

# 1. 绪论

迭代乃人工，递归方神通

To iterate is human, to recurse, divine.

(e) 迭代与递归

凡治众如治寡，分數是也

The control of a large force is  
the same principle as  
the control of a few men:  
it is merely a question of  
dividing up their numbers.

邓俊辉

deng@tsinghua.edu.cn

## 数组求和：迭代

- ❖ 问题：计算任意n个整数之和
- ❖ 实现：逐一取出每个元素，累加之

```
int SumI(int A[], int n) {  
    int sum = 0; //O(1)  
    for (int i = 0; i < n; i++) //O(n)  
        sum += A[i]; //O(1)  
    return sum; //O(1)  
}
```

- ❖ 无论A[]内容如何，都有：

$$T(n) = 1 + n*1 + 1 = n + 2 = O(n) = \Omega(n) = \Theta(n)$$

- ❖ 空间呢？

## 减而治之

### ❖ 【Decrease-and-conquer】

为求解一个大规模的问题，可以

将其划分为两个子问题：其一**平凡**，另一规模**缩减**

//单调性

分别求解子问题

由子问题的解，得到原问题的解



## 数组求和 : 线性递归

```
❖ sum(int A[], int n) {  
    return  
        (n < 1) ?  
            0 : sum(A, n-1) + A[n-1];  
}
```

❖ 递归跟踪 (recursion trace) 分析

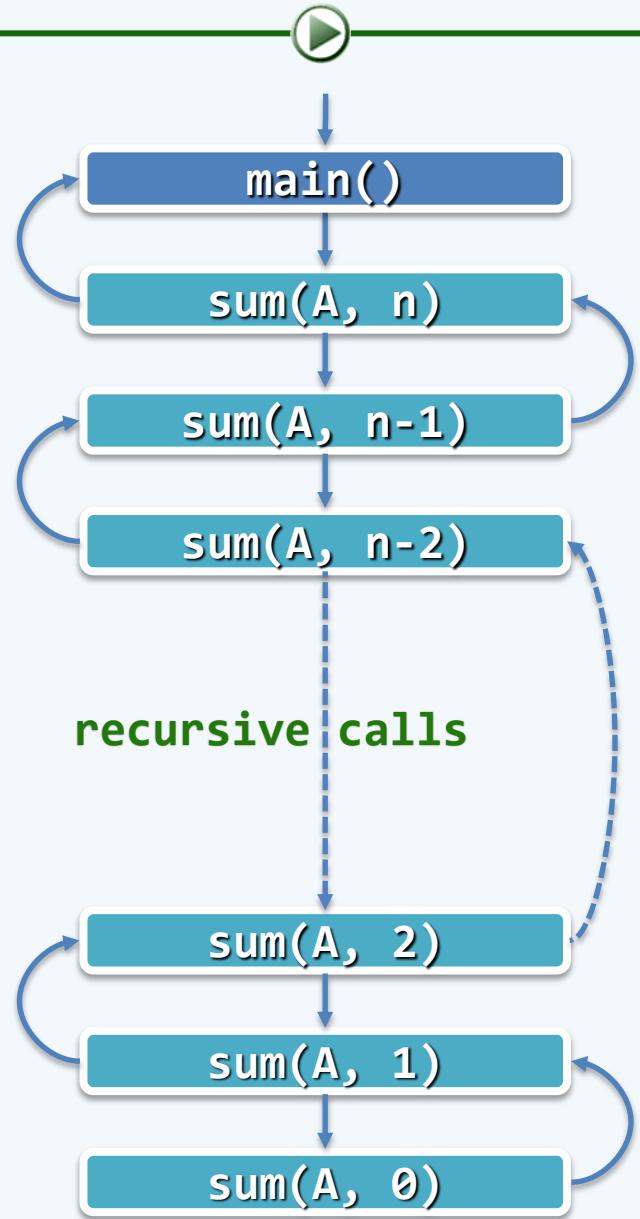
检查每个递归实例

累计所需时间 (调用语句本身, 计入对应的子实例)

其总和即算法执行时间

❖ 本例中, 单个递归实例自身只需  $\mathcal{O}(1)$  时间

$$T(n) = \mathcal{O}(1) * (n+1) = \mathcal{O}(n)$$



## 数组求和 : 线性递归

❖ 从递推的角度看，为求解 $\text{sum}(A, n)$ ，需

递归求解规模为 $n-1$ 的问题 $\text{sum}(A, n-1)$  // $T(n-1)$

再累加上 $A[n-1]$  // $O(1)$

递归基： $\text{sum}(A, 0)$  // $O(1)$

❖ 递推方程  $T(n) = T(n-1) + O(1)$  //recurrence

$T(0) = O(1)$  //base

❖ 求解  $T(n) - n = T(n-1) - (n-1) = \dots$

$$= T(2) - 2$$

$$= T(1) - 1$$

$$= T(0)$$

$$T(n) = O(1) + n = O(n)$$

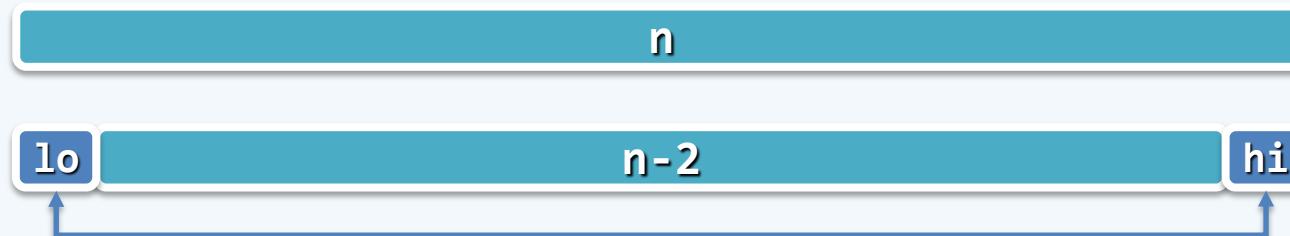
## 数组倒置

❖ 任给数组A[0, n) , 将其中的子区间A[lo, hi]前后颠倒

统一接口 : void reverse(int\* A, int lo, int hi);

❖ if (lo < hi) //问题规模的奇偶性不变，需要两个递归基 //递归版

```
{ swap(A[lo], A[hi]); reverse(A, lo + 1, hi - 1); }
```



❖ next: //迭代原始版

```
if (lo < hi)
{ swap(A[lo], A[hi]); lo++; hi--; goto next; }
```

❖ while (lo < hi) swap(A[lo++], A[hi--]); //迭代精简版

## 分而治之

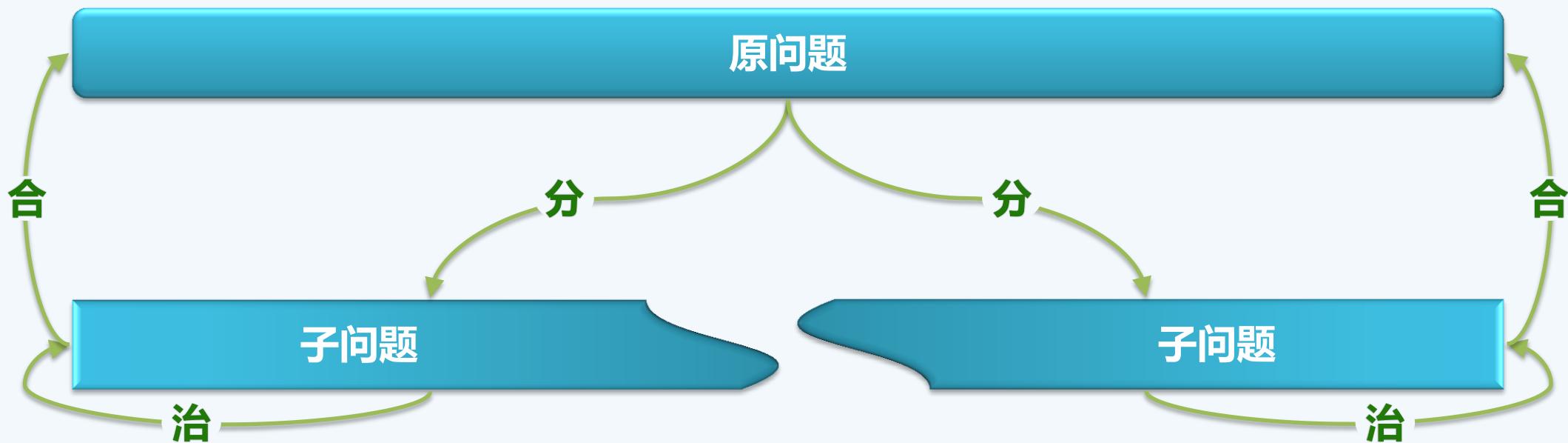
### ❖ 【Divide-and-conquer】

为求解一个大规模的问题，可以

将其划分为若干（通常两个）子问题，规模大体相当

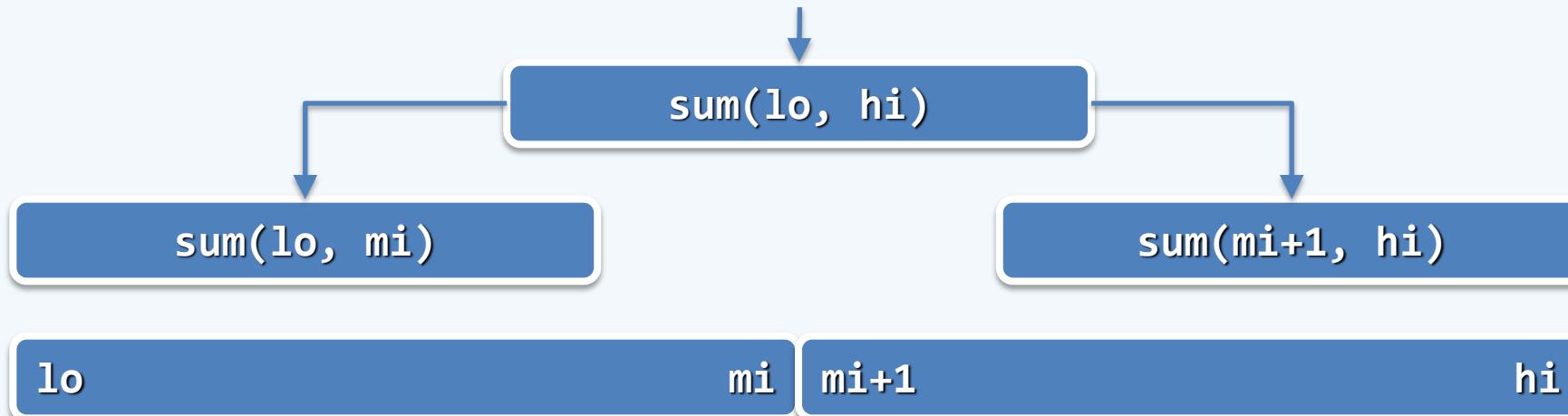
分别求解子问题

由子问题的解，得到原问题的解



## 数组求和：二分递归

```
❖ sum(int A[], int lo, int hi) { //区间范围A[lo, hi]  
    if (lo == hi) return A[lo];  
    int mi = (lo + hi) >> 1;  
    return sum(A, lo, mi) + sum(A, mi + 1, hi);  
} //入口形式为sum(A, 0, n-1)
```



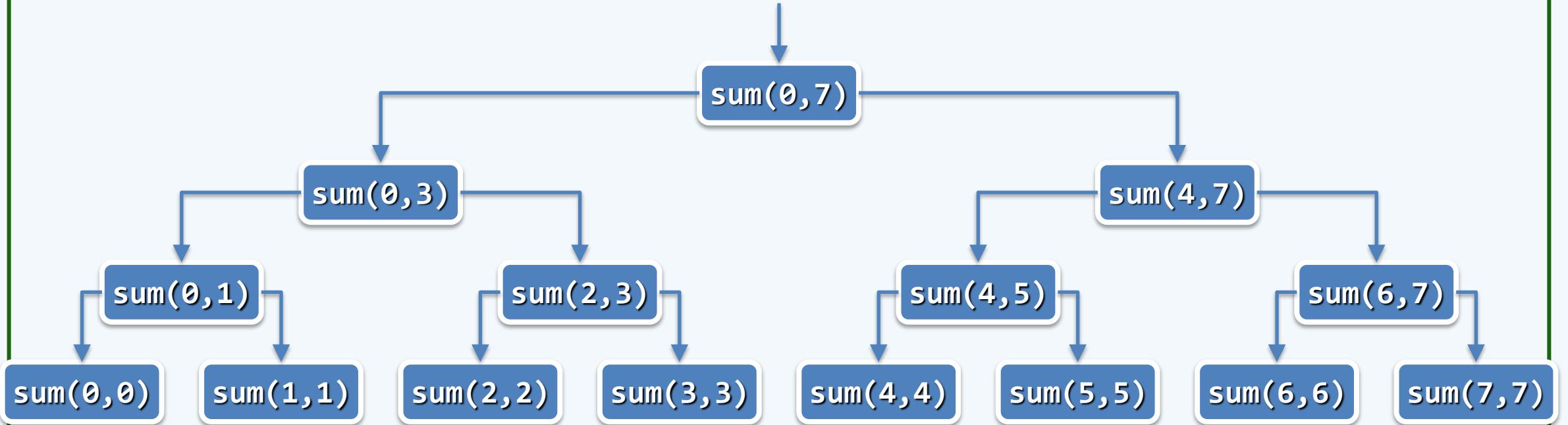
## 数组求和 : 二分递归

❖  $T(n) = \text{各层递归实例所需时间之和}$

// 递归跟踪

$$= O(1) \times (2^0 + 2^1 + 2^2 + \dots + 2^{\log n})$$

$$= O(1) \times (2^{\log n+1} - 1) = O(n)$$



## 数组求和 : 二分递归

❖ 从递推的角度看，为求解 $\text{sum}(A, \text{lo}, \text{hi})$ ，需

递归求解 $\text{sum}(A, \text{lo}, \text{mi})$ 和 $\text{sum}(A, \text{mi} + 1, \text{hi})$  // $2*T(n/2)$

进而将子问题的解累加 // $O(1)$

递归基： $\text{sum}(A, \text{lo}, \text{lo})$  // $O(1)$

❖ 递推关系  $T(n) = 2*T(n/2) + O(1)$

$$T(1) = O(1)$$

❖ 求解  $T(n) = 2*T(n/2) + c_1$

$$T(n) + c_1 = 2*(T(n/2) + c_1) = 2^2*(T(n/4) + c_1)$$

$$= \dots$$

$$= 2^{\log n} (T(1) + c_1) = n * (c_2 + c_1)$$

$$T(n) = (c_1 + c_2)n - c_1 = O(n)$$

## Max2 : 迭代1

- ❖ 从数组区间A[lo, hi)中找出最大的两个整数A[x1]和A[x2]  $\quad // A[x1] \geq A[x2]$   
元素比较的次数，要求尽可能地少

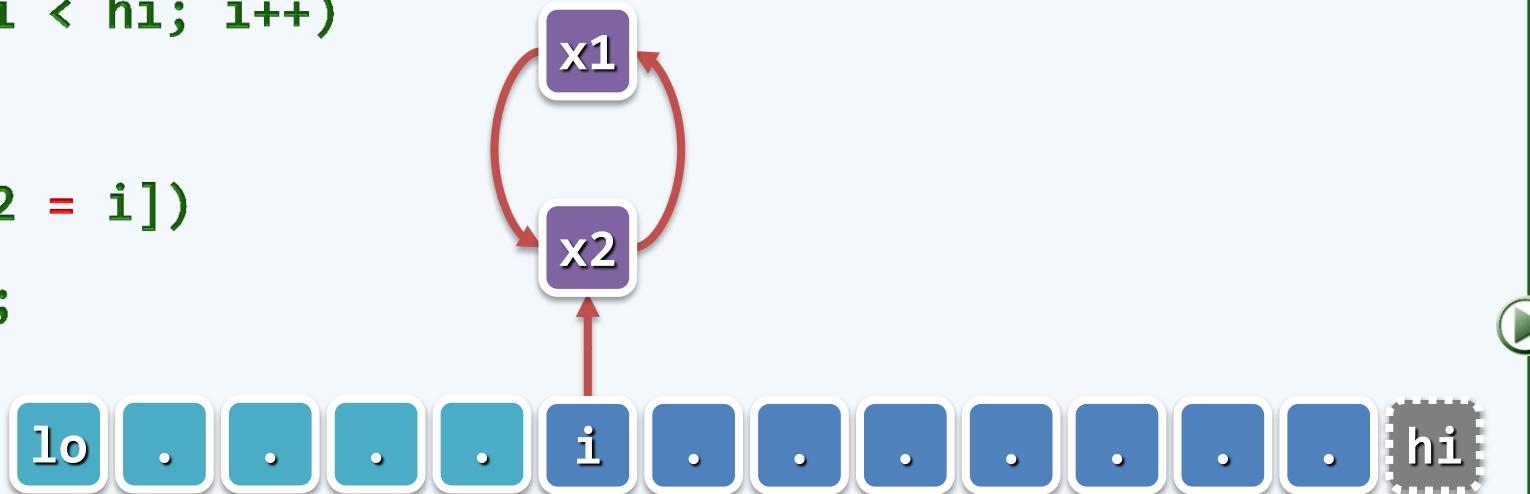
```
❖ void max2(int A[], int lo, int hi, int & x1, int & x2) { // 1 < n = hi - lo
    for (x1 = lo, int i = lo + 1; i < hi; i++) //扫描A[lo, hi), 找出A[x1]
        if (A[x1] < A[i]) x1 = i; // hi - lo - 1 = n - 1
    for (x2 = lo, int i = lo + 1; i < x1; i++) //扫描A[lo, x1)
        if (A[x2] < A[i]) x2 = i; // x1 - lo - 1
    for (int i = x1 + 1; i < hi; i++) //再扫描A(x1, hi), 找出A[x2]
        if (A[x2] < A[i]) x2 = i; // hi - x1 - 1
}
```



- ❖ 无论如何，比较次数总是 $\Theta(2n - 3)$

## Max2 : 迭代2

```
❖ void max2(int A[], int lo, int hi, int & x1, int & x2) { // 1 < n = hi - lo  
    if (A[x1 = lo] < A[x2 = lo + 1]) swap(x1, x2);  
    for (int i = lo + 2; i < hi; i++)  
        if (A[x2] < A[i])  
            if (A[x1] < A[x2 = i])  
                swap(x1, x2);  
}
```



❖ 最好情况 ,  $1 + (n - 2) * 1 = n - 1$

❖ 最坏情况 ,  $1 + (n - 2) * 2 = 2n - 3$

❖ 比较次数可否进一步减少呢 ? 分而治之 !

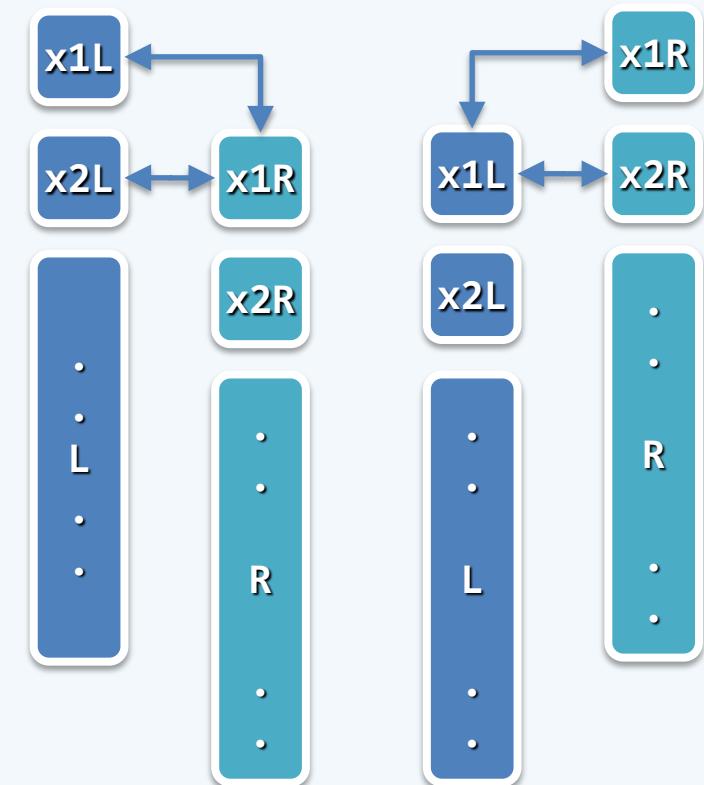
## Max2：递归 + 分治



```
❖ void max2(int A[], int lo, int hi, int & x1, int & x2) {  
    if (lo + 2 == hi) { /* ... */; return; } // T(2) = 1  
    if (lo + 3 == hi) { /* ... */; return; } // T(3) <= 3  
    int mi = (lo + hi)/2; //divide  
    int x1L, x2L; max2(A, lo, mi, x1L, x2L);  
    int x1R, x2R; max2(A, mi, hi, x1R, x2R);  
    if (A[x1L] > A[x1R]) {  
        x1 = x1L; x2 = (A[x2L] > A[x1R]) ? x2L : x1R;  
    } else {  
        x1 = x1R; x2 = (A[x1L] > A[x2R]) ? x1L : x2R;  
    } // 1 + 1 = 2  
} // T(n) = 2*T(n/2) + 2 <= 5n/3 - 2
```

The diagram illustrates the recursive division of an array segment [lo, hi] into four parts: x1L, x2L, x1R, and x2R. The segments are represented by blue boxes with arrows pointing to them from the corresponding lines of code. The segments are arranged vertically as follows:

- x1L
- x2L
- ⋮
- L
- ⋮



## 典型的递推方程

❖ 更多求解模式及规律：[AHU-74]， p64, (Master) Theorem 2.1

递推式	解	实例
$T(n) = T(n-1) + 1$	$\Theta(n)$	向量求和之线性递归版
$T(n) = T(n-1) + n$	$\Theta(n^2)$	列表起泡排序之线性递归版
$T(n) = 2*T(n-1) + 1$	$\Theta(2^n)$	Hanoi塔、Fibonacci数
$T(n) = 2*T(n-1) + n$	$\Theta(2^n)$	
$T(n) = T(n/2) + 1$	$\Theta(\log n)$	向量的二分查找
$T(n) = T(n/2) + n$	$\Theta(n)$	列表的二分查找
$T(n) = 2*T(n/2) + 1$	$\Theta(n)$	向量求和之二分递归版
$T(n) = 2*T(n/2) + n$	$\Theta(n \log n)$	归并排序

## 递归消除：尾递归

- ❖ 递归算法易于理解和实现，但空间（甚至时间）效率低  
在讲求效率时，应将递归改写为等价的迭代形式

- ❖ 尾递归：最后一步是递归调用  
最简单的递归模式，可便捷地改写

<pre>fac(n) { //递归     if (1 &gt; n) return 1;     else return n*fac(n-1); } //O(n) + O(n)</pre>	<pre>fac(n) { //迭代     int f = 1; //记录子问题的解     next: //转向标志，模拟递归调用     if (1 &gt; n) return f;     f *= n--;     goto next; //模拟递归返回 } //O(n) + O(1)</pre>	<pre>fac(n) { //迭代     int f = 1;     while (1 &lt; n)         f *= n--;     return f; } //O(n) + O(1)</pre>
--	---	--

多分支递归

二分递归

线性递归

尾递归

- ❖ 做递归跟踪分析时，为什么递归调用语句本身可不统计？
- ❖ 试用递归跟踪法，分析fib()二分递归版的复杂度  
通过递归跟踪，解释该版本复杂度过高的原因
- ❖ 递归算法的空间复杂度，主要取决于什么因素？
- ❖ 本节数组求和问题的两个（线性和二分）递归算法  
时间复杂度相同，空间呢？
- ❖ 自学递推式的一般性求解方法及规律  
`google("master theorem")`