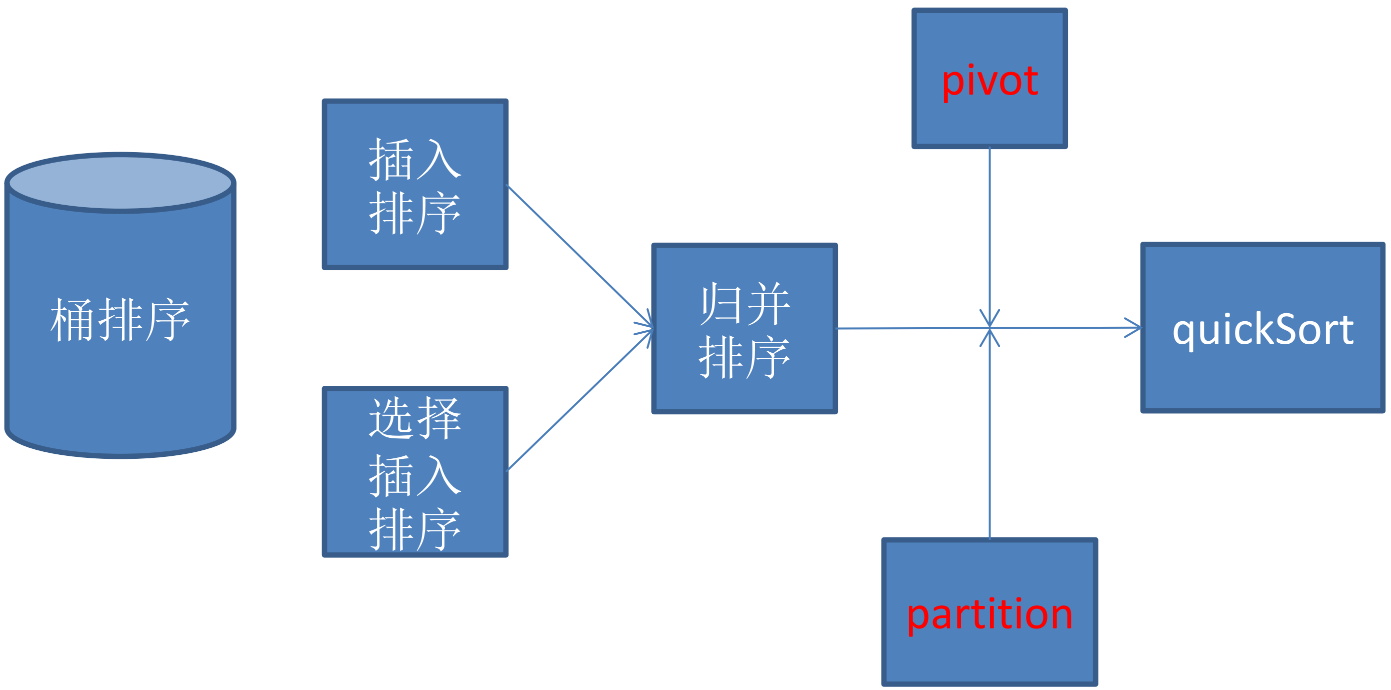
序列和集合的算法II

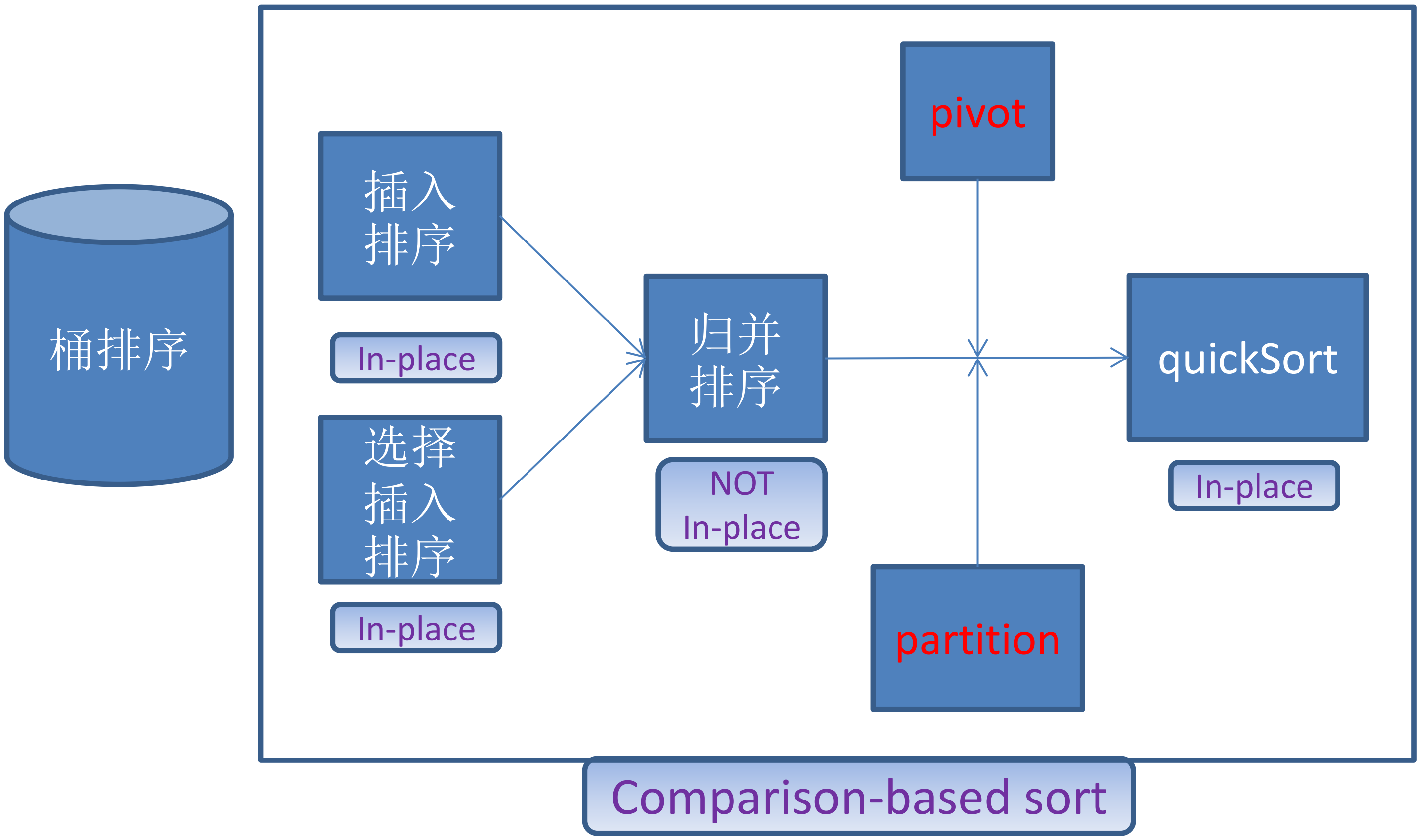
Instructor: Shizhe Zhou

Course Code:00125401

排序算法



排序算法


桶排序
插入排序
In-place
选择插入排序
In-place
pivot
归并排序
NOT In-place
quickSort
In-place
partition
Comparison-based sort

桶排序和基数排序

Bucket and Radix Sort

算法 ***Straight_Radix*** (X, n, k)

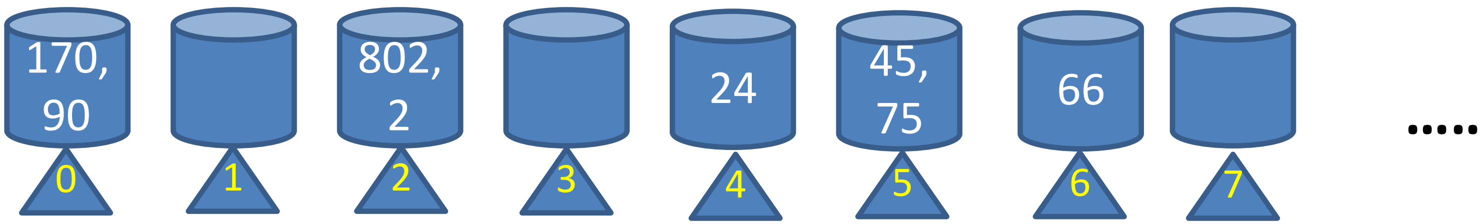
输入： X（元素下标从 1 至 n 的整数数组，每个元素有 k 位）

输出： X（排序后的数组）

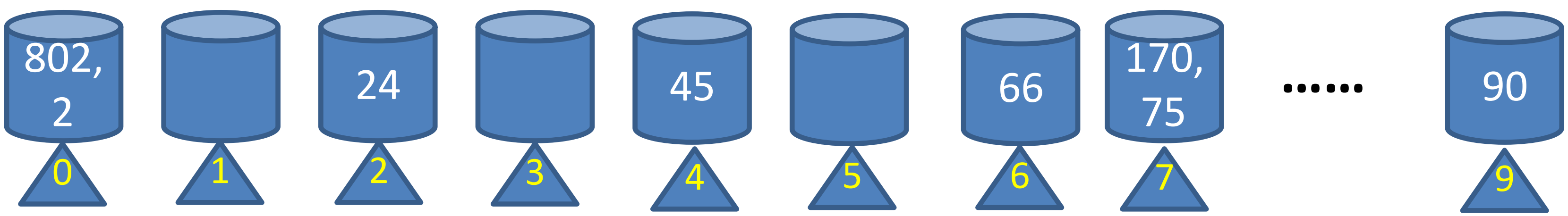
```
begin
    We assume that all elements are initially in a global queue GQ;
    {为简单起见，这里使用 GQ；当然也可以通过 X 本身来实现}
    for i := 1 to d do
    {d 是可能的数制，比如对于十进制，d = 10}
        Initialize queue Q[I] to be empty;
    for i := k downto 1 do                    ← 降序
        while GQ is not empty do              ┐
            pop x from GQ ;                   │
            d := the ith digit of x ;         ├ 分入桶
            insert x into Q[d] ;              ┘
        for t := 1 to d do                    ┐
            insert Q[t] into GQ ;             ┘ 收集
    for i := 1 to n do
        pop X[i] from GQ
end
```

图 6.6　算法 *Straight_Radix*

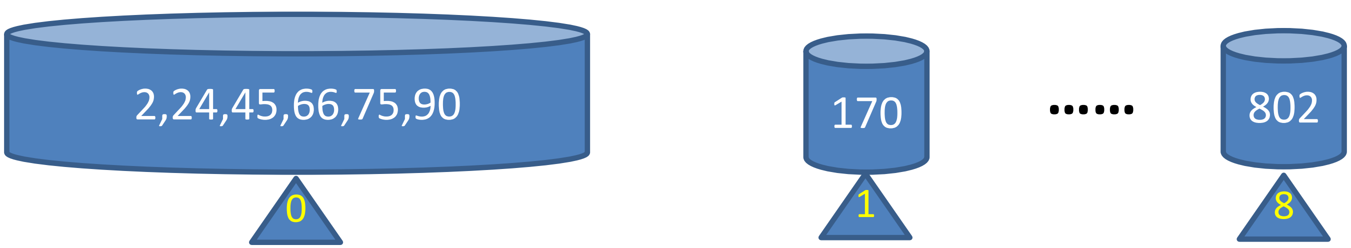
170, 45, 75, 90, 802, 2, 24, 66



170, 90, 802, 2, 24, 45, 75, 66

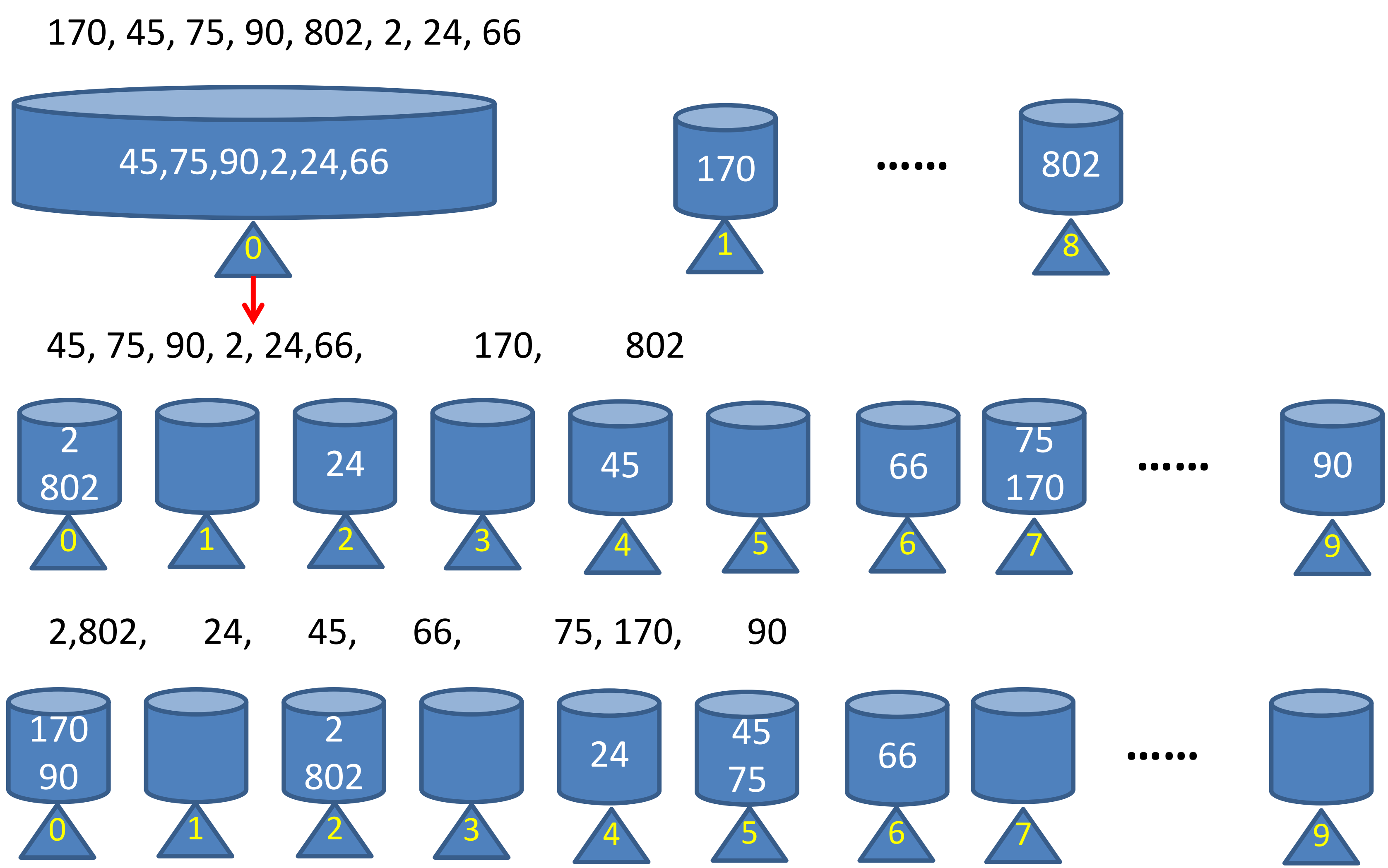


802, 2, 24, 45, 66, 170, 75, 90

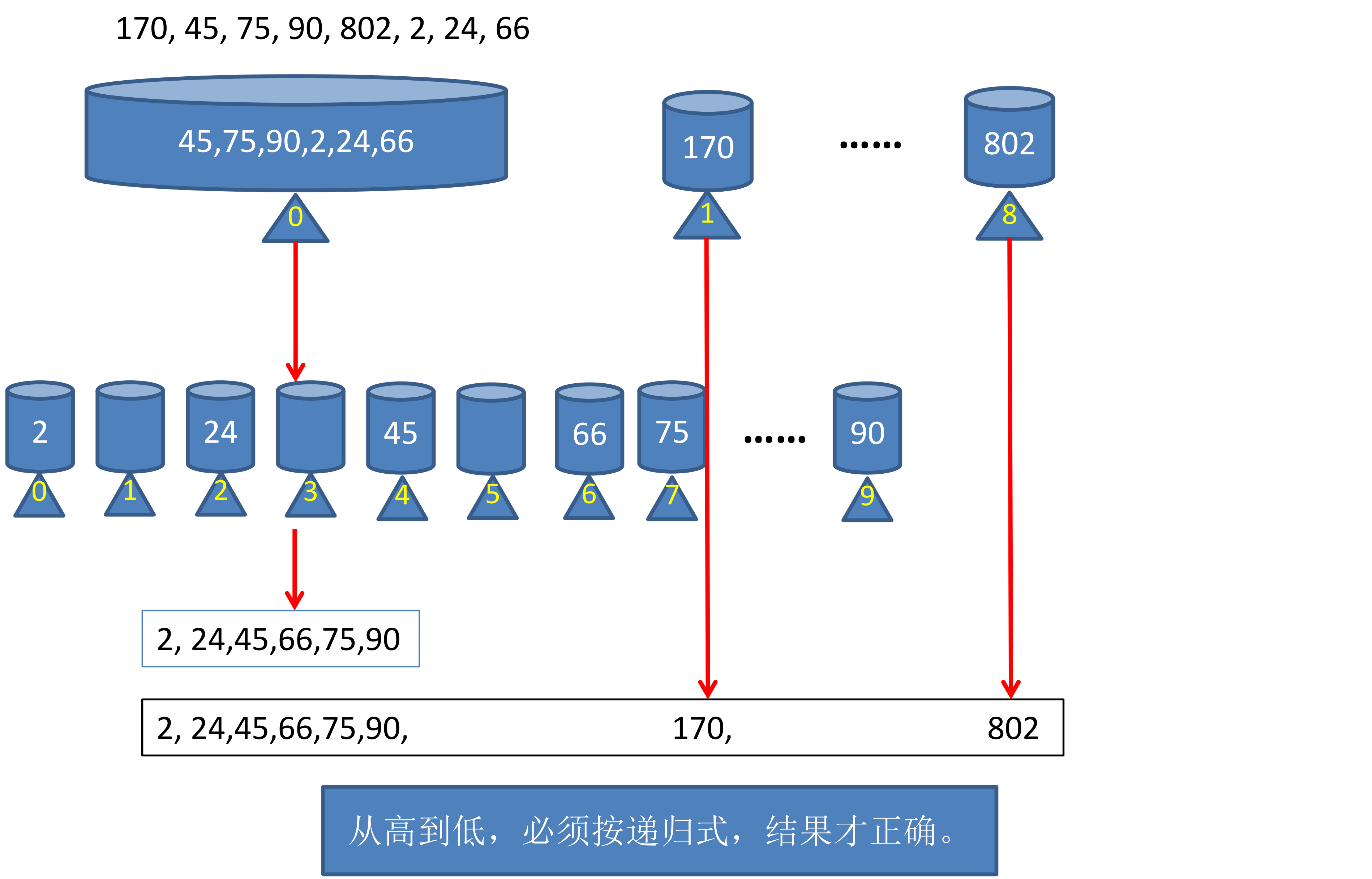


2, 24, 45, 66, 75, 90, 170, 802

对最高位相同的任意两个元素，在最后一步之前，他们已经按正确顺序排列了。



从高到低，如果直接按上一页slides的方法做，结果肯定是错的！必须要按递归做，见下一页,.


170, 45, 75, 90, 802, 2, 24, 66
45,75,90,2,24,66
0
170
1
......
802
8
2
0
1
24
2
3
45
4
5
66
6
75
7
......
90
9
2, 24,45,66,75,90
2, 24,45,66,75,90,
170,
802
从高到低，必须按递归式，结果才正确。

Analysis

- 重收集桶的循环执行K次，K是最大数的位数(e.g.10进制下)
- 需要全局队列GQ 和 分桶队列Q[0]...Q[K-1].
- O((n+n)k) = O(nk);

缺点：依赖于数据的结构性.

插入排序
insertion sort

- example

6 5 3 1 8 7 2 4

[Pseudocode](#)

```
for i ← 1 to length(A)
    j ← i
    while j > 0 and A[j-1] > A[j]
        swap A[j] and A[j-1]
        j ← j - 1
```

插入排序的一个改进
Shell 排序

- example

[Donald Shell 1959]

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}
input data:	62	83	18	53	07	17	95	86	47	69	25	28
after 5-sorting:	17	28	18	47	07	25	83	86	53	69	62	95
after 3-sorting:	17	07	18	47	28	25	69	62	53	83	86	95
after 1-sorting:	07	17	18	25	28	47	53	62	69	83	86	95

The running time of Shellsort is heavily dependent on the gap sequence it uses.
Best interval? open questions!

Shellsort is unstable(*changing relative order of equal elements*)
It is an adaptive sorting algorithm in that it executes faster when the input is partially sorted

the subarrays that Shellsort operates on are initially short; later they are longer but almost ordered:
In BOTH cases insertion sort works efficiently.

选择排序

selection sort

- example

64 25 12 22 11

11 25 12 22 64

11 12 25 22 64

11 12 22 25 64

11 12 22 25 64

基本思想:
每次都从右边剩下的n-k个数中选最小(大)的插入到第k个位置处.

输入list被分为两部分，前一部分总是排好序的sublist.

将64换成24,则最后一步也发生交换

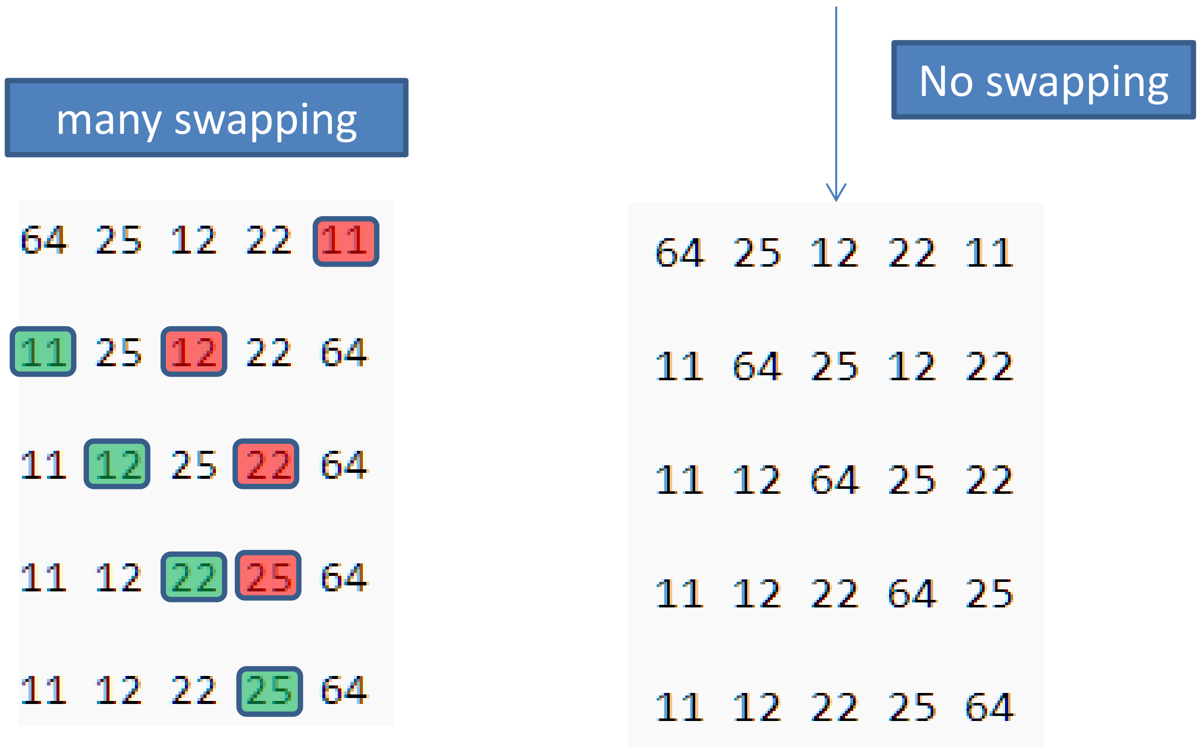
```
/* a[0] to a[n-1] is the array to sort */
int i,j;
int iMin;

/* advance the position through the entire array */
/*   (could do j < n-1 because single element is also min element) */
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */

    /* assume the min is the first element */
    iMin = j;
    /* test against elements after j to find the smallest */
    for ( i = j+1; i < n; i++) {
        /* if this element is less, then it is the new minimum */
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;
        }
    }

    /* iMin is the index of the minimum element. Swap it with the current position */
    if ( iMin != j ) {
        swap(a[j], a[iMin]);
    }
}
```

Another example

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

选择排序

- 如果输入数据结构是(linked)list,性能更好.



performance

- Finally, selection sort is greatly outperformed on larger arrays by Θ(n log n) divide-and-conquer algorithms such as mergesort.
- However, insertion sort or selection sort are both typically faster for small arrays (i.e. fewer than 10–20 elements).
- A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" sublists.

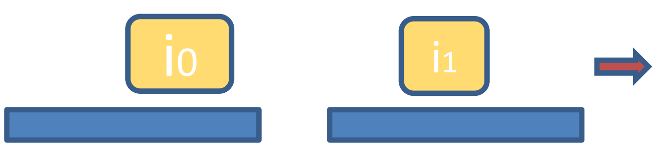
归并排序 merge sort

Top-Bottom

6 5 3 1 8 7 2 4

这是自顶向下递归，因此会分裂到长度为1的sublist再开始merge!

Major drawback:
Need a B for the merged list!



主函数

Merge into B

```
TopDownMergeSort(A[], B[], n)
{
    TopDownSplitMerge(A, 0, n, B);
}
CopyArray(B[], iBegin, iEnd, A[])
{
    for(k = iBegin; k < iEnd; k++)
        A[k] = B[k];
}
// iBegin is inclusive; iEnd is exclusive (A[iEnd] is not in the set)
TopDownSplitMerge(A[], iBegin, iEnd, B[])
{
    if(iEnd - iBegin < 2)                       // if run size == 1
        return;                                 //   consider it sorted
    // recursively split runs into two halves until run size == 1,
    // then merge them and return back up the call chain
    iMiddle = (iEnd + iBegin) / 2;              // iMiddle = mid point
    TopDownSplitMerge(A, iBegin,  iMiddle, B);  // split / merge left  half
    TopDownSplitMerge(A, iMiddle,    iEnd, B);  // split / merge right half
    TopDownMerge(A, iBegin, iMiddle, iEnd, B);  // merge the two half runs
    CopyArray(B, iBegin, iEnd, A);              // copy the merged runs back to A
}

//  left half is A[iBegin :iMiddle-1]
// right half is A[iMiddle:iEnd-1   ]
TopDownMerge(A[], iBegin, iMiddle, iEnd, B[])
{
    i0 = iBegin, i1 = iMiddle;

    // While there are elements in the left or right runs
    for (j = iBegin; j < iEnd; j++) {
        // If left run head exists and is <= existing right run head.
        if (i0 < iMiddle && (i1 >= iEnd || A[i0] <= A[i1]))
            B[j] = A[i0];
            i0 = i0 + 1;
        else
            B[j] = A[i1];
            i1 = i1 + 1;    }

}
```

Merge sort

- Bottom-Top

bottom up merge sort algorithm :
1 treats the list as an array of *n* sublists (called *runs* in this example) of size 1,
2 .iteratively merges sub-lists back and forth between two buffers.

主函数

```
/* array A[] has the items to sort; array B[] is a work array */
BottomUpSort(int n, int A[], int B[])
{
  int width;

  /* Each 1-element run in A is already "sorted". */

  /* Make successively longer sorted runs of length 2, 4, 8, 16... until whole array is sorted. */
  for (width = 1; width < n; width = 2 * width)
    {
      int i;

      /* Array A is full of runs of length width. */
      for (i = 0; i < n; i = i + 2 * width)
        {
          /* Merge two runs: A[i:i+width-1] and A[i+width:i+2*width-1] to B[] */
          /* or copy A[i:n-1] to B[] ( if(i+width >= n) ) */
          BottomUpMerge(A, i, min(i+width, n), min(i+2*width, n), B);
        }

      /* Now work array B is full of runs of length 2*width. */
      /* Copy array B to array A for next iteration. */
      /* A more efficient implementation would swap the roles of A and B */
      CopyArray(A, B, n);
      /* Now array A is full of runs of length 2*width. */
    }
}
BottomUpMerge(int A[], int iLeft, int iRight, int iEnd, int B[])
{
  int i0 = iLeft;
  int i1 = iRight;
  int j;

  /* While there are elements in the left or right lists */
  for (j = iLeft; j < iEnd; j++)
    {
      /* If left list head exists and is <= existing right list head */
      if (i0 < iRight && (i1 >= iEnd || A[i0] <= A[i1]))
        {
          B[j] = A[i0];
          i0 = i0 + 1;
        }
      else
        {
          B[j] = A[i1];
          i1 = i1 + 1;
        }
    }
}
```

Merge Sort的一个改进

- Natural merge sort

```
Start       : 3--4--2--1--7--5--8--9--0--6
Select runs : 3--4  2  1--7  5--8--9  0--6
Merge       : 2--3--4  1--5--7--8--9  0--6
Merge       : 1--2--3--4--5--7--8--9  0--6
Merge       : 0--1--2--3--4--5--6--7--8--9
```

any naturally occurring runs (sorted sequences) in the input are exploited!

Adavantage: 不需要像标准merge sort那样非得pass lbn 次.

MergeSort Performance

Class	Sorting algorithm
Data structure	Array
Worst case performance	O(*n* log *n*)
Best case performance	O(*n* log *n*) typical, O(*n*) natural variant
Average case performance	O(*n* log *n*)
Worst case space complexity	O(*n*) auxiliary

Quick Sort

- Best sorting algo ! Proposed by C.A.R Hoare in 1962
- Divide-&-Conquer algo
- In place (as insertion sort), comparison based
- Very practical (need a bit tuning though)

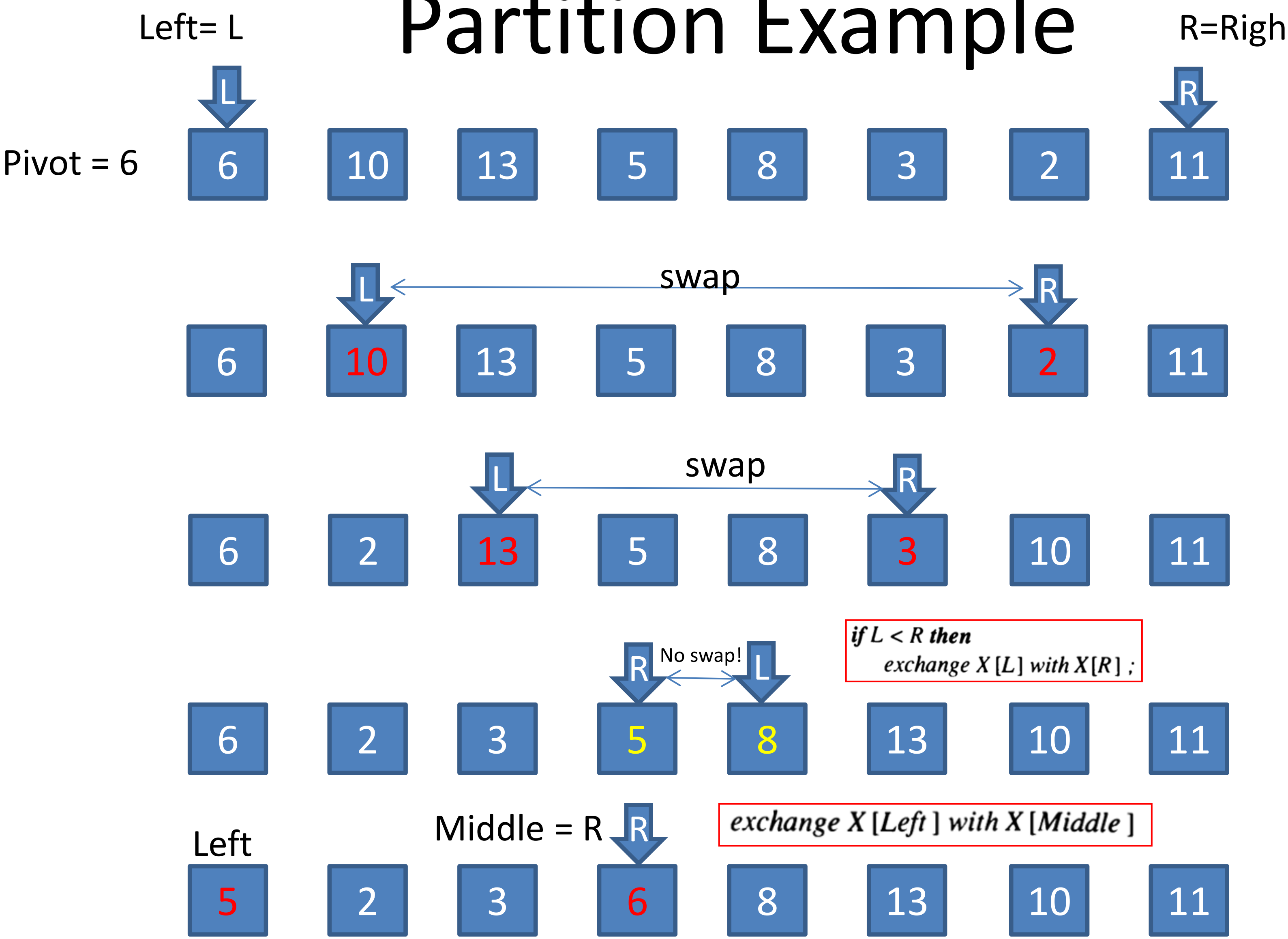
Partition (相向而行版本)

Algorithm Partition *(X, Left, Right) ;*
Input: *X* (an array), *Left* (the left boundary of the array), and *Right* (the right boundary).
Output: *X* and *Middle* such that $X[i] \le X[Middle]$ for all $i \le Middle$ and $X[j] > X[Middle]$ for all $j > Middle$.

```
begin
    pivot := X[Left] ;
    L := Left ;  R := Right ;
    while L < R do
        while X[L] ≤ pivot and L ≤ Right do L := L + 1 ;
        while X[R] > pivot and R ≥ Left do R := R - 1 ;
        if L < R then
            exchange X[L] with X[R] ;
    Middle := R ;
    exchange X[Left] with X[Middle]
end
```

Figure 6.9 Algorithm *Partition.*

Partition Example



Analysis

- Worst Case – input sorted(reversed sorted) – $O(n^2)$
- Best Case: uniform split 1:1 – O(nlgn)
- 2^{nd}-Worst Case: 1:9 split , using recursion tree – O(nlgn)
- Lucky-Unlucky switch case: O(nlgn)
- Average case(Randomized QuickSort) : Substitution method – O(nlgn).

To make things more practical

- Randomized QuickSort
  - Rearranging the element randomly
  - Selecting the pivot randomly
  - >=3 times faster than mergeSort!
  - Works well with cache and virtual mem.

Analysis of Randomized QuickSort

- We don't know which exactly split is made. Each recursive step uses random pivot!

- So, we compute the Expectation of the T(n) in terms of a random variable X_k.

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = \Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Expectation of T(n)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \quad\vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n) \right) \right]$$

Linearity of expectation. $= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) = \frac{2}{n}\sum_{k=1}^{n-1} E[T(k)] + \Theta(n)$$

Independence of X_k from other random choices.

Prove by substitution method

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \le a n \lg n$ for constant $a > 0$.

- Choose a large enough so that $a n \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact: $\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

Prove by substitution method

$$E[T(n)] \le \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\lg n - \left(\frac{an}{4} - \Theta(n)\right)$$

Express as ***desired* – *residual***.