1 Preliminaries

```c
void  X_Sort ( ElementType  A[ ],  int N )
```

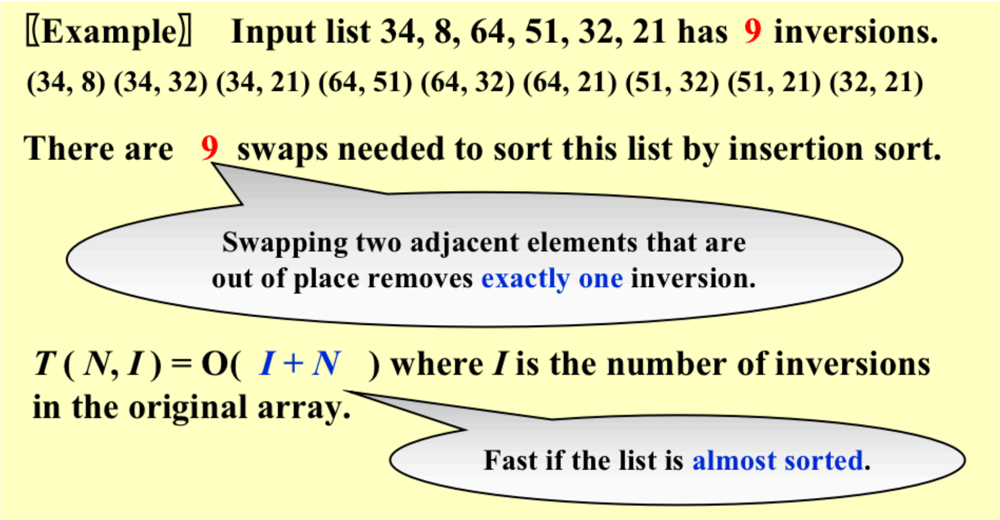
2 Insertion Sort

```c
void InsertionSort ( ElementType A[ ], int N )
{
      int  j, P;
      ElementType  Tmp;

      for ( P = 1; P < N; P++ ) {
        Tmp = A[ P ];  /* the next coming card */
        // 默认1-P已经排好了，0是第一个，根本不动
        for ( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
          // 这里经常错是A[j-1] 和 tmp比较的！！！
          A[ j ] = A[ j - 1 ];
        // 如果前面一个还大于Tmp
        /* shift sorted cards to provide a position for the new coming card */
        A[ j ] = Tmp;  /* place the new card at the proper position */
      } /* end for-P-loop */
}
```

$$
\begin{gathered}
T_{worst} = O(N^2) \\
T_{best} = O(N)
\end{gathered} \tag{8}
$$

3 A Lower Bound for Simple Sorting Algorithms

- 【Definition】An **inversion** 逆序对 in an array of numbers is any ordered pair (i, j) having the property that i < j but A[i] > A[j].



- 【Theorem】The average number of inversions in an array of N distinct numbers is $N(N-1)/4$. 平均的：最好的0 + 最坏的$N(N-1)/2$ 除以2
- 【Theorem】Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average. 任何算法只是交换相邻单元来排序，平均时间复杂度最好只能是N^2 。Bubble sort、Insertion Sort这些都是。因为每次只消除一对逆序对。

4 Shellsort

4.1 Donald Shell

〖Example〗 Sort:

	81	94	11	96	12	35	17	95	28	58	41	75	15
5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

- 先隔五个取一个数，进行Insertion Sort排序，消除了一些逆序对
- 再隔3个数，重复Insertion Sort，
- 上一轮的排序（消除的逆序对）可以保存下一轮
- 最后一步一定是 1-Sort ，增量为1，退化成Insertion Sort
- 本质上就是Insertion Sort执行的时候每次就是就交换两个数，相当于一次消一个逆序对
- 而Shell Sort是隔着很多个去交换，“冥冥之中”消除了很多个逆序对
- 也有可能“冥冥之中”一个逆序对都没消除，那么就还是和Insertion sort一样的N平方的复杂度

```c
void Shellsort( ElementType A[ ], int N )
{
      int  i, j, Increment;
      ElementType  Tmp;
      for ( Increment = N / 2; Increment > 0; Increment /= 2 )
        /*h sequence */
        for ( i = Increment; i < N; i++ ) { /* insertion sort */
          Tmp = A[ i ];
          for ( j = i; j >= Increment; j - = Increment )
            if( Tmp < A[ j - Increment ] )
              A[ j ] = A[ j - Increment ];
            else
              break;
            A[ j ] = Tmp;
        } /* end for-I and for-Increment loops */
}
```

Worst - Case Analysis

【Theorem】 The worst-case running time of Shellsort, using Shell's increments, is $\Theta(N^2)$.

〖Example〗 A bad case:

	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
8-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
4-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
2-sort	1	9	2	10	3	11	4	12	5	13	6	14	7	15	8	16
1-sort	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Pairs of increments are not necessarily relatively prime.
Thus the smaller increment can have little effect.

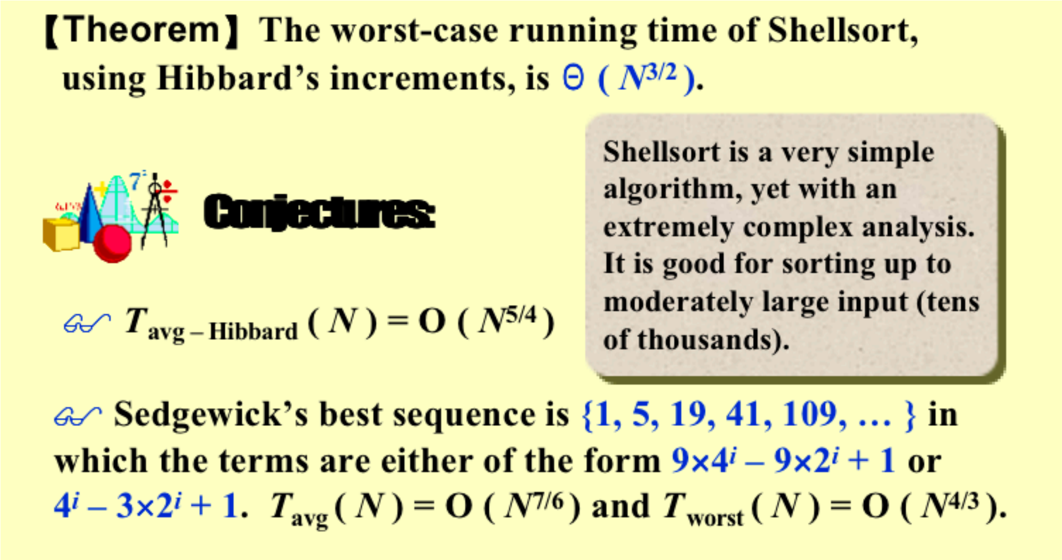
- 8 - Sort, 4 - Sort, 2 - Sort 什么事情都没做，只是在最后一轮把工作全做了
- Worst case就是Insertion Sort
- how to solve it: Shell Sort 中里面的increment 不要有公约数，不然容易出现上述的情况

4.2 Hibbard's Increment Sequence

$$h_k = 2^k - 1 \tag{9}$$

- Hibbard提出的: 最坏$\Theta(N^{3/2})$， 最好$O(N)$，平均$O(N^{5/4})$

4.3 Sedgewick's Increment Sequence



- Sedgewick提出的: 最坏$\Theta(N^{4/3})$， 最好$O(N)$，平均$O(N^{7/6})$

5 Heapsort

```
void Heapsort( ElementType A[ ], int N )
{  int i;
    for ( i = N / 2; i >= 0; i - - ) /* BuildHeap */
        PercDown( A, i, N );
    for ( i = N - 1; i > 0; i - - ) {
        Swap( &A[ 0 ], &A[ i ] ); /* DeleteMax */
        PercDown( A, 0, i );
    }
}
```

- 是建一个大顶堆，目的是每次换下去，就会把目前最大的扔到目前的最后面
- 这种方法会比再建一个 `Tmp[N]` 会省一倍空间
- 注意：现在从0单元开始，0单元也存数据，child也需要改变
- 【Theorem】The average number of comparisons used to heapsort a random permutation of N distinct items is

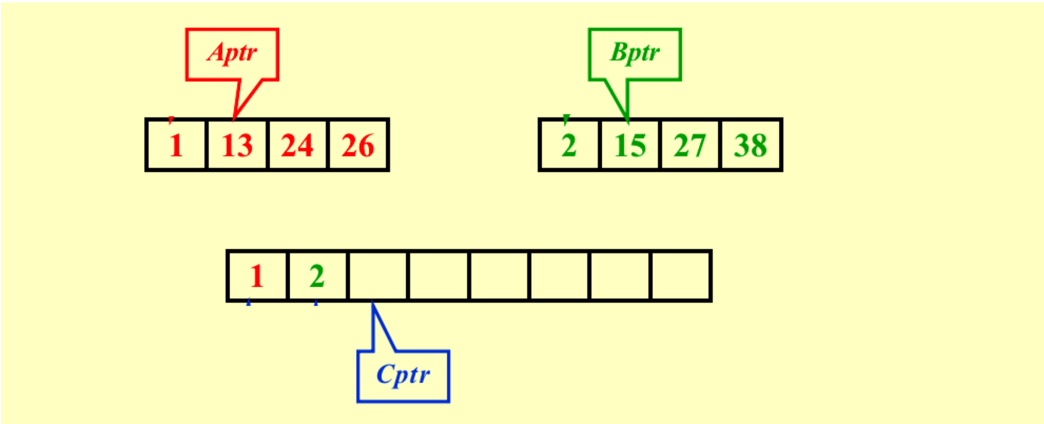
$$T_{avg} = 2NlogN - O(N\log\log N) \tag{10}$$

- Note: Although Heapsort gives the best average time, in practice it is slower than a version of Shellsort that uses Sedgewick's increment sequence.
- 实际中 Shellsort 会比Heapsort快，因为Heapsort交换的次数非常多

6 MergeSort

6.1 Merge two sorted lists

这一步是将两个已经排好序的小数组合并成一个大数组



$$T(N) = O(N) \tag{11}$$

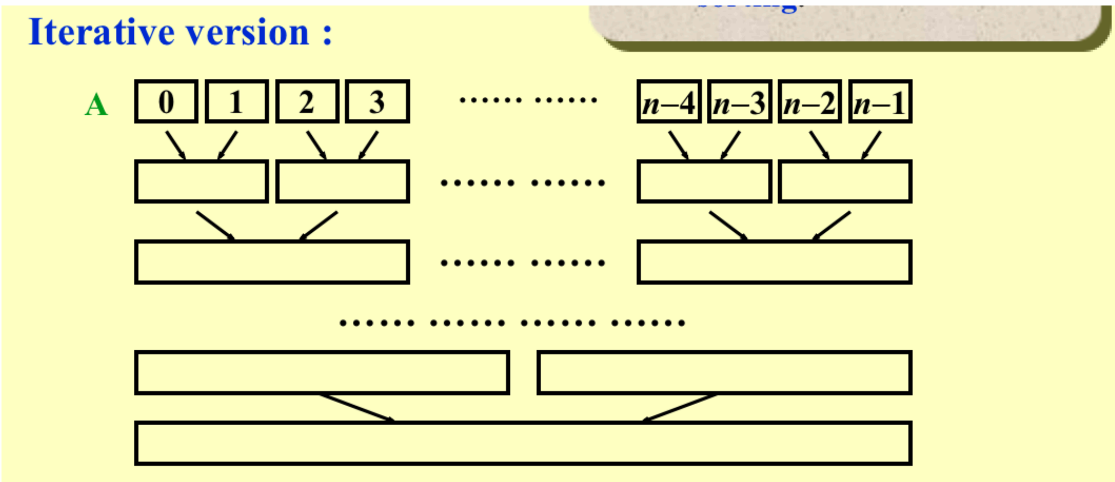
6.2 Mergesort

```c
void Mergesort( ElementType A[ ], int N )
{   ElementType  *TmpArray;  /* need O(N) extra space */
    TmpArray = malloc( N * sizeof( ElementType ) );
    if ( TmpArray != NULL ) {
      MSort( A, TmpArray, 0, N - 1 );
      free( TmpArray ); // Space can be recovered
    }
    else  FatalError( "No space for tmp array!!!" );
}

void MSort( ElementType A[ ], ElementType TmpArray[ ], int Left, int Right )
{   int  Center;
    if ( Left < Right ) {  /* if there are elements to be sorted */
      Center = ( Left + Right ) / 2;
      MSort( A, TmpArray, Left, Center );              /* T( N / 2 ) */
      MSort( A, TmpArray, Center + 1, Right );         /* T( N / 2 ) */
      Merge( A, TmpArray, Left, Center + 1, Right );  /* O( N ) */
    }
```

```c
}

/* Lpos = start of left half, Rpos = start of right half */
void Merge( ElementType A[ ], ElementType TmpArray[ ], int Lpos, int
Rpos, int RightEnd )
{   int  i, LeftEnd, NumElements, TmpPos;
    LeftEnd = Rpos - 1;
    TmpPos = Lpos;
    NumElements = RightEnd - Lpos + 1;
    while( Lpos <= LeftEnd && Rpos <= RightEnd ) /* main loop */
        /* 不管谁先到底，只要到底，就退出 */
        if ( A[ Lpos ] <= A[ Rpos ] )
          TmpArray[ TmpPos++ ] = A[ Lpos++ ];
        else
          TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    // 下面两步就是把没排好的加到进去
    while( Lpos <= LeftEnd ) /* Copy rest of first half */
        TmpArray[ TmpPos++ ] = A[ Lpos++ ];
    while( Rpos <= RightEnd ) /* Copy rest of second half */
        TmpArray[ TmpPos++ ] = A[ Rpos++ ];

    for( i = 0; i < NumElements; i++, RightEnd - - )
        /* Copy TmpArray back */
        A[ RightEnd ] = TmpArray[ RightEnd ];
}
```

- 如果 Msort 中每次的 TmpArray 都会被动态分配（每次完都会回收），$S(N) = (N \log N)$

$$T(N) = O(N + N \log N) \tag{12}$$



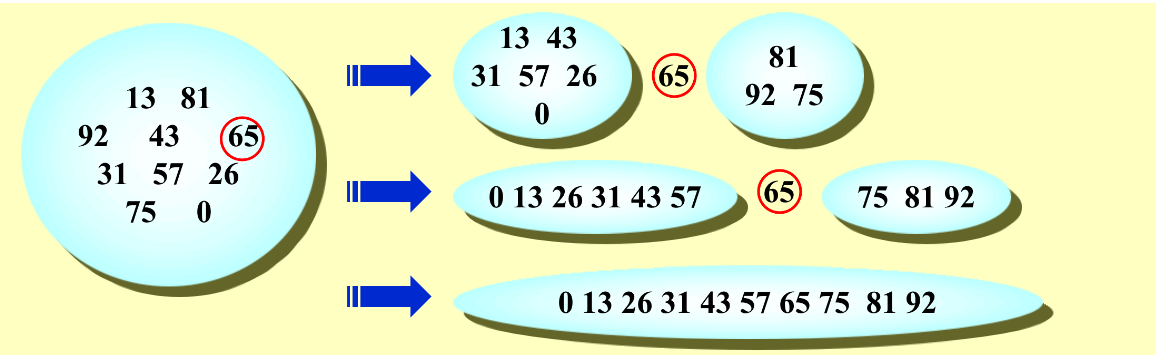
- Note: Mergesort requires linear extra memory, and copying an array is slow. It is hardly ever used for internal sorting, but is quite useful for external sorting.

7 Quicksort

The fastest known sorting algorithm in practice

7.1 The algorithm

```
void Quicksort ( ElementType A[ ], int N )
{
     if ( N < 2 )  return;
     pivot = pick any element in A[ ];
     Partition S = { A[ ] \ pivot } into two disjoint sets:
        A1={ a 属于 S | a <= pivot } and A2={ a 属于 S | a >= pivot };
     A = Quicksort ( A1, N1 ) U { pivot } U Quicksort ( A2, N2);
}
```



- The best case $T(N) = O(N \log N)$

7.2 Picking the Pivot

☞ **A Wrong Way:** **Pivot = A[0]**

The worst case: **A[ ] is presorted – quicksort will take $O(N^2)$ time to do nothing** ☹

☞ **A Safe Maneuver:** **Pivot = random select from A[ ]**

☹ **random number generation is expensive**

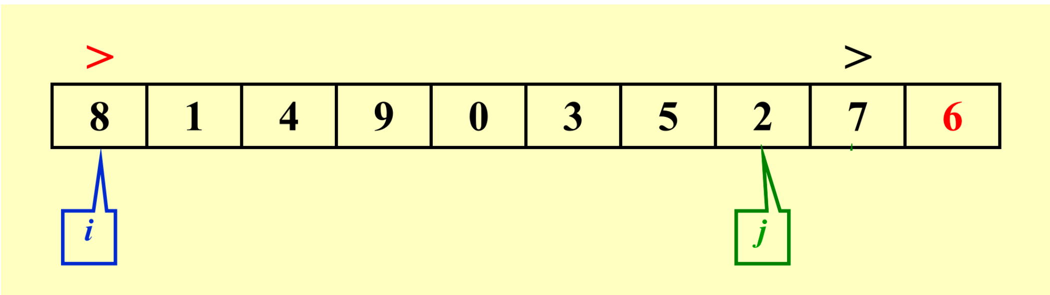
☞ **Median-of-Three Partitioning:**

Pivot = median (left, center, right)

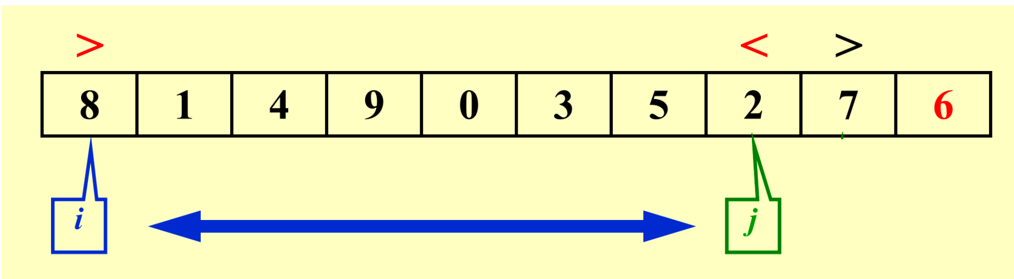
Eliminates the bad case for sorted input and actually reduces the running time by about 5%.

- ✖ 选第一个作为Pivot，其实不好，如果遇到排好了的sorted list，那么就要花费 $O(N^2)$ 的时间
- ✖ 那随机数呢：随机数的产生花时间
- ☑取中位数：取最左边，最右边，中间的中位数

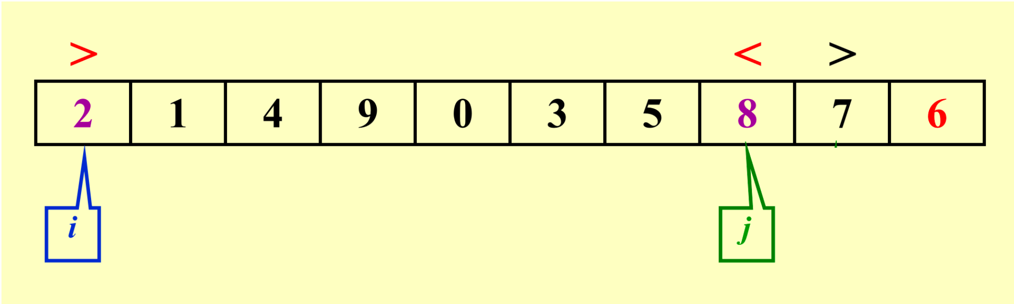
7.3 Partitioning Strategy



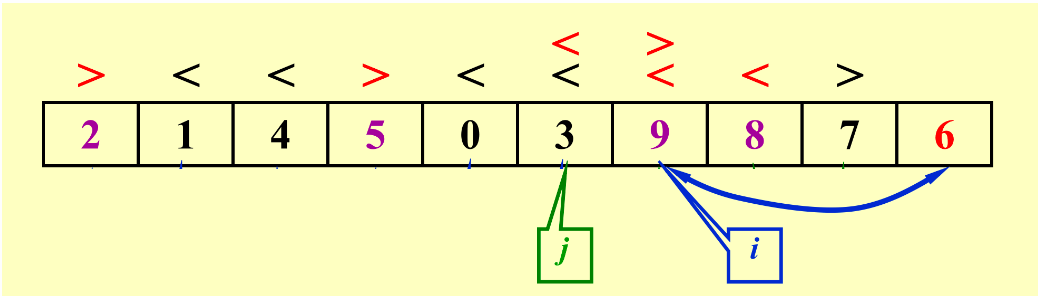
- 先选好pivot，从最左边和最右边（pivot-1的位置）



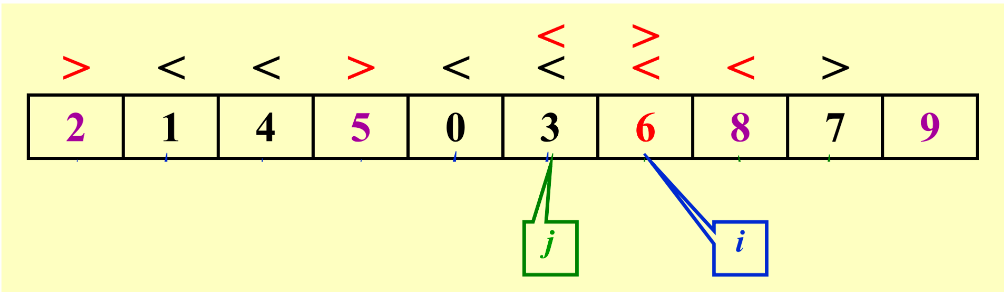
- i指针判断当前值是否小于pivot，如果大于，i停止
- j指针判断当前值是否大于pivot，如果小于，j停止



- 两个指针停止后，交换两者元素



- 继续这样做，这样交换，直到i和j已经交汇



- 此时交换i的元素和Pivot
- 思考一下：也就是i踏足过的地方，都小于pivot
- j踏足过的地方都大于pivot
- 那么就消除了很多逆序对
- 每**run**一次，就有一个位置的数据到了它的**final**位置，可以用此性质判断到了第几次或至多几次

7.4 Small Arrays

Problem: Quicksort is slower than insertion sort for small N (≤ 20).

Solution: Cutoff when N gets small (e.g. $N = 10$) and use other efficient algorithms (such as insertion sort).

7.5 Implementation

考试要求

```
void  Quicksort( ElementType A[ ], int N )
{
  Qsort( A, 0, N - 1 );
  /* A:    the array   */
  /* 0:    Left index  */
  /* N – 1: Right index */
}
```

```
/* Return median of Left, Center, and Right */
/* Order these and hide the pivot */

ElementType Median3( ElementType A[ ], int Left, int Right )
{
    int  Center = ( Left + Right ) / 2;

    if ( A[ Left ] > A[ Center ] )
        Swap( &A[ Left ], &A[ Center ] );

    if ( A[ Left ] > A[ Right ] )
        Swap( &A[ Left ], &A[ Right ] );

    if ( A[ Center ] > A[ Right ] )
        Swap( &A[ Center ], &A[ Right ] );
    // 实际上三个比较，把最大的数换到最右边
    // center 是 中位数

    /* Invariant: A[ Left ] <= A[ Center ] <= A[ Right ] */
    Swap( &A[ Center ], &A[ Right - 1 ] ); /* Hide pivot */
    /* only need to sort A[ Left + 1 ] … A[ Right – 2 ] */

    return  A[ Right - 1 ];  /* Return pivot */
}
```

```c
void  Qsort( ElementType A[ ], int Left, int Right ) // 考到几率很大
{   int  i,  j;
    ElementType  Pivot;
    if ( Left + Cutoff <= Right ) {  /* if the sequence is not too short */
        // 如果太短了的话，直接去Insertion Sort
        Pivot = Median3( A, Left, Right );  /* select pivot */

        i = Left;     j = Right - 1;  /* why not set Left+1 and Right-2? */

        for( ; ; ) {
          // 一次最少能消除2对逆序对

          while ( A[ ++i ] < Pivot ) { }  /* scan from left  , i 指针扫描*/
          while ( A[ --j ] > Pivot ) { }  /* scan from right , j 指针扫描*/

          if ( i < j ) // i >= j 说明已经已经完全scan了
            Swap( &A[ i ], &A[ j ] );  /* adjust partition */
          else  break;  /* partition done */

        }

        Swap( &A[ i ], &A[ Right - 1 ] ); /* restore pivot */
        // Pivot 不需要再排
        Qsort( A, Left, i - 1 );      /* recursively sort left part */
        Qsort( A, i + 1, Right );   /* recursively sort right part */

    }  /* end if - the sequence is long */
    else /* do an insertion sort on the short subarray */
        // 数组长度过短，执行Insertion Sort
        InsertionSort( A + Left, Right - Left + 1 );
}
```

- QSort每次一定能排对一个

7.6 Analysis

$$T(N) = T(i) + T(N-i-1) + cN$$

☞ **The Worst Case:**

$$T(N) = T(N-1) + cN \quad \Rightarrow \quad T(N) = O(N^2)$$

☞ **The Best Case:** [… …] • [… …]

$$T(N) = 2T(N/2) + cN \quad \Rightarrow \quad T(N) = O(N \log N)$$

☞ **The Average Case:**

Assume the average value of $T(i)$ for any i is $\frac{1}{N}\left[\sum_{j=0}^{N-1} T(j)\right]$

$$T(N) = \frac{2}{N}\left[\sum_{j=0}^{N-1} T(j)\right] + cN \quad \Rightarrow \quad T(N) = O(N \log N)$$

> 〖Example〗Given a list of N elements and an integer k. Find the kth largest element.
>
> 堆去做；
>
> QuickSort - Q Select： 判断k和i的关系，就能确定比i大还是比i小
>
> 求中位数可以得到线性复杂度

8 Sorting Large Structures

> *Problem: Swapping large structures can be very much expensive.*
>
> *Solution: Add a pointer field to the structure and swap pointers instead – indirect sorting. Physically rearrange the structures at last if it is really necessary.*

〖Example〗 **Table Sort**

list	[0]	[1]	[2]	[3]	[4]	[5]
key	d	b	f	c	a	e
table	0	1	2	3	4	5

- 首先建一个table，里面代表了Key

〖Example〗 Table Sort

list	[0]	[1]	[2]	[3]	[4]	[5]
key	d	b	f	c	a	e
table	4	1	3	0	5	2

- 然后根据Key对table进行排序：4-a 1-b 3-c 0-d 5-e 2-f

Note: Every permutation is made up of disjoint cycles.

list	[0]	[1]	[2]	[3]	[4]	[5]
key	d	b	f	c	a	e
table	4	1	3	0	5	2

temp = d
current = 0
next = 4

- 但是我还想物理上也排好
- 就采用如上的算法，current代表目前的位置，next代表真正所指数据存储位置

Note: Every permutation is made up of disjoint cycles.

list	[0]	[1]	[2]	[3]	[4]	[5]
key	a	b	f	c	a	e
table	0	1	3	0	5	2

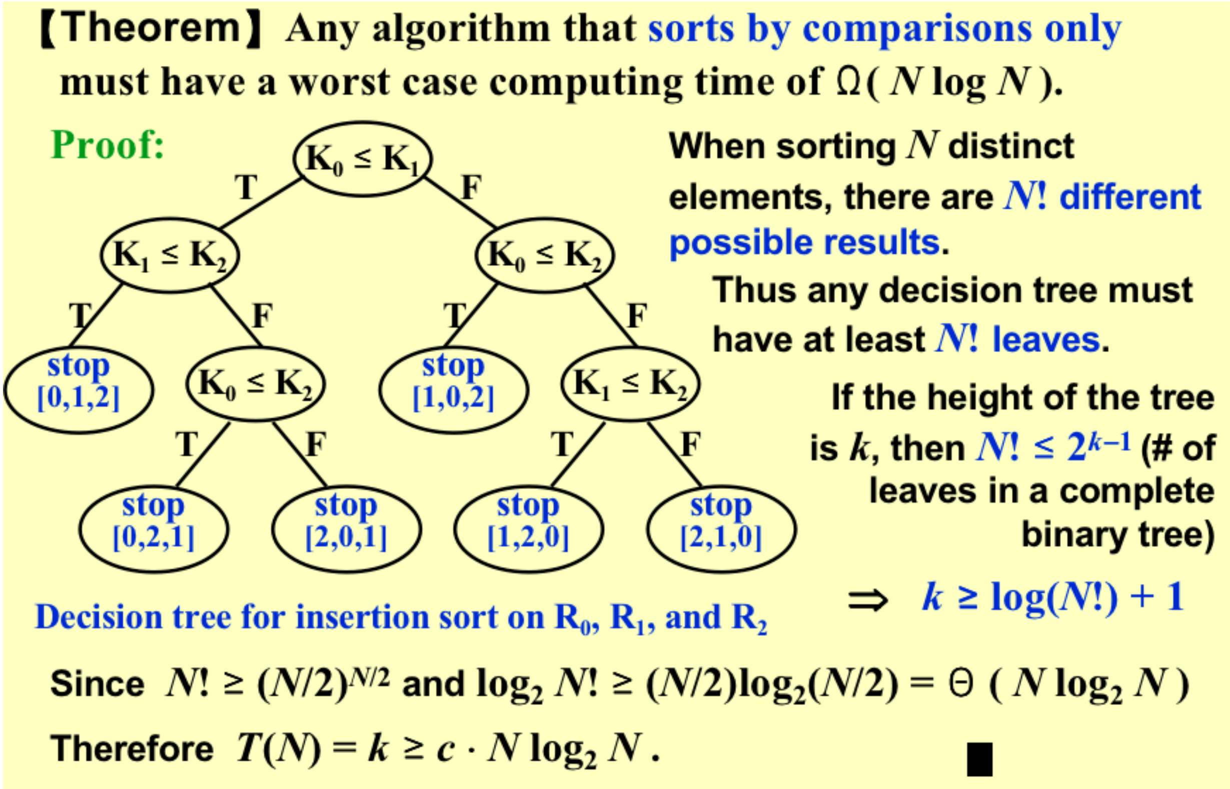
temp = d
current = 4
next = 5

- 找到next = 4代表的list[next] = a，然后换到list[current] = list[0] 的位置
- 然后 current = next，next = list[current]

In the worst case there are $\lfloor N/2 \rfloor$ cycles and requires $\lfloor 3N/2 \rfloor$ record moves.

$T = O(m\, N)$ where m is the size of a structure.

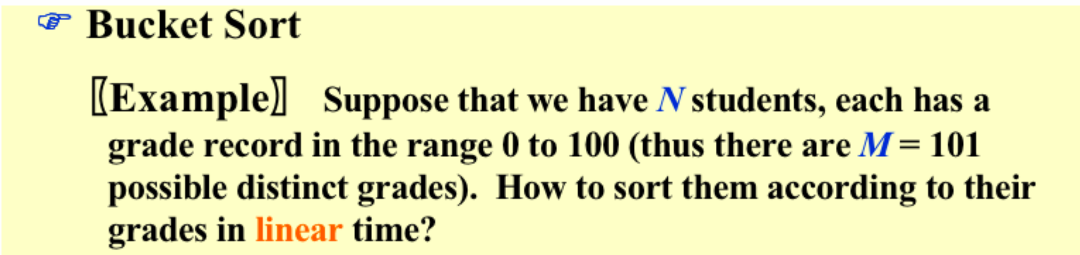
9 A General *Lower Bound* for Sorting



- 非叶节点都在做比较，叶节点是排序结果
- 这么多叶节点，只有一个是对的
- 这里说的是基于比较的算法，最坏的情况下也只需要走 $O(N\log N)$

$$T_{based\ on\ comparison} = \Omega(N\log N) \tag{13}$$

10 Bucket Sort and Radix Sort



- 设一个list（开辟一堆桶），有一个数进来就扔到对应的桶里

```
Algorithm
{
    initialize count[ ];
    while (read in a student's record)
        insert to list count[stdnt.grade];
    for (i=0; i<M; i++) {
        if (count[i])
            output list count[i];
    }
}
```

$$T(N, M) = O(N + M) \tag{14}$$

〖Example〗 Given N = 10 integers in the range 0 to 999 (M = 1000)
Is it possible to sort them in linear time?

What if we sort according to the Most Significant Digit first?

☞ **Radix Sort**

Input: 64, 8, 216, 512, 27, 729, 0, 1, 343, 125

Sort according to the Least Significant Digit first.

Bucket	0	1	2	3	4	5	6	7	8	9
Pass 1	0	1	512	343	64	125	216	27	8	729
Pass 2	0	512	125		343		64			
	1	216	27							
	8		729							
Pass 3	0	125	216	343		512		729		
	1									
	8									
	27									
	64									

T=O(P(N+B)) where P is the number of passes, N is the number of elements to sort, and B is the number of buckets.

Output: 0, 1, 8, 27, 64, 125, 216, 343, 512, 729

- KeyPoint: Sort according to the Least Significant Digit first.
- 第一轮根据个位来装桶
- 第二轮根据十位来装桶
- 第三轮根据百位来装桶

Suppose that the record R_i has r keys.

- K_i^j ::= the j-th key of record R_i
- K_i^0 ::= the **most** significant key of record R_i
- K_i^{r-1} ::= the **least** significant key of record R_i
- A list of records $R_0, \ldots, R_{n-1}$ is **lexically sorted** with respect to the keys $K^0, K^1, \ldots, K^{r-1}$ iff

$$(K_i^0, K_i^1, \cdots, K_i^{r-1}) \le (K_{i+1}^0, K_{i+1}^1, \cdots, K_{i+1}^{r-1}),\ 0 \le i < n-1.$$

That is, $K_i^0 = K_{i+1}^0, \ldots, K_i^l = K_{i+1}^l, K_i^{l+1} < K_{i+1}^{l+1}$ for some $l < r - 1$.

〖Example〗 A deck of cards sorted on 2 keys

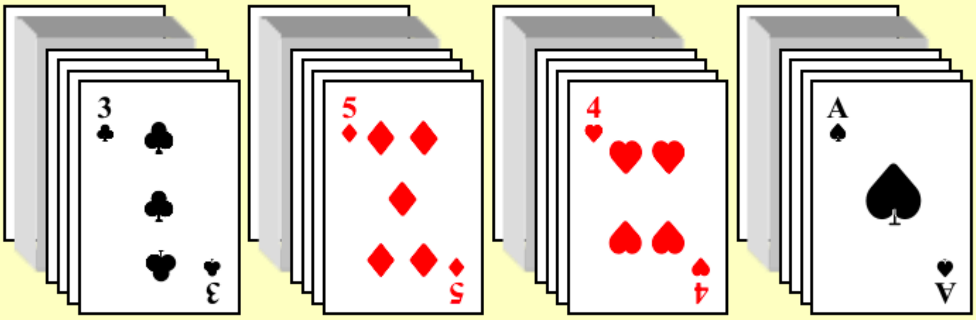
K^0 [Suit] ♣ < ♦ < ♥ < ♠

K^1 [Face value] 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A

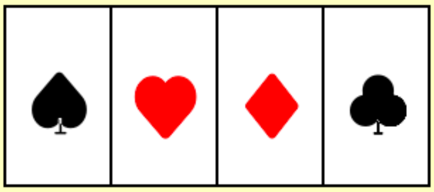
Sorting result : 2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

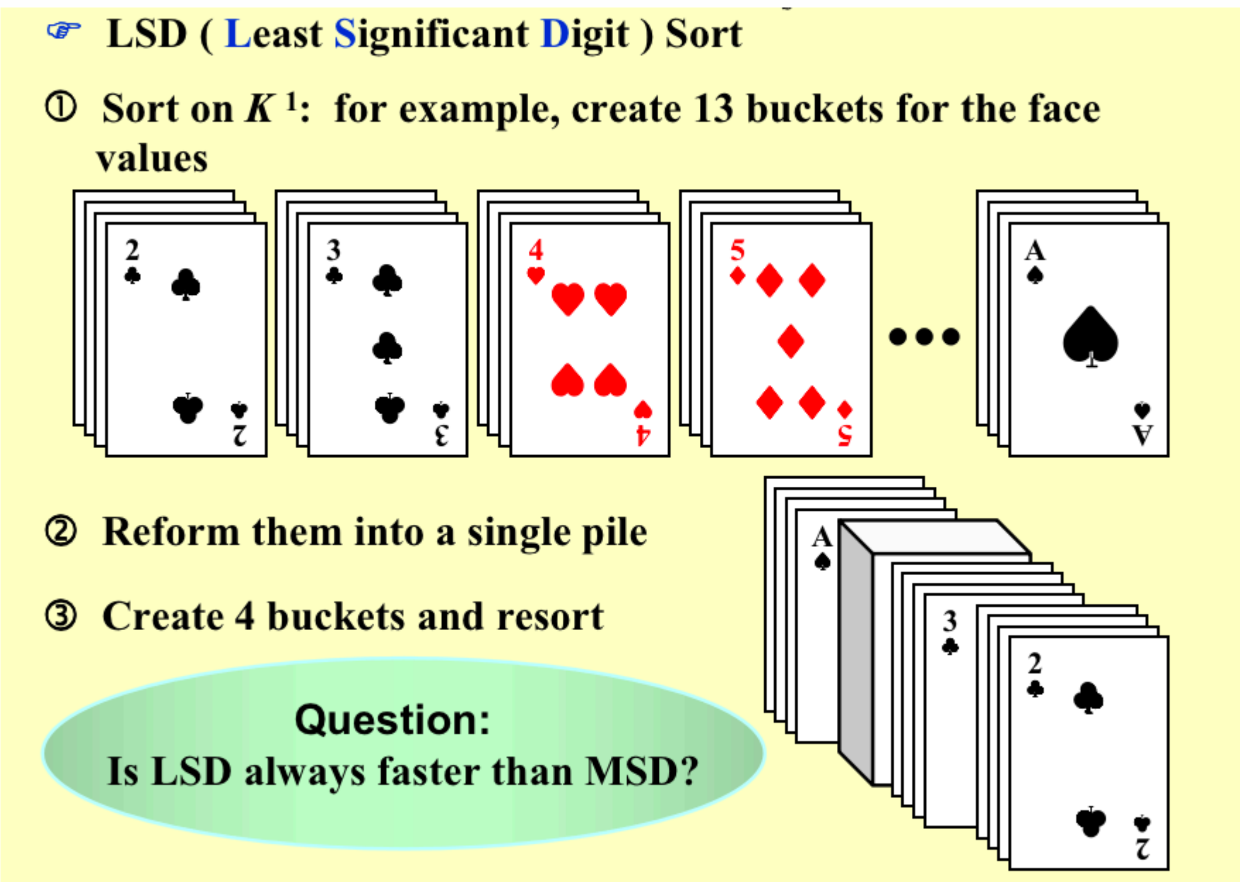
☞ MSD (Most Significant Digit) Sort

① Sort on K^0: for example, create 4 buckets for the suits



② Sort each bucket independently (using any sorting technique)





- LSD实际上执行两次桶排序