

排序

快速排序

算法原理

采用分而治之的策略：

❖ 将元素序列分为两个子序列： $S = S_L + S_R$ $//O(n)$

子序列规模缩小，而且

各自内部的排序互相独立： $\max(S_L) \leq \min(S_R)$

❖ 递归地对子序列分别排序

$T(n_L) + T(n_R) \approx 2 \times T(n/2)$

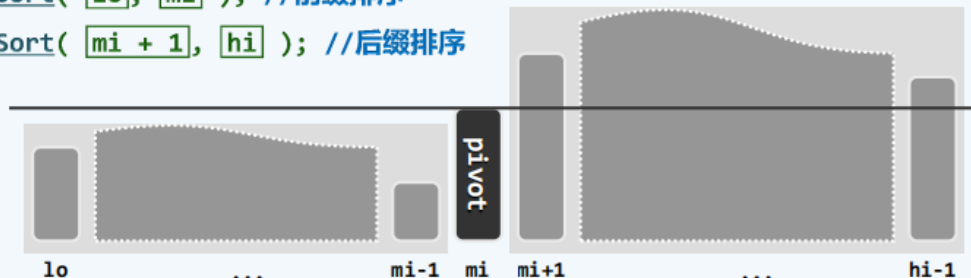
❖ 平凡解：只剩单个元素时，本身就是解

❖ 合并结果 $//O(1)$

以上策略的重点在于找到将原序列分裂为两个子序列的轴点，且在轴点左侧/右侧的元素都不比它大/小，因此轴点必定是序列中已然归位的一个元素。如果没有这样一个现成的轴点，我们就需要构造出这样一个轴点，或者说是让轴点归位，然后以轴点为界将原序列分成左右两个独立的子序列，并在子序列中递归做相同的处理（让轴点归位和分裂序列），当所有轴点归位后原序列就完成排序。主体算法如下所示：

一个序列完全有序，则其中所有元素皆是轴点，反之亦然。

```
❖ template <typename T> void Vector<T>::quickSort( Rank lo, Rank hi ) {  
    if ( hi - lo < 2 ) return; //单元素区间自然有序，否则  
    Rank mi = partition( lo, hi - 1 ); //先构造轴点，再  
    quickSort( lo, mi ); //前缀排序  
    quickSort( mi + 1, hi ); //后缀排序  
}
```



如何构造轴点

即partition函数如何实现。

构造轴点

❖ 任取一候选者（比如[0]）

❖ 2个指针：将序列分为3段

前缀L：≤ 轴点，初始为空

后缀G：≥ 轴点，初始为空

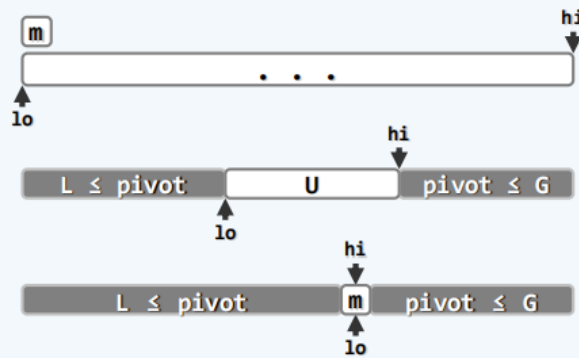
中段U：? 待确定，初始为全集

❖ 交替地向内移动lo和hi，并

检查所指元素：若更小/大，则转移归入L/G

❖ 当lo = hi时，只需将候选者嵌入于L、G之间，它即是轴点！

❖ 整个过程中，各元素最多移动一次（候选者两次）——累计 $O(n)$ 时间



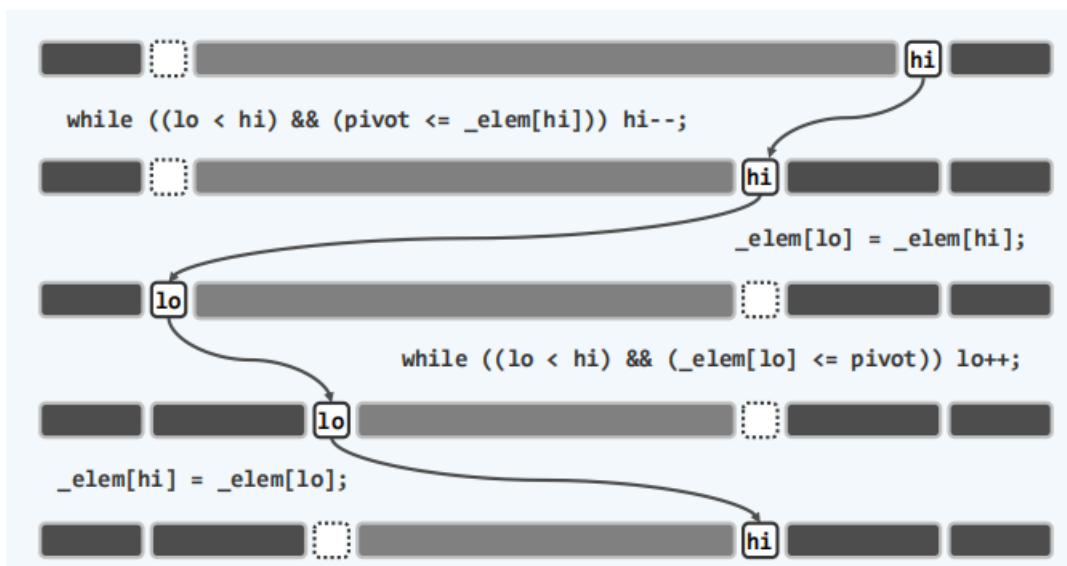
算法实现：

算法A：实现

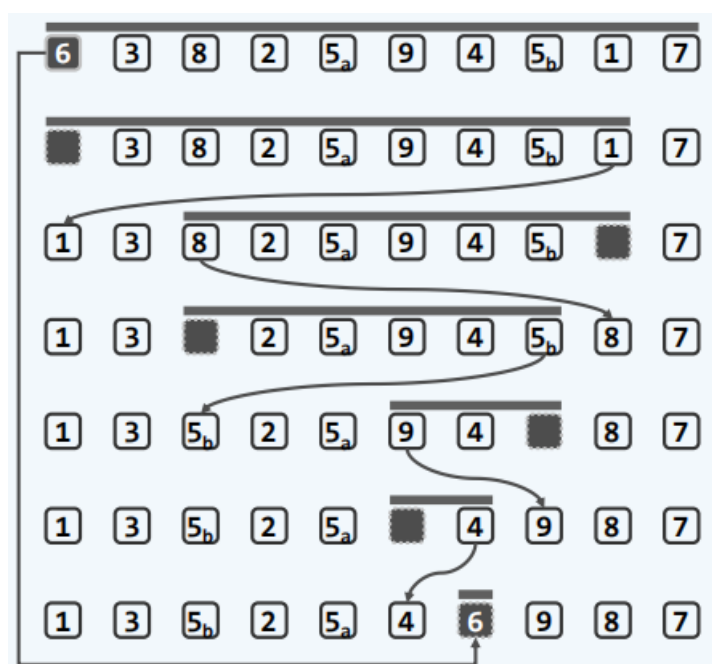
```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { //[lo, hi]
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); //随机交换
    T pivot = _elem[ lo ]; //经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { //从两端交替地向中间扫描，彼此靠拢
        while ( lo < hi && pivot <= _elem[ hi ] ) hi--; //向左拓展G
        _elem[ lo ] = _elem[ hi ]; //凡小于轴点者，皆归入L
        while ( lo < hi && _elem[ lo ] <= pivot ) lo++; //向右拓展L
        _elem[ hi ] = _elem[ lo ]; //凡大于轴点者，皆归入G
    } //assert: lo == hi
    _elem[ lo ] = pivot; return lo; //候选轴点归位；返回其秩
}
```

算法原理图：

- 不变性： $L < \text{PIVOT} < G$ ；U[lo]和U[hi]交替空闲；
- 单调性：U不断减小；L和G不断增大。



举例说明：



性能分析

- ❖ **不稳定**：lo/hi的移动方向相反，左/右侧的大/小重复元素可能前/后**颠倒**
- ❖ **就地**：只需 $O(1)$ 附加空间——时间呢？
- ❖ **最好情况**：每次划分都（接近）**平均**，轴点总是（接近）**中央**

$$T(n) = 2 \times T\left(\frac{(n-1)}{2}\right) + O(n) = O(n \log n) \quad // \text{到达下界！}$$
- ❖ **最坏情况**：每次划分都**极不均衡** **//比如，轴点总是最小/大元素**

$$T(n) = T(n-1) + T(0) + O(n) = O(n^2) \quad // \text{与起泡排序相当！}$$
- ❖ 即便采用**随机选取**、（Unix）**三者取中**之类的策略
也只能**降低**最坏情况的概率，而无法**杜绝**

平均性能

❖ $O(n \log n)$ ——以均匀独立分布为例...

$$\begin{aligned} \text{❖ } T(n) &= (n + 1) + \left(\frac{1}{n}\right) \times \sum_{k=1}^n [T(k - 1) + T(n - k)] \\ &= (n + 1) + \left(\frac{2}{n}\right) \times \sum_{k=1}^n T(k - 1) \end{aligned}$$

$$\text{❖ } nT(n) - (n - 1)T(n - 1) = 2n + 2T(n - 1)$$

$$\begin{aligned} \text{❖ } T(n)/(n + 1) &= 2/(n + 1) + T(n - 1)/n \\ &= 2/(n + 1) + 2/n + T(n - 2)/(n - 1) \\ &= 2/(n + 1) + 2/n + 2/(n - 1) + T(n - 3)/(n - 2) \\ &= 2/(n + 1) + 2/n + 2/(n - 1) + \dots + 2/2 + T(0)/1 \\ &= 1.39 \times \log n \end{aligned}$$

另一版本

算法原理：

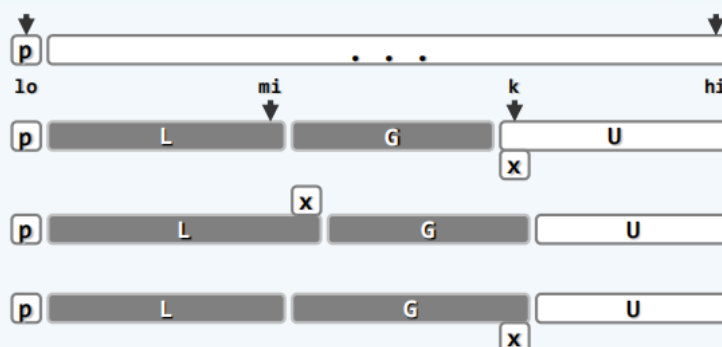
算法c：思路

❖ $[lo, hi]$
 $[lo]$ $[L] \ (lo, mi]$ $[G] \ (mi, k)$ $[U] \ [k, hi]$

❖ 考察 $[k]$ ：若小于轴点，则经 $swap(mi + 1, k)$ ， $[L]$ 拓展、 $[G]$ 滚动平移；否则，直接 $[G]$ 拓展

$[k]$ 不小于轴点 ? 直接 $[G]$ 拓展 : $[G]$ 滚动后移， $[L]$ 拓展

$pivot \leq S[k]$? $k++$: $swap(S[mi], S[k])$



算法实现：

算法C：实现

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { //[lo, hi]

    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo + 1 ) ] ); //随机交换

    T pivot = _elem[ lo ]; int mi = lo;

    for ( int k = lo + 1; k <= hi; k++ ) //自左向右考查每个[k]

        if ( _elem[ k ] < pivot ) //若[k]小于轴点，则将其

            swap( _elem[ ++mi ], _elem[ k ] ); //与[mi]交换，L向右扩展

    swap( _elem[ lo ], _elem[ mi ] ); //候选轴点归位（从此名副其实）

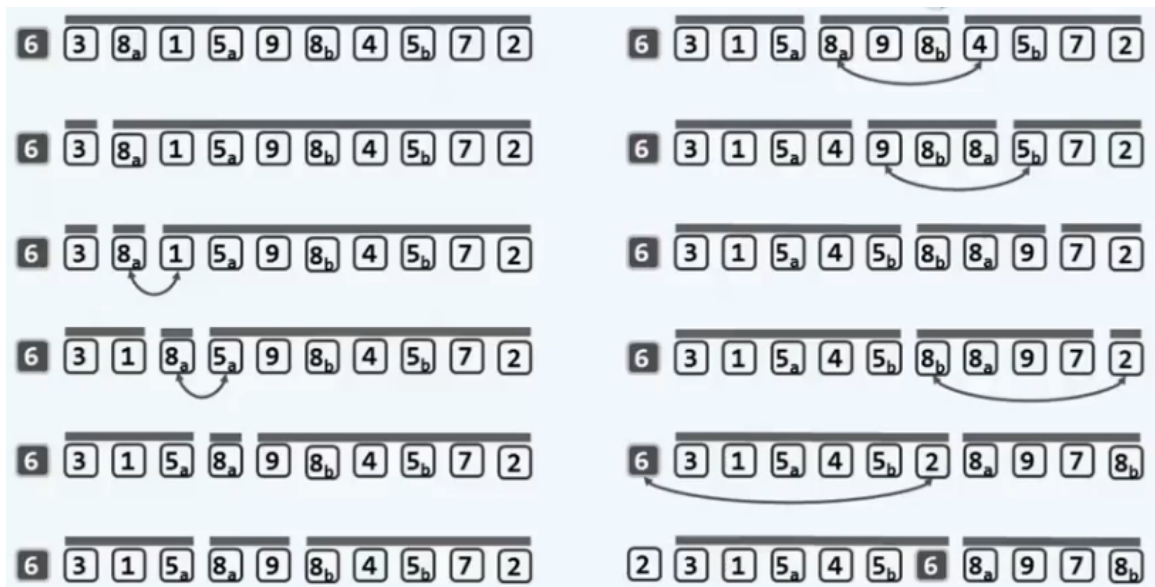
    return mi; //返回轴点的秩

}
```

Data Structures & Algorithms (Fall 2013), Tsinghua University

16

举例说明：



选取算法

1. K选取：选取排在第K位的元素
2. 中位数选取：选取排在中间 $\lfloor n/2 \rfloor$ 的元素
3. 众数选取：选取总数超过一半的元素

众数选取

三种策略

1. 众数必是中位数，因此只要找出中位数，然后进行验证即可：

```
template <typename T> bool majority( Vector<T> A, T & maj )

{ return majEleCheck( A, maj = median( A ) ); }
```

2. 众数必是频繁数，因此只要找出频繁数，然后进行验证即可：

```
template <typename T> bool majority( Vector<T> A, T & maj )
{ return majEleCheck( A, maj = mode( A ) ); }
```

3. 盐水策略，即通过减治找出唯一候选者，然后进行验证：

```
template <typename T> bool majority( Vector<T> A, T & maj )
{ return majEleCheck( A, maj = majEleCandidate( A ) ); }
```

盐水策略

盐水策略的基本原理：

众数：减而治之

❖ 若在向量A的前缀P (|P|为偶数) 中，元素x出现的次数恰占半数，则A有众数仅当，对应的后缀A - P有众数m，且m就是A的众数

❖ 既然最终总要花费 $O(n)$ 时间做验证，故而只需考虑A的确含有众数的两种情况：

1. 若 $x = m$ ，则在排除前缀P之后，m与其它元素在数量上的差距保持不变
(从浓度50%的盐水中析出50%的一部分，剩余部分的浓度仍50%)
2. 若 $x \neq m$ ，则在排除前缀P之后，m与其它元素在数量上的差距不致缩小

❖ 因此，可采用减而治之策略，逐步缩减原问题的规模，直至平凡

Data Structures & Algorithms (Fall 2013), Tsinghua University

4

根据以上原理，可知该策略的执行过程就是：不断找出存在半数者的前缀并删去，在最后剩余序列中超过半数的那个元素就是众数的唯一候选者。

1. 确定删除前缀：将首元素作为当前的众数候选者x，向后扫描直到x占据一半，此时的扫描区间就是需要删除的前缀；
2. 删除前缀并选取当前序列的首元素作为新的众数候选者，继续步骤1；
3. 重复步骤1和2，直到序列没有可以删除的前缀时，此时序列中的超过半数者就是真正的众数候选者。

算法实现

```
❖ template <typename T> T majEleCandidate( Vector<T> A ) {
    T maj; //众数候选者
    // 线性扫描：借助计数器c，记录maj与其它元素的数量差额
    for ( int c = 0, i = 0; i < A.size(); i++ )
        if ( 0 == c ) { //每当c归零，都意味着此时的前缀P可以剪除// 此时表示前缀已经删除，因此进入
            maj = A[i]; c = 1; //众数候选者改为新的当前元素      新的序列重新寻找前缀进行删除。
        } else //否则
            maj == A[i] ? c++ : c--; //相应地更新差额计数器
    return maj; //至此，原向量的众数若存在，则只能是maj —— 尽管反之不然
}
```

K选取

直观尝试

❖ 蛮力

0

...

k

...

n-1

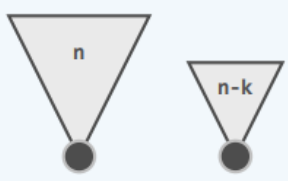
对A排序 $//O(n\log n)$

从前向后行进k步 $//O(k) = O(n)$

❖ 堆(A)

将所有元素组织为小顶堆 $//O(n)$


连续调用k次 `delMin()` $//O(k\log n)$



❖ 堆(B)

任选（比如前）k个元素，组织为大顶堆 $//O(k)$

对于剩余的 $n - k$ 个元素，各调用一次 `insert()` 和 `delMax()` $//O(2*(n - k)*\log k)$



使用两个堆实现，即包含k个元素的大顶堆和包含剩余元素的小顶堆：如果大顶堆的堆顶元素大于小顶堆的堆顶元素，就将其交换并重新下滤调整。循环这个过程直到大顶堆的堆顶元素不再大于小顶堆的堆顶元素，这时小顶堆的堆顶元素g就是要求的第k个元素（因为此时小顶堆存在n-k-1个元素比g大，而大顶堆的k个元素比g小，两者相夹，g就恰好处于第k个元素的位置上）。

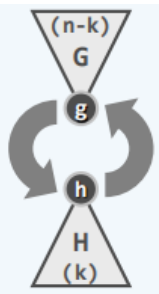
❖ H：任取k个元素，组织为大顶堆 $//O(k)$

G：其余 $n - k$ 个元素，组织为小顶堆 $//O(n - k)$

反复地：比较h和g $//O(1)$

如有必要，交换之 $//O(2 \times (\log k + \log(n - k)))$

直到： $h \leq g$ $//O(\min(k, n - k))$



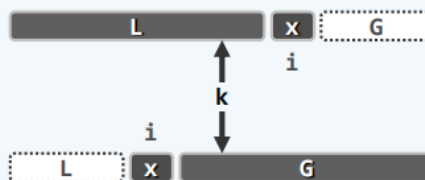
快速选取

这是快速排序的一种利用，其重点在于减治策略。其过程如下：

1. 随机选取一个轴点P进行归位；
2. 如果目标元素K的秩小于P，则可以将选取范围缩减至比P小的区间L；如果目标元素K的秩大于P，则可以将选取范围缩减至比P大的区间G；如果目标元素K的秩等于P，则可以直接返回；
3. 在新的区间内重复步骤1和2，直到区间缩减为单元素的情况，此时该元素就是所求目标元素。

quickSelect()

```
template <typename T> void quickSelect( Vector<T> & A, Rank k ) {
    for ( Rank lo = 0, hi = A.size() - 1; lo < hi; ) {
        Rank i = lo, j = hi; T pivot = A[lo];
        while ( i < j ) { //O(hi - lo + 1) = O(n)
            while ( i < j && pivot <= A[j] ) j--; A[i] = A[j];
            while ( i < j && A[i] <= pivot ) i++; A[j] = A[i];
        } //assert: i == j
        A[i] = pivot;
        if ( k <= i ) hi = i - 1;
        if ( i <= k ) lo = i + 1;
    } //A[k] is now a pivot
}
```



Data Structures & Algorithms (Fall 2013), Tsinghua University

3

线性选取

这是基于快速选取的一种改进，即更加有效的缩减目标元素的选取区间。其过程如下：

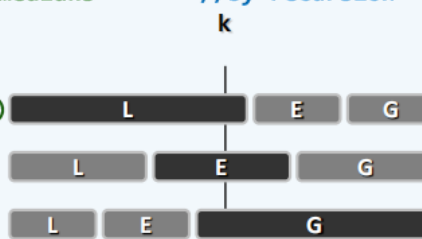
1. 选取一个适当且足够小的值 Q ；
2. 如果目标向量的规模 n 比 Q 还小，则可以直接用平凡算法（直接排序）求得目标元素 k ；
3. 如果目标向量的规模足够大，则将目标向量切分成 n/Q 个规模为 Q 的子向量；
4. 对这 n/Q 个子向量进行排序以求得它们各自的中位数，并通过递归的方式求得这些中位数的中位数 M ；
5. 按照所求的中位数 M 将原向量进行划分为三个区间：大于 M 的区间、小于 M 的区间和等于 M 的区间；
6. 此时便可以按照目标元素与中位数 M 之间的大小，将选取区间缩减至对应的小区间上，并按照上述步骤递归处理该子区间，直至平凡的情况。但如果目标元素等于其中的某一个中位数 M ，则可以直接返回。

总体来说就是，找到合适的划分点将选取区间缩减至更小的范围内，并不断重复这个过程，直到区间够小，可以直接排序求得目标元素的秩或刚好划分点就是所求元素。

linearSelect()

Let Q be a small constant

0. if ($n = |A| < Q$) return trivialSelect(A, k)
1. else divide A evenly into n/Q subsequences (each of size Q)
2. Sort each subsequence and determine n/Q medians //e.g. by insertionsort
3. Call linearSelect to find M , median of the medians //by recursion
4. Let $L/E/G = \{ x \mid x < / = / > M \mid x \in A \}$
5. if ($k \leq |L|$) return linearSelect(L, k)
- if ($k \leq |L| + |E|$) return M
- return linearSelect($G, k - |L| - |E|$)

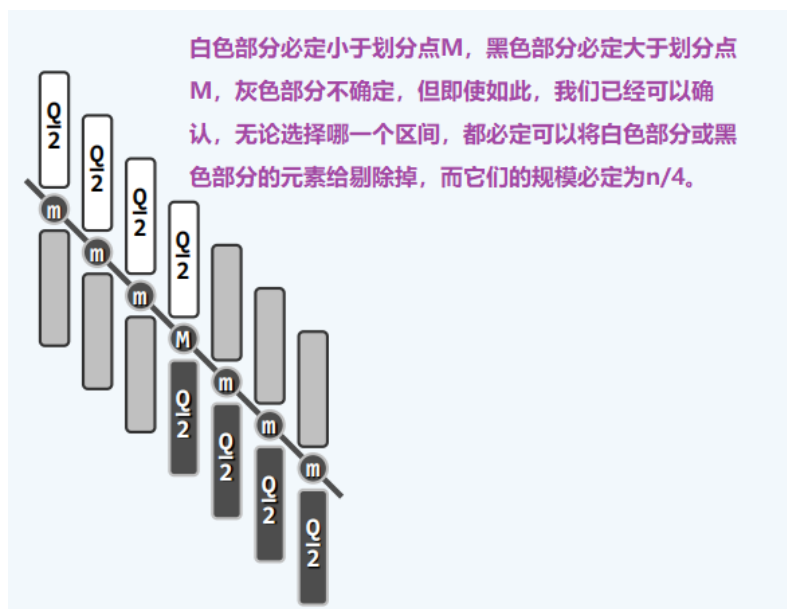


Data Structures & Algorithms (Fall 2013), Tsinghua University

4

规模的有效缩减

经过每次划分，都可以将规模至少缩减至原来的3/4：



复杂度分析

- ❖ 将`linearSelect()`算法的运行时间记作 $T(n)$
- ❖ 第0步： $O(1)$ = $O(Q \log Q)$ //递归基：序列长度 $|A| \leq Q$
- ❖ 第1步： $O(n)$ //子序列划分
- ❖ 第2步： $O(n)$ = $O(1) \times n/Q$ //子序列各自排序，并找到中位数
- ❖ 第3步： $T(n/Q)$ //从 n/Q 个中位数中，递归地找到全局中位数
- ❖ 第4步： $O(n)$ //划分子集L/E/G，并分别计数 —— 一趟扫描足矣
- ❖ 第5步： $T(3n/4)$ // 已知规模至少可以缩减至 $3n/4$

综合各步可得总体的时间复杂度为：

$$\diamond T(n) = O(n) + T(n/Q) + T(3n/4)$$

为使之解作线性函数，只需保证

$$n/Q + 3n/4 < n$$

或等价地

$$1/Q + 3/4 < 1$$

❖ 比如，若取 $Q = 5$ ，则存在常数 c ，使得

$$T(n) = cn + T(n/5) + T(3n/4)$$


$$T(n) = O(20cn) = O(n)$$

希尔排序

基本原理

Shellsort

- ❖ Donald L. Shell, 1959：将整个序列视作一个矩阵，逐列各自排序 **w-sorting**
- ❖ 递减增量 **diminishing increment**
 重排矩阵，使其更窄，再次逐列排序 **w-ordered**
 如此往复，直至矩阵变成一列 **1-sorting**
- ❖ 步长序列 **step sequence**：由各矩阵宽度构成的逆序列
 $\mathcal{W} = \{ w_1 = 1, w_2, w_3, \dots, w_k, \dots \}$
- ❖ 正确性：最后一次迭代，等同于全排序
1-ordered = sorted



$w_1 = 1$

Data Structures & Algorithms (Fall 2013), Tsinghua University

通俗点讲就是，将序列分成多个组（即矩阵的多个列），然后进行组内排序，等所有组排完序后，再将整个序列重新分成组数更小（即矩阵更窄）的多个组，再进行组内排序。不断重复这个过程，直到组数变为1并进行一次全排序（即1-sorting），从而结束排序。总结一句话就是不断分组排序，直到组数为1。

步长序列：对应每次排序的组数所组成的序列。

举例说明

Example : $w_5 = 8$

80 23 19 40 85 1 18 92 71 8 96 46 12

按每行至多8个元素排成矩阵
(或者说将序列分成8组)

80 23 19 40 85 1 18 92
71 8 96 46 12

每列一组
进行排序

71 8 19 40 12 1 18 92
80 23 96 46 85

排序后, 按每列(组)原本在序列中的位置重新组合成完整的序列(实际上在排序过程中各组在序列中的位置并没有变化, 变化的只是组内元素之间的位置)

71 8 19 40 12 1 18 92 80 23 96 46 85

宽度减少并继续以上步骤

Example : $w_4 = 5$

71 8 19 40 12 1 18 92 80 23 96 46 85

71 8 19 40 12
1 18 92 80 23
96 46 85

1 8 19 40 12
71 18 85 80 23
96 46 92

1 8 19 40 12 71 18 85 80 23 96 46 92

Example : $w_3 = 3$

1 8 19 40 12 71 18 85 80 23 96 46 92

| | | | | | | |
|----|----|----|--|----|----|----|
| 1 | 8 | 19 | | 1 | 8 | 19 |
| 40 | 12 | 71 | | 18 | 12 | 46 |
| 18 | 85 | 80 | | 23 | 85 | 71 |
| 23 | 96 | 46 | | 40 | 96 | 80 |
| 92 | | | | 92 | | |

1 8 19 18 12 46 23 85 71 40 96 80 92

Data Structures & Algorithms (Fall 2013), Tsinghua University

4

Example : $w_2 = 2$

1 8 19 18 12 46 23 85 71 40 96 80 92

| | | | | |
|----|----|--|----|----|
| 1 | 8 | | 1 | 8 |
| 19 | 18 | | 12 | 18 |
| 12 | 46 | | 19 | 40 |
| 23 | 85 | | 23 | 46 |
| 71 | 40 | | 71 | 80 |
| 96 | 80 | | 92 | 85 |
| 92 | | | 96 | |

1 8 12 18 19 40 23 46 71 80 92 85 96

Data Structures & Algorithms (Fall 2013), Tsinghua University

5

Example : $w_1 = 1$

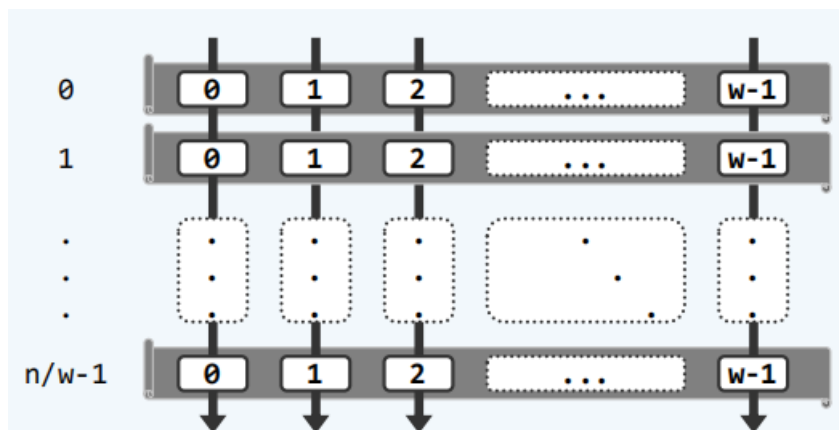
| | |
|----|----|
| 1 | 1 |
| 8 | 8 |
| 12 | 12 |
| 18 | 18 |
| 19 | 19 |
| 40 | 23 |
| 23 | 40 |
| 46 | 46 |
| 71 | 71 |
| 80 | 80 |
| 92 | 85 |
| 85 | 92 |
| 96 | 96 |

Data Structures & Algorithms (Fall 2013), Tsinghua University

6

如何分组

实际上并不需要真的进行分组，各组的排序其实都可以在原序列上进行。假设此时宽度为 w （组数为 w ），则第 i 组的元素的秩就是 $A[i+kw]$ （ $0 \leq k < n/w$ ），所以我们只需根据这些元素的秩在逻辑上完成重排，而无需在物理上实现分组。能够这样做的原因就在于向量循秩访问的特性。



组内排序

每组内部的排序必须采用输入敏感的算法（即输入的序列越有序，所花费的时间就越少），以保证有序性持续改善，且总体成本足够低廉。

对此，插入排序就是一个非常适合的算法，其实际运行时间完全取决于输入序列所含的逆序对总数。

但无论采用哪种组内排序算法，希尔排序的总体效率还是取决于具体使用何种步长序列。

希尔序列

这是由希尔提出的一个效率不太高的步长序列。

Shell's Sequence

- ❖ Shell 1959: $\mathcal{H}_{shell} = \{1, 2, 4, 8, \dots, 2^k, \dots\}$
- ❖ 实际上，采用 \mathcal{H}_{shell} ，在最坏情况下需要运行 $\Omega(n^2)$ 时间...
- ❖ 考查由子序列 $A = \text{unsort}[0, 2^{N-1})$ 和 $B = \text{unsort}[2^{N-1}, 2^N)$ 交错而成的序列

11
4
14
3
10
0
15
1
9
6
8
7
13
2
12
5

- ❖ 在做 2-sorting 时，A、B 各成一列；故此后必然各自有序

8
0
9
1
10
2
11
3
12
4
13
5
14
6
15
7

- ❖ 然而其中的逆序对依然很多，最后的 1-sorting 仍需 $1 + 2 + 3 + \dots + 2^{N-1} = \Omega(n^2/4)$ 时间
- ❖ 根源在于， \mathcal{H}_{shell} 中各项并不互素，甚至相邻项也非互素

Data Structures & Algorithms, Tsinghua University

内容拓展

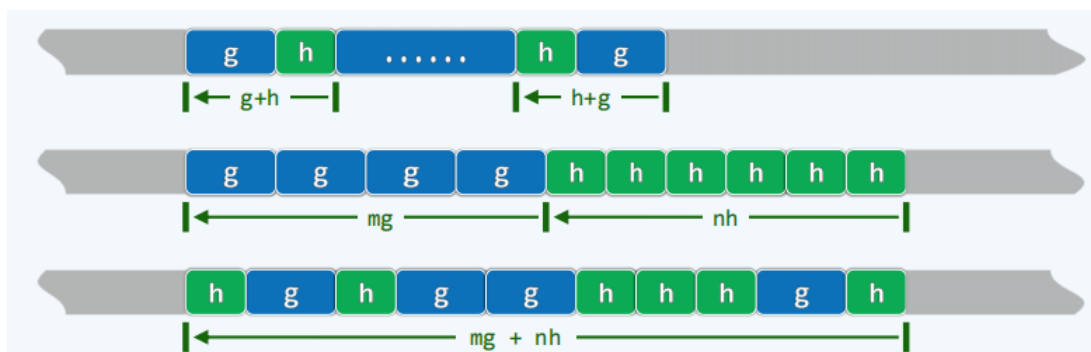
1. 一些概念：

h -ordered：指在该序列中距离间隔 h 的元素之间呈顺序性；

1-ordered：指距离间隔为1的元素之间都具有顺序性，即全排序；

h -sorting：使序列变成 h -ordered的过程，即希尔排序的一次迭代。

2. 一个 g -ordered 的序列经过 h -sorting 后，它既是 g -ordered，也是 h -ordered，称为 (g, h) -ordered，或严格地表示成 $(mg+nh)$ -ordered。也就是说，凡是任何两个元素之间的间隔能表示成 $(mg+nh)$ ，它们之间就是顺序的。



3. 假设存在一个 (g, h) -ordered 的序列，考察其中某个元素 i ，它与哪些元素具有顺序性？
- 如果 g 和 h 是互素的，根据定理“不能用 $(mg+nh)$ 表示的最大数值为 $(g-1)(h-1)-1$ ”可知，与元素 i 间隔大于 $(g-1)(h-1)-1$ 的所有元素都与 i 具有顺序性。反过来说，间隔小于该数值的元素都有可能与 i 构成逆序对，它们的总数影响组内插入排序的效率，但随着希尔排序每一次迭代，该数值会越来越小，使得插入排序的效率越来越高；
 - 如果 g 和 h 是非互素的，不能用 $(mg+nh)$ 表示的数值存在于任何范围，也就是序列中的任何元素都有可能与 i 构成逆序对，它们的总数无法确定，而且随着希尔排序的进行，该总数还可能增加，这也是希尔序列效率不高以及步长序列要求互素的原因。
4. 由上可知，随着希尔排序的不断迭代，序列中的逆序对会不断减少，而插入排序的时间复杂度正比于序列的逆序对数，所以插入排序花费的时间也会不断减少，这也是希尔排序效率高的原因。然而，在采用希尔序列的希尔排序中，有可能导致逆序对数不降反增的趋势，这也是希尔序列效率低的原因，所以采用适当的步长序列非常重要，最好保证步长序列中各项之间是互素的。