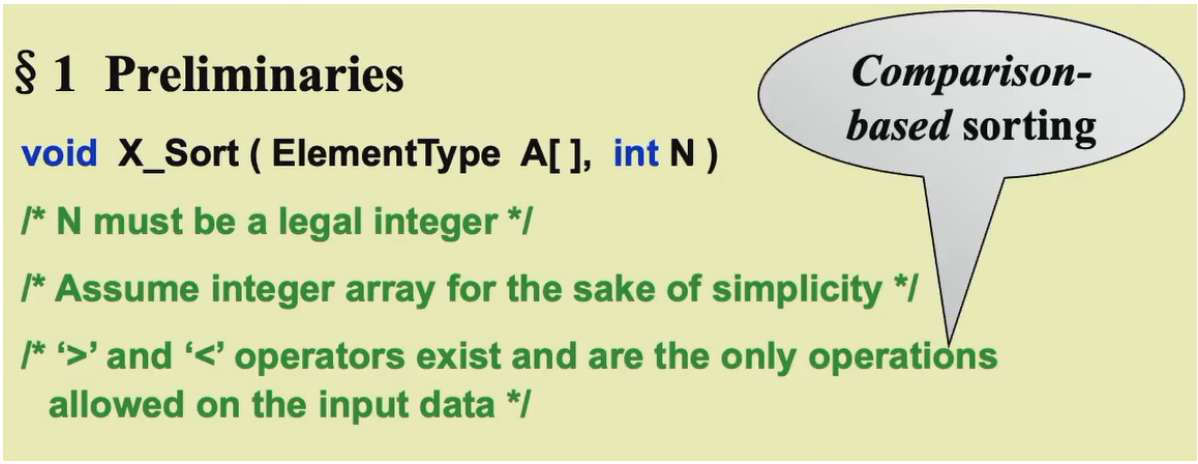
Sorting

Preliminaries



- internal sorting: 可以在内存中加载所有数据执行的排序
- external sorting

Insertion Sort

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

    for ( P = 1; P < N; P++ ) {
        Tmp = A[ P ];  /* the next coming card */
        for ( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
            A[ j ] = A[ j - 1 ];
            /* 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 */
}
```

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

    for ( P = 1; P < N; P++ ) {
        Tmp = A[ P ];  /* the next coming card */
        for ( j = P; j > 0 && A[ j - 1 ] > Tmp; j-- )
            A[ j ] = A[ j - 1 ];
            /* 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 */
}
```

The worst case: Input A[ ] is in reverse order. $T(N) = O(N^2)$

The best case: Input A[ ] is in sorted order. $T(N) = O(N)$

A Lower Bound for Simple Sorting Algorithm

§ 3 A Lower Bound for Simple Sorting Algorithms

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

〖Example〗 Input list 34, 8, 64, 51, 32, 21 has 9 inversions.

(34, 8) (34, 32) (34, 21) (64, 51) (64, 32) (64, 21) (51, 32) (51, 21) (32, 21)

There are 9 swaps needed to sort this list by insertion sort.

Swapping two adjacent elements that are out of place removes exactly one inversion.

> 排序的过程本质就是消去逆序对的过程
>
> 交换相邻的元素一次只能消去一次逆序对

$T(N, I) = O(I + N)$

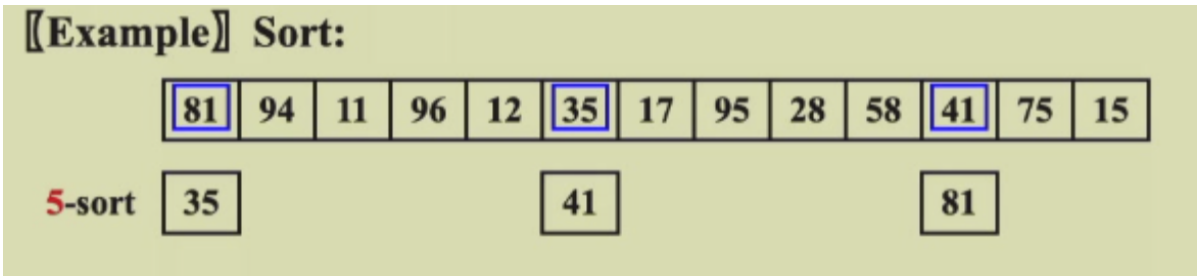
【Theorem】The average number of inversions in an array of N distinct numbers is $N(N-1)/4$.

> Average = (Worst + Best) / 2

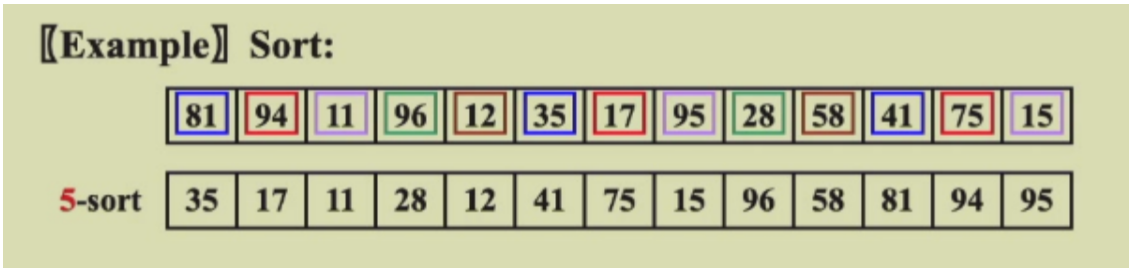
【Theorem】Any algorithm that sorts by exchanging adjacent elements requires $\Omega(N^2)$ time on average.

> Simple sorting algorithm : sort by exchanging adjacent elements

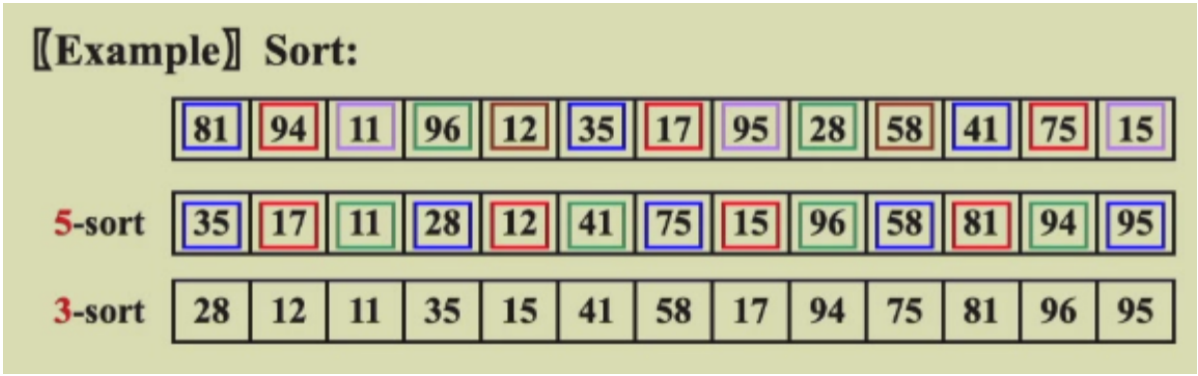
Shellsort



> 5-sort：每隔5个取元素，在这几个元素之间进行比较



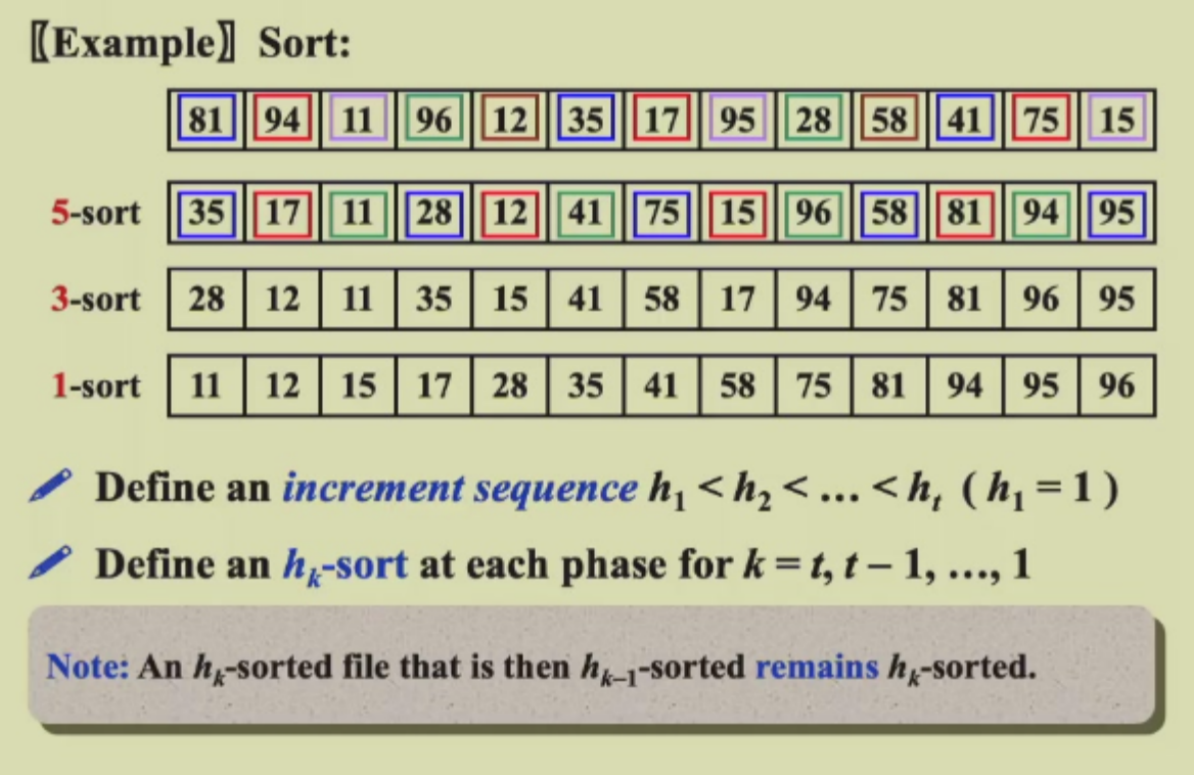
第一轮结束后，进行3-sort



第二轮结束后，进行1-sort

> 本质上是多轮的Insertion sort

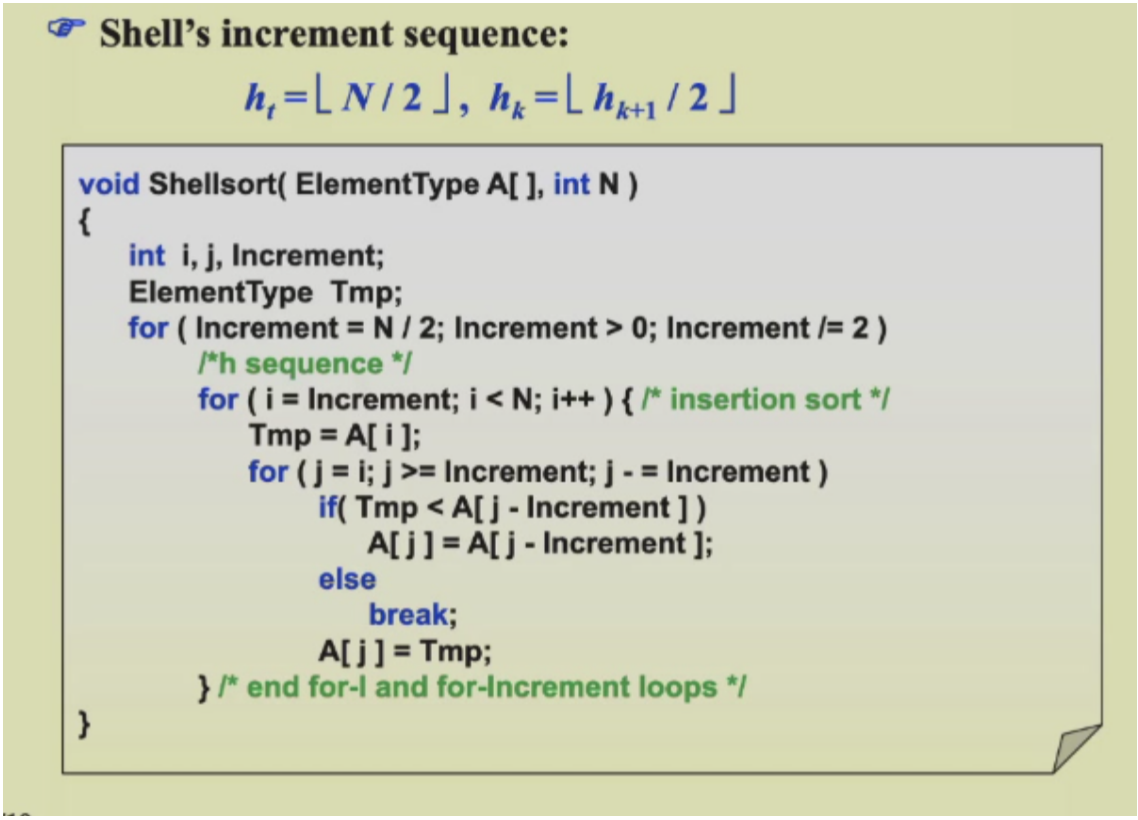
> 最后一轮一定是1-sort，就是Insertion sort，保证结果的正确性



> key：sequence的设计

Sequence的设计

- Shell's Increment Sequence



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.

> worst case: 前面的几次排序啥都没干，原因是存在公约数，浪费了很多次比较
>
> 但使用素数也不太行，每次的增量太小了 没啥效果

- Hibbard's Increment Sequence

$h_k = 2^k - 1$ ---- consecutive increments have no common factors.

【Theorem】The worst-case running time of Shellsort, using Hibbard's increments, is $\Theta(N^{3/2})$.

$$T_{avg-Hibbard}(N) = O(N^{5/4})$$

- Sedgewick's best sequence

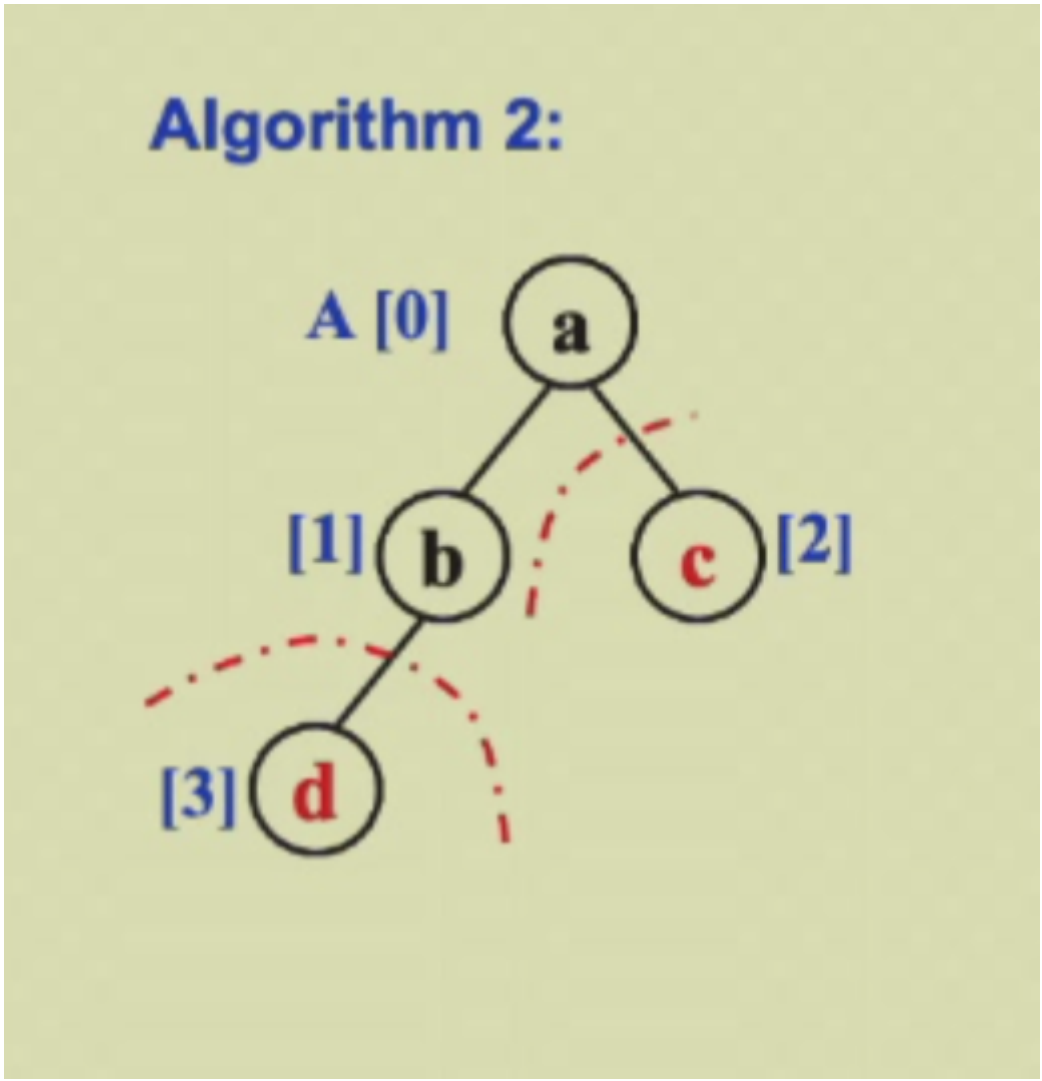
Sedgewick's best sequence is {1, 5, 19, 41, 109, … } in which the terms are either of the form $9\times4^i - 9\times2^i + 1$ or $4^i - 3\times2^i + 1$. $T_{avg}(N) = O(N^{7/6})$ and $T_{worst}(N) = O(N^{4/3})$.

Heapsort

```
Algorithm 1:
{
    BuildHeap( H );
    for ( i=0; i<N; i++ )
        TmpH[ i ] = DeleteMin( H );
    for ( i=0; i<N; i++ )
        H[ i ] = TmpH[ i ];
}
```

> - heapsort 保证$O(N \log N)$
> - 算法1空间复杂度太高了

- 算法二：使用MaxHeap



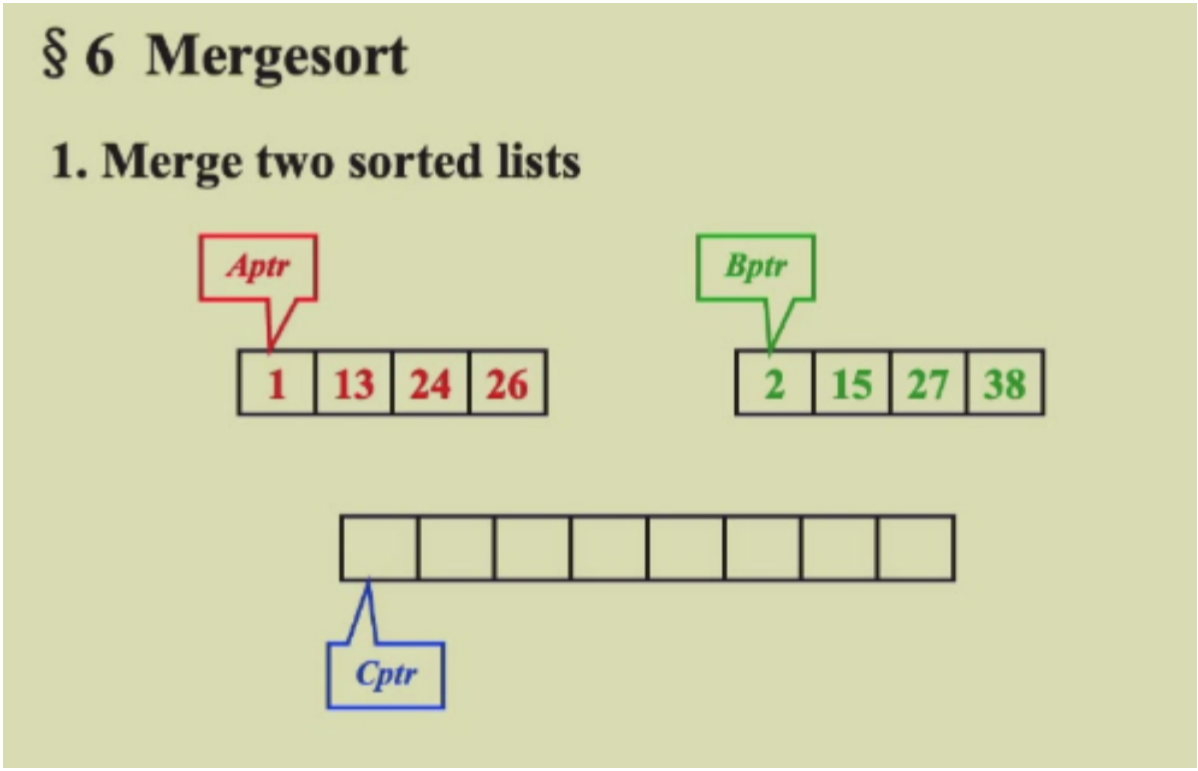
DeleteMax并把弄出来的元素塞到后面去

```
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 );
   }
}
```

> Time Complexity: $O(N \log N)$
>
> 【Theorem】The average number of comparisons used to heapsort a random permutation of N distinct items is $2N \log N - O(N \log \log N)$.
>
> Note: Although Heapsort gives the best average time, in practice it is slower than a version of Shellsort that uses Sedgewick's increment sequence.
>
> Heapsort比较少用，非全排序可以考虑Heapsort

Mergesort

external sort



时间复杂度：$O(N)$

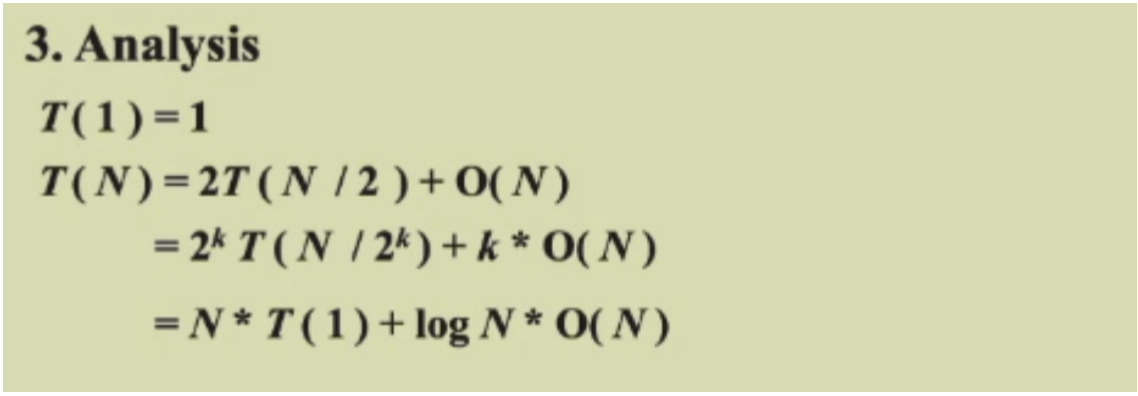
```
void MSort( ElementType A[ ], ElementType TmpArray[ ],
                    int Left, int Right )
{  int  Center;
   if ( Left < Right ) {  /* if there are elements to be sorted */
           Center = ( Left + Right ) / 2;
           MSort( A, TmpArray, Left, Center );          /* T( N / 2 ) */
 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

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 );
   }
   else  FatalError( "No space for tmp array!!!" );
}
```

> 在外部申请TmpArray: $O(N)$
>
> 每次Merge都申请TmpArray：$O(N \log N)$

```
/* 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 */
      if ( A[ Lpos ] <= A[ Rpos ] )
          TmpArray[ TmpPos++ ] = A[ Lpos++ ];
      else
          TmpArray[ TmpPos++ ] = A[ Rpos++ ];
   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 ];
}
```

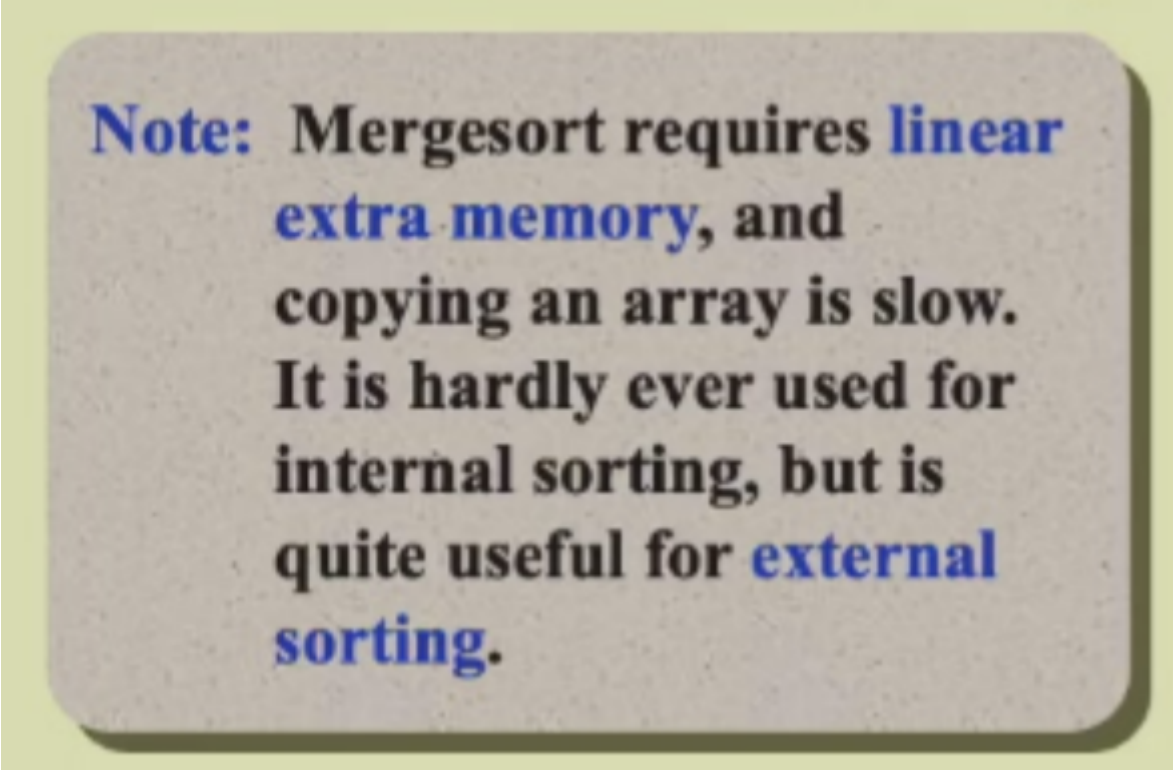
- Analysis



时间复杂度：$O(N + N \log N)$

- Iterative Version:

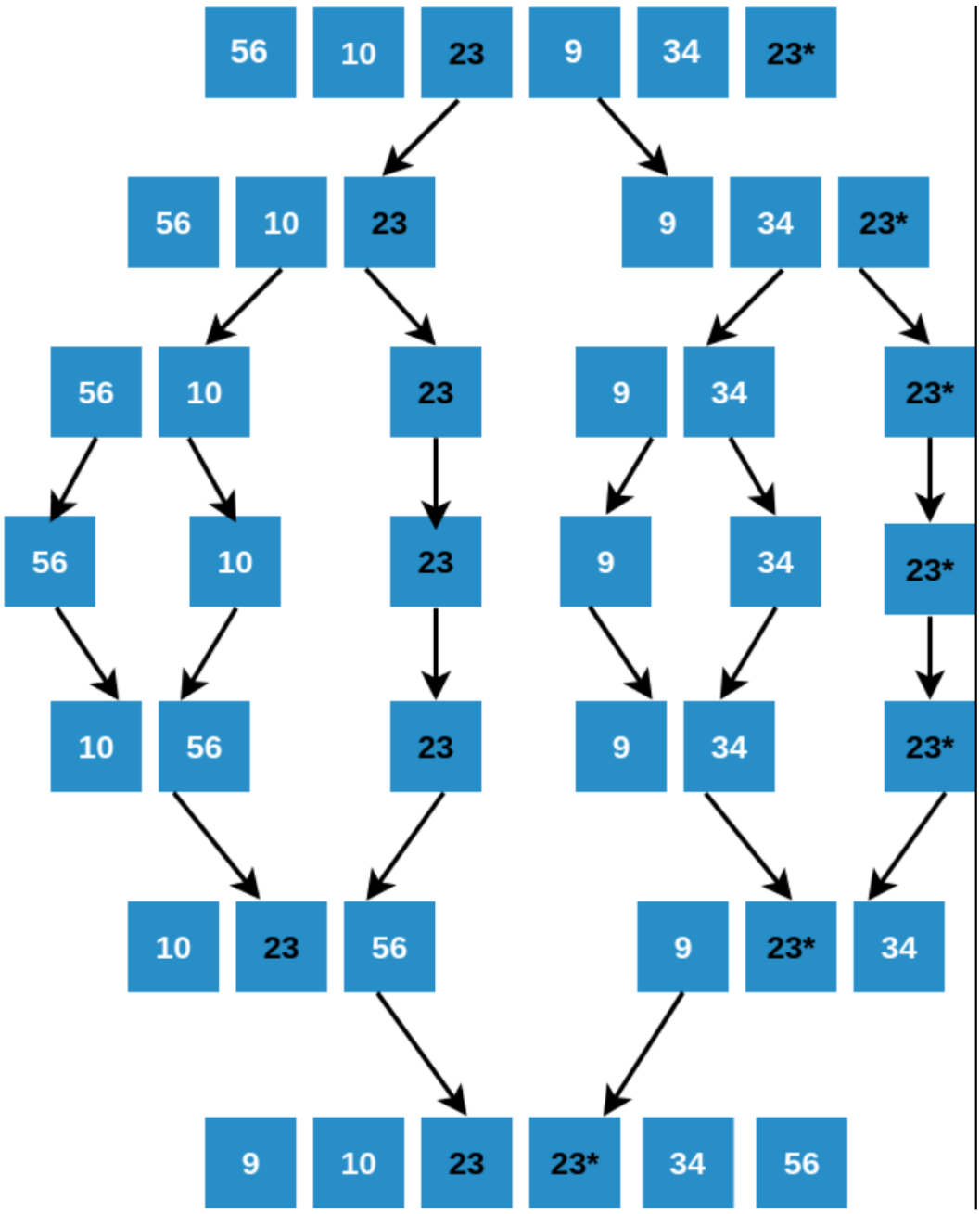
  Conquer不变，Divide变成for循环，去计算下标



Stable

是否会改变相同key的元素的相对位置，不改变就是stable

- Insertion sort是stable的，他不改变已经排好序的元素的相对位置
- Shellsort是unstable的，因为在前期排序的时候可能会调换相同key元素的相对位置
- Mergesort是stable的



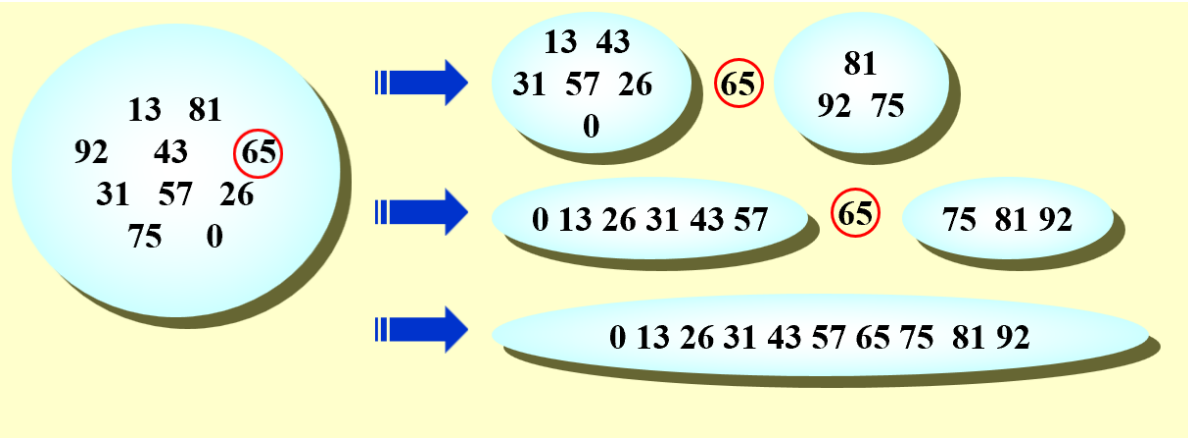
- Heapsort是unstable的，在每次调整树结构的时候难以保证他们两个的相对顺序

QuickSort

Algorithm

已知最快的算法

```
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∈S | a ≤ pivot } and A2={ a∈S | a ≥ pivot };
    A = Quicksort ( A1, N1) ∪ { pivot } ∪ Quicksort ( A2, N2);
}
```



1. 选个支点
2. 分成三部分：小于支点的集合，支点，大于支点的集合
3. 对左右两部分进行qsort

- 一次至少排好一个数：pivot

选择pivot

- worst case

> ☞ A Wrong Way: Pivot = A[0]
>
> The worst case: A[ ] is presorted – quicksort will take O(N^2) time to do nothing ☹

- 随机数：不现实

> ☞ A Safe Maneuver: Pivot = random select from A[ ]
>
> ☹ random number generation is expensive

- 用头中尾的中位数去模拟总体的中位数

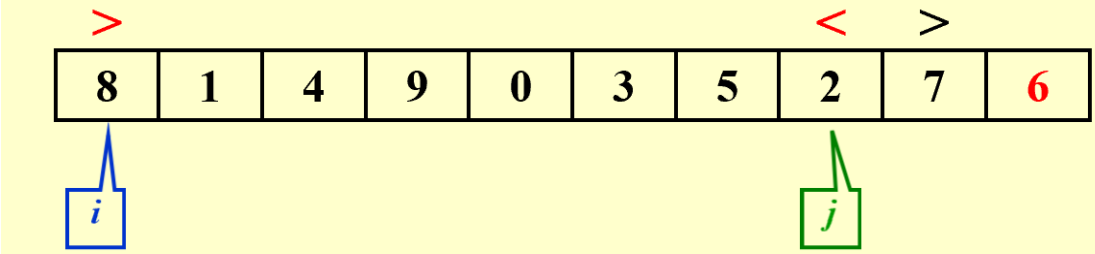
☞ **Median-of-Three Partitioning:**

Pivot = median (left, center, right)

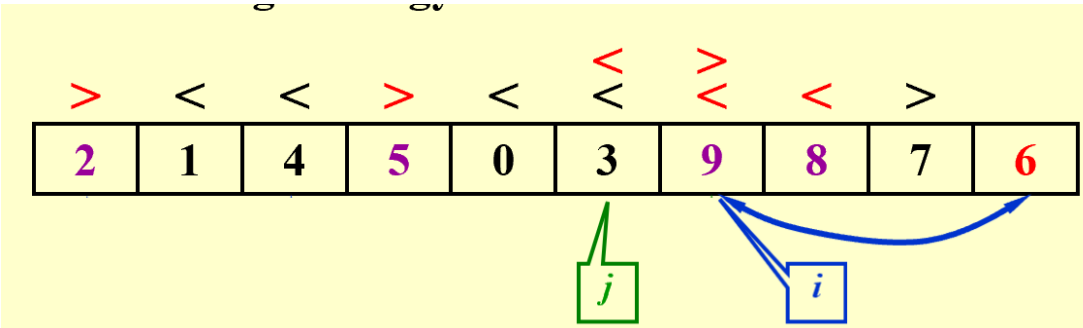
Eliminates the bad case for sorted input and actually reduces the running time by about 5%.

进行partition

1. pivot放在末尾
2. 使用头尾两个指针，分别向前向后遍历，直至都找到不符合要求的元素，然后进行交换



3. 直到j小于i的时候停下，不发生交换，将末尾的pivot换到正确的位置



- 当key=pivot时，需要停下，即使会进行不必要的swap

对于小输入

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).**

当序列长度小到一定程度时，进行普通的算法替代Qsort

Implementation

```
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 ] );
   /* 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 */
}
```

```
void  Qsort( ElementType A[ ], int Left, int Right )
{  int  i,  j;
   ElementType  Pivot;
   if ( Left + Cutoff <= Right ) {  /* if the sequence is not too short */
      Pivot = Median3( A, Left, Right );  /* select pivot */
      i = Left;     j = Right – 1;  /* why not set Left+1 and Right-2? */
      for( ; ; ) {
            while ( A[ + +i ] < Pivot ) { }  /* scan from left */
            while ( A[ – –j ] > Pivot ) { }  /* scan from right */
            if ( i < j )
              Swap( &A[ i ], &A[ j ] );  /* adjust partition */
            else     break;  /* partition done */
      }
      Swap( &A[ i ], &A[ Right - 1 ] ); /* restore pivot */
      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 */
      InsertionSort( A + Left, Right - Left + 1 );
}
```

Analysis

§ 7

6. Analysis

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

☞ **The Worst Case:**

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

☞ **The Best Case:** [... ...] • [... ...]

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

☞ **The Average Case:**

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

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

〖Example〗 Given a list of N elements and an integer k. Find the kth largest element.

使用Qsort的算法，在递归的时候比较i和k，只排序k所在的那一个序列

> Best case: $T(N) = T(N/2) + cN,\ O(N)$

Table Sort

交换大结构体代价太大，改为交换pointer

〖Example〗 Table Sort

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

The sorted list is

list [table[0]], list [table[1]], ……, list [table[n–1]]

这样使得下标构成一个环，能够更好地在对pointer排序后进行对大结构体的排序

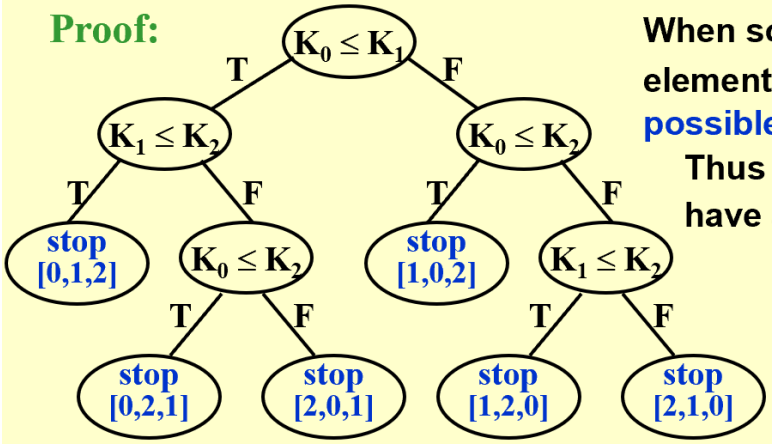
In the worst case there are $\lfloor N/2 \rfloor$ cycles and requires $\lfloor 3N/2 \rfloor$ record moves.

$T = O(mN)$ where m is the size of a structure.

A General Lower Bound for Sorting

【Theorem】Any algorithm that sorts by comparisons only must have a worst case computing time of $\Omega(N \log N)$.

Proof:



Decision tree for insertion sort on R_0, R_1, and R_2

When sorting N distinct elements, there are $N!$ different possible results.

Thus any decision tree must have at least $N!$ leaves.

If the height of the tree is k, then $N! \le 2^{k-1}$ (# of leaves in a complete binary tree)

$\Rightarrow\ k \ge \log(N!) + 1$

Since $N! \ge (N/2)^{N/2}$ and $\log_2 N! \ge (N/2)\log_2(N/2) = \Theta(N \log_2 N)$

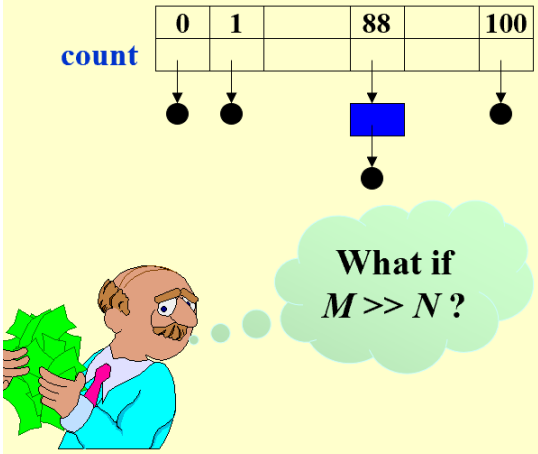
Therefore $T(N) = k \ge c \cdot N \log_2 N$. ■

Bucket Sort & Radix Sort

桶之间的排序关系已知，将element依次塞入对应的桶里。

Bucket Sort

〖Example〗 Suppose that we have N students, each has a grade record in the range 0 to 100 (thus there are $M = 101$ possible distinct grades). How to sort them according to their grades in linear time?




```
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];
   }
}
```

$T(N, M) = O(M+N)$

> 当M>>N时，时间复杂度对N来说就不太行

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

每一个run的输出作为下一个run的输入，这样保证了每一次run桶内的排序是符合上一个run的排序结果的

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 $R_0, ..., R_{n-1}$ is lexically sorted with respect to the keys $K^0, K^1, ..., K^{r-1}$ iff

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

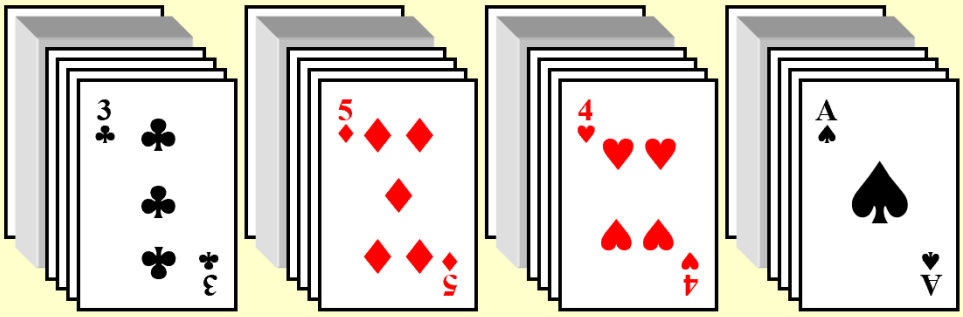
That is, $K_i^0 = K_{i+1}^0$, ... , $K_i^l = K_{i+1}^l$, $K_i^{l+1} < K_{i+1}^{l+1}$ for some $l < r-1$.

〖Example〗 A deck of cards sorted on 2 keys

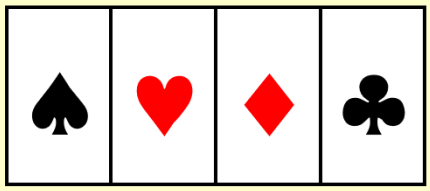
K^0 [Suit]	♣ < ♦ < ♥ < ♠
K^1 [Face value]	2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < J < Q < K < A
Sorting result :	2♣ ... A♣ 2♦ ... A♦ 2♥ ... A♥ 2♠ ... A♠

☞ MSD (Most Significant Digit) Sort

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

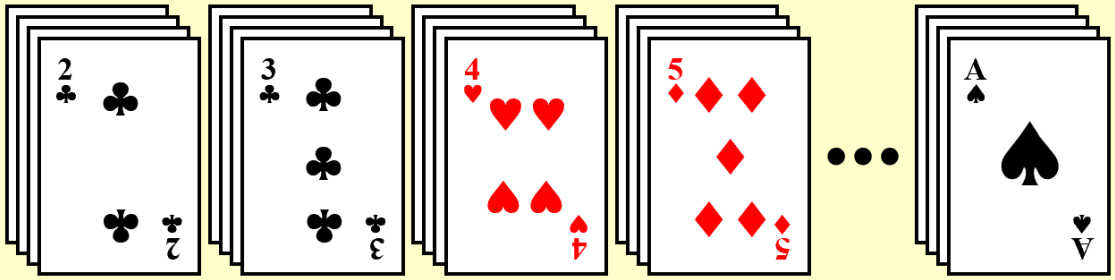


② Sort each bucket independently (using any sorting technique)



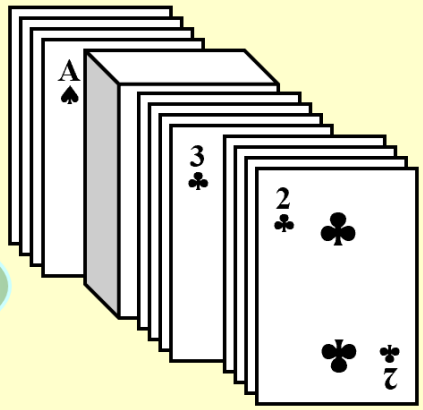
☞ **LSD (Least Significant Digit) Sort**

① **Sort on K^1: for example, create 13 buckets for the face values**



② **Reform them into a single pile**

③ **Create 4 buckets and resort**



Question:
Is LSD always faster than MSD?