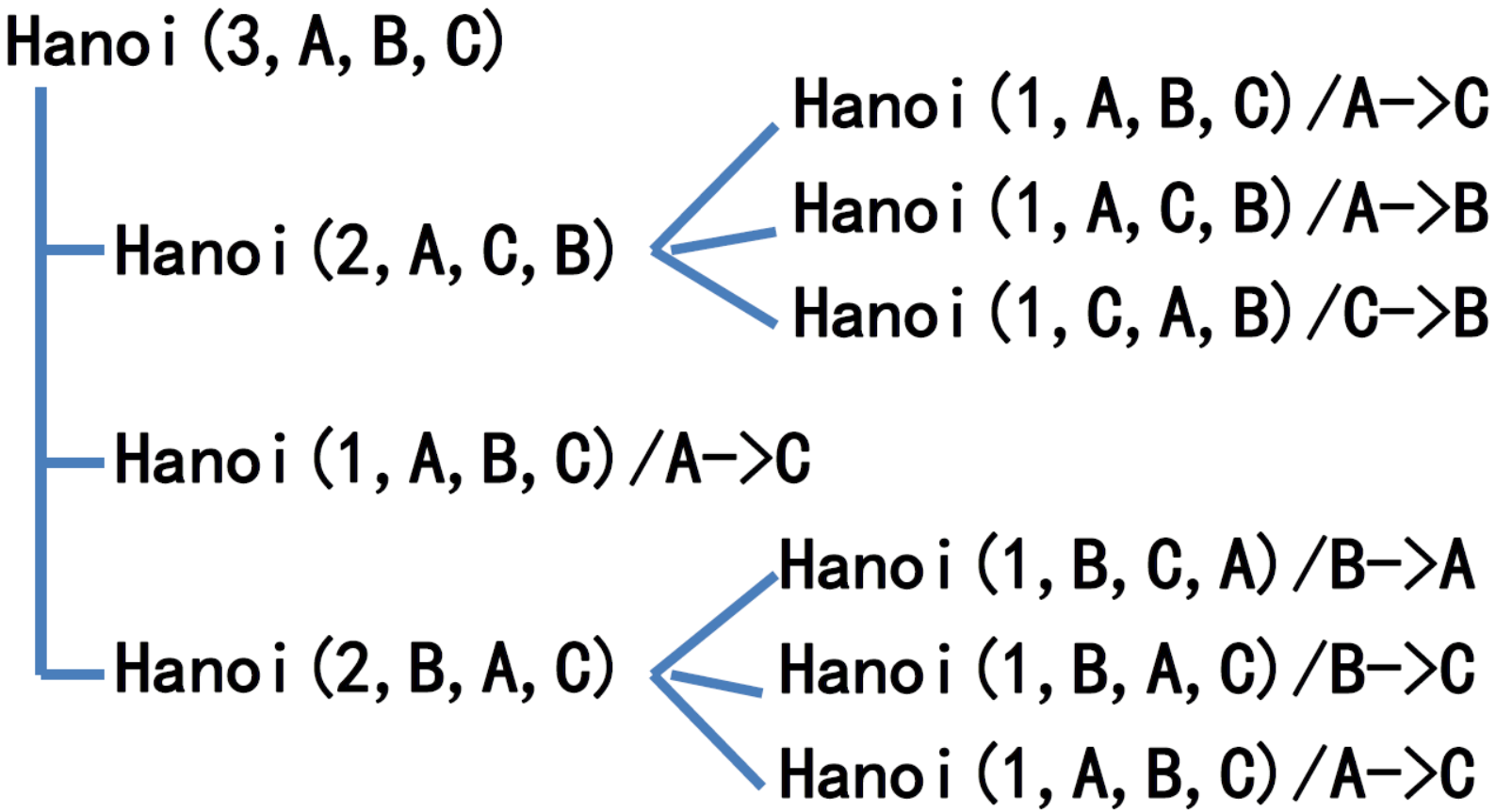
- Hanoi(n-1, A, C, B)
- Hanoi(1, A, B, C)/A->C
- Hanoi(n-1, B, A, C)



汉诺塔问题


Hanoi (3, A, B, C)
Hanoi (2, A, C, B)
Hanoi (_____)
Hanoi (_____)
Hanoi (_____)
Hanoi (1, A, B, C)
Hanoi (2, B, A, C)
Hanoi (_____)
Hanoi (_____)
Hanoi (_____)

汉诺塔问题


Hanoi (3, A, B, C)
Hanoi (2, A, C, B)
Hanoi (1, A, B, C)/A->C
Hanoi (1, A, C, B)/A->B
Hanoi (1, C, A, B)/C->B
Hanoi (1, A, B, C)/A->C
Hanoi (2, B, A, C)
Hanoi (1, B, C, A)/B->A
Hanoi (1, B, A, C)/B->C
Hanoi (1, A, B, C)/A->C

汉诺塔问题

```
void main(void)
{
    int n;
    char A =‘A’, B =‘B’, C =‘C’;
    printf("Enter the number of disks:");
    scanf("%d", &n);
    printf("The soltion for n=%d\n", n);
    Hanoi(n, A, B, C);
}
```

汉诺塔问题

```
void Hanoi(int n, char a, char b, char c)
{
    if (n == 1)
        printf(“move % c to % c\n”, a, c);
    else
    {
        Hanoi(n - 1, a, c, b);
        printf(“move % c to % c\n”, a, c);
        Hanoi(n - 1, b, a, c);
    }
}
```

递归

- 递归调用中，参数如何传递?
- 返回地址保存在哪里?

递归

- 为了保证函数调用（包括递归调用)的正确执行，必须解决调用时的参数传递和返回地址保存问题。用工作栈来解决这个问题。
- 每次函数调用时，将返回地址、实参、本层（被调用函数）的局部变量组成一个工作记录；该工作记录存入工作栈。每次调用返回就从工作栈退出一个工作记录。

递归

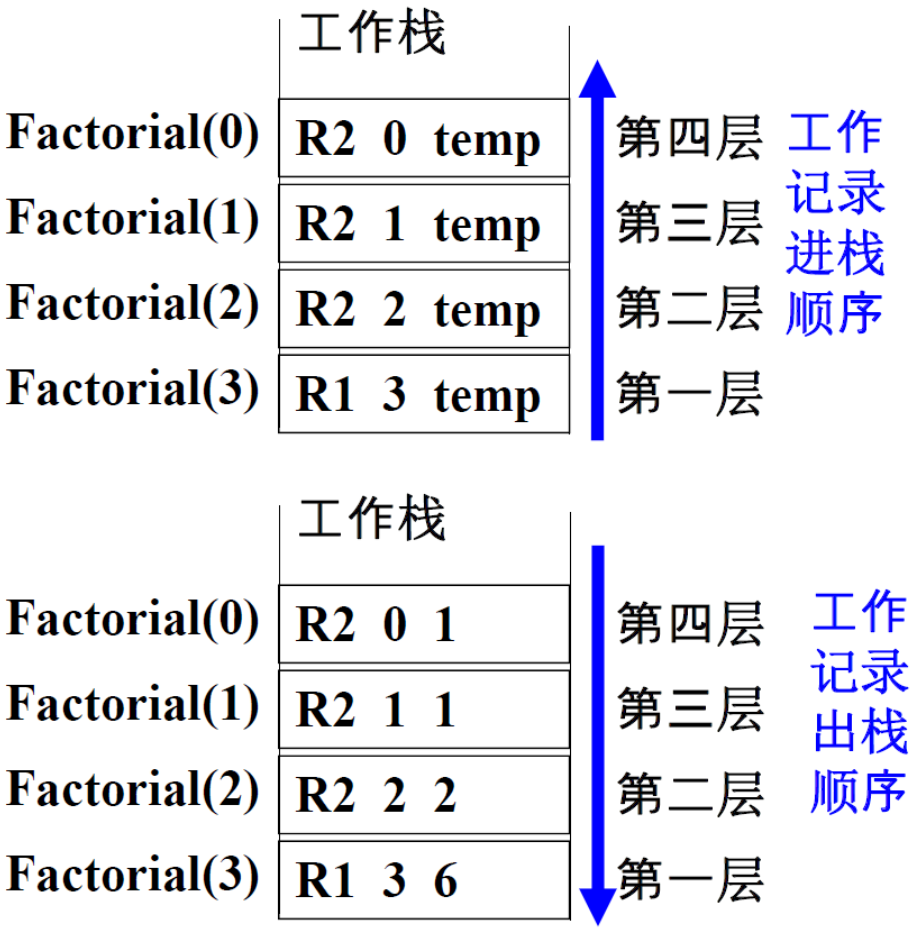
```
void main(void)
{
   int n;
   n = Factorial(3);
}
```


假设其返回地址为R1

```
int Factorial(int n)
{
int temp;
   if (n == 0)
       temp = 1;
   else
       temp = n * Factorial(n - 1);
return temp;
}
```

假设其返回地址为R2

递归


工作栈
Factorial(0)
R2 0 temp
Factorial(1)
R2 1 temp
Factorial(2)
R2 2 temp
Factorial(3)
R1 3 temp
第四层
第三层
第二层
第一层
工作记录进栈顺序
工作栈
Factorial(0)
R2 0 1
Factorial(1)
R2 1 1
Factorial(2)
R2 2 2
Factorial(3)
R1 3 6
第四层
第三层
第二层
第一层
工作记录出栈顺序

汉诺塔的递归分析

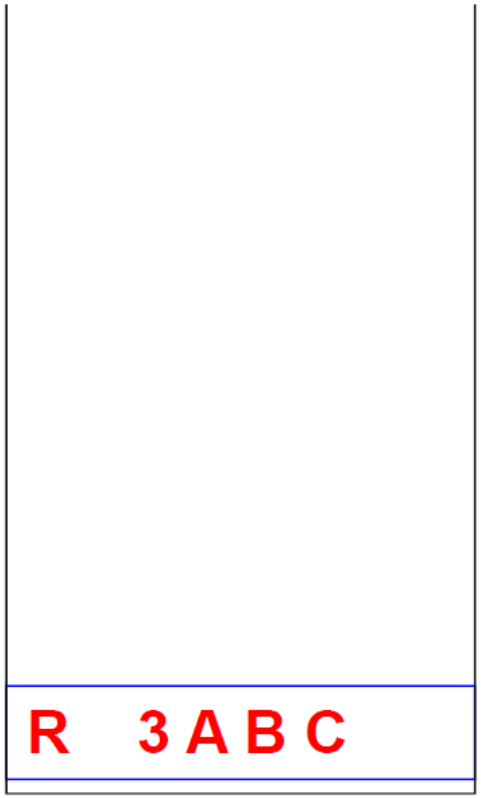
对n=3的汉诺塔问题，分析工作栈的变化过程，按出栈顺序写出出栈的记录。

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);          R1
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);          R2
   }
}
```

```
void main(void)
{
   int n;
   char A =‘A’, B =‘B’, C =‘C’;
   printf("Enter the number of disks:");
   scanf("%d", &n);
   printf("The soltion for n=%d\n", n);
   Hanoi(n, A, B, C);                 R
}
```

汉诺塔的递归分析

```
void main(void)
{
   int n;
   char A =‘A’, B =‘B’, C =‘C’;
   printf("Enter the number of disks:");
   scanf("%d", &n);
   printf("The soltion for n=%d\n", n);
   Hanoi(n, A, B, C);                    R
}
```

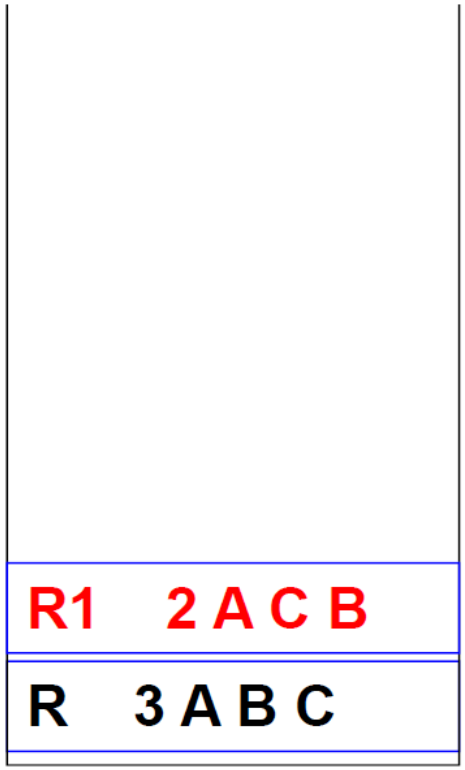


汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf("move % c to % c\n", a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf("move % c to % c\n", a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2

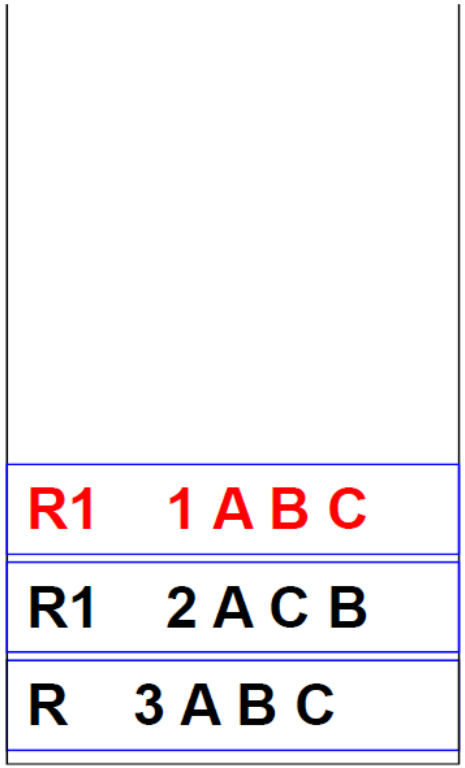


汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2

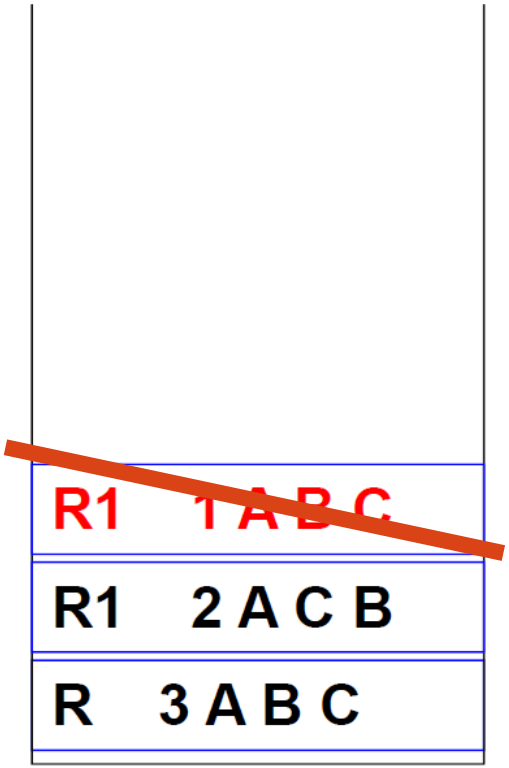


汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2

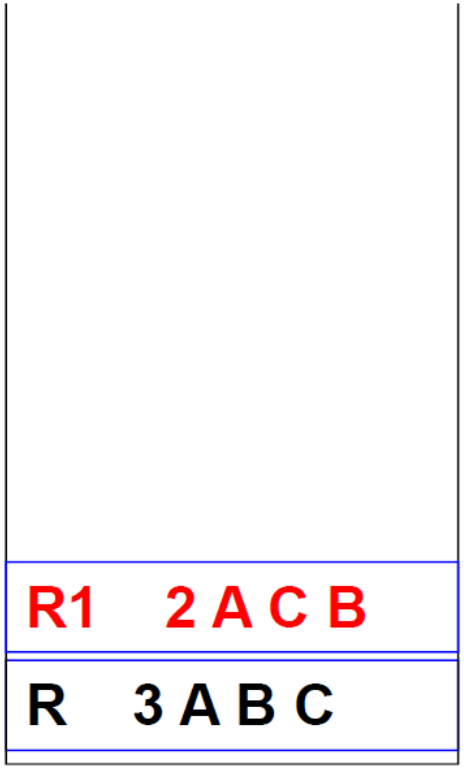


汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



A to C

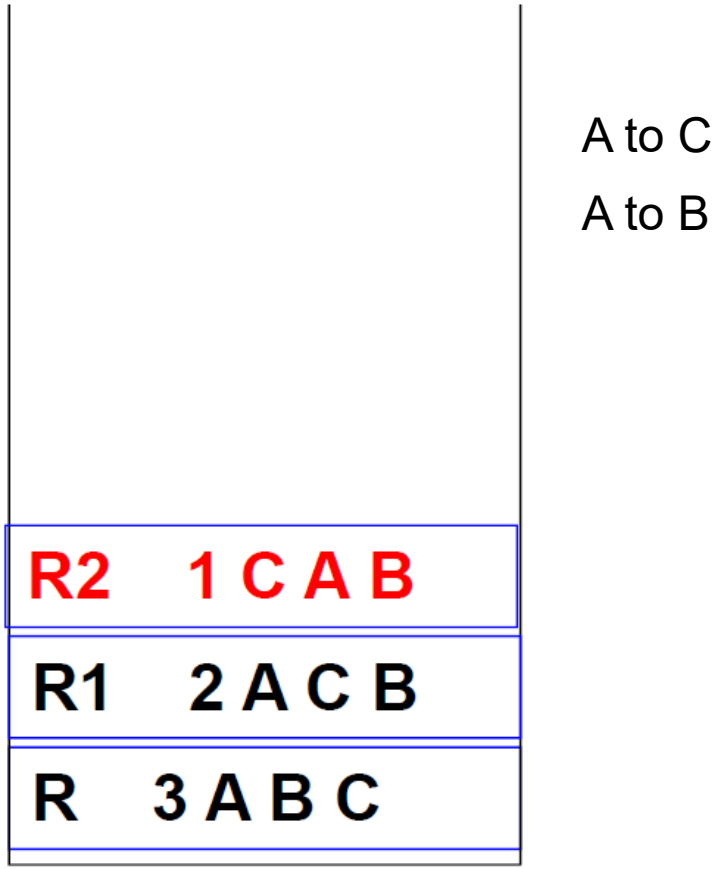
A to B

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2

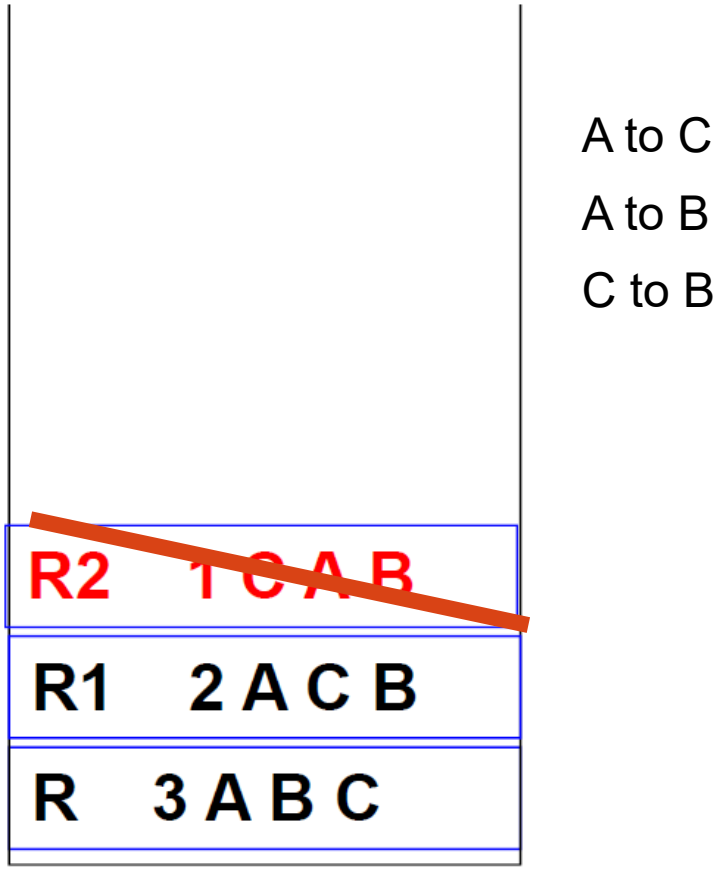


汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2

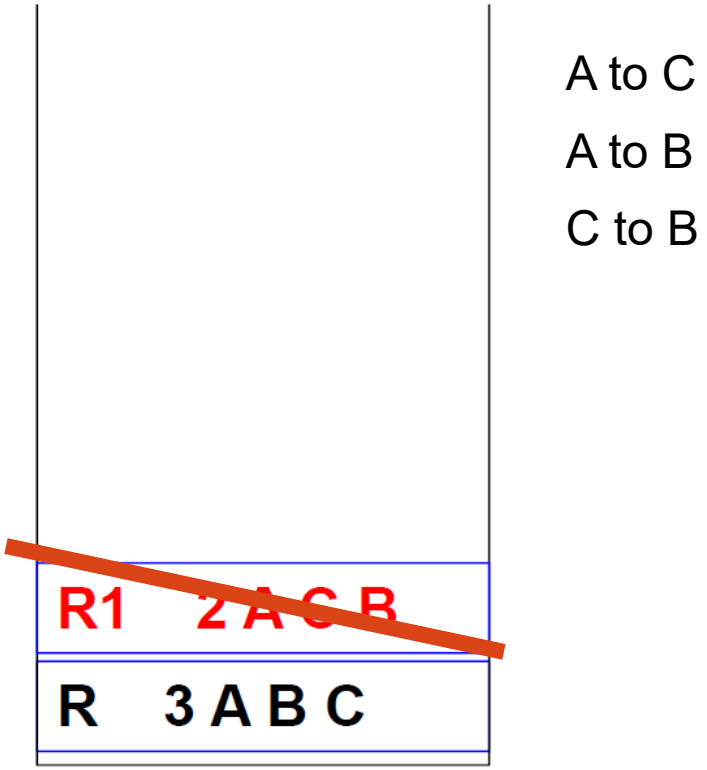


汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



A to C

A to B

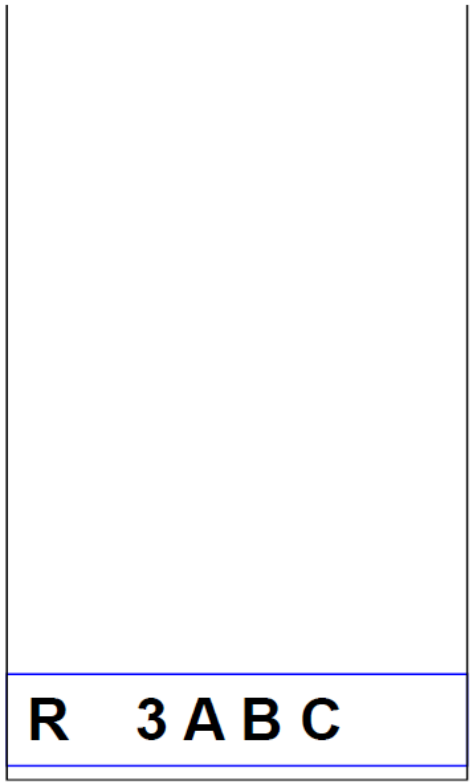
C to B

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



A to C

A to B

C to B

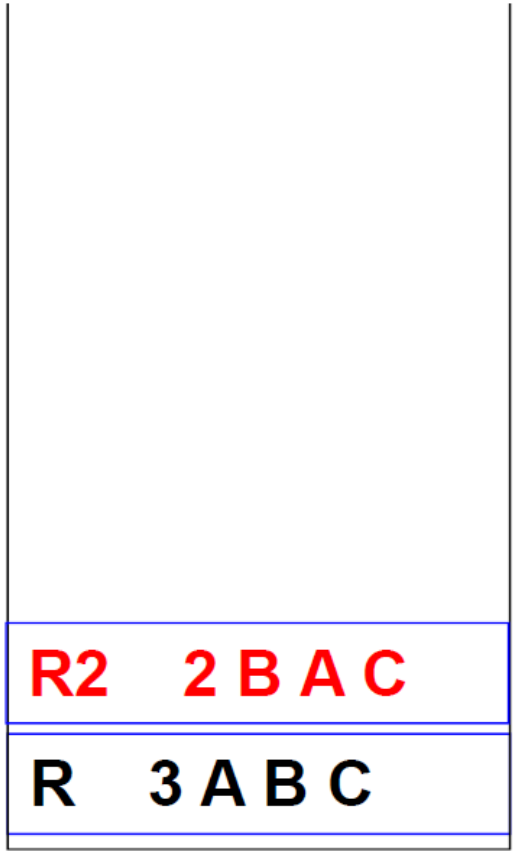
A to C

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



A to C

A to B

C to B

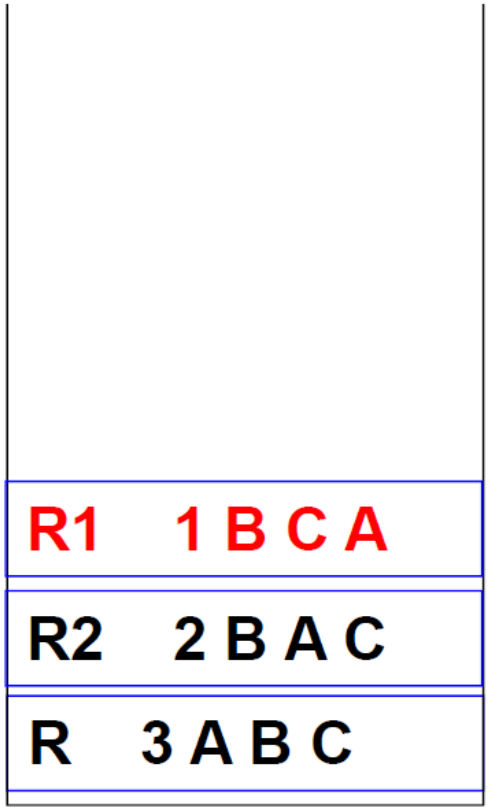
A to C

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf("move % c to % c\n", a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf("move % c to % c\n", a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



A to C

A to B

C to B

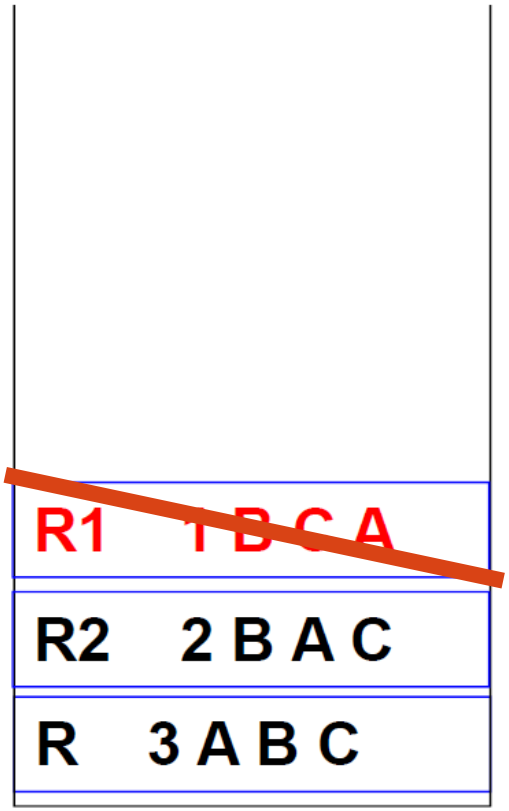
A to C

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf("move % c to % c\n", a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf("move % c to % c\n", a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



A to C

A to B

C to B

A to C

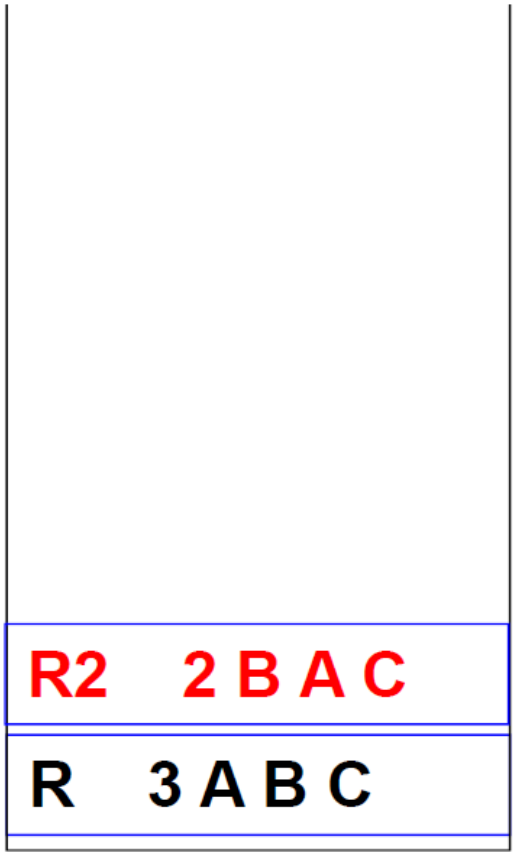
B to A

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



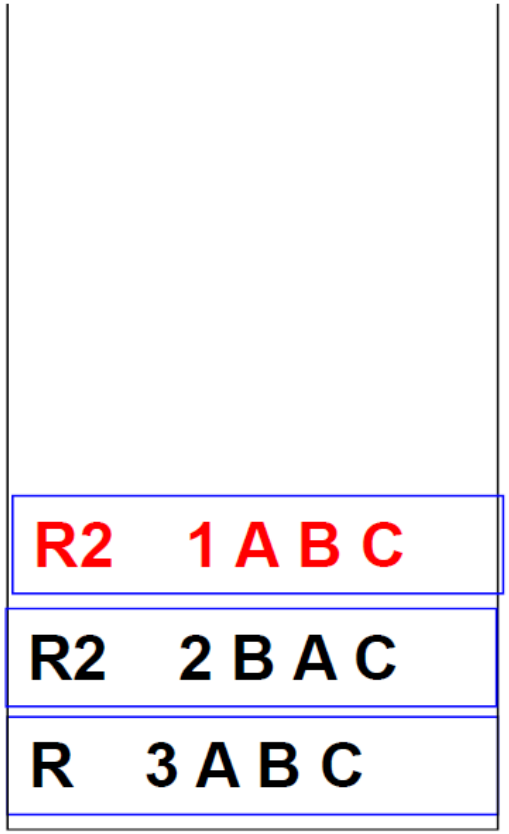
A to C
A to B
C to B
A to C
B to A
B to C

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



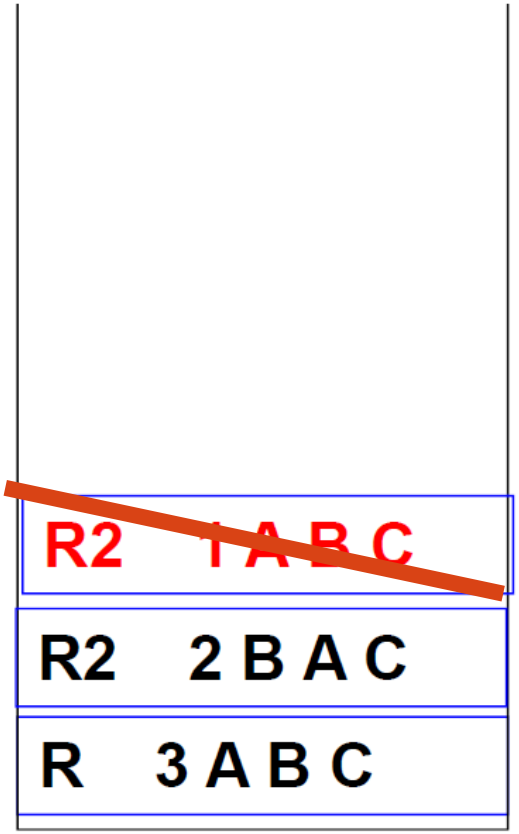
A to C
A to B
C to B
A to C
B to A
B to C

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf("move % c to % c\n", a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf("move % c to % c\n", a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2



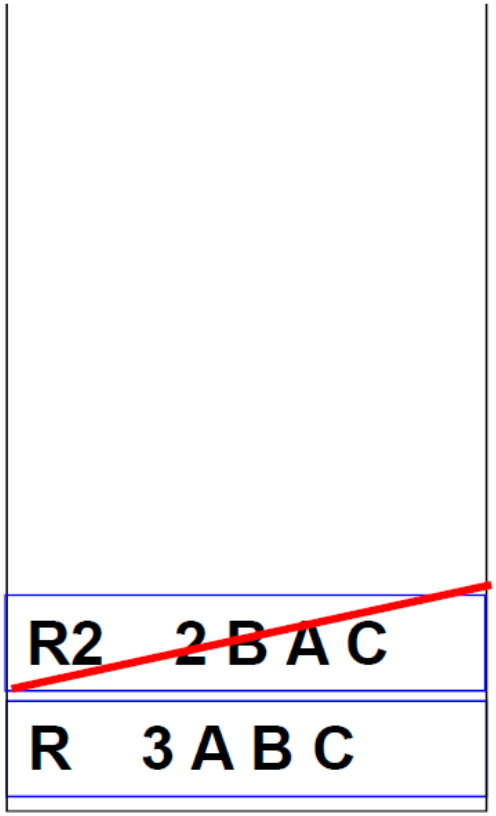
A to C
A to B
C to B
A to C
B to A
B to C
A to C

汉诺塔的递归分析

```
void Hanoi(int n, char a, char b, char c)
{
   if (n == 1)
      printf(“move % c to % c\n”, a, c);
   else
   {
      Hanoi(n - 1, a, c, b);
      printf(“move % c to % c\n”, a, c);
      Hanoi(n - 1, b, a, c);
   }
}
```

R1

R2

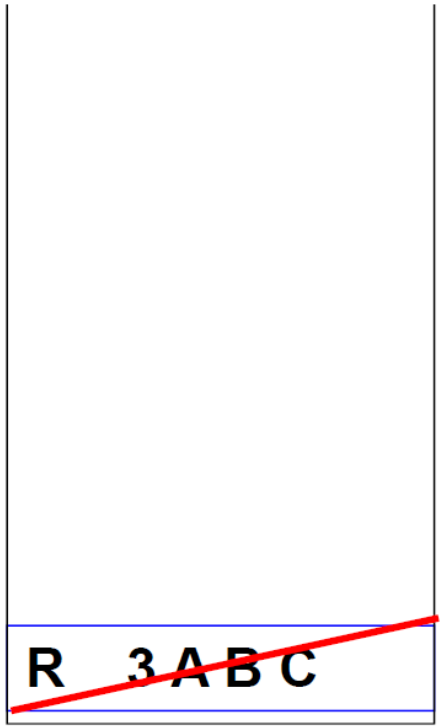


A to C
A to B
C to B
A to C
B to A
B to C
A to C

汉诺塔的递归分析

```
void main(void)
{
   int n;
   char A =‘A’, B =‘B’, C =‘C’;
   printf("Enter the number of disks:");
   scanf("%d", &n);
   printf("The soltion for n=%d\n", n);
   Hanoi(n, A, B, C);
}
```

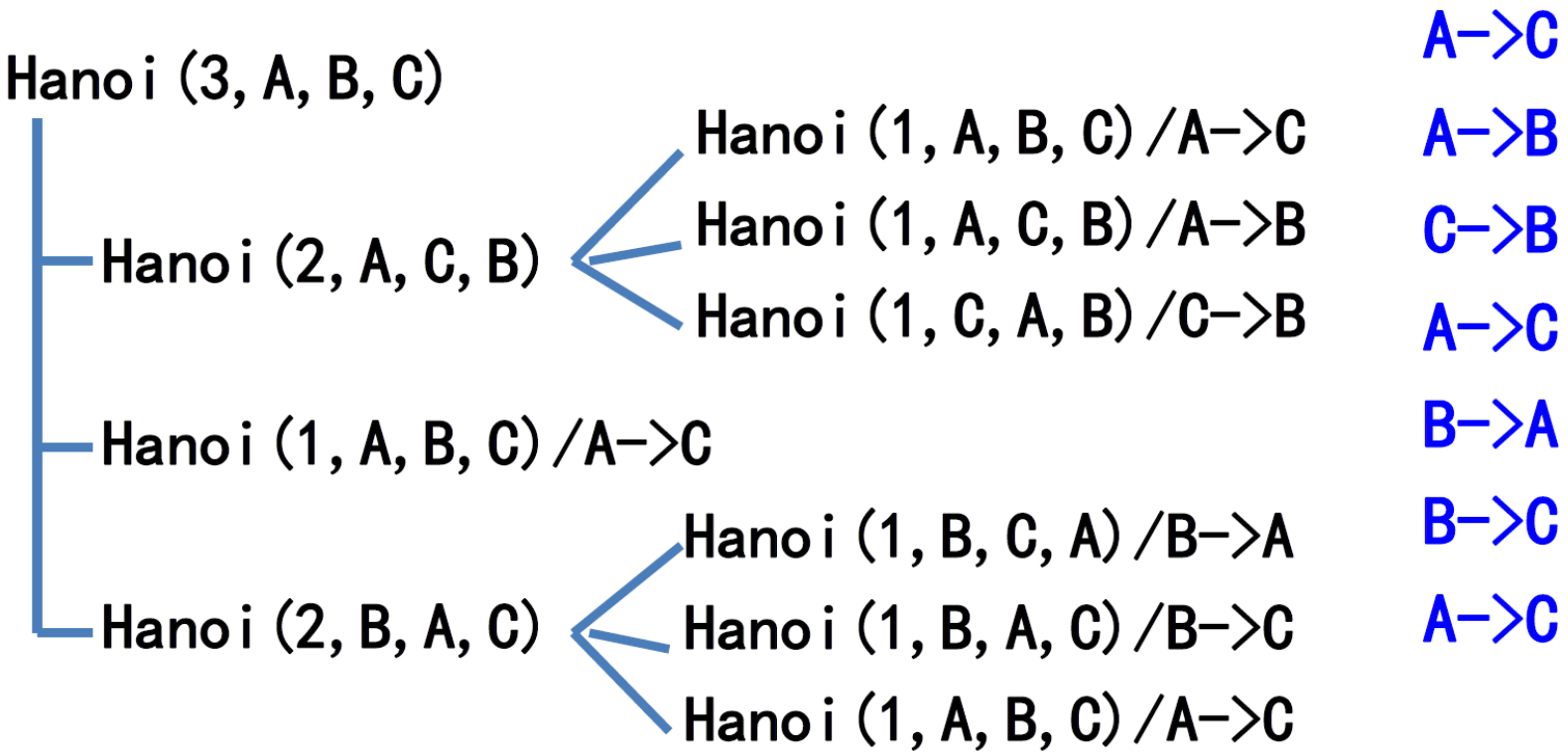
R



A to C
A to B
C to B
A to C
B to A
B to C
A to C

汉诺塔的递归分析

- 运行结果与递归分解树是什么关系？



递归

- 为了保证函数调用（包括递归调用)的正确执行，必须解决调用时的参数传递和返回地址保存问题。用工作栈来解决这个问题。
- 每次函数调用时，将返回地址、实参、本层（被调用函数）的局部变量组成一个工作记录；该工作记录存入工作栈。每次调用返回就从工作栈退出一个工作记录。

汉诺塔的时效分析

- 算法 $Hanoi(n, A, B, C)$
  - $T(n) = 2T(n-1) + 1,\ T(1) = 1$

$$T(n) = 2^n - 1$$

1 秒移 1 次，64 个盘子要多少时间？

5000 亿年！

千万亿次/秒，4 个多小时

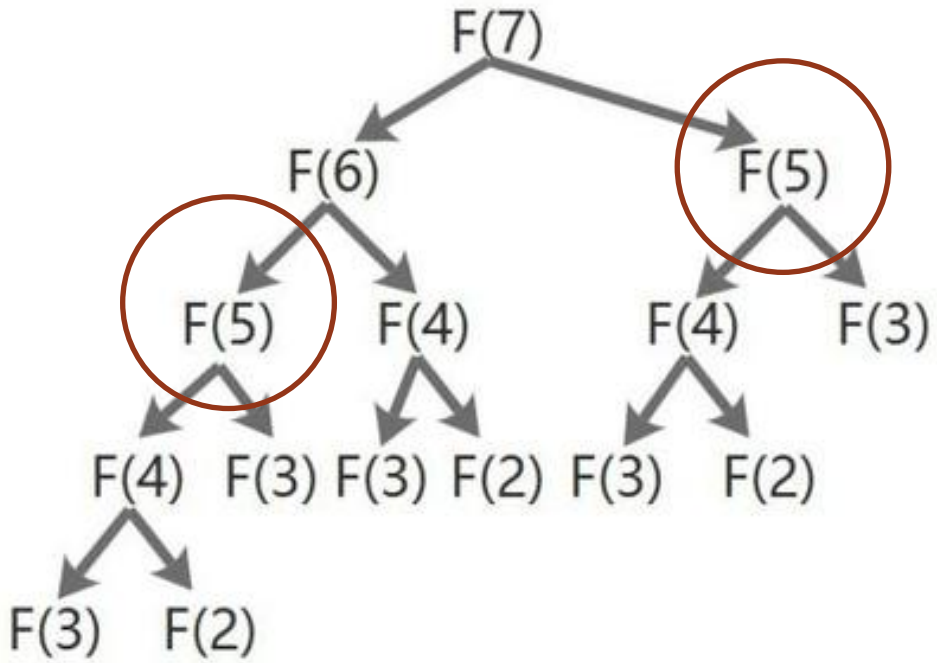
递归的缺点?

- 我们考虑斐波拉契数列问题

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

递归的缺点?

- 我们考虑斐波拉契数列问题



递归的缺点?

- 我们考虑斐波拉契数列问题
- 假如重复的计算我们都只做一次，那么请问Fib(n)的递归解法浪费了多少时间?