递归与分治

1. 递归分治的概念与示例

1.1. 递归概述

1 递归算法=调用自身的算法，递归函数=调用自己的函数

2 递归二要素：边界条件+递归方程，必须有着两个要素才能有限次后得到递归结果

3 递归的缺点：

1. 允许效率低，耗时/内存占用远高于非递归算法
2. 改进方法：用递推来实现递归函数，将一些递归转化为尾递归

1.2. 分治概述

1 分治的步骤

1. 将问题分为若干小规模相同子问题(前提是该问题有最优子结构)
2. 子问题规模小到可以轻易解决，各个子问题间相互独立(无公共子问题)
3. 将子问题的解合并为问题的解

2 分治算法的设计

```
divideAndConquer(P)
{
    if (size(P) <= n0)
        return solveDirectly(P); // 直接解决小规模问题

    // 将问题P分解为更小的子问题P1, P2, ..., Pk
    divide P into smaller subinstances P1, P2, ..., Pk;

    // 对每个子问题递归地应用分治法
    for (i = 1; i <= k; i++)
        y[i] = divideAndConquer(Pi);

    // 合并子问题的解以形成原问题的解
    return merge(y[1], ..., y[k]);
}
```

1.3. 递归分治问题举例

1 阶乘函数

边界条件

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

递归方程

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

2 Fibonacci数列

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

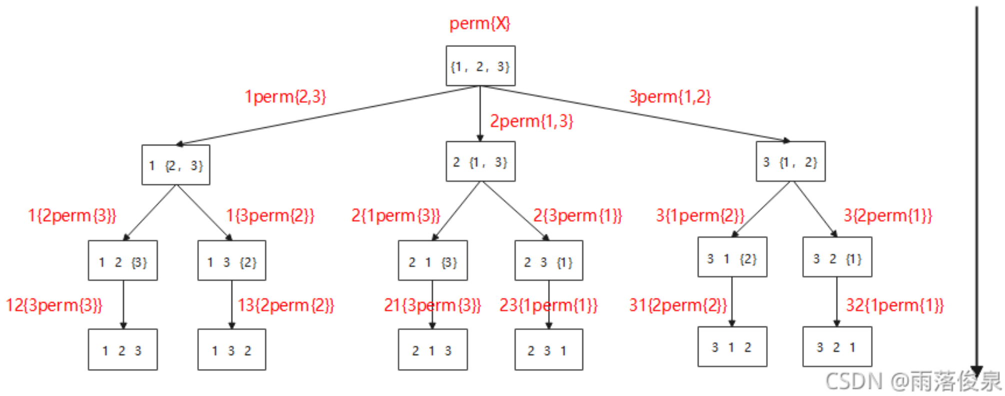
边界条件

递归方程

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

3 排列问题：递归生成元素$R=\{r_1,r_2,...,r_n\}$的全排

1. 符号表示
   - $R_i=R-\{r_i\}$：集合R中，去掉某元素r_i
   - $perm(X)$：集合X全排
   - $(r_i)perm(X)$：全排$perm(X)$前加上r_i前缀得到的排列
2. R的全排可归结为：
   - R只含元素r时：$perm(R)=(r)$
   - R含n个元素时：$perm(R)$由$(r_1)perm(R_1)$，$(r_2)perm(R_2)$，...，$(r_n)perm(R_n)$构成
3. 以$X=\{1,2,3\}$全排为例



4. 代码

```
/*传入：代全排的数组，排列考虑的开始and终止位置，result二维数组(每行代表一种结果)*/
/*其中：resultIndex用于追踪result数组中已经填充的行数*/
void permute(int nums[], int start, int end, int** result, int& resultIndex)
{
```

```
    if (start == end) // 开始结束位置相同时，意味着找到了一种排列，将其复制到结果数组中
    {

        for (int i = 0; i <= end; i++)
        {
            result[resultIndex][i] = nums[i];
        }
        resultIndex++; // 更新结果数组的索引
        return;
    }
    else
    {
        for (int i = start; i <= end; i++)
        {
            // 交换当前开始位置的元素与第i个元素，使前缀不断变化
            std::swap(nums[start], nums[i]);
            // 递归调用，对剩余元素进行排列
            permute(nums, start + 1, end, result, resultIndex);
            // 递归回来时还原到本层的初始模样，以便于下一次k++继续做递归时出发点是一样的
            std::swap(nums[start], nums[i]);
        }
    }
}
```

4 整数划分

1. 问题描述：将正整数n表示为$n = n_1 + n_2 + \ldots + n_k$，其中$n_1 \geq n_2 \geq \ldots \geq n_k \geq 1$，$k \geq 1$

   示例：3的划分——3，2+1，1+1+1

2. 符号：将数字n分解，分解得到的所有加数不大于m，按这样规则的能有$q(n, m)$种划分

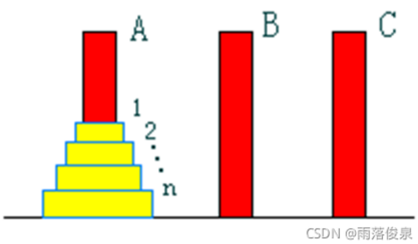
   例如$q(4, 2) = 3$，对应2+2，2+1+1，1+1+1+1

3. $q(n, m)$的性质
   - $q(n, 1) = 1, n \geq 1$，$n = \underbrace{1 + 1 + \ldots + 1}_{n}$
   - $q(n, m) = q(n, n), m \geq n$
   - $q(n, n) = q(n, 1) + q(n, n-1) = 1 + q(n, n-1)$
   - $q(n, m) = q(n, m-1) + q(n-m, m), n > m > 1$

4. 代码表示

```
int q(int n, int m)
{
    // 当所有加数都不大于1时，只有一种划分
    if (m == 1 || n == 1) return 1;
    // 当最大加数m大于等于n时，q(n, m)等于q(n, n)
    if (n <= m) return q(n, n);
    // 当n等于m时，q(n, n)等于1加上q(n, n-1)
    if (n == m) return 1 + q(n, n - 1);
    // 当n大于m且m大于1时，q(n, m)等于q(n, m-1)加上q(n-m, m)
    if (n > m && m > 1) return q(n, m - 1) + q(n - m, m);
    return 0;
}
```

5 Hanoi塔



1. 问题描述：塔A上有一叠共n个圆盘自上而下从小到大堆叠，要把n个圆盘自上而下从小到达移到B
2. 规则
    - 每次只移1圆盘
    - 任何时刻都不允许大盘压小盘
    - 满足前两条规则前提下，可将圆盘移至ABC任一塔上(C为辅助塔)
3. 代码实现

```
void hanoi(int n, int a, int b, int c)
{
    if (n > 0)
    {
        hanoi(n-1, a, c, b);//设法将n-1个较小的圆盘依照移动规则从塔座a移至塔座c
        move(a,b);            //将塔座a上编号为n的圆盘移到b上
        hanoi(n-1, c, b, a);//设法将n-1个较小的圆盘依照移动规则从塔座c移至塔座b
    }
}
```

2. 递归-分治的复杂性分析

2.1. 复杂性

设置一个阈值n_0，n为问题规模

1 $n_0 = n$时，问题可以在常数时间内立马解决

2 $n_0 > n$时，将问题分解为k个规模为n/m的子问题，将子问题合并的时间为$f(n)$

$$T(n)=\begin{cases}O(1), n=n_0\\ kT(n/m)+f(n), n>n_0\end{cases}$$

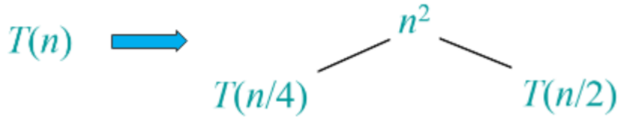
2.2. 递归树分析法(普适)

2.2.1. 方法示例

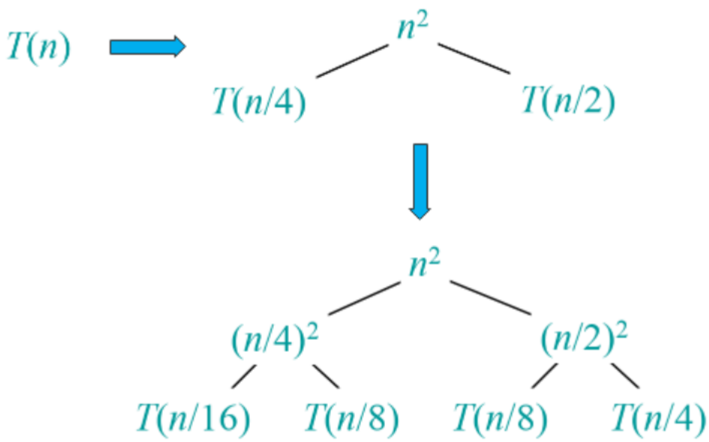
基于$T(n)=T(n/4)+T(n/2)+n^2$分析，即可分解为$T(n/4)$和$T(n/2)$子问题，合并成本为n^2

1 递归树的生成：一个问题分解后，合并成本留在原地，生成子问题变成子节点

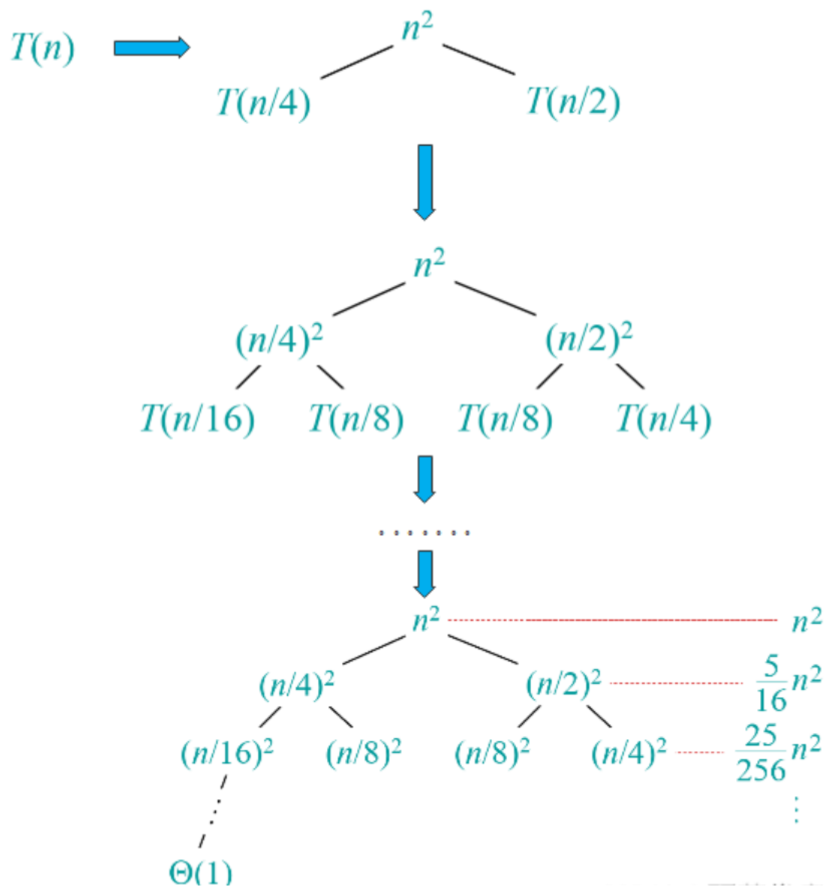
1. $T(n)=T(n/4)+T(n/2)+n^2$



2. $T(n/4)=T(n/16)+T(n/8)+(n/4)^2$，
   $T(n/2)=T(n/8)+T(n/4)+(n/2)^2$



3. 以此类推直到分解到$\Theta(1)$的子问题，然后标出每一层的合并成本总和



2 递归树的高度：

1. 本例种要从$T(n) \rightarrow \Theta(1)$有两种途径：

   $T(n) \rightarrow T(n/2) \rightarrow \ldots \rightarrow \Theta(1)$，$T(n) \rightarrow T(n/4) \rightarrow \ldots \rightarrow \Theta(1)$

   显然前者变换的速度更慢，产生的层数更多，所以$n/2^h = 1$，即$h = log_2 n$

2. 总结就是$T(n) = kT(n/m) + f(n)$中，$h = log_{m_{max}} n$

3 复杂度求解：分为两个部分，要解决所有分解得到的$O(1)$子问题，所有合并的成本

1. 解决所有分解得到的$O(1)$子问题：$O(1)$子问题的数量不可能大于n，故复杂度为$O(n)$

2. 合并的成本：将所有非叶节点相加，如下，可知复杂度为$O(n^2)$

   $t_1 \leq n^2[1 + \frac{5}{16} + \frac{25}{256} + \ldots + (\frac{5}{16})^{\log_2 n - 1}] = \frac{(\frac{5}{16})^{\log_2 n} - 1}{\frac{5}{16} - 1} < \frac{16}{11} n^2$

二者相加后总的复杂度还是$O(n^2)$

2.2.2. 递归树法有关结论

对于$f(n) = af\left(\frac{n}{b}\right) + d(n)$

1 当$d(n)$为常数时：

$$f(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a \neq 1 \\ O(\log n) & \text{if } a = 1 \end{cases}$$

2 当$d(n) = cn$时：

$$f(n) = \begin{cases} O(n) & \text{if } a < b \\ O(n \log n) & \text{if } a = b \\ O(n^{\log_b a}) & \text{if } a > b \end{cases}$$

2.3. 主方法

2.3.1. 主方法所依赖的定理

对于$a \geq 1, b > 1$，$T(n) = aT(\frac{n}{b}) + f(n)$，则$T(n)$有如下渐进界

条件	条件的含义	$\frac{f(n)}{n^{\log_b a}}$	结论
1. $f(n) = O(n^{\log_b a - \epsilon})$	$f(n)$渐进小于$n^{\log_b a}$	$\frac{1}{n^\epsilon}$	$T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b a})$	$f(n)$渐进等于$n^{\log_b a}$	1	$T(n) = \Theta(n^{\log_b a} \lg n)$
3. $f(n) = \Omega(n^{\log_b a + \epsilon})$	$f(n)$渐进大于$n^{\log_b a}$	n^ϵ	$T(n) = \Theta(f(n))$

1 此外情况3还需满足一个条件：$\exists c < 1$和n，$af(n/b) \leq cf(n)$

2 注意，$\epsilon > 0$

3 关键是算出$\frac{f(n)}{n^{\log_b a}}$

2.3.2. 示例

1 $T(n) = 9T(n/3) + n$：算出$\frac{f(n)}{n^{\log_b a}} = \frac{1}{n}$，视为情况1 $\Longrightarrow T(n) = \Theta(n^2)$

2 $T(n) = T(2n/3) + 1$：算出$\frac{f(n)}{n^{\log_b a}} = 1$，视为情况2 $\Longrightarrow \ T(n) = \Theta(\lg n)$

3 $T(n) = 3T(n/4) + n \lg n$:

1. 算出$\frac{f(n)}{n^{\log_b a}} = n^{0.2} \lg n$，忽略$\lg n$(其渐进增长率远小于幂函数)后视为情况3
2. 验证$\exists c < 1$和n，$af(n/b) \leq cf(n)$，即$(3-4c)\lg n \leq 3 \lg 4$，让$c = \frac{3}{4}$即可
3. 最后得到$T(n) = \Theta(n \lg n)$

4 $T(n) = 2T(n/2) + n \lg n$：算出$\frac{f(n)}{n^{\log_b a}} = \lg n$，不属于任何一种情况，只能用递归树

3. 分治算法设计与技巧

3.1. 二分查找：分治策略，$O(\log n)$

```
int BinarySearch(int[] nums,int target)
{
 //找到target时返回其在数组中的位置，否则返回-1
 int left =0,right=nums.length-1;
 while(left<=right)
 {
     int middle= left + (right - left) / 2; // 防止计算时溢出
     if(nums[middle]<target) {left=middle+1;}
     else if(nums[middle]>target) {right=middle-1;}
     else {return middle;}
 }
 return -1;
}
```

3.1.1. 算法思想

将一升序数组一分为二，中间的数$\xleftrightarrow{\text{比较}}$目标

1. 中间的数=目标：找到目标
2. 中间的数>目标：目标在左边，在左边一半继续寻找
3. 中间的数<目标：目标在右边，在右边一半继续寻找

3.1.2. 复杂度分析

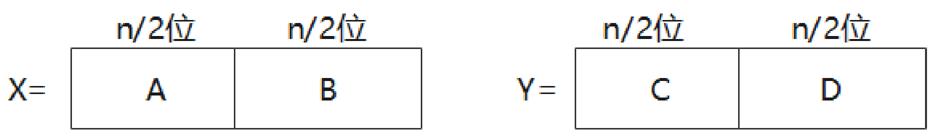
1 每执行一次while，待搜索数组大小就减半

2 最坏情况下，while也被执行了$\log_2 n$次

3 循环体内运算需要$O(1)$，所以复杂度为$O(\log n)$

3.2. 大数乘法：分治

1 大数乘法概念：对于n位二进制数，原本复杂度为$O(n^2)$



$X = A \times 2^{n/2} + B$

$Y = C \times 2^{n/2} + D$

$XY = AC \times 2^n + (AD + BC) \times 2^{n/2} + BD$

修改为：

$XY = AC \times 2^n + ((A - B)(D - C) + AC + BD) \times 2^{n/2} + BD$

2 复杂度分析：

1. 原本需要进行一次n位乘法，故为$T(n)$

2. 现在需要进行三次$n/2$位乘法AC，$(A-B)(D-C)$，BD即$3T(n/2)$
3. 将三次$n/2$位乘法结果处理得到XY，需相加+位移操作，处理成本为$O(n)$
   - 相加的复杂度为$O(1)$
   - 位移因为是移动了n与$n/2$位，所以复杂度为$O(n)$

$T(n)=3T(n/2)+O(n) \implies T(n)=O(n^{\log 3})=O(n^{1.59})$

3.3. Strassen矩阵乘法：分治

1 传统矩阵乘法：$C=AB$

1. C的每个元素是由A的一行乘以B的一列，得到一个C的元素的复杂度是$O(n)$
2. C有n^2个元素，所以总的复杂度为$O(n^3)$

2 什么是Strassen矩阵乘法

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \implies \begin{cases} C_{11} & =A_{11}B_{11}+A_{12}B_{21} \\ C_{12} & =A_{11}B_{12}+A_{12}B_{22} \\ C_{21} & =A_{21}B_{11}+A_{22}B_{21} \\ C_{22} & =A_{21}B_{12}+A_{22}B_{22} \end{cases}$$

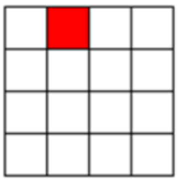
3 复杂度分析：计算一次n阶矩阵乘→计算八次$n/2$阶矩阵乘+计算四次$n/2$阶矩阵加

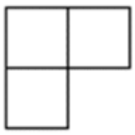
1. 一次n阶矩阵乘：$T(n)$
2. 八次$n/2$阶矩阵乘：$8T(n/2)$
3. 四次$n/2$阶矩阵加：$O(n^2)$

一步Strassen方法可以转化为七次乘法运算，即$T(n)=7T(n/2)+O(n^2)$

3.4. 棋盘覆盖：分治

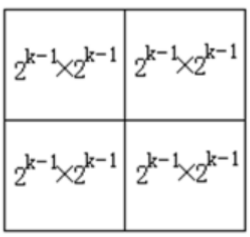
1 问题描述

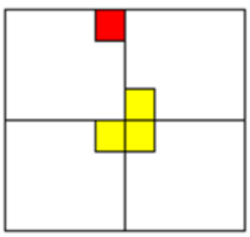




1. 在$2^k \times 2^k$ 方格棋盘中，有一个特殊方格
2. 要用L型骨牌，覆盖除特殊方格外所有格子

2 分治策略





1. 将$2^k \times 2^k$ 盘，分为4个$2^{k-1} \times 2^{k-1}$ 子盘
2. 必定有一个盘含有特殊方格，然后用一个L骨牌将剩下三个盘也改为含特殊方块，这一步为$O(1)$
3. 原问题转化为4个较小规模的棋盘覆盖问题

3 复杂度：$T(k)=4T(k-1)+O(1)$，解得$T(k)=O(4^k)$

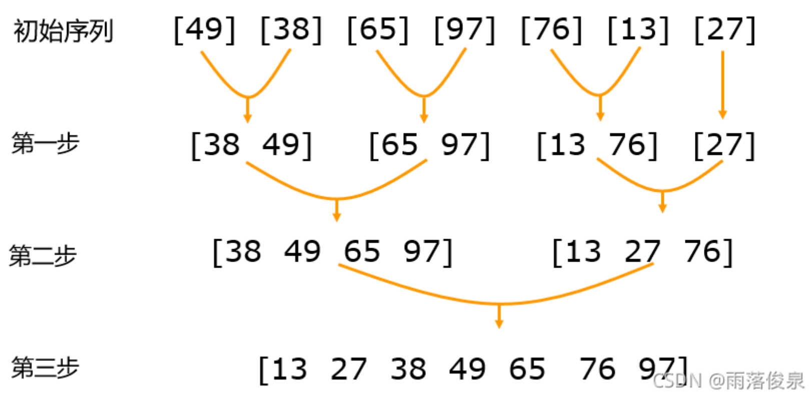
3.5. 快速排序：分治&递归，$O(n\log n)$

```
template<class Type>
 void MergeSort(Type a[], int left, int right)
{
 if (left<right) {//至少有2个元素
     int i=(left+right)/2;  //取中点
     mergeSort(a, left, i);
     mergeSort(a, i+1, right);
     merge(a, b, left, i, right);  //合并到数组b
     copy(a, b, left, right);    //复制回数组a
 }
}
```

1 算法思想：

1. 应用递归：将待排元素分成大小相同的两半→每一半分别排序，最终将排好序的集合合并
2. 复杂度分析：$T(n)=2T(n/2)+O(n)$，解得$T(n)=O(n\log n)$，属于渐进最优

2 用分治消除递归的方法：不断两两相邻配对



3.6. 快速排序

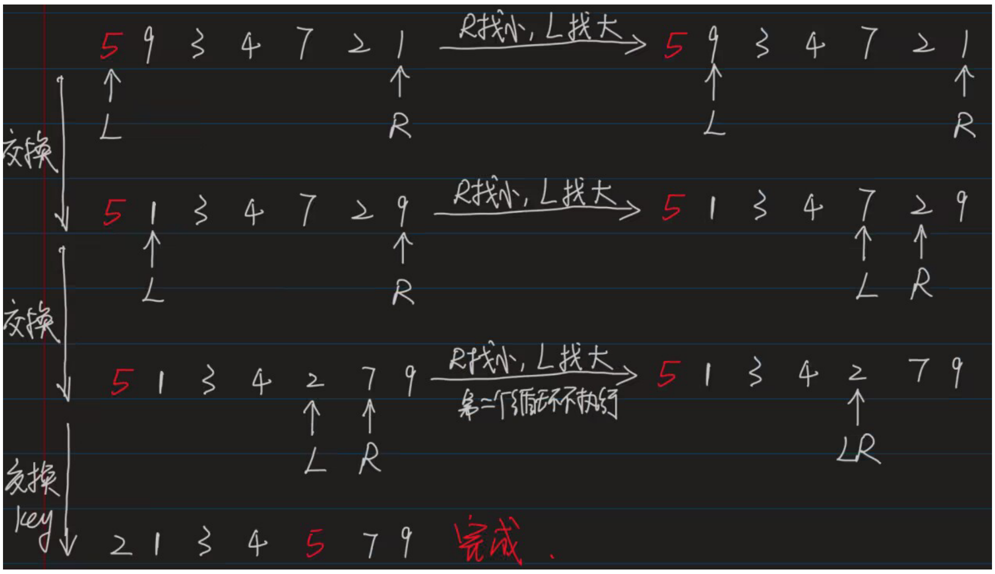
3.6.1. 算法步骤

1 分解：以`a[q]`为基准，以`a[p:q-1]`全部元素<`a[q]`<`a[q+1:r]`全部元素，将`a[q]`分为三份

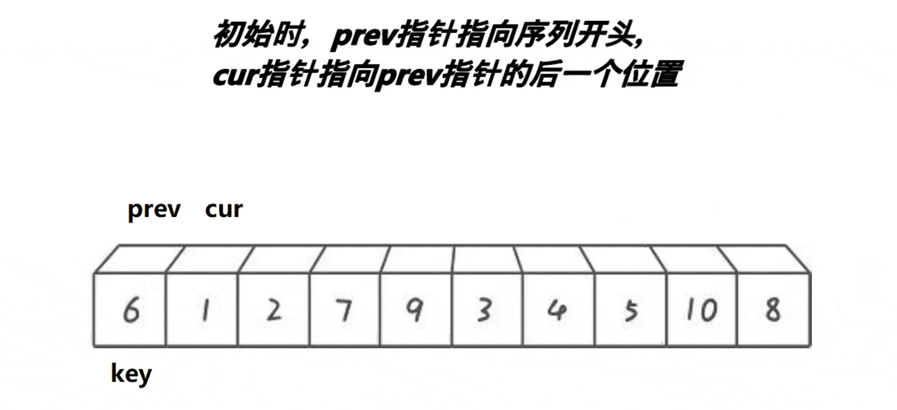
1. 方法1：hoare法

```
int Partion(int* a, int left, int right)
{
    int keyI = left;
    //left == right两个指针相遇，退出循环
    while (left < right)
    {
        //right先找，right找小
        while (left < right && a[right] >= a[keyI]) {right--;}
        //left找大
        while (left < right && a[left] <= a[keyI]) {left++;}
        //都找到了，交换
        Swap(&a[left], &a[right]);
```

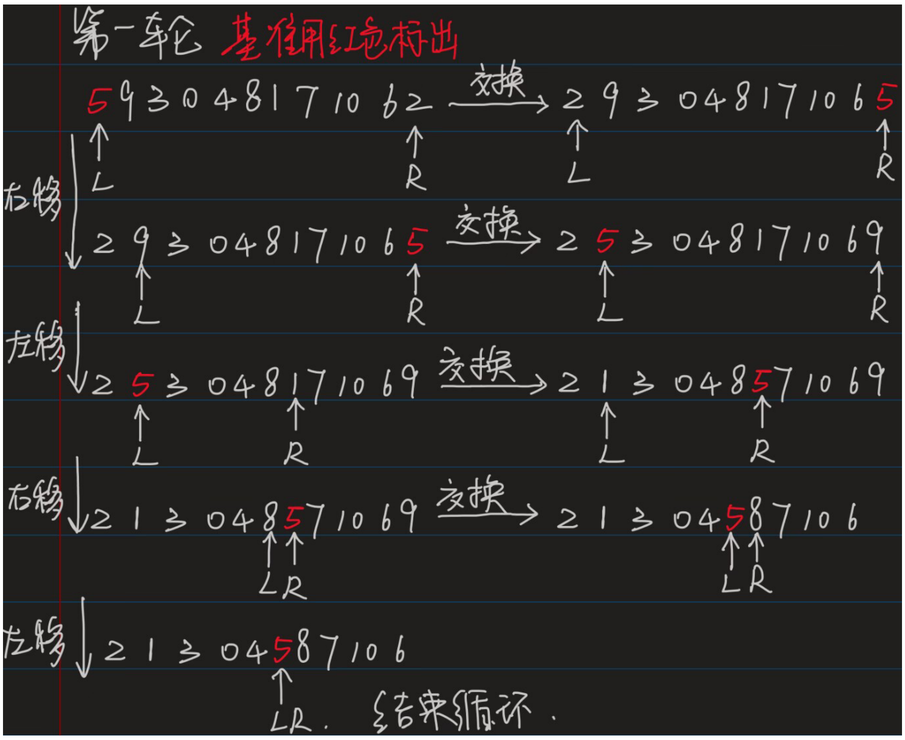
```
    }
    //left和right相遇，交换key和相遇位置元素
    Swap(&a[keyI], &a[left]);
    return left;
}
```



2. 方法2：前后指针法

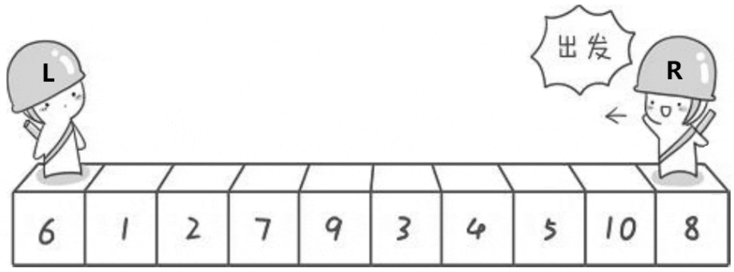


3. 方法3：挖坑法



先将第一个数据存放在临时变量 key 中，形成一个坑位

key =



2 递归排序

```
void quickSort(int[] a, int p, int r) {
    if (p < r) {
        int q = partition(a, p, r);
        quickSort(a, p, q - 1);
        quickSort(a, q + 1, r);
    }
}
```

3.6.2. 算法分析

1 注意事项：取最后一个元素作为基准，且其是数组中最大元素，那么会陷入死循环

2 划分算法的复杂性分析

1. 每次划分都化成(n-1)+1两份：$T(n) = T(n-1) + O(n)$，Partition的计算时间为 $O(n)$
2. 每次划分都砍一半：$T(n) = 2T(n/2) + O(n)$

3.7. 线性时间选择

3.7.1. 引例：寻找n个数中第k小的元素

```
template<class Type>
Type RandomizedSelect(Type a[],int p, int r, int k)
{
    if(p==r)
        return a[p];
 /*数组a[p:r]被划为a[p:i]和a[i+1:r],每个a[p:r]中元素都小于a[i+1:r]中元素*/
    int i = RandomizePartition(a,p,r);
 /*计算子数组a[p:i]中元素个数j*/
    j=i-p+1;
 /*如果k≤j,则a[p:r]中第k小元素落在子数组a[p:i]中: 左半边再来一次*/
    if(k<=j)
     return RandomizedSelect(a,p,i,k);
 /*如果k>j,则要找的第k小元素落在子数组a[i+1:r]中: 全体第k小变成右半边第k-j小*/
 else
     return RandomizedSelect(a,i+1,r,k-j);
}
```

1 复杂度分析：

1. 最坏情况：每次都只排除一个元素，即a[p:i]包含k然后a[i+1:r]只含一个元素，复杂度为$O(n^2)$
2. 平均情况：每次递归，问题规模减少的一半，复杂度为$O(n)$

2 对于最坏情况的解决：**关键在于划分基准**

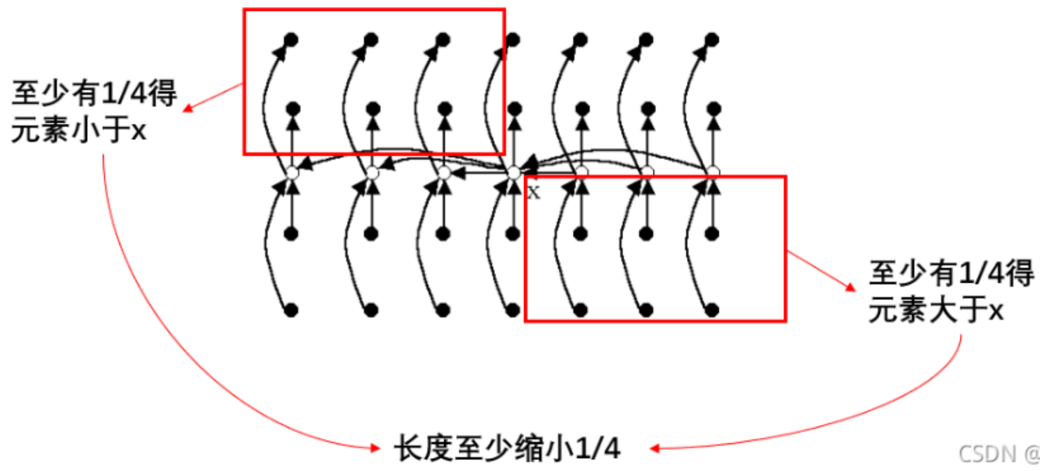
1. 如果能在线性时间内找到一个划分基准
2. 按这个基准所划分出的2个子数组长度，都至少为原数组长度的$\varepsilon \in (0,1)$倍
3. 那**最坏情况下**，用$O(n)$时间完就可以成选择任务

3.7.2. 寻找划分基准示例

1 问题：n个元素五个一组划分为$\frac{n}{5}$组(一组不足5个)，找出每组共$\frac{n}{5}$个中位数，再找出这些中位数的中位数

2 问题分析：

1. 至少有$\dfrac{\frac{n}{5}-1}{2}$个组，这些组中至少3个元素小于x，共计$\dfrac{3(n-5)}{10}$个元素小于基准x，大于x的元素数量同理也是这么多
2. $n \geq 75$时$\dfrac{3(n-5)}{10} \geq \dfrac{n}{4}$，此时按照这种策略砍掉可以确定的大于/小于$x$的元素，数组长度至少缩短1/4
3. **将每一组的大小定为5，并选取75作为是否作递归调用的分界点**，这使得递归式中两个自变量之和$n/5+3n/4=19n/20=\varepsilon n$，$0<\varepsilon<1$



白点代表每组的中位数，黑点代表每组其他数，$A \rightarrow B$代表$A > B$

3 代码

```
template<class Type>
    Type Select(Type a[], int p, int r, int k)
{
    /*当问题规模小于75的时，就直接开始排序*/
    if (r-p<75) {
        Sort(Type a[], int p, int r);
        return a[p+k-1];
    };

    /*当问题规模大于75的时，考虑递归*/
    //将a[p+5*i]至a[p+5*i+4]的第3小元素与a[p+i]交换位置
    for ( int i = 0; i<=(r-p-4)/5; i++ )
    {
        //找中位数的中位数，r-p-4即上面所说的n-5
        Type x = Select(a, p, p+(r-p-4)/5, (r-p-4)/10);
    }
    int i=Partition(a,p,r, x),
    j=i-p+1;
    if (k<=j) return Select(a,p,i,k);
    else return Select(a,i+1,r,k-j);
}
```

3 复杂度分析：以下公式解得$T(n) = O(n)$

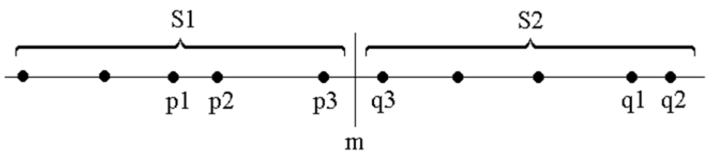
$$T(n) \leq \begin{cases} C_1 & \text{if } n < 75 \\ C_2 n + T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) & \text{if } n \geq 75 \end{cases}$$

3.8. 最接近点对问题

会写伪码，典型的简答题

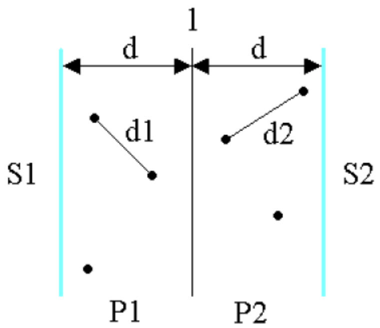
1 问题描述：在平面上n个点中，找到最接近的一对点

2 一维情形分析

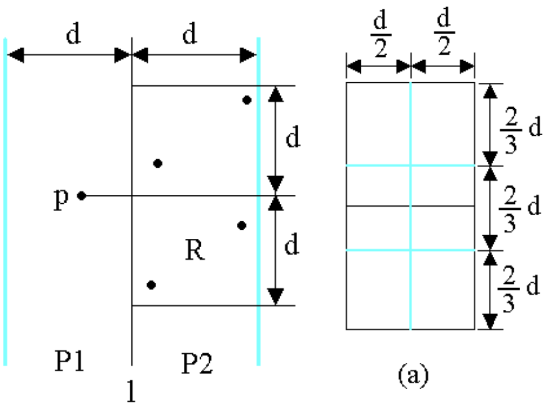


1. 用所有点的中位数m划分，构成两个子问题
2. 递归地在S_1和S_2上找出其最接近点对$\{p_1, p_2\}$和$\{q_1, q_2\}$，$d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$
3. 考虑m边界处的两点，最接近点对只可能是$\{p_1, p_2\}$，$\{q_1, q_2\}$，$\{p_3, q_3\}$
4. 可以在线性时间内找到p_3, q_3，讲解如下
    - $p_3 \in (m-d, m]$，$q_3 \in (m, m+d]$
    - S_1中d长度内，至多只可以包含一个点
    - $(m-d, m]$至多只包含一个点，并且如果包含了，那这个点一定是S_1中最大点
    - $(m, m+d]$至多只包含一个点，并且如果包含了，那这个点一定是S_2中最小点

3 二维情况



1. 用$l: x = m_{\text{各点横坐标中位数}}$分割为$S_1, S_2$，递归地在$S_1, S_2$上找到$d_1, d_2$，设$d = \min\{d_1, d_2\}$
2. 最接近的点要么是d，要么是边界上两边某点的连线$\{p, q\}$
3. 可以在线性时间内找到p, q，讲解如下



- 对任$p \in P_1$，要找到$q \in P_2$使得$\text{distance}(p, q) < d$，则$P_2$一定落在一个$d \times 2d$的矩形$R$中
- 矩形R中最多只有6个S中的点，由此最多需要检查$6 \times n/2 = 3n$个候选者

4 伪代码

```
function cpair2(S)
{
    n = |S|; // 集合S的大小
```

```
    if (n < 2) return ∞; // 如果点的数量小于2，则没有最近点对，返回无穷大

    /*步骤1：找到x坐标的中位数*/
    m = S中所有点x坐标的中位数;
    构造S1和S2: S1={p∈S|x(p)≤m}, S2={p∈S|x(p)≤m}

    /*步骤2：递归寻找S1和S2中最近的点对*/
    d1 = cpair2(S1);
    d2 = cpair2(S2);

    /*步骤3：从S1和S2找到的点对中选取最小距离*/
    dm = min(d1, d2);

    /*步骤4：创建P1和P2集合，包含在距离分割线l dm以内的所有点*/
    P1 = S1中距分割线l,距离在dm内的所有点;
    P2 = S2中距分割线l,距离在dm内的所有点;
    按照y坐标对P1和P2中的点进行排序;
    设X和Y为对应的已排序点列;

    /*步骤5：通过扫描X，检查Y中与X中每个点距离dm内的所有点(最多6个)来合并*/
    dl = 按照这种扫描方式找到的点对间的最小距离;
    //当X中的扫描指针逐次向上移动时，Y中的扫描指针在宽度为2dm的区间内移动;

    // 步骤6：确定最小距离
    d = min(dm, dl);
    return d;
}

```

5 复杂度分析：设对于n个点的平面点集S，算法耗时$T(n)$

$$T(n)=\begin{cases}O(1) & \text{if } n<4\\ 2T\left(\frac{n}{2}\right)+O(n) & \text{if } n\geq 4\end{cases}$$

第1步	第2步	第3步	第4步	第5步	第6步
$O(n)$	$2T(n/2)$	$O(1)$	最坏$O(n\log n)$，采用预排序后为$O(n)$	$O(n)$	$O(1)$

解得：$T(n)=O(n\log n)$