算法第一讲——分治

乔扬

2015-11-21

本章主要内容

1. 第一个例子：归并排序

- 用循环不变量的方法证明程序的正确性
- 递归算法的时间复杂度

2. 其他例子：逆序对计数，最近点对，乘法和快速傅里叶变换。

3. 分治算法和随机算法结合在一起有很好的应用：快速排序算法，选择数算法。 分治算法的优势：有些问题使用暴力破解算法已经是多项式时间复杂度，但使用分治算法使得复杂度还能降低。比如最近点对的算法复杂度。

1 排序问题

1.1 算法策略

如果一个问题可以分成更小的子问题，我们有以下2种策略：

(1) 增量式：一个数排序，增加一个数，2个数排序，直到n个数排序。

(2) 分治算法：大刀阔斧的分，n个问题分两半，n/2的问题不会，则再分成n/4，直到能解决为止。

1.1.1 增量式策略

基本思想：假设我们有一个部分解，例如 A[0..j-1]已经被正确排序，将A[j]加入进来并放入正确的位置，那么A[0..j]就被排好序了。

举例分析：插入排序。具体代码如下：

InsertionSort(A)

```
1: for j = 0 to n − 1 do
2:     key = A[j];
3:     i = j − 1;
4:     while i ≥ 0 and A[i] > key do
5:         A[i + 1] = A[i];
6:         i − −;
7:     end while
8:     A[i + 1] = key;
```

9: **end for**

- 最差的情况：数组是倒序排列的，那么每一个数加进来后，由于比前面所有的数都小，前面的数都要向后挪一个位置，因此$T(n) = 1 + 2 + 3 + ... + n$
- 插入排序的时间复杂度分析: $O(n^2)$.
- 迭代表达式表示为：$T(n) = T(n-1) + cn = O(n^2)$.

1.1.2 分治策略

归并排序：第一次是被冯诺依曼在1940s提出的。
重要观察：一个问题能被划分成2个独立的子问题。具体步骤如下：

- Divide：将n个元素的序列分成2个分别有n/2个元素的序列
- Conquer：假设每个子问题能被解决，实现方法是使用递归调用。
- Merge：将2个排序好的子序列合并成一个序列，得到原始问题的解。

算法伪代码如下：

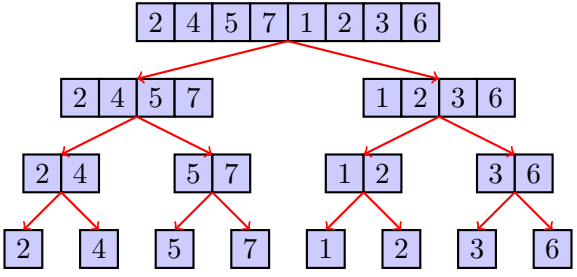
MERGESORT(A, l, r)
1: /* To sort part of the array $A[l..r]$. */
2: **if** $l < r$ **then**

3: $m = (l + r)/2$; // m denotes the middle point;
4: MergeSort(A, l, m);
5: MergeSort(A,m, r);
6: Merge(A, l, m, r); // combining the sorted subsequences;
7: **end if**

第4步和第5步是将数组A[l..r]分成A[l..m]和A[m..r]，并对两个子问题递归调用求解。第6步将2个部分解合并成最终解。

举例分析归并排序的具体做法

(一)怎么分?



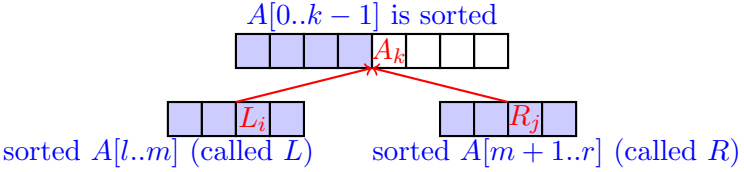
对于不会解的问题，将其分解到能被解决为止，在这里2个数的比较是算法的基础解，能被解决。

(二)怎么合并? 合并算法如下所示:

Merge (A, l, m, r)

1: /* to merge $A[l..m]$(named as L) and $A[m + 1..r]$ (named as R). */

```
2: i = 0; j = 0;
3: for k = l to r do
4:    if L[i] < R[j] then
5:       A[k] = L[i];
6:       i++;
7:    else
8:       A[k] = R[j];
9:       j++;
10:   end if
11: end for
```



左边一半存入数组L和右边一半存入数组R都是已经排好的子序列，如何将2者合并成一个序列呢?

每次取两个数组最小的元素，取2者最小的填入数组A中，然后将数组下标右移一位。

a) 证明merge策略的正确性：循环不变量技术

- 循环不变量：类似于数学归纳法技术。
- 初始情况：$k = l$.因为A[l..k-1]为空，所以循环不变量成立。

- 特征保持：假设$L[i] < R[j]$，而且A[l..k-1]有k-l个最小的数，那么把L[i]放到A[k]，A[l..k]就是拥有k-l+1个最小的数

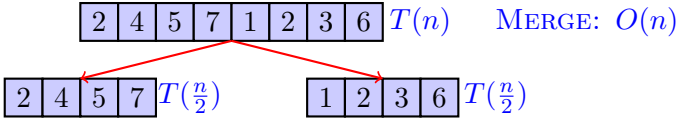
b) Merge算法的时间复杂度

For循环最多执行n次，故时间复杂度是$O(n)$。

(三) 归并排序的时间复杂度

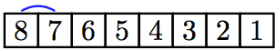
假设T(n)是整个问题的时间复杂度，将其分成2半，左边需要$T(n/2)$，右边需要$T(n/2)$，那么可以写成如下:

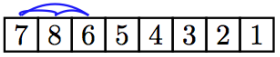
$$T(n) = \begin{cases} c & n = 2 \\ T(n/2) + T(n/2) + cn & otherwise \end{cases} \tag{1}$$

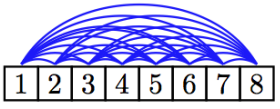


从插入排序到归并排序，复杂度从$O(n^2)$到$O(n \log n)$，那么节省的究竟是什么呢?

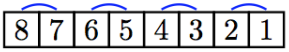
下图表示的箭头数量是2种算法排序过程中需要进行的比较次数。



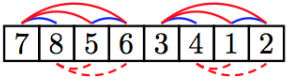




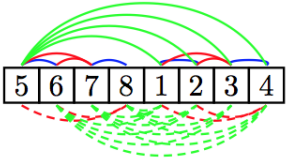
INSERTSORT: 28 ops



MERGESORT step 1: 4 ops



MERGESORT step 2: 4 ops, save: 4 ops



MERGESORT step 3: 4 ops, save: 12 ops

我们可以看出，插入排序需要28次比较。而归并排序中第2步省去了4次比较，第3步省去了12次比较。例如：第3步中，5和7比较之后，不需要再和8进行比较。

1.1.3 分治算法的复杂度

归并排序的时间复杂度：如何精确计算分治算法的时间复杂度?

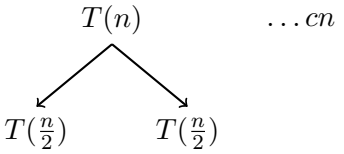
我们已经得到算法复杂度的递归表达式，如何将递归表达式写成精确地结果？主要有以下3种方法。

- 将表达式展开，展开一定程度之后就会观察到结果的模型。
- 猜测并证明猜测的正确性。
- 产生函数

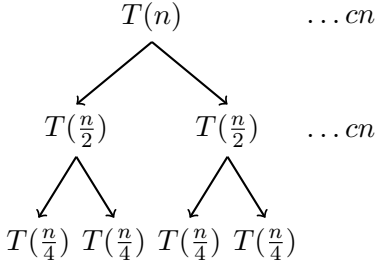
本书只讲前2种方法。

（1）复杂度分析方法－－展开

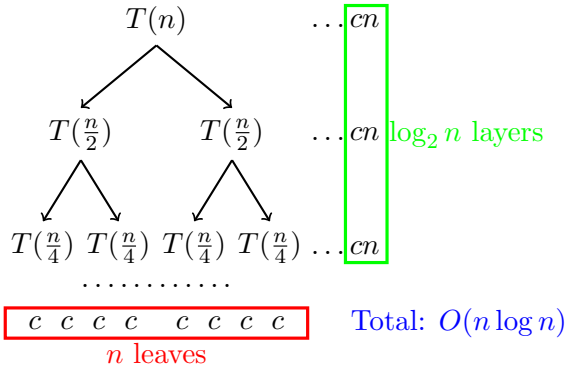
- 将$T(n)$分成$T(n/2)$的表达式，如果还看不出结果的话继续展开。



- 如果还不能解决，将$T(n/2)$的问题展开成 $T(n/4)$的问题，直到能被解决为止。



- 直到展开的子问题的时间是常数。那么我们只需要计算最后一层常数时间的子问题的个数。



那么时间复杂度就由2部分组成：合并的时间和最后一层比较要花的时间。

那么究竟有多少层呢，很显然要层数是$O(\log_2 n)$。那么合并花的时间就是$cn * \log_2 n$。 可以看出这棵树总共有n个叶子节点，花费时间是cn。

因此时间复杂度是$O(n \log n)$

(二)复杂度分析方法2－－先猜测后证明

- 猜测一个解决方案并替换掉设定的系数，证明公式的正确性。
- 猜测: $T(n) \leq cn \log_2 n$ for all $n \geq 2$;
- 证明:
  - 当 $n = 2$: $T(2) = c \leq cn \log_2 n$;

– 当 $n > 2$: 假设对所有 $m \leq n$都有$T(m) \leq cm \log_2 m$ 。那么

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn && (2)\\
&\leq 2c(n/2)\log_2(n/2) + cn && (3)\\
&= 2c(n/2)\log_2 n - 2c(n/2) + cn && (4)\\
&= cn \log_2 n && (5)
\end{aligned}
$$

但事实上，我们经常猜的没有那么准，我们可能只猜到 $T(n) = O(n \log n)$，或者写成$T(n) = k \log_b n$。K和b我们不清楚，再去求解。

所以我们可能会猜测一个比较模糊的表达式，如下：

- 猜测和替换，我们只能猜测k和b可以等于多少，然后去替换掉。
- 模糊猜测: $T(n) = O(n \log n)$。时间复杂度这样描述 $T(n) = k \log_b n$, k, b 是待定系数

$$
\begin{aligned}
T(n) &= 2T(n/2) + cn \\
&\leq 2k(n/2)\log_b(n/2) + cn \quad \text{(set b=2 for simplification)} \\
&= 2k(n/2)\log_2 n - 2k(n/2) + cn \\
&= kn\log_2 n - kn + cn \quad \text{(set k=c for simplification again)} \\
&= cn\log_2 n
\end{aligned}
$$

后来，大家把递归表达式总结成一个定理。定理描述如下:
假设 $T(n)$ 定义为$T(n) = aT(n/b) + f(n)$, 那么 $T(n)$ 会有如下性质:

1. If $f(n) = O(n^{\log_b a-\epsilon})$, then $T(n) = \Theta(n^{\log_b a})$;
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a}) \log n$;
3. If $f(n) = \Omega(n^{\log_b a+\epsilon})$ and $af(n/b) \leq cf(n)$, then $T(n) = \Theta(f(n))$. Here, ϵ denotes a small, positive number.

举例分析如下

- Example 1: $T(n) \leq 3T(n/2) + cn$ (see a figure)
  $T(n) = O(n^{\log_2 3}) = O(n^{1.585})$

- Example 2: $T(n) \leq 2T(\frac{n}{2}) + cn^2$ (see a figure)

  $T(n) = \sum_{j=0}^{\log n} \frac{cn^2}{2^j} = cn^2 \sum_{j=0}^{\log n} \frac{1}{2^j} = 2cn^2$
  (Note: not $O(n^2 \log n)$)

- Example 3: $T(n) \leq T(n/3) + T(2n/3) + cn$ (see a figure)

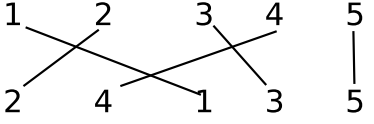
2 逆序数计数问题

2.1 实际应用

1. 识别2个人的相似度，比如比较2人对书本，电影等的排序。
2. meta search engine, 它把搜索请求提交给其他搜索引擎，比较各个搜索引擎的排序结果，计算它们的相似度。

逆序数问题的形式化表示

- 输入：一组n个不同的数的序列
- 输出：逆序对数，如果$i < j$，而$a_i > a_j$，那么就是逆序数对。



如上图表示，将正确的排好的序列1,2,3,4,5和序列2,4,1,3,5相同的数连起来，那么连线交叉的数量就是逆序数的对数。比如1和2,1和4,3和4总共3对。

2.1.1 逆序数的具体应用

- 基因序列的比较

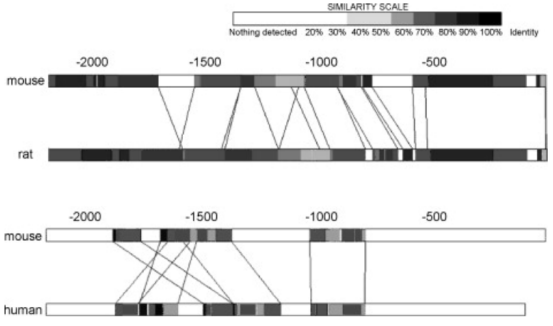


Figure 1: Sequence comparison of the 5' flanking regions of mouse, rat and human ERβ.

在生物信息领域，比较2个基因组是否一样，基因组有很多基因的排序，假设有n个基因，上图是mouse和rat之间基因序列的比较，相同的基因连成一条线，交叉的地方就是逆序。老鼠和人的基因的相似情况同样如此。

- 求2个数列的相似度

常用方法：1.Pearson系数。2.Spearman相关系数。 新的策略：

$$W_1 = \sum_{i=1}^{n-k+1}(I_i^+, I_i^-)$$

表达式的含义是求相同长度子序列的相似度。举例分析

$$X: \ 1\ 3\ 4\ 2\ 5$$
$$Y: \ 1\ 4\ 5\ 2\ 3$$

$W_1 = 2$ when $k = 3$.当设定比对的序列长度（滑动窗口）是3时，（1，3，4）和（1，4，5）是没有逆序的，（3，4，2）和（4，5，2）是没有逆序的，（4，2，5）和（5，2，3）是有逆序的。那么相似度就是2。这种算法的结果比Pearson系数要好。

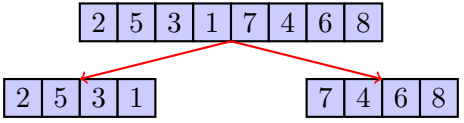
求逆序数的算法：暴力方法：枚举所有的点对，然后得到逆序数的结果，时间复杂度是$O(n^2)$。那么有改善的算法吗?

2.2 分治策略

重要观察：逆序数问题能够被划分成几个子问题。

分治策略:

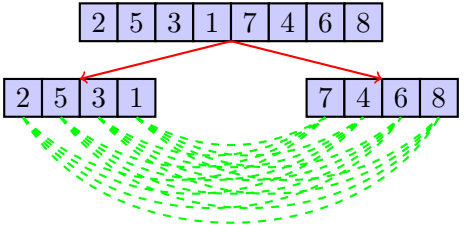
1. Divide: 分成2个序列: $A[0..n/2]$ 和 $A[n/2+1...n-1]$;
2. Conquer: 递归调用计算各一半的逆序数;
3. Combine：怎么比较分别位于左边和右边的数呢?



（一）合并策略1

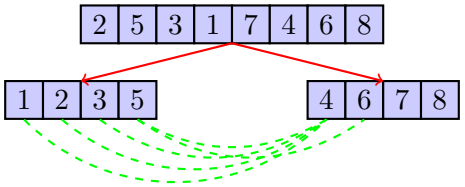
简单的枚举算法的话，需要$\frac{n^2}{4}$ 次比较。因此$T(n) = 2T(\frac{n}{2}) + \frac{n^2}{4} = O(n^2)$.

对于通用的分治算法的递归表达式$T(n) = aT(\frac{n}{b}) + f(n)$，由于这里的合并时间消耗$f(n) = \frac{n^2}{4}$，所以时间复杂度太大。



（二）合并策略2：假设左一半和右一半已经排好序了，那么会大大减少比较的次数。

如果合并策略花费 $O(n)$ 时间，那么整个分治算法的时间花费 $O(n \log n)$ 。



如图：1和4比较，$1 < 4$，那么1不需要在和4后面的数比较。

实际上我们只需要在归并排序的基础上就行修改就可得到逆序数的解决方法。其算法伪代码如下：

SORT-AND-COUNT(A)

Divide A into two sub-sequences L and R;
(RC_L, L) = SORT-AND-COUNT(L);
(RC_R, R) = SORT-AND-COUNT(R);
(C, A) = MERGE-AND-COUNT(L, R);
return $(RC = RC_L + RC_R + C, A)$;

MERGE-AND-COUNT (L, R)

$RC = 0$; $i = 0$; $j = 0$;
for $k = 0$ to $\|L\| + \|R\| - 1$ **do**
  if $L[i] > R[j]$ **then**
    $A[k] = R[j]$;
    $j++$;
    $RC += (\frac{n}{2} - i)$;

```
    else
        A[k] = L[i];
        i++;
    end if
end for
return (RC, A);
```

Time complexity: $T(n) = O(n \log n)$.

在合并的函数中，比较左边的数L[i]和右边子数组的R[j]，如果$L[i] > R[j]$，那么L[i]后面的数比R[j]都要大，因此逆序数个数增加了$\frac{n}{2} - i$。

逆序数问题的讨论：

排序的过程实际上就是减少逆序数的过程，假设我们记录了排序过程中减少的逆序数的数量，那么就得到了所有的逆序数的数量。

3 快速排序

3.1 快速排序的思想

快速排序和上次课讲的归并排序的思路比较像，但是分的办法不一样。

快速排序的分法：随便选择一个元素$A[j]$，所有的其他元素和这个元素比较一次，那么就把比它小的放到 S_-，比它大的元素放到S_+。然后就是对S_-排序，对S_+排序。接着就输出S_-，$A[j]$，S_+。

快速排序比归并排序的优势之一：代码比较简单。 虽然代码写起来容易，但是分析就比较难了。理想情况下，是一次分2半，但是因为选择的随机性，我们不知道将数组分成的比例是多少。

- **Worst-case:** 选择了最大或者最小的元素作枢纽元； 我们每次选择的都是数组中最大或者最小的数，分类操作需要一次循环，因此合并操作需要$O(n)$的时间。那么复杂度就是$O(n^2)$。

$$T(n) \leq T(n-1) + cn \Rightarrow T(n) = O(n^2)$$

- **Best-case:** 选择了中间元素作枢纽元;那么复杂度是$O(n \log n)$。

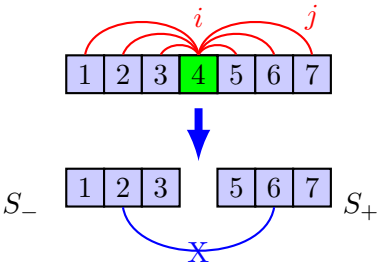
$$T(n) \leq 2T(n/2) + cn \Rightarrow T(n) = O(n \log n)$$

- **Most cases:** 事实上，我们选择的情况不是最好也不是最差，只会是一般的情况。我们接下来证明期望值是$T(n) = O(n \log n)$。

3.2 快速排序复杂度$O(n \log n)$的证明

我们在计数过程中只关心两两比较的次数，用X表示对大小为N的数组排序比较的次数。

1. 第一个观察：任意i和j 最多比较一次，假如选择了元素4，4和所有元素比了一次，那么4以外的元素都被划分到了2个子数组，4不会再和其他元素比较。下面计算快速排序算法比较次数的期望值。



- 定义 $X_{ij} = I\{A[i] \text{ is compared with } A[j]\}$.如果$A[i]$ 和$A[j]$比较了，$X_{ij} = 1$，否则$X_{ij} = 0$。

- 因此 $X = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}$.

$$\begin{aligned} E[X] &= E[\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} X_{ij}] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} E[X_{ij}] \\ &= \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} Pr\{A[i] \text{ is compared with } A[j]\} \end{aligned}$$

2. 第二个观察：当我们在处理$A[i, i+1, ..., j]$数组的时候，A[i] 和 A[j]比较发生的条件是：$A[i]$或者 $A[j]$其中一个元素被选择作为枢纽元（划分数组的元素）。如果都没有被选中过，那么不会被比较。

   例如：对于1，2，3，4，5这个数组，1和5比较的条件是1或者5被选择，假如选择2，3，4，那么1和5会被分到2个不同的子数组，就不会被比较。而且1和5比较的概率此时是2/5。当i,j不是数组的边界时，这个概率小于$\frac{2}{j-i+1}$，因此$Pr\{A[i] \text{ is compared with } A[j]\} \leq \frac{2}{j-i+1}$。

那么我们重新计算前面的比较次数期望。

$$\begin{aligned} E[X] &= \sum_{i=1}^{n}\sum_{j=i+1}^{n} Pr\{A[i] \text{ is compared with } A[j]\} \\ &\leq \sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{2}{j-i+1} \\ &= \sum_{i=1}^{n}\sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n}\sum_{k=1}^{n} \frac{2}{k+1} \\ &= O(n\log n) \end{aligned}$$

3.3 改造快速排序

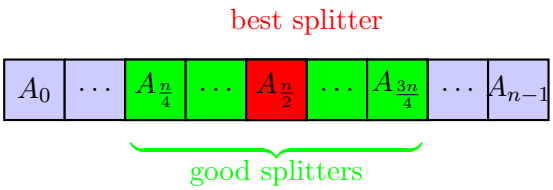
我们把算法改造一下，让我们的分析更加简单。

ModifiedQuickSort(A)

1: **while** TRUE **do**
2: randomly choose a splitter $A[j]$;
3: **for** $i = 0$ to $n-1$ **do**
4: Put $A[i]$ in S_- if $A[i] < A[j]$;
5: Put $A[i]$ in S_+ if $A[i] > A[j]$;
6: **end for**
7: **if** $\|S_+\| > \frac{n}{4}$ and $\|S_-\| > \frac{n}{4}$ **then**
8: break;

9: **end if**
10: **end while**
11: ModifiedQuickSort(S_+);
12: ModifiedQuickSort(S_-);
13: Output S_-, then $A[j]$, and finally S_+;

我们在前面加了一个死循环，随机选择一个元素，并将整个数组分成2部分：S_-和S_+。如果2个子数组的大小都比n/4要大，那么我们决定选择这个元素作枢纽；否则我们重新选择一个元素直到满足条件。



1. 选中中间元素$A[n/2]$ 的概率是$\frac{1}{n}$
2. 选中中间部分（绿色部分）的概率是$\frac{1}{2}$。2项分布的概率是$\frac{1}{2}$，那么期望是2.也就是循环2次就可以找到中间部分的枢纽元。那么合并的时间就是2n。$T(n) = T(n/4) + T(3n/4) + 2n$的复杂度是$O(n \log n)$。
3. 总结:
   - 迭代深度是$O(\log_{\frac{4}{3}} n)$.

- 每一次迭代找枢纽元的时间是$O(n)$.
- $T(n) = O(n \log_{\frac{4}{3}} n)$.

4 乘法问题

一般做法如下图所示:

$$\begin{array}{cccc} & & 1 & 2 \\ \times & & 3 & 4 \\ \hline & & 4 & 8 \\ 3 & 6 & & \\ \hline 4 & 0 & 8 & \end{array}$$

提出问题：$O(n^2)$的复杂度是最优的吗?

4.1 解决方法1

- 重要观察：2个整数x,y可以表示成n位的二进制数，可以分成2部分。
- 分治策略:

1. **Divide:** $x = x_h \times 2^{\frac{n}{2}} + x_l$, $y = y_h \times 2^{\frac{n}{2}} + y_l$,
2. **Conquer:** calculate $x_h y_h$, $x_h y_l$, $x_l y_h$, and $x_l y_l$;
3. **Combine:**

$$\begin{aligned} xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (6) \\ &= x_h y_h 2^n + (x_h y_l + x_l y_h) 2^{\frac{n}{2}} + x_l y_l && (7) \end{aligned}$$

举例分析：计算 12×34

- Objective: to calculate 12×34
- $x = 12 = 1 \times 10 + 2$, $y = 34 = 3 \times 10 + 4$
- $x \times y = (1 \times 3) \times 10^2 + ((1 \times 4) + (2 \times 3)) \times 10 + 2 \times 4$

这个算法的复杂度：我们将其分成4个子问题，每个子问题的规模是$\frac{n}{2}$，3次加法。$T(n) = 4T(n/2) + cn \Rightarrow T(n) = O(n^2)$。如下表所示。

$\times$	y_h	y_l
x_h	$x_h y_h$	$x_h y_l$
x_l	$x_l y_h$	$x_l y_l$

我们可以发现以下问题:

- 我们的目标是计算 $x_h y_h 2^n + (x_h y_l + x_l y_h)2^{\frac{n}{2}} + x_l y_l$。
- 我们发现没必要分别计算 $x_h y_l$ 和 $x_l y_h$，只需要计算 $(x_h y_l + x_l y_h)$。
- 很明显 $(x_h y_l + x_l y_h) + (x_h y_h + x_l y_l) = (x_h + x_l) \times (y_h + y_l)$。 所以$(x_h y_l + x_l y_h)$ 只需要一次额外的乘法就能计算出来。

4.2 新的分治算法

- **Divide:** $x = x_h \times 2^{\frac{n}{2}} + x_l$, $y = y_h \times 2^{\frac{n}{2}} + y_l$,
- **Conquer:** 计算 $x_h y_h$, $x_l y_l$, 和 $P = (x_h + x_l)(y_h + y_l)$;
- **Combine:**

$$
\begin{aligned}
xy &= (x_h \times 2^{\frac{n}{2}} + x_l)(y_h \times 2^{\frac{n}{2}} + y_l) && (8)\\
&= x_h y_h 2^n + (x_h y_l + x_l y_h)2^{\frac{n}{2}} + x_l y_l && (9)\\
&= x_h y_h 2^n + (P - x_h y_h - x_l y_l)2^{\frac{n}{2}} + x_l y_l && (10)
\end{aligned}
$$

还是刚才那个例子：计算12 $\times$ 34

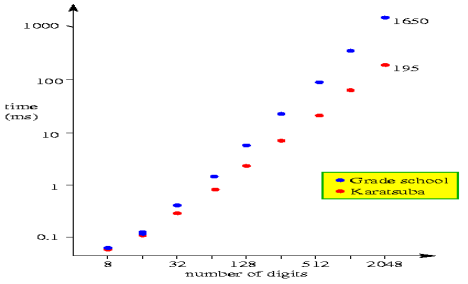
1. Objective: to calculate 12×34
2. $x = 12 = 1 \times 10 + 2$, $y = 34 = 3 \times 10 + 4$
3. $P = (1 + 2) \times (3 + 4)$
4. $x \times y = (1 \times 3) \times 102 + (P - 1 \times 3 - 2 \times 4) \times 10 + 2 \times 4$

这个算法可以分成3个子问题，6次加法，2次移位。 那么算法的复杂度变成$T(n) = 3T(n/2) + cn \Rightarrow T(n) = O(n^{\log_2 3}) = O(n^{1.585})$。

4.3 算法性能的比较

那么这个算法到底怎么样呢?

对于n很大时，Karatsuba的算法效果是很明显的。但是当n的规模比较小时，额外的移位和加法操作将使这个算法变慢。 当使用快速傅立叶变换的技术，乘法时间复杂度是$O(n \log n)$ 。



4.4 扩展：快速除法

1. 问题: 给定2个n位的数 s 和 t, 计算$q = s/t$ 和$r = smodt$。
2. 方法:
   - 先用牛顿法 $x_{i+1} = 2x_i - t \times x_i^2$ 计算 $x = 1/t$ 。
   - 最多需要 $\log n$ 次迭代。
   - 除法和乘法速度一样快。
3. 牛顿迭代法的细节
   - $f(x) = (t - \frac{1}{x})$，x 是 $f(x) = 0$ 的解。

- 牛顿迭代法:

$$
\begin{aligned}
x_{i+1} &= x_i - \frac{f(x_i)}{f'(x_i)} && (11)\\
&= x_i - \frac{t - \frac{1}{x_i}}{\frac{1}{x_i^2}} && (12)\\
&= -t \times x_i^2 + 2x_i && (13)
\end{aligned}
$$

- 牛顿迭代法的迭代速度：迭代次数只需要 $\log \log t = O(\log n)$。

4. 举例分析：计算 $\frac{1}{13}$

#Iteration	x_i	ϵ_i
0	0.018700	-0.058223
1	0.032854	-0.044069
2	0.051676	-0.025247
3	0.068636	-0.008286
4	0.076030	-0.000892
5	0.076912	-1.03583e-05
6	0.076923	-1.39483e-09
7	0.076923	-2.77556e-17
8	...	...

5 矩阵乘法

问题: 给定2个 $n \times n$矩阵 A 和B, 计算 $C = AB$。

5.1 解决方法1

一般的方法： A的第i行 和B的第j列相乘得到C_{ij}，复杂度是$O(n^3)$。因为根据定义矩阵C中每个元素都需要$O(n)$次乘法，总共有n个元素。

关键观察：将矩阵分成4块，每一块是 $\frac{n}{2} \times \frac{n}{2}$ 的大小。

分治算法:

1. **Divide:** 把 A, B, 和 C分别划分成4个小矩阵 ;
2. **Conquer:** 计算子矩阵的乘积;
3. **Combine:**

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C_{11} = (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \tag{14}$$
$$C_{12} = (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \tag{15}$$
$$C_{21} = (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \tag{16}$$
$$C_{22} = (A_{21} \times B_{12}) + (A_{22} \times B_{22}) \tag{17}$$

算法的复杂度：每个问题分成8个子问题和4次加法。每次加法花费$O(n^2)$时间。 所以时间复杂度是：$T(n) = 8T(n/2) + cn^2 \Rightarrow T(n) = O(n^3)$

问题：能降低矩阵乘法的复杂度吗?

5.2 Strassen 算法

Strassen 算法：第一次提出的算法比O(n3)要快。

5.2.1 算法的主要思想

减少求解子问题的数目：8个子问题的结果并不一定都需要，只需要算出7个矩阵的结果就能表示出来。

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$P_1 = A_{11} \times (B_{12} - B_{22}) \tag{18}$$

$$P_2 = (A_{11} + A_{12}) \times B_{22} \tag{19}$$

$$P_3 = (A_{21} + A_{22}) \times B_{11} \tag{20}$$

$$P_4 = A_{22} \times (B_{21} - B_{11}) \tag{21}$$

$$P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \tag{22}$$

$$P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \tag{23}$$

$$P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \tag{24}$$

$$C_{11} = P_4 + P_5 + P_6 - P_2 \tag{25}$$
$$C_{12} = P_1 + P_2 \tag{26}$$
$$C_{21} = P_3 + P_4 \tag{27}$$
$$C_{22} = P_1 + P_5 - P_3 - P_7 \tag{28}$$

5.2.2 Strassen算法的复杂度

那么Strassen算法的复杂度是$T(n) = 7T(n/2) + cn^2 \Rightarrow T(n) = O(n^{\log_2 7}) = O(n^{2.807})$。

5.2.3 Strassen算法的优劣

- **优势:**
  1.比一般的算法速度快很多。

  2.Strassen算法可以用来解决其他问题：矩阵求逆，行列式求值，图的三角形个数计数。

- **劣势:**
  1.只对于规模较大的问题复杂度有所提升。

  2.稳定性不足。

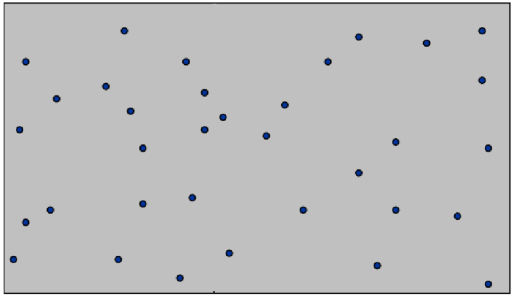
  3.相对于其他算法要求更多的内存资源。

5.3 快速矩阵算法的进展

- multiply two 2×2 matrices: 7 scalar sub-problems: $O(n^{\log_2 7}) = O(n^{2.807})$ [Strassen 1969]
- multiply two 2×2 matrices: 6 scalar sub-problems: $O(n^{\log_2 6}) = O(n^{2.585})$ (impossible)[Hopcroft and Kerr 1971]
- multiply two 3×3 matrices: 21 scalar sub-problems: $O(n^{\log_3 21}) = O(n^{2.771})$ (impossible)
- multiply two 20×20 matrices: 4460 scalar sub-problems: $O(n^{\log_{20} 4460}) = O(n^{2.805})$
- multiply two 48×48 matrices: 47217 scalar sub-problems: $O(n^{\log_{48} 47217}) = O(n^{2.780})$
- Best known: $O(n^{2.376})$ [Coppersmit-Winograd, 1987]
- Conjecture: $O(n^{2+\epsilon})$ for any $\epsilon > 0$;

6 最近点对问题

- **INPUT:** 平面上的n个点
- **OUTPUT:** 欧式距离最近的点对。

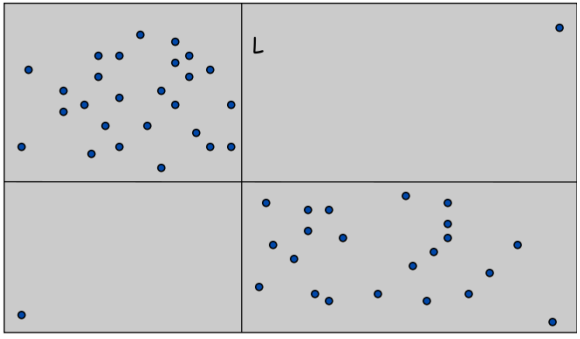


- 直线上:将点进行排序，找出2个距离最近的点，复杂度是O(n log n) 。
- 平面上:比较所有的点对，需要O(n2) 的时间。

问题：能够找出更快的算法吗?

6.1 解决方案

6.1.1 方案1：分成4个子问题



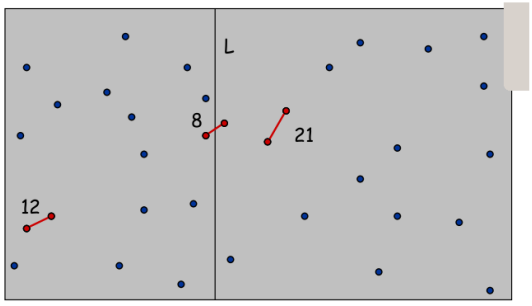
困难：分成的子集可能是不均衡的。我们无法保证每个子集都有差不多$n/4$个点。因此可能需要$O(n^2)$ 的时间去合并子集，最后的算法迭代表达式可能是这样的： $T(n) = 2T(\frac{n}{2}) + O(n^2)$ 。

6.1.2 方案2:分成2个子问题

解决方法：很容易将所有的点安装x坐标进行排序，然后用$x_{\lfloor \frac{n}{2} \rfloor}$ 将所有点分成2半。

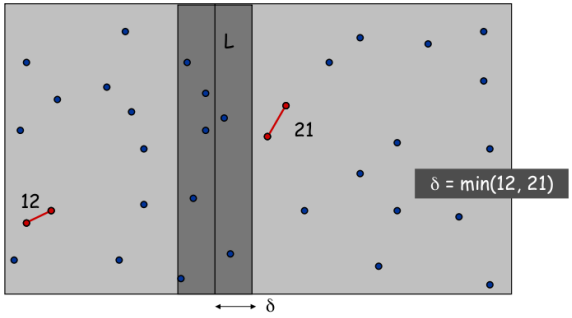
- **Divide:** 将所有点分成2半;
- **Conquer:** 找出每个子问题的解：左半部分的最近点对，右半部分的最近点对;

- **Combine:** 最近点对可能存在于边界左边一点和右边一点的情况。

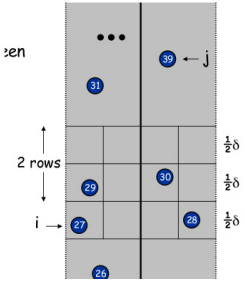


如果对左半边的每个点和右半边的每个点都做计算，那么时间开销很大：需要$O(n^2)$的时间。

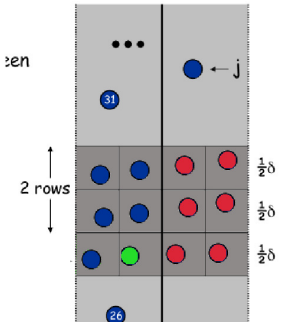
观察1：实际上，我们只需要观察L条带里面的点对的距离。那么条带的范围是2δ，δ 是取左边的最近点对和右边的最近点对的距离的较小者。如图所示，$\delta = \min(12, 21)$。



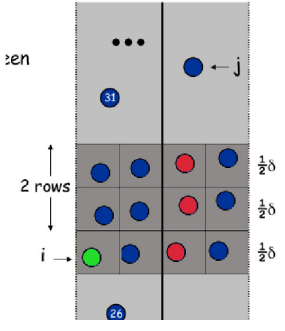
观察2:即使在条带里面也没必要两两之间去计算。对中间的条带进行分割，每个格子的边长是$\delta/2$。显然，一个格子里面只能装一个点。原因：当2个点在一个格子里面时，它们的距离将会小于δ。

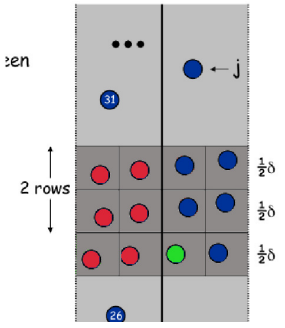


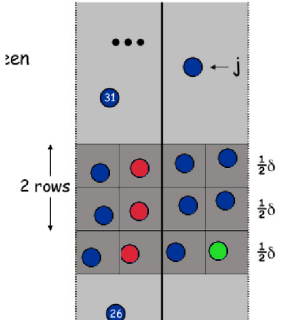
接下来分析可能出现的最近点对的计算，下面4种情况是可能出现的最近点对的情况。每幅图中



绿色的点只需要和红色的点继续比较即可。







对于左边的点，只需要对右边的最多6个点比较即可。我们将条带中的点按照y坐标进行排序，那么只需要找出临近的11个点即可，因为距离它最近的点一定在这11个点里面。 算法伪代码表示如下：

ClosestPair($p_i, ..., p_j$) /* $p_i, ..., p_j$ have already been sorted according to x-coordinate; */

1: **if** $j - i == 1$ **then**
2: return $d(p_i, p_j)$;
3: **end if**
4: Use the x-coordinate of $p_{\lfloor \frac{i+j}{2} \rfloor}$ to divide $p_i, ..., p_j$ into two halves;
5: δ_1 = ClosestPair(left half); $T(\frac{n}{2})$
6: δ_2 = ClosestPair(right half); $T(\frac{n}{2})$
7: $\delta = \min(\delta_1, \delta_2)$;
8: Sort points within the 2δ strip by y-coordinate; $O(n \log(n))$
9: Scan points in y-order and calculate distance between each point with its next 11 neighbors. Update δ if finding a distance less than δ; $O(n)$

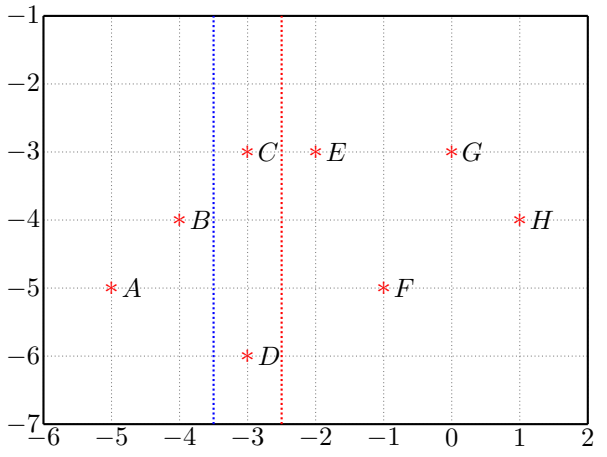
Time-complexity: $T(n) = 2T(\frac{n}{2}) + O(n \log n) = O(n \log^2(n))$.

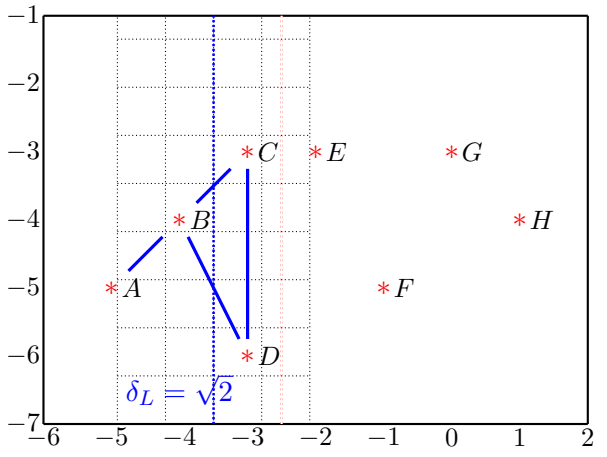
6.1.3 算法的提升

上面算法的合并时间：边界附近的点进行排序需要 $O(n \log n)$ 时间。

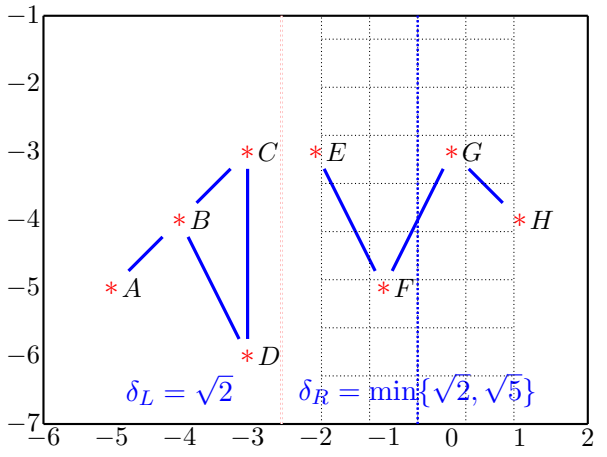
如果每一次迭代保持2个序列，一个按照x排序的序列，一个按照y排序的序列。像归并排序一样，将预先排序好的2个方向的序列合并起来，实际上只需要$O(n)$的时间。因此时间复杂度是 $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

6.1.4 举例分析：8个点的最近点对

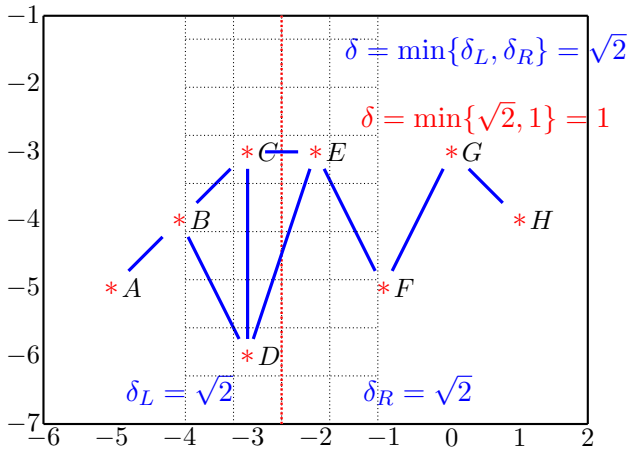




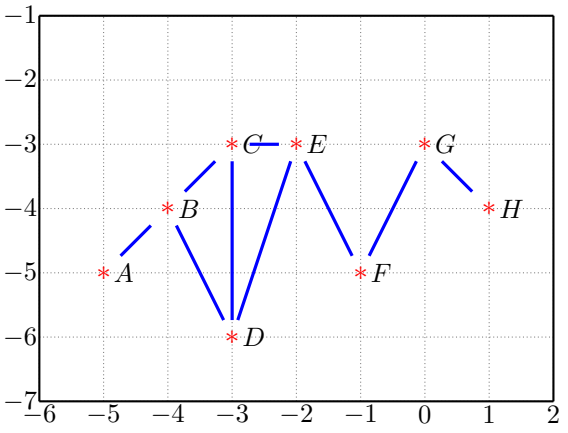
- Pair 1: $d(A, B) = \sqrt{2}$;
- Pair 2: $d(C, D) = 3$; $\Rightarrow$ min $= \sqrt{2}$; Thus, it suffices to calculate:
- Pair 3: $d(B, C) = \sqrt{2}$;
- Pair 4: $d(B, D) = \sqrt{5}$; $\Rightarrow \delta_L = \sqrt{2}$.



- Pair 5: $d(E, F) = \sqrt{5}$;
- Pair 6: $d(G, H) = \sqrt{2}$; $\Rightarrow$ min $= \sqrt{2}$; Thus, it suffices to calculate:
- Pair 7: $d(G, F) = \sqrt{5}$; $\Rightarrow \delta_R = \sqrt{2}$.



- Pair 8: $d(C, E) = 1$;
- Pair 9: $d(D, E) = \sqrt{10}$; $\Rightarrow \delta = 1$.



- 我们只计算了9对点。 剩下的19对是冗余的，原因如下：
  - 至少有一个点位于2δ条带之外。
  - 虽然2个点都在 2δ条带内, 但是它们之间的距离超过了2排格子 (格子大小: $\frac{\delta}{2} \times \frac{\delta}{2}$) 。