1.5 绪论-迭代与递归

备注	课后作业！
学习日	@2020/03/08
视频起始	https://www.bilibili.com/video/av75509584?p=18
讲义	01.Introduction.E.Iteration_Recursion.pdf

▼ 01-E-1 迭代与递归

❖ 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)
}
```

对于时间复杂度的计算：

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

▼ 01-E-2 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

分别求解子问题

由子问题的解，得到原问题的解

规模会不断缩减，或者其中变成复杂度为O(1)的子问题

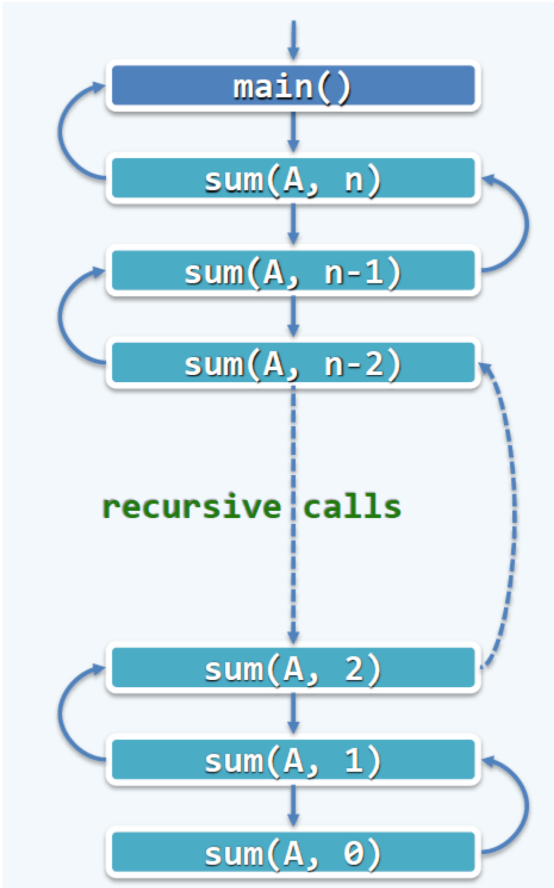
▼ 01-E-3 递归跟踪，递归方程

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

- 递归跟踪 recursion trace

  检查每一个递归示例 → 相当于是一个子问题；累计所需的时间；然后计算所得总和即为整体算法的时间

  递归计算一个数组的和，那么整体算法顺序图则为：



因此，对应的算法复杂度如下：

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

- 递推方程

❖ **从递推的角度看，为求解sum(A, n)，需**

　　递归求解规模为n-1的问题sum(A, n-1)　　//T(n-1)

　　再累加上A[n-1]　　//O(1)

递归基：sum(A, 0)　　//O(1)

然后解方程求解，得到显式的递归方程：

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

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

❖ **求解** $T(n) - n = T(n-1) - (n-1) = \ldots$

$= T(2) - 2$

$= T(1) - 1$

$= T(0)$

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

▼ 01-E-4 例-数组倒置

❖ **任给数组A[0, n)，将其前后颠倒** //更一般地，子区间A[lo, hi]

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

❖ **递归版**

```
if (lo < hi) //问题规模的奇偶性不变，需要两个递归基
   { swap(A[lo], A[hi]); reverse(A, lo + 1, hi - 1); }
```

一步步减而治之，问题会不断缩减为原来n-2的范围

进一步地，因为问题可能会有奇偶性的问题，所以最终会有两个递归基

```
void reverse ( int* A, int lo, int hi ) { //数组倒置（多递归基递归版）
   if ( lo < hi ) {
      swap ( A[lo], A[hi] ); //交换A[lo]和A[hi]
      reverse ( A, lo + 1, hi - 1 ); //递归倒置A(lo, hi)
   } //else隐含了两种递归基
} //O(hi - lo + 1)
```

除此之外，数组倒置还有其他的精简方法，可见：

- 迭代原始版

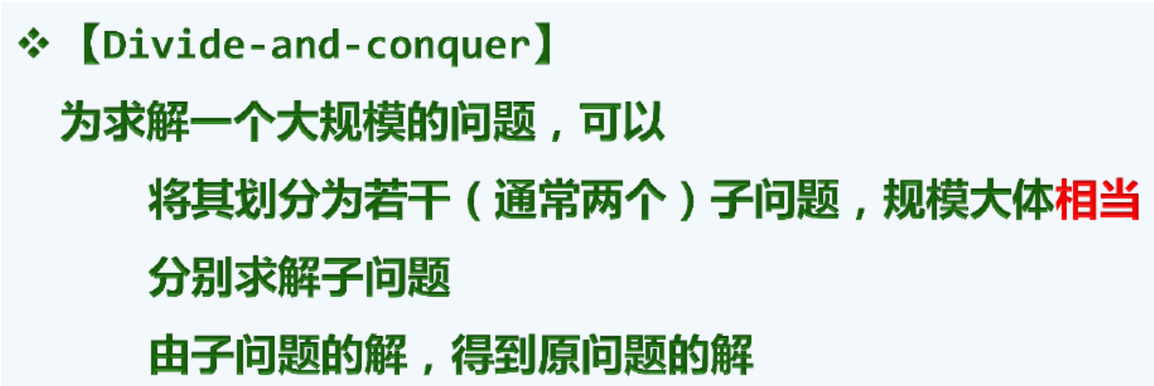
https://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/reverse/reverse-iterative-0.cpp.htm

- 递归精简版

reverse-iterative-1.cpp
https://dsa.cs.tsinghua.edu.cn/~deng/ds/src_link/reverse/reverse-iterative-1.cpp.htm

▼ 01-E-5 分而治之

divide-and-conquer

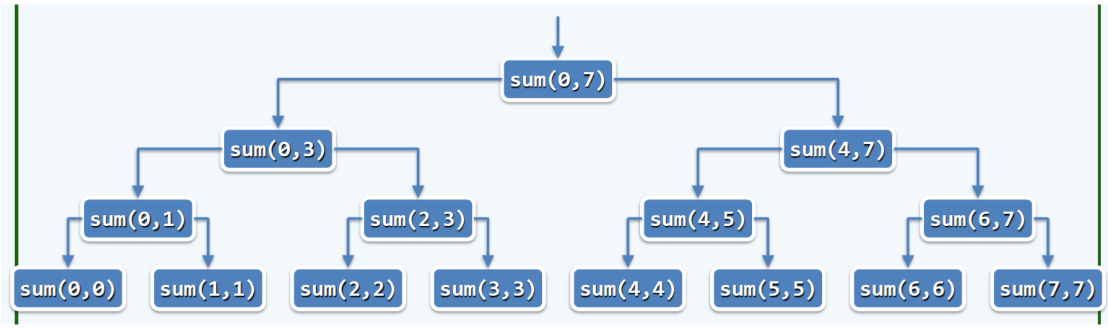


▼ 01-E-6 例-数组求和-分而治之

```
int sum ( int A[], int lo, int hi ) { //数组求和算法（二分递归版，入口为sum(A, 0, n))
   if ( hi - lo < 2 ) return A[lo]; //递归基：区间宽度不足2
   int mi = ( lo + hi ) >> 1; // （否则）均分原区间
   return sum ( A, lo, mi ) + sum ( A, mi, hi ); //递归求和，然后合计
} //O(hi - lo)，线性正比于区间的长度
```

采用分而治之的方法，就是将求和分为一半一半的子问题进行求解！

- 递归跟踪，整体的算法复杂度等于示例所需时间之和



$$T(n) = \mathcal{O}(1) \times (2^0 + 2^1 + \ldots + 2^{\log n}) = \mathcal{O}(1) \times (2^{\log n+1} - 1) = \mathcal{O}(n)$$

- 递推方程

  两个子问题 2*T(n/2) + 一个累加 O(1)

❖ **递推关系** T(n) = 2*T(n/2) + O(1)

T(1) = $\mathcal{O}$(1)

❖ **求解** T(n) = 2*T(n/2) + c_1

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

$$= \ldots$$

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

$$T(n) = (c_1+c_2)n - c_1 = \mathcal{O}(n)$$

▼ 01-E-7 例-MAX2

从数组区间A[lo, hi)种找出最大的两个整数A[x1]和A[x2]

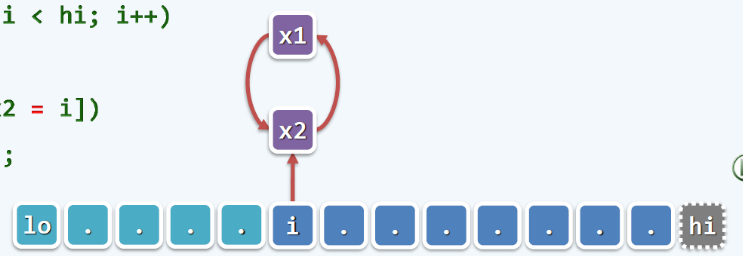
- 迭代1：第一个方法是穷举，写找出最大，在找出次大

```
❖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
```

找出最大，n-1次，找出次大，n-2次 → 无论如何，比较次数总是O(2n-3)

- 迭代2：维护两个指针，x1>x2，然后逐一扫描（没有实质改进！）
  1. 初始化，比较x1=lo, x2 =lo+1
  2. 注意扫描，先于小的数进行比较；若再大，则和大元素比较

```
❖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);
 }
```

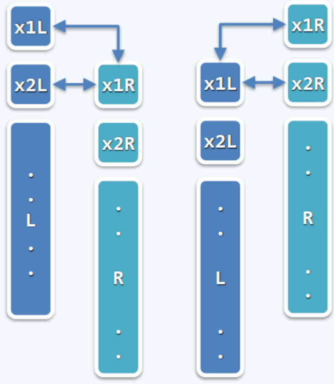


本算法，最好的情况1+n-2=O(n-1)；最差的情况1+2x(n-2)=O(2n-3)

- 递归和分治

分而治之，先进行分序列的比较，得到**子序列的最大和次大，然后最大和次大再相互比较！**

```
❖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
```



- **其他的PPT与课后作业！**
  - 但是递归算法存在空间（时间）效率低的缺陷，因而建议写成等价的迭代形式！如阶乘算法为例：
    - 递归版本

    ```
    __int64 facR ( int n ) { return ( n < 1 ) ? 1 : n * facR ( n - 1 ); } //阶乘运算（递归版）
    ```

    - 迭代版本

    ```
    __int64 facI ( int n ) { __int64 f = 1; while ( n > 1 ) f *= n--; return f; } //阶乘运算（迭代版）
    ```

  - 其他递归方程

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

- 课后作业！

❖ 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")