Data Structures and Algorithms

Lecture 8: Sorting

Outline of Today's Lecture

- Internal sorting
  - Three basic sorting algorithms $\longleftarrow$ $\Theta(n^2)$
    - Insertion / bubble / selection sorts
  - One medium sorting algorithms $\longleftarrow$ $\Theta(n^{1.5})$
    - Shell sort
  - Three fast sorting algorithms $\longleftarrow$ $\Theta(n \log n)$
    - Merge / quick / heap sorts
  - Two special cases $\longleftarrow$ $\Theta(n)$
    - Bin / radix sorts
- External sorting

Sorting

- **Motivation**: Suppose the record of student consists of student name, ID, course name, score, we sort *n* students by their scores.
- Given a set of records $r_1, r_2, \dots, r_n$ with key values $k_1, k_2, \dots, k_n$, the **Sorting Problem** is to arrange the records in non-decreasing order by their keys.
- Measures of algorithm cost:
  - Comparisons and Swaps are two main operations in sorting.

Sorting terminology

- Input is a set of records stored *in an array*.
- A sorting algorithm is **stable**, if it does not change the relative ordering of records with identical key values.
- **Internal sorting** *vs.* **External sorting**
  - In interval sorting, all records can be loaded into a computer memory
  - External sorting, there are too many records to be sorted → cannot be loaded into the memory

Internal Sorting Algorithms

- Three $\Theta(n^2)$ sorting algorithms
  - Insertion / bubble / selection sorts
- Shell sort -- $O(n^{1.5})$ in average case
- Three quick sorting algorithms-- $\Theta(n \log n)$
  - Merge / quick / heap sorts
- Two $\Theta(n)$ sorting algorithms for special cases of record keys
- Lower bounds for sorting

Insertion Sort (1)

- Assume you have sorted the first *i* (e.g., *i*=2) numbers, consider the (*i*+1)th number 36, insert the number in order so that the first *i*+1 numbers are sorted.

Before insert:	**[27**	**53]**	**36**	**15**	**69**	**42**
After insert :	**[27**	**36**	**53]**	**15**	**69**	**42**

Insertion Sort (2)

- Traverse *i* from 1 to n-1, do the insertion

i=1:	[53]	27	36	15	69	42
i=2:	[27	53]	36	15	69	42
i=3:	[27	36	53]	15	69	42
i=4:	[15	27	36	53]	69	42
i=5:	[15	27	36	53	69]	42
	[15	27	36 42	53	69]	

Insertion Sort (3)

```
template <class E>
void insertSort(E A[], int n) {
  for (int i=1; i<n; i++)
    for (int j=i; j>0 && A[j] < A[j-1]; j--)
      swap(A, j, j-1);
}
```

Best Case Analysis of Insertion Sort

- The best case occurs when the initial list of number are already sorted
- Best Case: 0 swap, $n - 1$ comparisons

15	**27**	**36**	**42**	**53**	**69**

Worst Case Analysis of Insertion Sort

- The worst case occurs when the initial list of number are reversely sorted
- At i-th iteration, performs i comparisons and swaps
- Total: $\Sigma i = n^2/2$ swaps and comparisons

69	**53**	**42**	**36**	**27**	**15**

Average Case Analysis of Insertion Sort

- At *i*-th iteration, performs *i/2* comparisons and swaps on average
- Total: $\Sigma i/2 = n^2/4$ swaps and comparisons

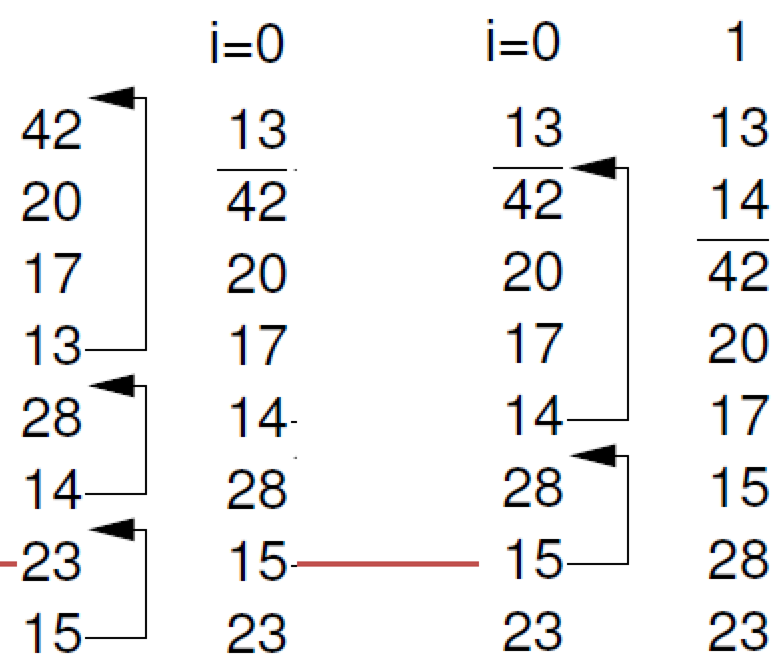
Before insert:	**[27**	**53]**	**36**	**15**	**69**	**42**
After insert :	**[27**	**36**	**53]**	**15**	**69**	**42**

Insertion Sort

- Best Case: 0 swap, $n - 1$ comparisons
- Worst Case: $n^2/2$ swaps and comparisons
- Average Case: $n^2/4$ swaps and comparisons
- Insertion Sort is suitable for the cases where the records in the input array are almost sorted, e.g.,
  - Many records are already been sorted initially, but some a few new records are added

Bubble Sort (1)

- Scan from the bottom to the top, compare each adjacent values K[j-1] and K[j], swap them if the K[j] < K[j-1]. After the scan, the *smallest* value is at the top (bubble up)
- Do the 2nd scan from the bottom to the top-2
- …



Bubble Sort (2)

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	42	14	14	14	14	14	14
17	20	42	15	15	15	15	15
13	17	20	42	17	17	17	17
28	14	17	20	42	20	20	20
14	28	15	17	20	42	23	23
23	15	28	23	23	23	42	28
15	23	23	28	28	28	28	42

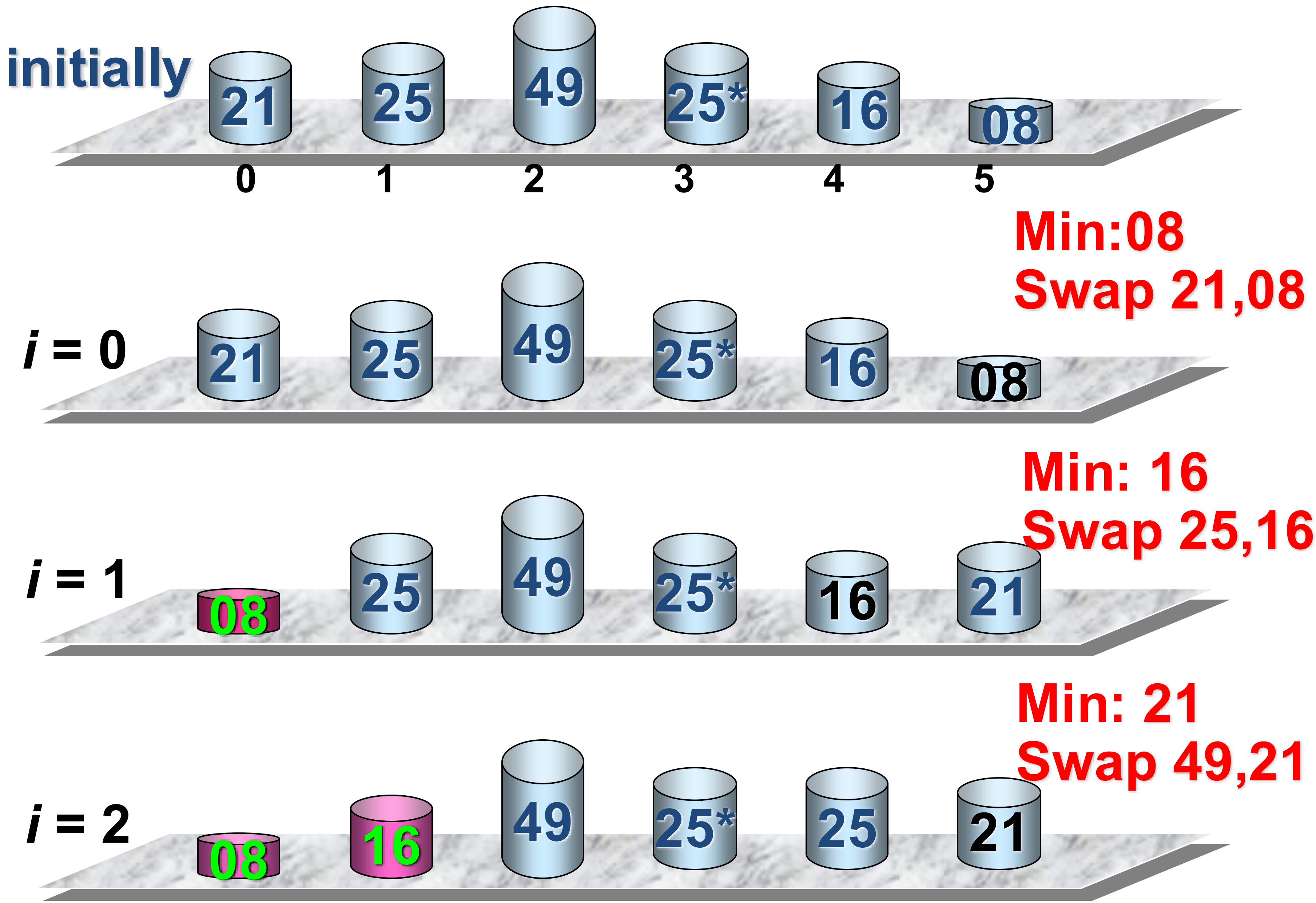
Bubble Sort (3)

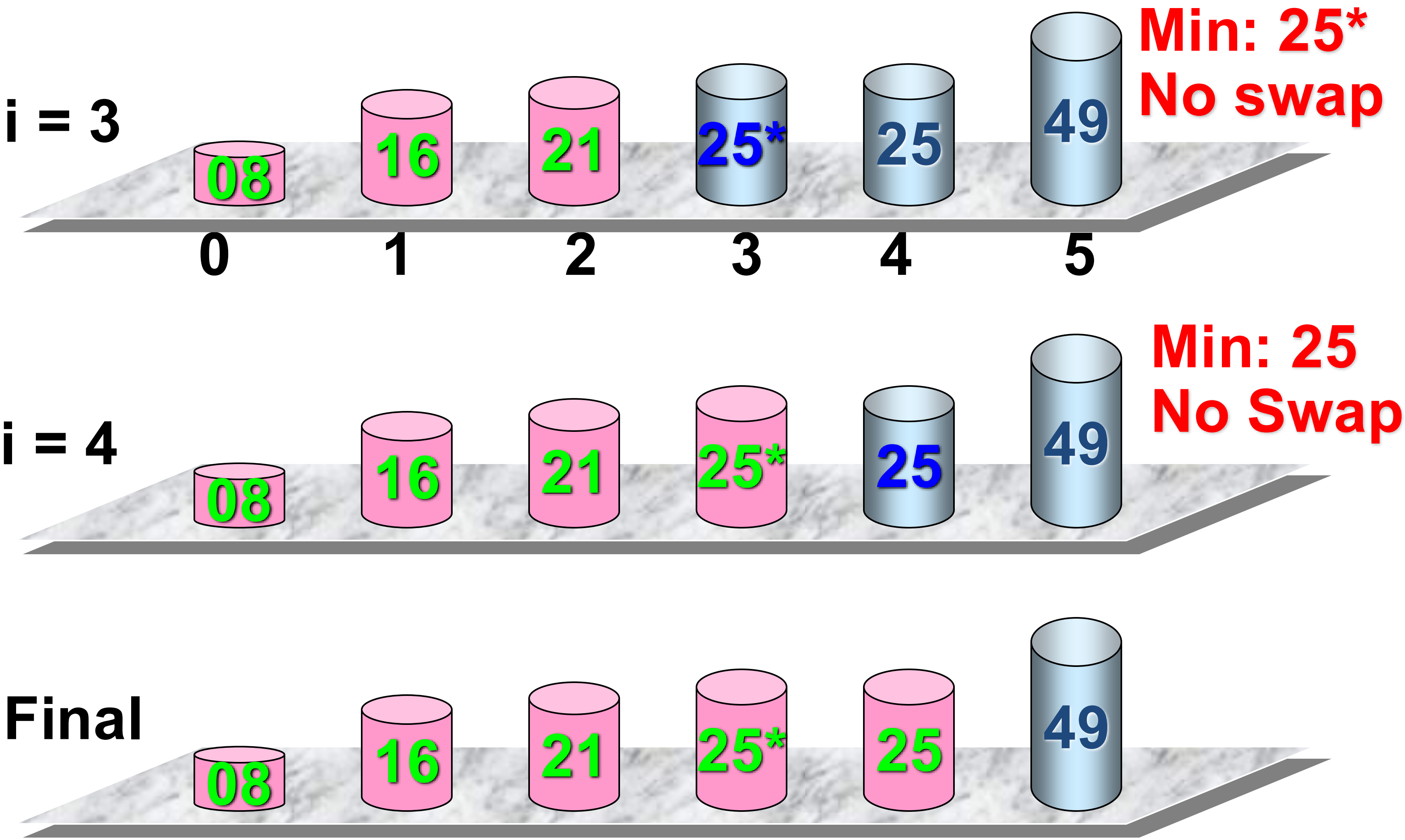
```
template <class E>
void bubbleSort(E A[], int n) {
  for (int i=0; i<n-1; i++)
    for (int j=n-1; j>i; j--)
      if ( A[j] < A[j-1] )
        swap(A, j, j-1);
}
```

- Best Case: 0 swaps, $n^2/2$ comparisons
- Worst Case: $n^2/2$ swaps and comparisons
- Average Case: $n^2/4$ swaps and $n^2/2$ comparisons

Selection Sort

- Basic idea:
- First, select the smallest value, store it at the first location in the array
- Select the 2^{nd} smallest value, store it at the 2^{nd} location in the array
- …
- The array is sorted after *n* iterations


initially
21 25 49 25* 16 08
0 1 2 3 4 5
Min:08
Swap 21,08
i = 0
21 25 49 25* 16 08
Min: 16
Swap 25,16
i = 1
08 25 49 25* 16 21
Min: 21
Swap 49,21
i = 2
08 16 49 25* 25 21


i = 3
08
16
21
25*
25
49
Min: 25*
No swap
0
1
2
3
4
5
i = 4
08
16
21
25*
25
49
Min: 25
No Swap
Final
08
16
21
25*
25
49

Selection Sort (2)

```
template <class E>
void selectionSort(E A[], int n) {
   for (int i=0; i<n-1; i++) {
      int lowindex = i; // Remember its index
      for (int j=n-1; j>i; j--) // Find least
         if ( A[j] < A[ lowindex ])
            lowindex = j; // Put it in place
      swap(A, i, lowindex);
   }
}
```

- Best case: n-1 swaps, $n^2/2$ comparisons.
- Worst case: n - 1 swaps and $n^2/2$ comparisons.
- Average case: n-1 swaps and $n^2/2$ comparisons.

Summary of three $\Theta(n^2)$ sorting algorithms

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Running time comparisons (n=100k)

- Random input

```
100000 numbers to be sorted:
Sort with InsertionSort, Time consumed:  1.438 (seconds)
Sort with bubble sort, Time consumed:    17.360 (seconds)
Sort with SelectionSort, Time consumed:  3.813 (seconds)
```

- The input is already sorted

```
100000 numbers to be sorted:
Sort with InsertionSort, Time consumed:  0.000 (seconds)
Sort with bubble sort, Time consumed:    3.766 (seconds)
Sort with SelectionSort, Time consumed:  3.905 (seconds)
```

- The input is reversely sorted

```
100000 numbers to be sorted:
Sort with InsertionSort, Time consumed:  2.984 (seconds)
Sort with bubble sort, Time consumed:    9.692 (seconds)
Sort with SelectionSort, Time consumed:  3.872 (seconds)
```

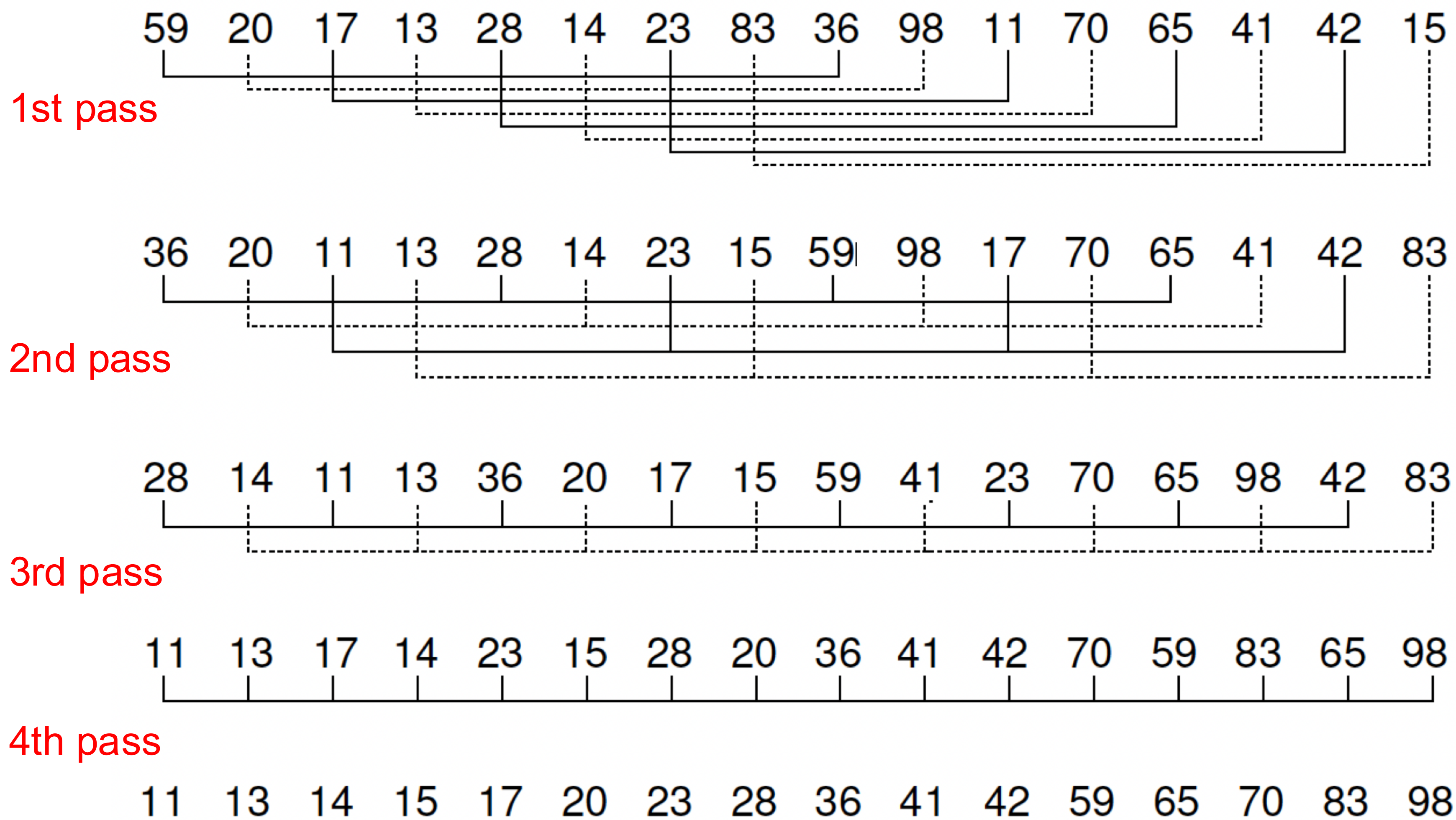
Shell sort

- Shellsort, named after its invertor, D.L. Shell.
  - Sometimes called the diminishing increment sort.
  - $O(n^{1.5})$ on average-case
- Its strategy is to make the list "mostly sorted" so that a final Insertion Sort can finish the job.
- Main steps:
  - Break the list into sublists
  - Sort them
  - Then, recombine the sublists

Shell sort process

- During each iteration/pass, Shellsort breaks the list into disjoint sublists so that each element in a sublist is a fixed number of postions aparts. e.g.,
  - Let us assume for convenience that *n*, the number of values to be sorted, is *a power of two*.
  - Shellsort will begin by breaking the list into *n/2* sublists of 2 elements each, where the array index of the 2 elements in each sublist differs by *n/2*.

Shell sort - an example (1)



Shell sort - an example (2)

- Some choices for increments would make Shellsort run more efficiently.
  - In particular, the choice of increments described above $(2^k, 2^{k-1}, \ldots, 2,1)$ turns out to be relatively inefficient.
  - A better choice is the following series based on devision by three: $(\ldots, 121,40,13,4,1)$.

Shellsort Implementation

```
// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
  for (int i=incr; i<n; i+=incr)
    for (int j=i; (j>=incr) &&
          (Comp::prior(A[j], A[j-incr])); j-=incr)
      swap(A, j, j-incr);
}

template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
  for (int i=n/2; i>2; i/=2) // For each increment
    for (int j=0; j<i; j++) // Sort each sublist
      inssort2<E,Comp>(&A[j], n-j, i);
  inssort2<E,Comp>(A, n, 1);
}
```

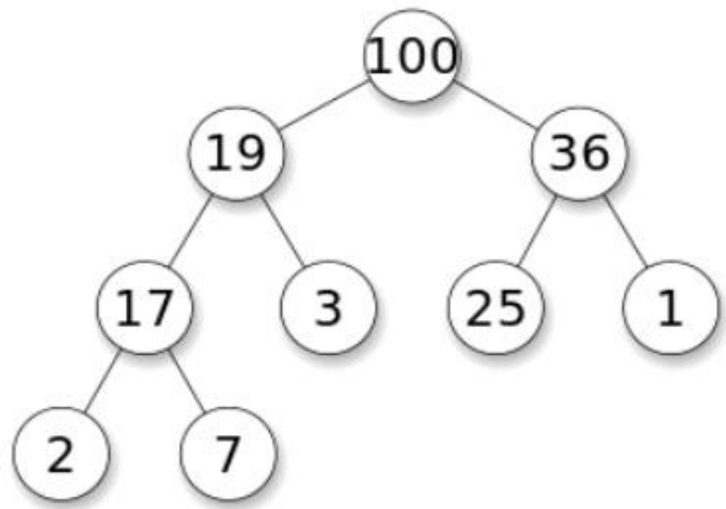
Three fast sorting algorithms

- Heap sort
  - $\Theta(n \log n)$ for the worst, best, average cases
- Merge sort
  - $\Theta(n \log n)$ for the worst, best, average cases
- Quick sort
  - $\Theta(n \log n)$ for the best and average cases
  - $\Theta(n^2)$ for the worst case

Heap – a special binary tree (Ch. II.5)

Heap: Complete binary tree with the heap property:

- Max-heap: each value in a node is no less than its children values
- The values in the tree are partially ordered.
  - The left child may less or greater than its right child

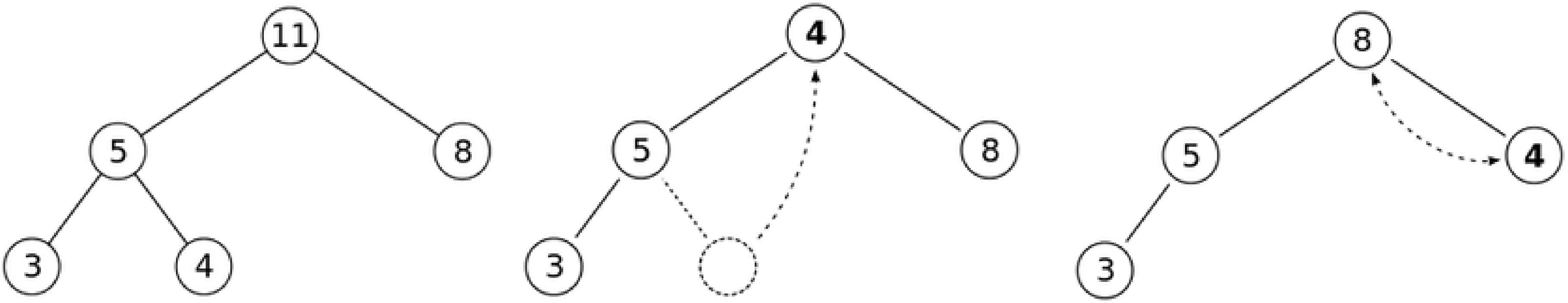


Heap Sort

- Given an array, build a max-heap
- Remove the maximum number from the heap
- Remove the next maximum number
- …
- Continue until no numbers are left in the heap

Heap -- removeMax

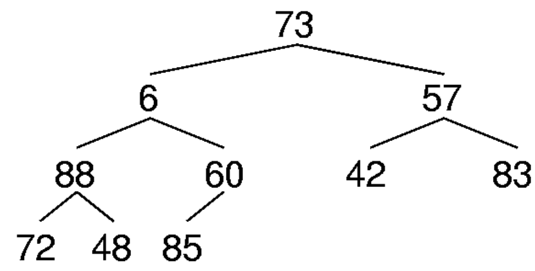
- Replace the root of the heap with the last element on the last level
- Compare the new root with its children (shift down operation)
  - if the new root is larger than its children, stop.
  - If not, swap the element with its largest children, and return to the previous step
  - Worst time complexity Θ(log n)



HeapSort Example (1)

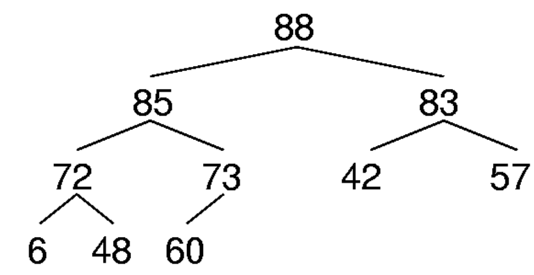
Original Numbers

73	6	57	88	60	42	83	72	48	85



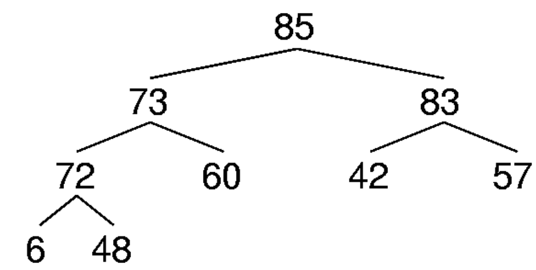
Build Heap

88	85	83	72	73	42	57	6	48	60



Remove 88

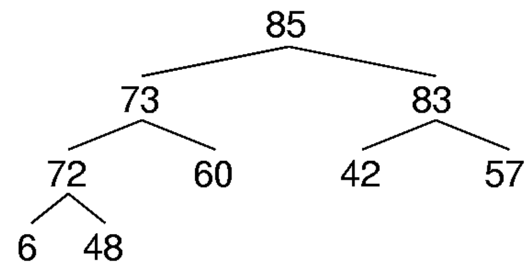
85	73	83	72	60	42	57	6	48	88



HeapSort Example (2)

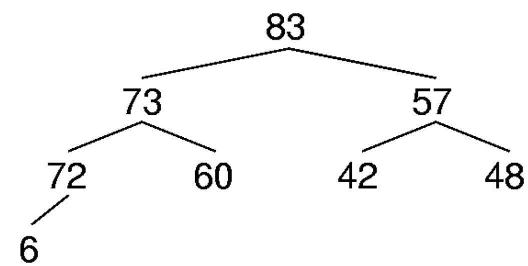
Remove 88

85	73	83	72	60	42	57	6	48	88



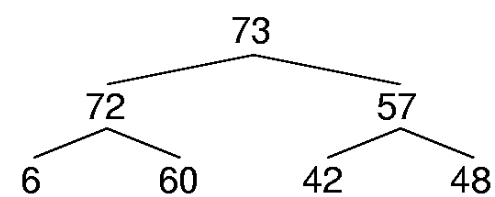
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Heapsort

```
template <class E>
void heapSort(E A[], int n) { // Heapsort
  E mval;
  maxheap<E> H(A, n, n);
  for (int i=0; i<n; i++)   // Now sort
    H.removemax(mval);      // Put max at end
}
```

Analysis of Heap Sort

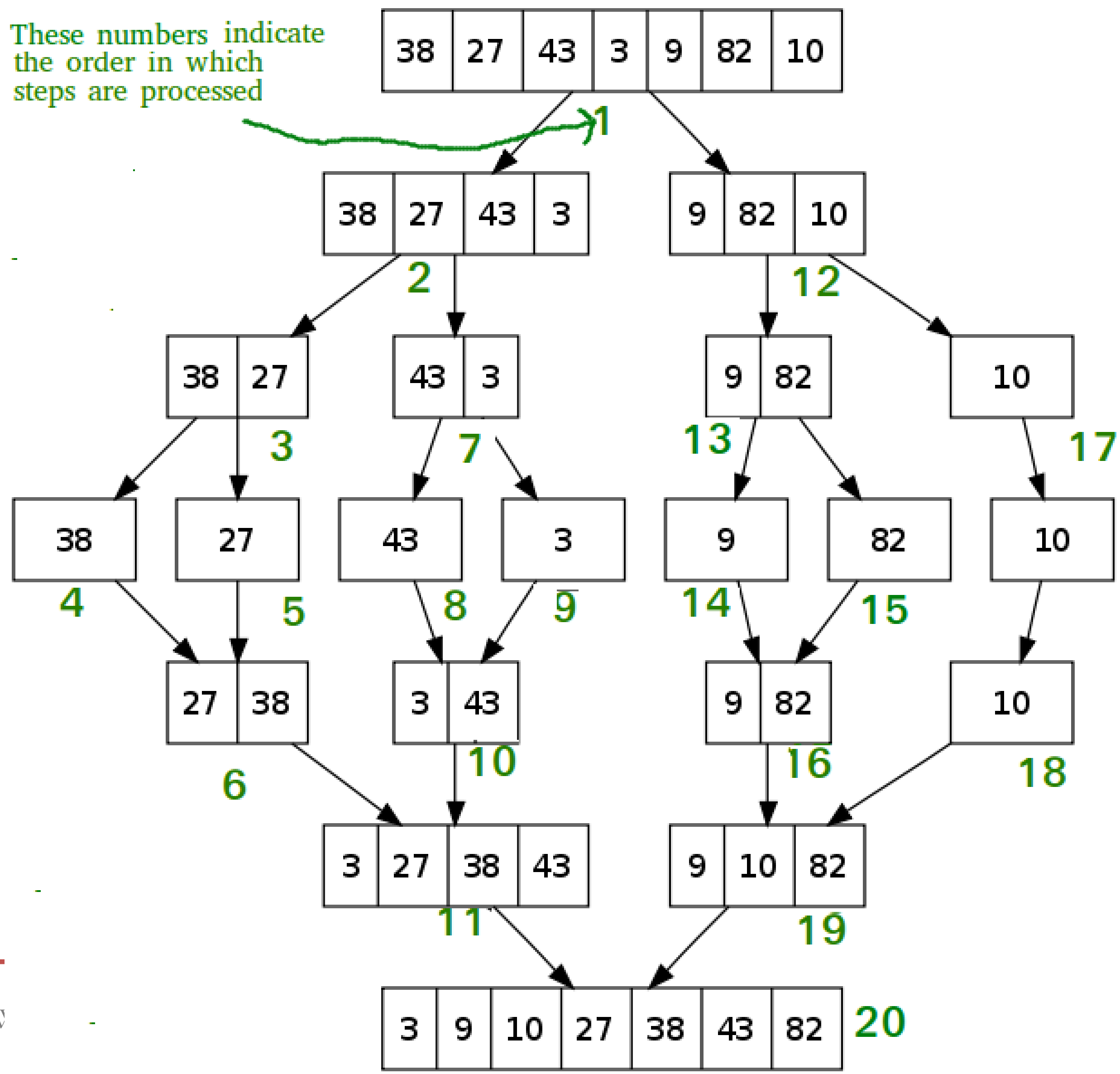
- Build a heap takes time $\Theta(n)$
- Remove the maximum value takes $\Theta(\log n)$, as heap is a complete tree
- Total time is $\Theta(n) + n\,\Theta(\log n) = \Theta(n \log n)$

Merge Sort

- Basic idea: divide and conquer

1. Given a list of numbers to be sorted
2. Split the list into two sub-lists with the identical length
3. Recursively sort the sub-lists, respectively
4. Merge the two sorted sub-lists

Merge Sort


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge sort with an array-based list (1)

- An array A[left, …,right], with the index range: left -- right
- How to split ?
- Let mid = (left + right)/2
- Left sub-list = A[left,…,mid]
- Right sub-list=A[mid+1,…, right]

Merge Sort with an array-based list (2)

- How to merge two sorted sub-lists A[left,…,mid], A[mid+1, …, right] ?

- An extra array temp[left…, right] is needed

- Step 1: move the smallest value of the first numbers of the two-sublists to array temp
  - If one sub-list is exhausted, just move the first number of the other sublist
- Continue until no numbers are left
- Copy back: A[left,…, right]=temp[left,…,right]

Merge Sort Implementation

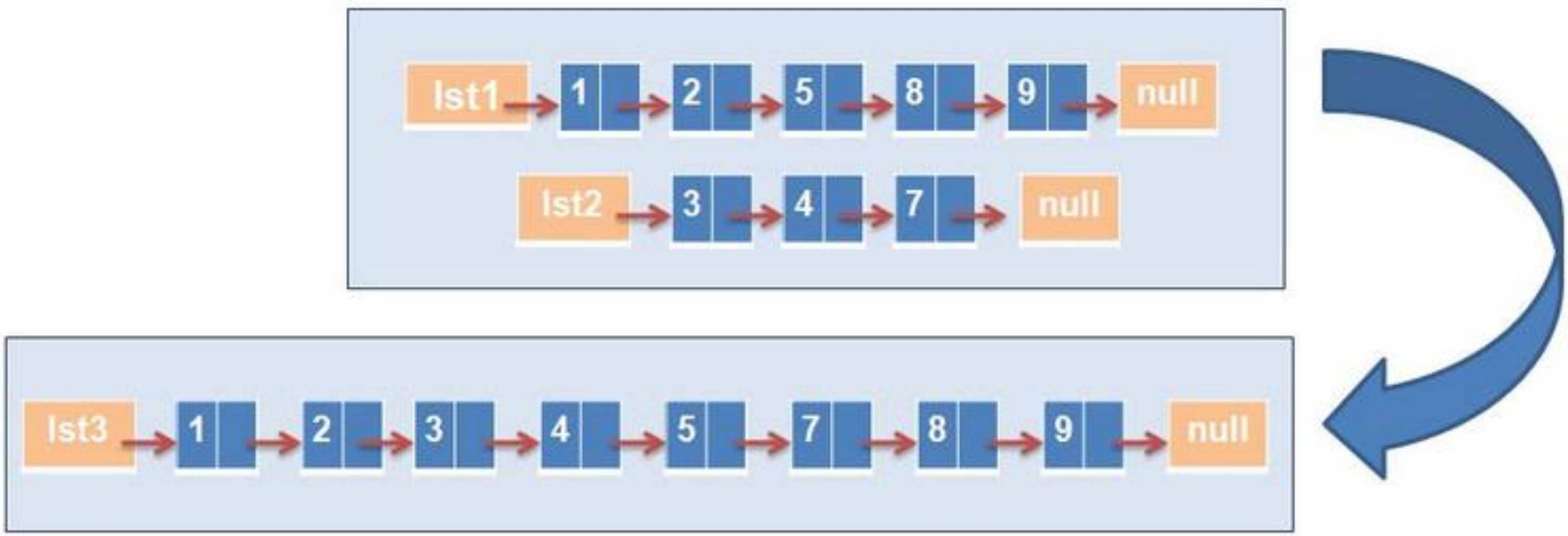
```
template <class E>
void mergeSort(E A[], E temp[],
                 int left, int right) {
  if (left == right) return;
  int mid = (left+right)/2;
  mergesort<E>(A, temp, left, mid);
  mergesort<E>(A, temp, mid+1, right);
  //merge two sorted sublists
  int i1 = left; int i2 = mid + 1;
  for (int curr=left; curr<=right; curr++) {
    if (i1 == mid+1)        // Left exhausted
      temp[curr] = A[i2++];
    else if (i2 > right)    // Right exhausted
      temp[curr] = A[i1++];
    else if (A[i1] < A[i2])
      temp[curr] = A[i1++];
    else temp[curr] = A[i2++];
  }
  for (int i=left; i<=right; i++) // Copy back
    A[i] = temp[i];
}
```

Merge Sort based on a linked list (1)

- How to split ?
- Given a singly linked list of numbers
- Need to scan half of numbers in the list

How to merge two sorted sub-lists ?

- Similar to the array-based version
- But no extra memory is needed



Time complexity of Merge Sort

- Let $T(n)$ be the running time for **n** numbers
- Split: $\Theta(1)$ for the array-based list
- Recursively sorting two sub-lists: $2 * T(n/2)$
- Merge: $\Theta(n)$
- $T(n)= 2\,T(n/2) + \Theta(n)$
- Expand the recurrence relationship, we have:
- $T(n)= \Theta(n \log n)$

Quick Sort

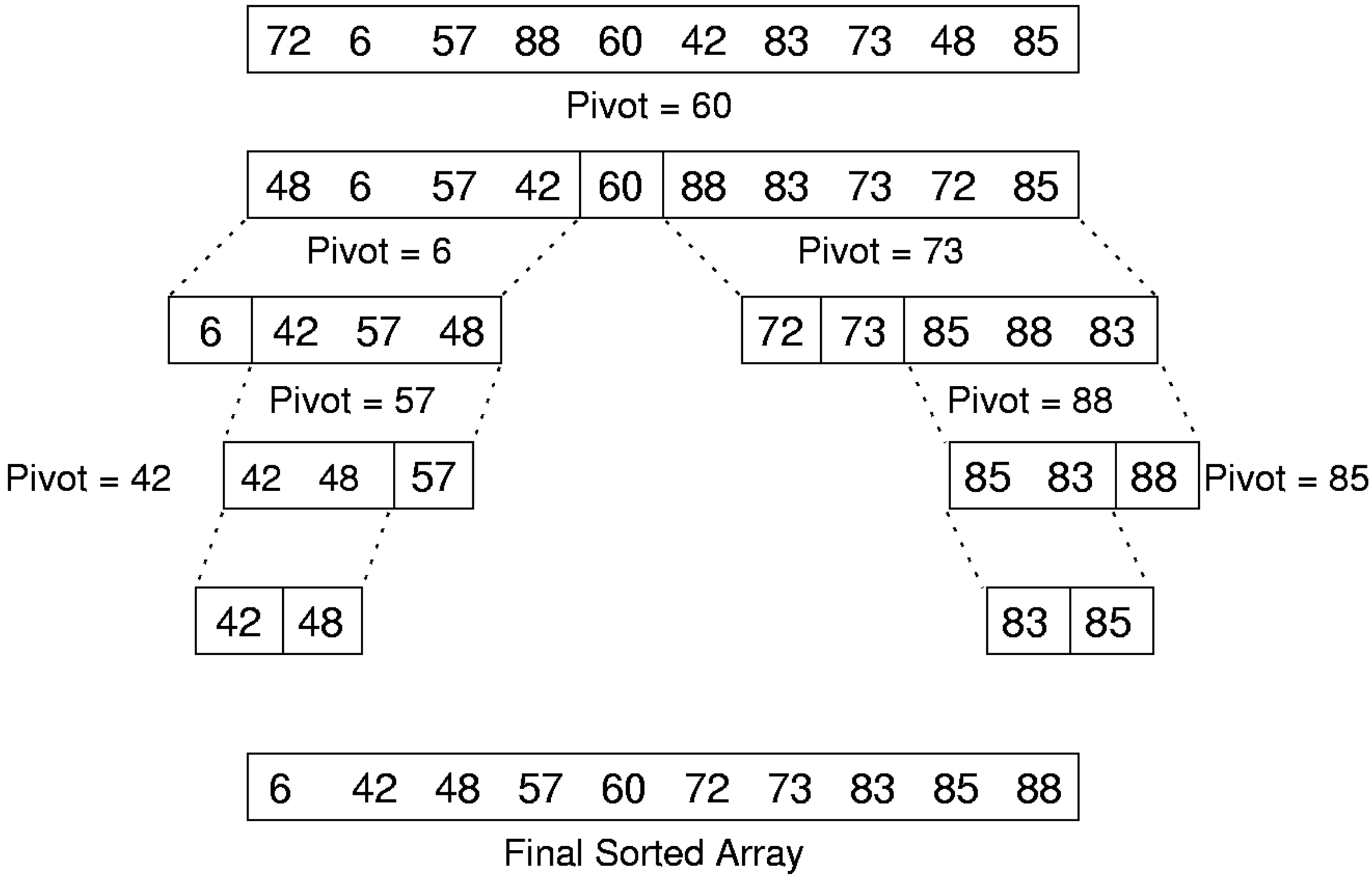
- Given an array of numbers A[left, …, right]
- Pick a value in the array as a pivot

- Partition the array into three parts

1. The numbers in the left part are < pivot 54
2. The pivot itself in place
3. The numbers in the right part are ≥ pivot 54

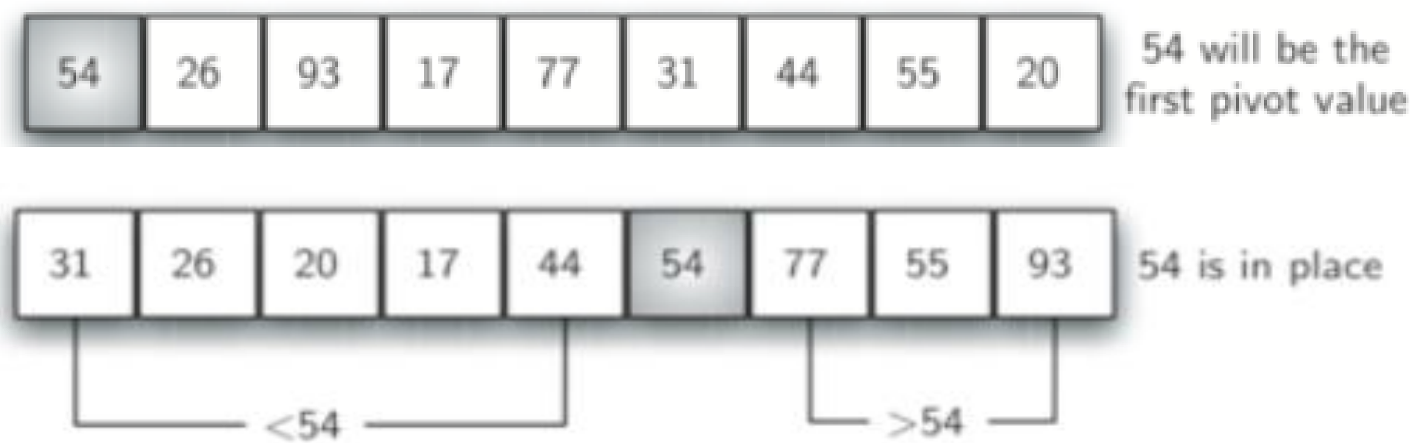
- Recursively sort the left and right parts

QuickSort Example



Two key problems in Quick Sort

- How to choose the pivot, such that the left and right parts are roughly balanced ?
  - The number of records in the left part is more or less the number in the right part
- How to efficiently partition an array by the pivot?



Solutions to the choice of a pivot

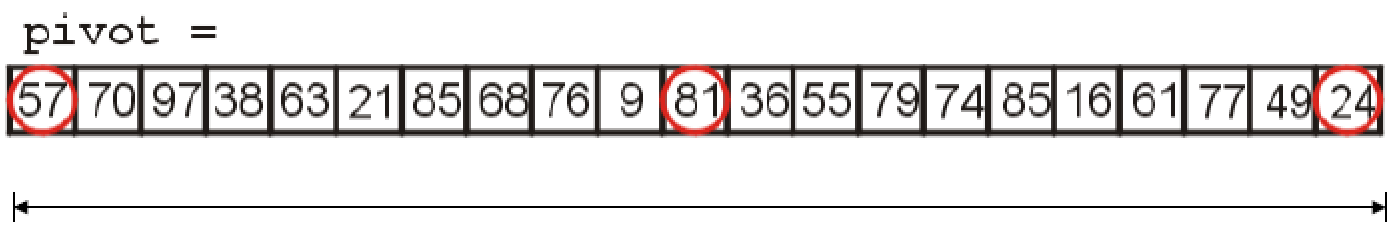
1. Traditionally, choose the first or the last number in the array
   - This is bad if the given array are already (or nearly) sorted, or reversely sorted, one part has 0 number, the other part has (n-1) numbers
2. Choose the middle number
   - mid = (left+right)/2; pivot = A[mid];
   - a better choice
3. median of three
   - Choose the pivot as the median of the first, middle and last numbers
   - Much better

Solutions to the choice of a pivot

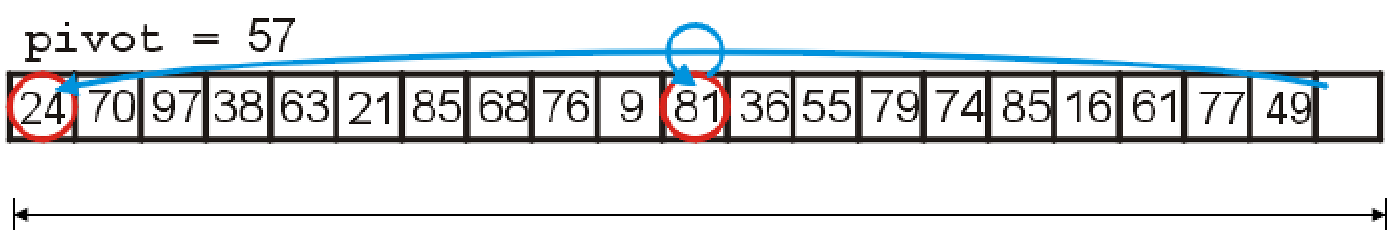
4. Randomly choose a number as the pivot
   - It is unlikely that a randomly chosen number is the smallest or the largest ones
   - Can combine with the 3^{rd} solution, i.e., randomly choose three numbers, and select the median of the three as the pivot

Partition an array by a given pivot

- Assume that the pivot is the median of the first, middle, and last numbers

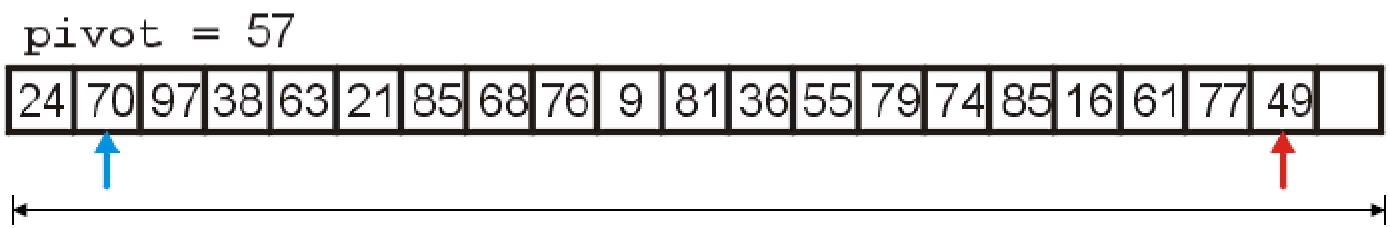


- First swap the pivot with the last number

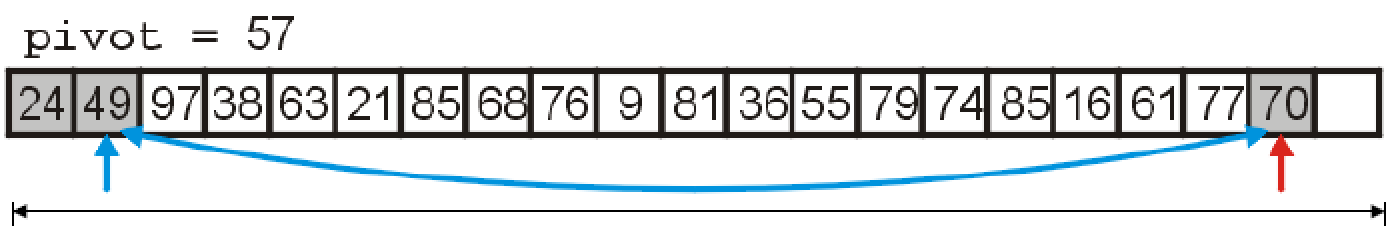


Partition an array by a given pivot

1. Start from the 1st location, search forward until we find a value ≥ pivot, e.g., 70 > 57
2. Start from the 2nd last location, search backward until we find a value < pivot, e.g., 49 < 57



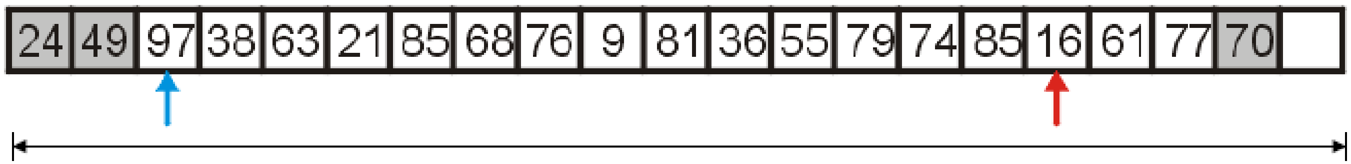
3. 70 and 49 are out of order, swap them



- We continue step1—step 3, see the next slides

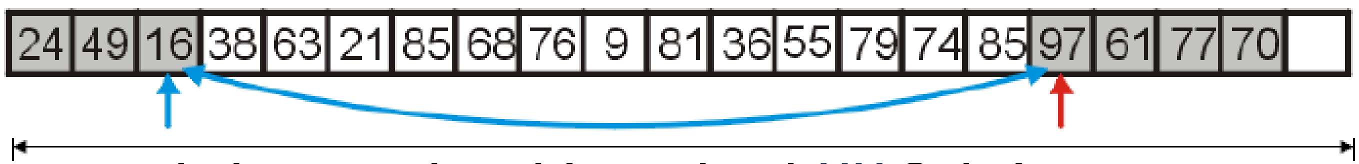
4. search forward until we find 97 ≥ 57
5. search backward until we find 16 < 57

pivot = 57



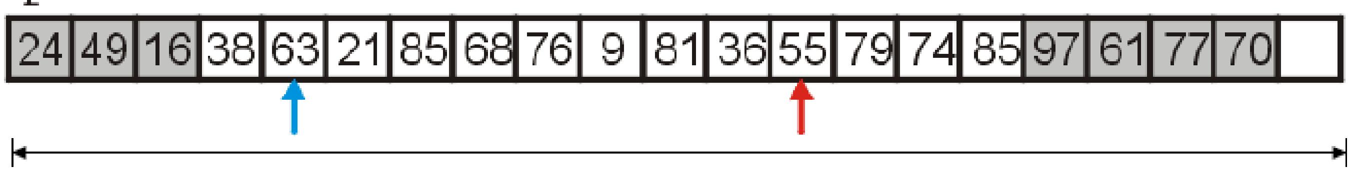
6. Swap 97 and 16

pivot = 57



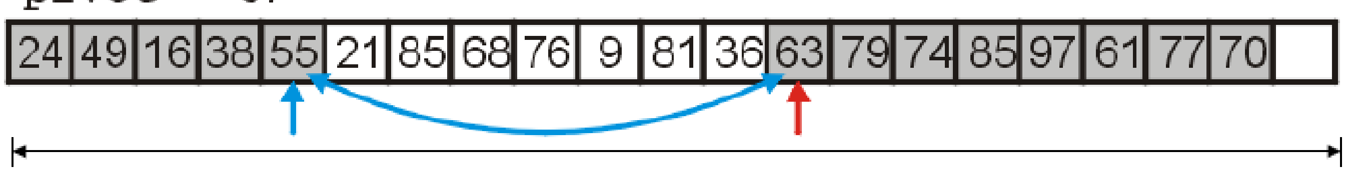
7. search forward until we find 63 ≥ 57
8. search backward until we find 55 < 57

pivot = 57

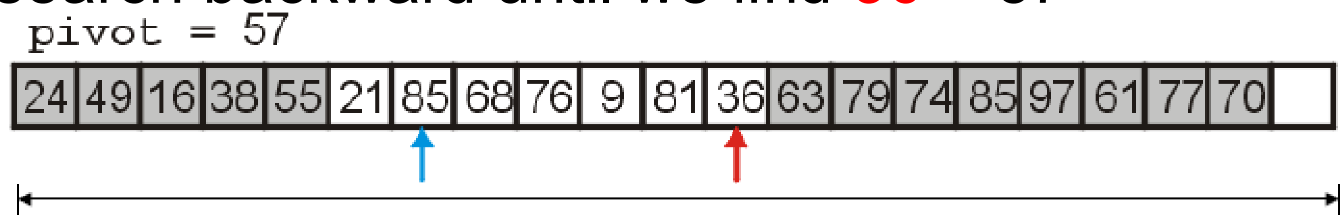


9. Swap 63 and 55

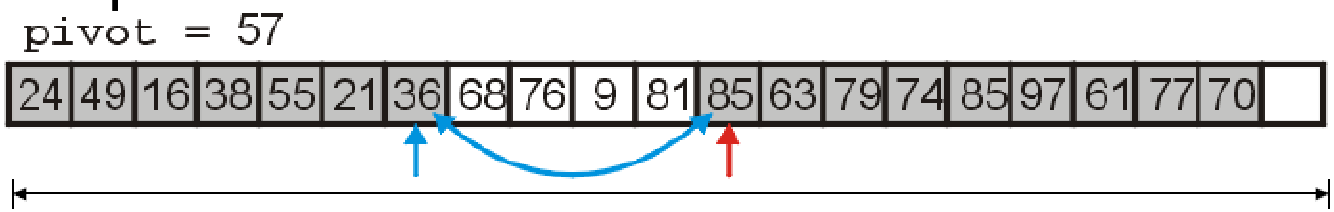
pivot = 57



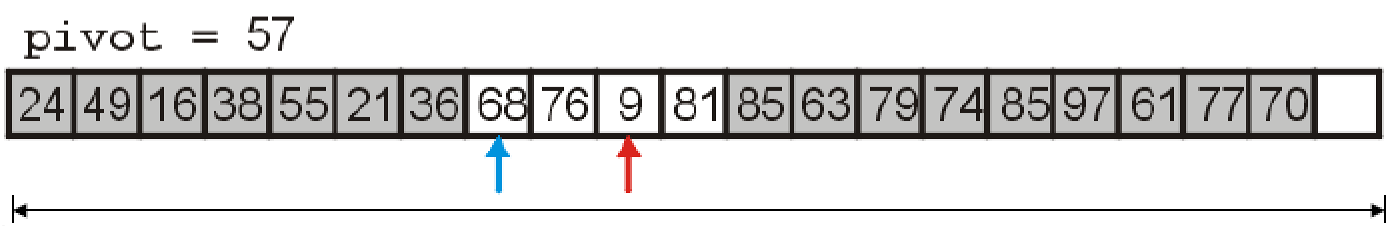
10. search forward until we find 85 ≥ 57
11. search backward until we find 36 < 57



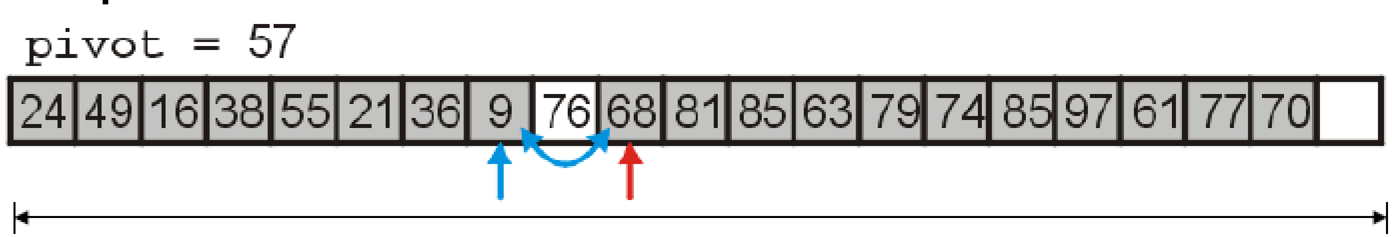
12. Swap 85 and 36



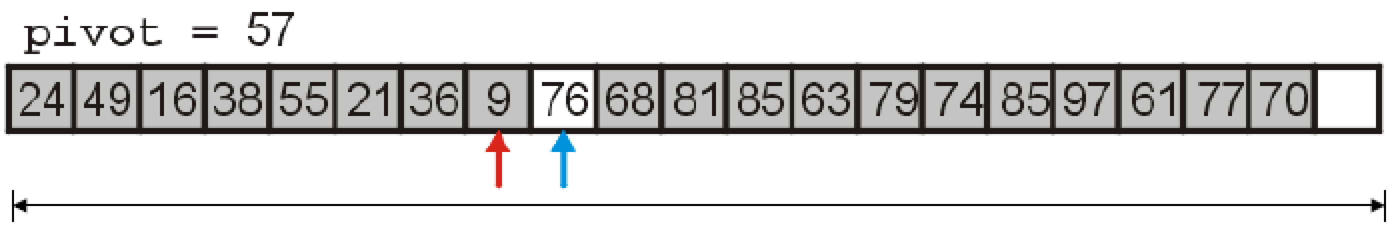
13. search forward until we find 68 ≥ 57
14. We search backward until we find 9 < 57



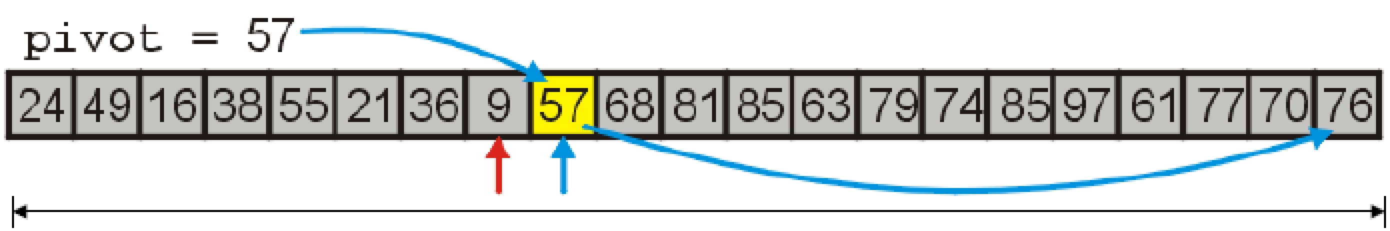
15. Swap 68 and 9



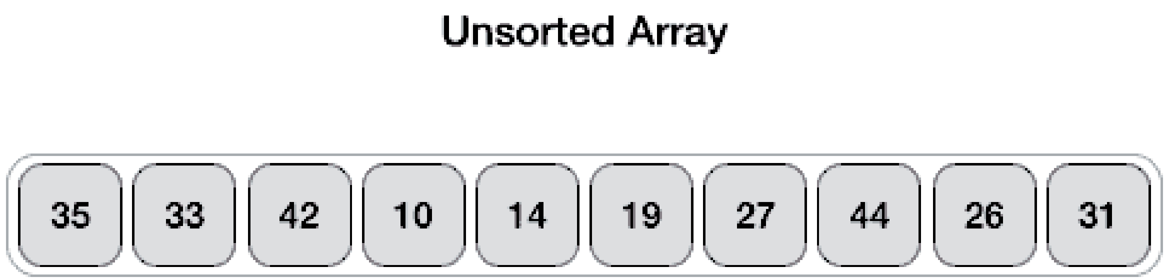
16. search forward until we find 76 ≥ 57
17. search backward until we find 9 < 57
   - The indices are out of order, stop



- Move the first value larger than pivot, i.e., 76, to the last location of the array
- Fill the empty location with the pivot 57
- The pivot is in the correct location



Another example of the partition (animation)



Time complexity of Quick Sort

- Finding the pivot takes time $\Theta(1)$
- Partitioning an array takes time $\Theta(n)$
- Worst case time complexity
  - For each partition, one part has 0 number, the other has n-1 numbers
  - $T(n)= \Theta(n) + T(n-1)$
  - $T(n)= \Theta(n^2)$

Time complexity of Quick Sort

- Best case analysis
  - The best case occurs if the left and right parts are balanced, each has about $n/2$ numbers
  - $T(n) = \Theta(n) + 2T(n/2)$
  - $T(n) = \Theta(n \log n)$

Average time complexity of quick sort

- Consider all cases of the lengths of the two parts
  - Left: 0 number, right: n-1 numbers
  - Left: 1 number, right: n-2 numbers
  - Left: 2 number, right: n-3 numbers
  - …
  - Left: n-1 number, right: 0 numbers
- Assume the probabilities of different cases are equal, i.e., 1/n, we have

$$\mathrm{T}(n) = cn + \frac{1}{n}\sum_{k=0}^{n-1}[\mathrm{T}(k) + \mathrm{T}(n-1-k)], \quad \mathrm{T}(0) = \mathrm{T}(1) = c.$$

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

$$\begin{aligned}
nT(n) &= cn^2 + 2\sum_{k=0}^{n-1} T(k) \\
(n-1)T(n-1) &= c(n-1)^2 + 2\sum_{k=0}^{n-2} T(k)
\end{aligned}$$

$$nT(n) - (n-1)T(n-1) = c(2n-1) + 2T(n-1)$$

$$nT(n) = (n+1)T(n-1) + c(2n-1)$$

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{c(2n-1)}{n(n+1)} \\
&\le \frac{T(n-1)}{n} + \frac{c2n}{n(n+1)} \\
&= \frac{T(n-1)}{n} + \frac{2c}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2c}{n} + \frac{2c}{n+1} \\
&\vdots \\
&= \frac{T(1)}{2} + \sum_{i=2}^{n} \frac{2c}{i+1} \\
&\le 2c\sum_{i=1}^{n-1}\frac{1}{i} \approx 2c\int_1^n \frac{1}{x}\mathrm{d}x = 2c\ln n
\end{aligned}$$

$$T(n) = \Theta(n \log n)$$

Running time comparisons (n=100k)

- Random input

```
100000 numbers to be sorted:
Sort with InsertionSort, Time consumed:  1.438 (seconds)
Sort with bubble sort, Time consumed:    17.360 (seconds)
Sort with SelectionSort, Time consumed:  3.813 (seconds)
Sort with shellSort, Time consumed:      0.016 (seconds)
Sort with heapSort, Time consumed:       0.000 (seconds)
Sort with mergeSort, Time consumed:      0.015 (seconds)
Sort with quickSort, Time consumed:      0.016 (seconds)
```

- The input is already sorted

```
100000 numbers to be sorted:
Sort with InsertionSort, Time consumed:  0.000 (seconds)
Sort with bubble sort, Time consumed:    3.766 (seconds)
Sort with SelectionSort, Time consumed:  3.905 (seconds)
Sort with shellSort, Time consumed:      0.002 (seconds)
Sort with heapSort, Time consumed:       0.005 (seconds)
Sort with mergeSort, Time consumed:      0.004 (seconds)
Sort with quickSort, Time consumed:      0.000 (seconds)
```

- The input is reversely sorted

```
100000 numbers to be sorted:
Sort with InsertionSort, Time consumed:  2.984 (seconds)
Sort with bubble sort, Time consumed:    9.692 (seconds)
Sort with SelectionSort, Time consumed:  3.872 (seconds)
Sort with shellSort, Time consumed:      0.004 (seconds)
Sort with heapSort, Time consumed:       0.005 (seconds)
Sort with mergeSort, Time consumed:      0.005 (seconds)
Sort with quickSort, Time consumed:      0.001 (seconds)
```

Running time comparisons (n=3M)

- Random input

```
3000000 numbers to be sorted:
Sort with shellSort, Time consumed:      1.087 (seconds)
Sort with heapSort, Time consumed:       0.860 (seconds)
Sort with mergeSort, Time consumed:      0.421 (seconds)
Sort with quickSort, Time consumed:      0.334 (seconds)
```

- The input is already sorted

```
3000000 numbers to be sorted:
Sort with shellSort, Time consumed:      0.196 (seconds)
Sort with heapSort, Time consumed:       0.220 (seconds)
Sort with mergeSort, Time consumed:      0.156 (seconds)
Sort with quickSort, Time consumed:      0.033 (seconds)
```

- The input is reversely sorted

```
3000000 numbers to be sorted:
Sort with shellSort, Time consumed:      0.306 (seconds)
Sort with heapSort, Time consumed:       0.217 (seconds)
Sort with mergeSort, Time consumed:      0.179 (seconds)
Sort with quickSort, Time consumed:      0.042 (seconds)
```

Two $\Theta(n)$ sorting algorithms

- Only applicable for special cases, but not general cases
- BinSort
- Radix Sort

BinSort Motivation

- Consider n=5 integers to be sorted:
  - A[5]=1, 5, 4, 9, 2
  - Notice that the maximum number is < 2n = 10
- Allocate an array Bin[10]
- Place A[i] to Bin[A[i]], e.g.,
  - Place A[1] = 5 to Bin[5] by setting Bin[5] = **1**
  - The other values in Bin are **0**

	0	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

BinSort Motivation

- A[5]=1, 5, 4, 9, 2

	0	**1**	**1**	**0**	**1**	**1**	**0**	**0**	**0**	**1**
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

- BinSort has three steps:

1. Set Bin[j]=0 for 0≤j ≤9
2. Scan array A, set Bin[A[i]]=1 for 0≤i ≤5
3. Scan array Bin from the leftmost to rightmost, if Bin[j] = 1, number j is in array A, and output j

- The output is the sequence:1, 2, 4, 5, 9

Binsort

- A[0, …, n-1],
- Assume that A[i]≥0 and A[i] < c*n, c is a constant, e.g., c = 2
- Allocate an array B with size c*n

```
for (j=0; j<c*n; j++)
    B[j]=0;
for (i=0; i<n; i++)
  ++B[ A[i] ]; // may have duplicate numbers
i=0; // the ith sorted number
for (j=0; j<c*n; j++)//number j appears B[j] times
  for (k=0; k<B[j]; k++, i++)
    A[i] = j;
```

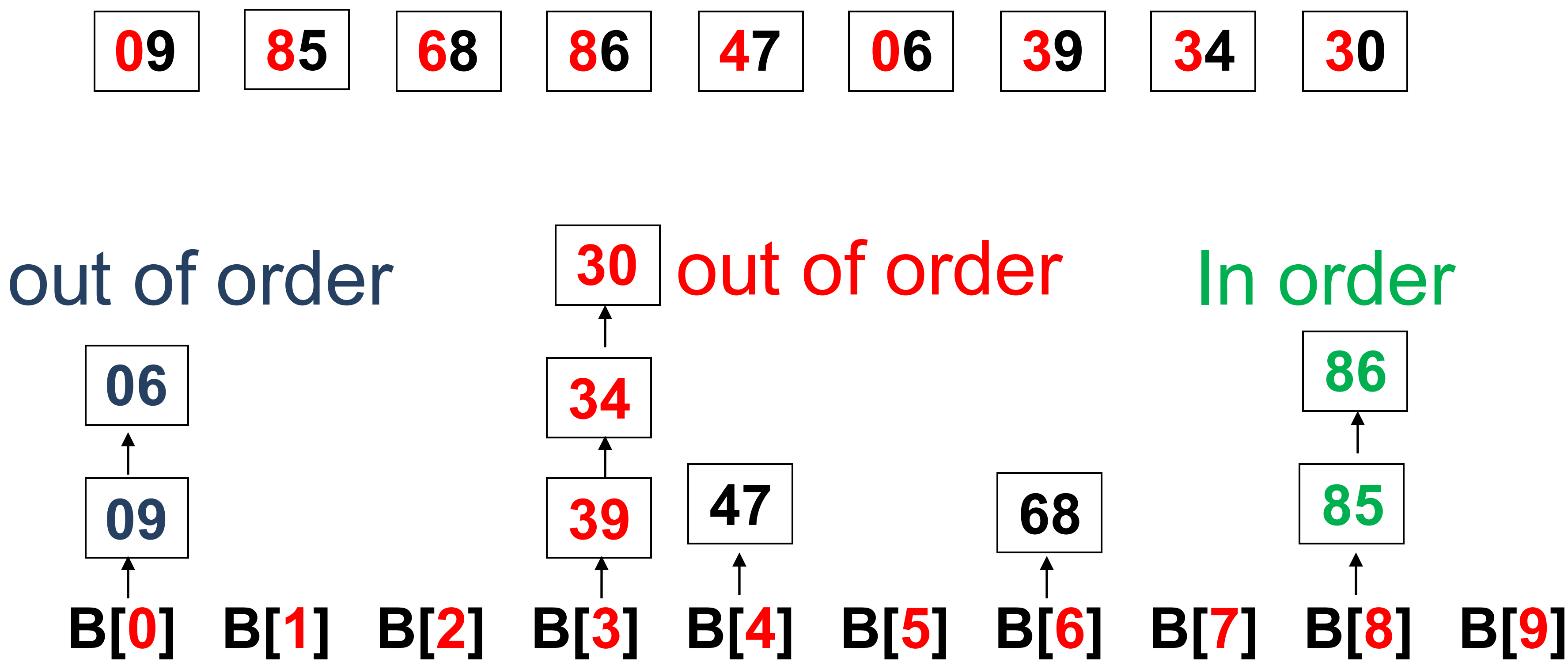
- Time complexity
  - $\Theta(cn) + \Theta(n) + \Theta(cn+n)$
  - $= \Theta(cn) = \Theta(n)$, as c is a constant

The application of BinSort is limited

- A[0, …, n-1],
- BinSort is applicable when A[i] < $c*n$
- Consider another example with $n=9$ numbers
  - 09, 85, 68, 86, 47, 06, 39, 34, 30
  - The maximum number 86 is about 10 times larger than n, $\geq n^2=81$
  - If BinSort is applied, an array B with size $87 \geq n^2$ is needed
  - The time complexity then is in $\Omega(n^2)$

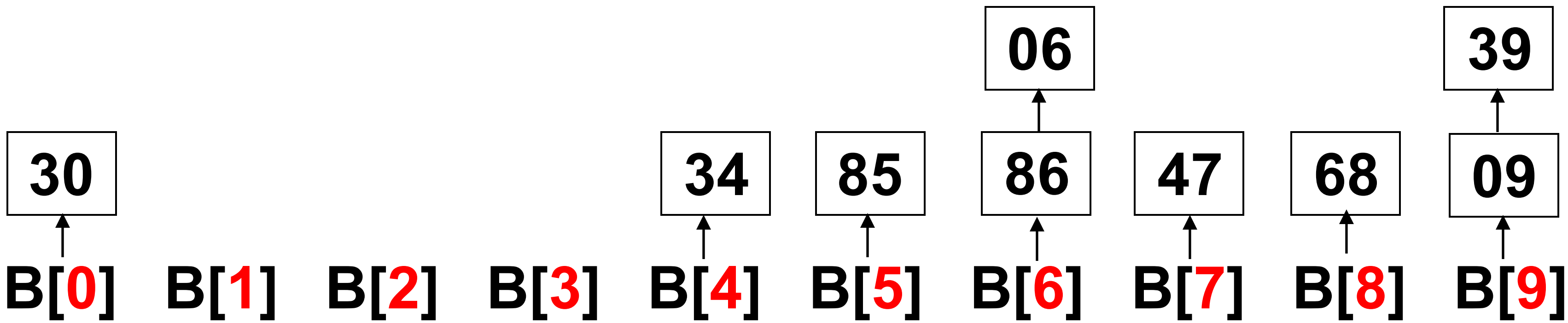
Radix Sort -- Extend BinSort

- Some examples of radix or base
- Radix 10: the values of each digit may be 0, 1, 2, …, 9
  - 5_{10}, 16_{10}, 20_{10},…
- Radix 2: each digit is 0 or 1
  - 101_2, 10000_2, 10100_2,…
- Radix 26 (26 letters): a, b, c, …, x, y, z
  - Strings \`type', 'alpha', \`go'



- Each number has two digits
- If we first sort by the highest digit:
- But the numbers in the same bin may be out of order

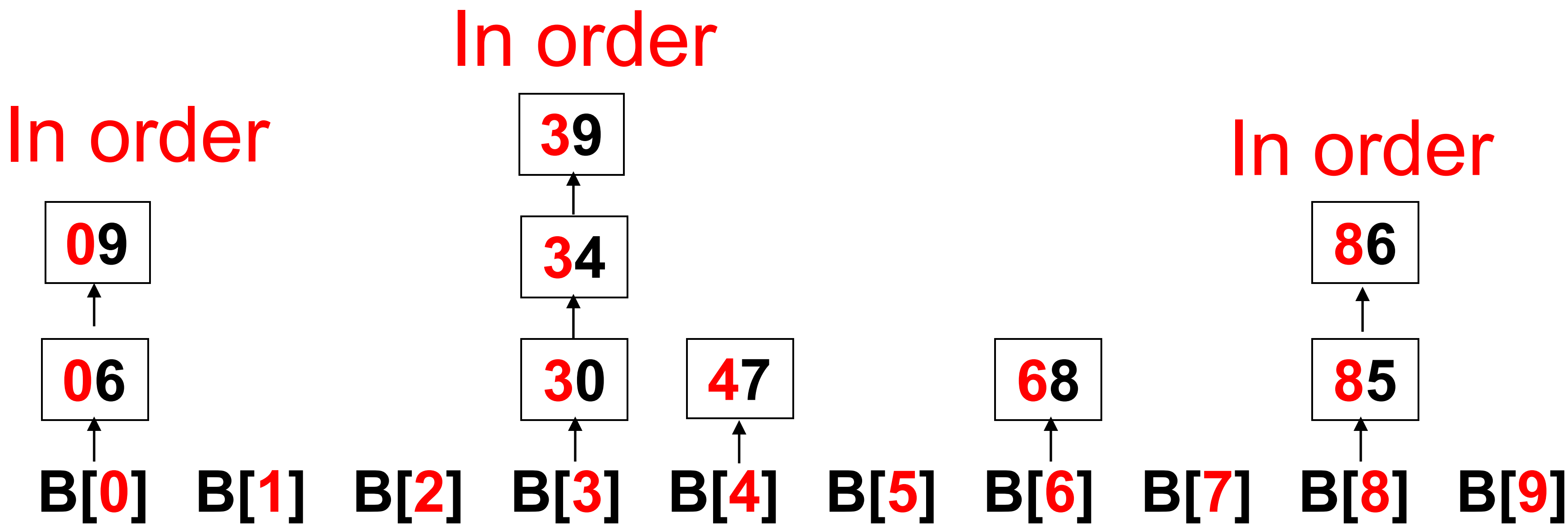
09 85 68 86 47 06 39 34 30



- What if we first sort by the lowest digit
- We then collect the numbers in the bins
- The numbers with the same highest digit are in order, see the numbers with the same color

30 34 85 86 06 47 68 09 39

30 34 85 86 06 47 68 09 39

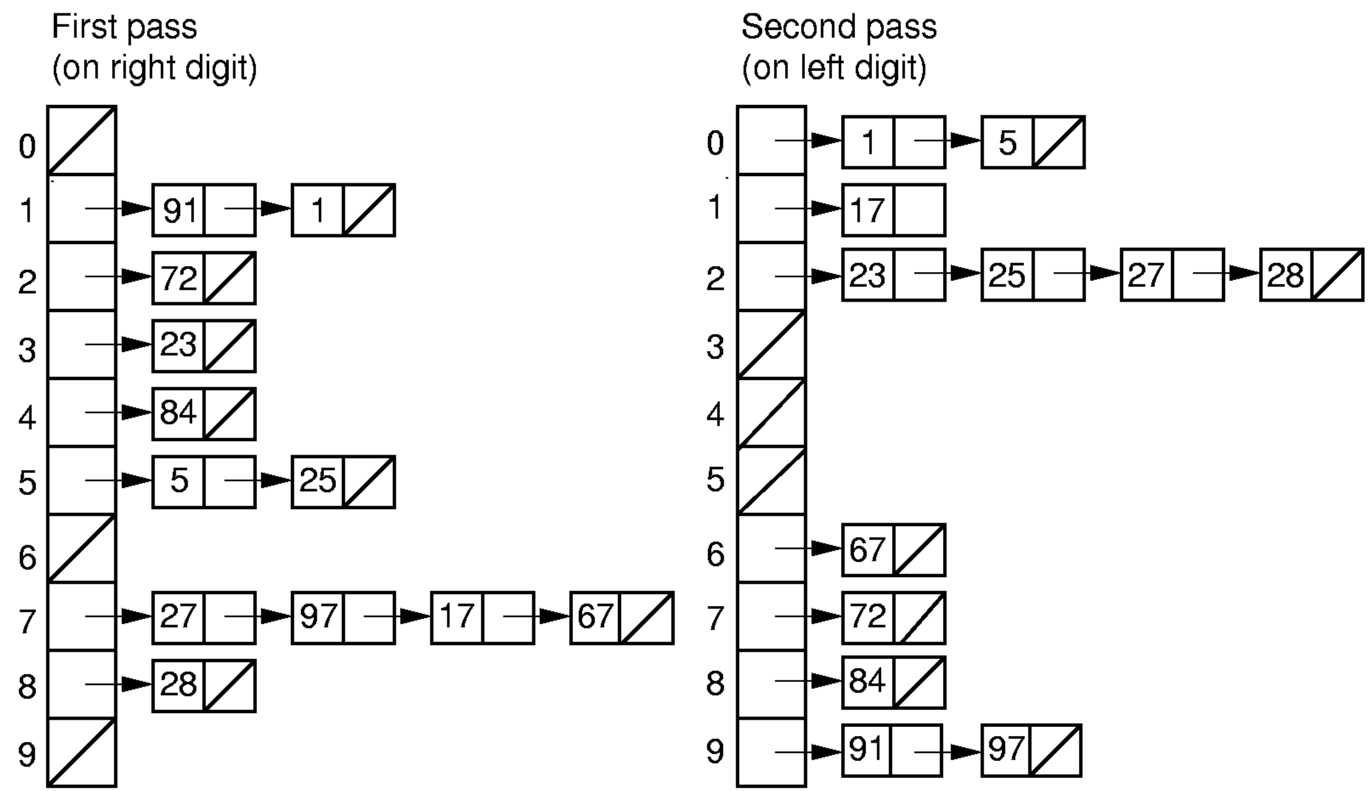


- We then sort the numbers by the highest digit
- Collect the numbers in the bins again
- The numbers are in order now

06 09 30 34 39 47 68 85 89

RadixSort: sort from the lowest digit to the highest digit

Initial List: 27 91 1 97 17 23 84 28 72 5 67 25



Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Radix Sort Cost

- Consider n numbers A[0,1, …,n-1] with radix r, each number has no more than k digits
- Has k BinSorts, from the lowest to the highest
- Each sort takes time $\Theta(n+r)$
- Total Cost: $\Theta(k(n+r))$
- If n numbers are distinct, $k >= \log_r n$
- If r is small, e.g., r=2, radixSort is in $\Theta(n \log n)$
- We usually use large values of r, e.g., r=1K, 1M, or even, n

Running time comparisons (n=3M)

- random input
- RadixSort is faster if r is larger, 10 ≤r≤100k
- But does not improve any more when r approaches to n

```
3000000 numbers to be sorted:
Sort with shellSort, Time consumed:      1.062 (seconds)
Sort with heapSort, Time consumed:       0.953 (seconds)
Sort with mergeSort, Time consumed:      0.438 (seconds)
Sort with quickSort, Time consumed:      0.328 (seconds)
Sort with radixSort (r=10), Time consumed:       0.313 (seconds)
Sort with radixSort (r=100), Time consumed:      0.156 (seconds)
Sort with radixSort (r=1000), Time consumed:     0.141 (seconds)
Sort with radixSort (r=10000), Time consumed:    0.093 (seconds)
Sort with radixSort (r=100000), Time consumed:   0.078 (seconds)
Sort with radixSort (r=1000000), Time consumed:  0.079 (seconds)
```

Running time comparisons (n=3M)

- The input is already sorted
  - quickSort is faster than radixSort

```
3000000 numbers to be sorted:
Sort with shellSort, Time consumed:      0.172 (seconds)
Sort with heapSort, Time consumed:       0.234 (seconds)
Sort with mergeSort, Time consumed:      0.156 (seconds)
Sort with quickSort, Time consumed:      0.031 (seconds)
Sort with radixSort (r=10), Time consumed:       0.329 (seconds)
Sort with radixSort (r=100), Time consumed:      0.156 (seconds)
Sort with radixSort (r=1000), Time consumed:     0.156 (seconds)
Sort with radixSort (r=10000), Time consumed:    0.141 (seconds)
Sort with radixSort (r=100000), Time consumed:   0.109 (seconds)
Sort with radixSort (r=1000000), Time consumed:  0.063 (seconds)
```

- The input is reversely sorted

```
3000000 numbers to be sorted:
Sort with shellSort, Time consumed:      0.281 (seconds)
Sort with heapSort, Time consumed:       0.234 (seconds)
Sort with mergeSort, Time consumed:      0.172 (seconds)
Sort with quickSort, Time consumed:      0.032 (seconds)
Sort with radixSort (r=10), Time consumed:       0.313 (seconds)
Sort with radixSort (r=100), Time consumed:      0.156 (seconds)
Sort with radixSort (r=1000), Time consumed:     0.156 (seconds)
Sort with radixSort (r=10000), Time consumed:    0.156 (seconds)
Sort with radixSort (r=100000), Time consumed:   0.110 (seconds)
Sort with radixSort (r=1000000), Time consumed:  0.078 (seconds)
```

The limitation of RadixSort

- Only applicable to sorting integers
- But inapplicable for
  - real numbers
  - Strings has arbitrarily length
    - E.g., short string `a’, long string `dfdfldlfdfdfldjfdlfjslfjsdfdfdfdoojll’
  - …

Lower Bound for Sorting

- We would like to know a lower bound for all possible sorting algorithms
- Sorting is $O(n \log n)$ (average, worst cases) because we know algorithms with this upper bound, e.g., MergeSort or HeapSort
- Sorting takes $\Omega(n)$ time, as each number must be accessed at least once
- Is there any one better than $\Theta(n \log n)$?
- It is proved that sorting is $\Omega(n \log n)$
- MergeSort and HeapSort are asymptotically optimal !

Chapter III-8. File Processing and External Sorting

Primary vs. Secondary Storage

- Primary storage: Main memory (RAM)
  - volatile, i.e., data is lost if powered off
  - Usually a few GB
  - Expensive (unit: $/MB), fast
- Secondary Storage: Peripheral devices
  - Hard Disk, Solid State Drive (SSD), USB, CD, Tape,…
  - Non-volatile
  - Hundreds of GB, or TB
  - Cheap and slow

Performance Comparisons (typical values)

	Sequential read	seq. write	Random read	Random write
RAM	5 GB/s	4 GB/s	300 MB/s	250 MB/s
Hard Disk	80 MB/s	80 MB/s	0.3 MB/s	0.5 MB/s
SSD	200 MB/s	80 MB/s	25 MB/s	70 MB/s

- Performance of hard disks is terribly poor for random read and write

Golden Rule of File Processing

- Minimize the number of disk accesses!
  - Arrange information so that you get what you want with few disk accesses
    - Store data on adjacent tracks, rather than randomly
  - Arrange information to minimize future disk accesses
    - Cache

External Sorting

- Problem: Sorting data sets too large to fit into main memory.
  - Assume data are stored on disk drive.
- To sort, portions of the data must be brought into main memory, processed, and returned to disk.
- An external sort should minimize disk accesses.

Model of External Computation

- As sequential access is much more efficient than random access to the file
  - adjacent logical blocks of the file must be physically adjacent.

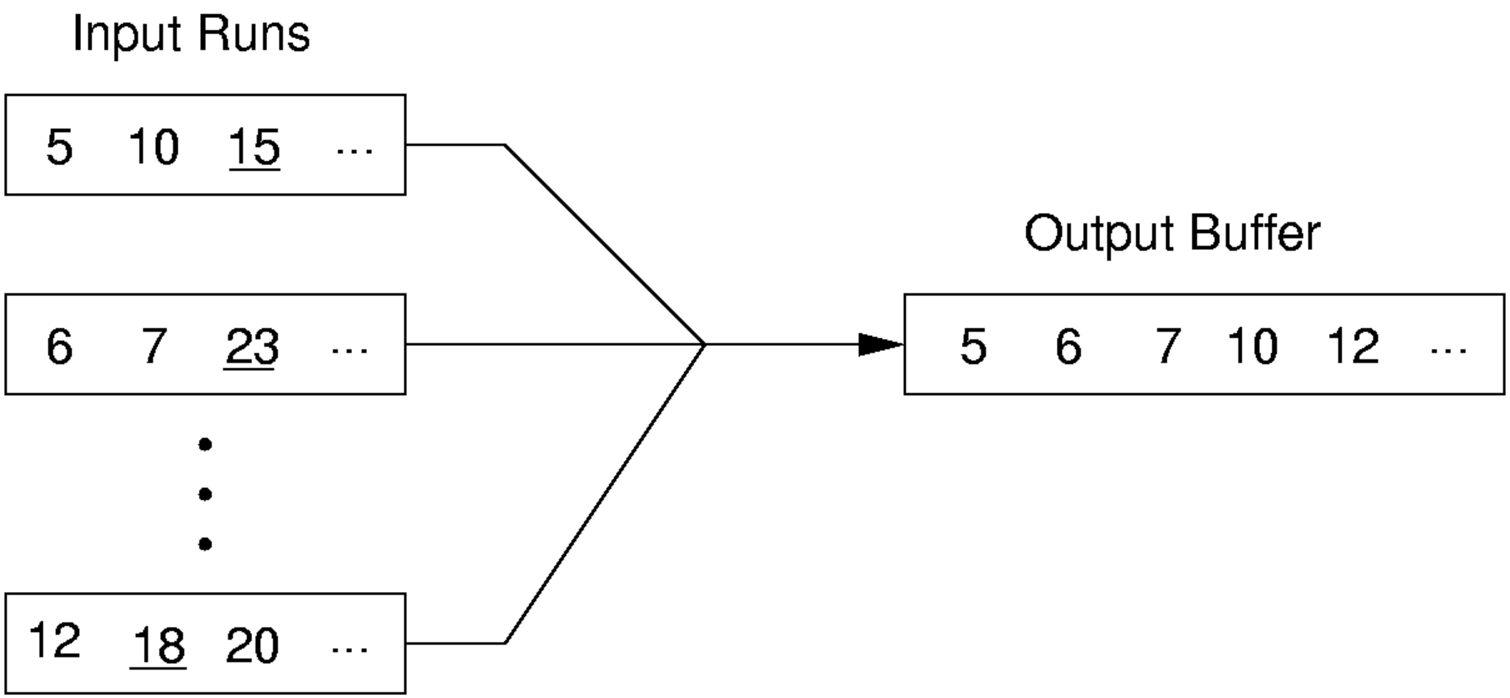
External Sorting

- Three Steps:

1. Break a large file into multiple small initial blocks, so that each block can be fit into memory
   - E.g., break a 10 GB file into 10 blocks with each being 1 GB
2. Sorting the blocks by a fast internal sorting algorithm one by one, and write back to hard disks
3. Merge the sorted blocks together to form a single sorted file.

Multiway Merge

- Merge multiple blocks together, not just two blocks as the internal MergeSort



Summary of Today's Lecture

- Internal sorting
  - THREE basic sorting algorithms ⟵ $\Theta(n^2)$
    - Insertion / bubble / selection sorts
  - ONE medium sorting algorithms ⟵ $\Theta(n^{1.5})$
    - Shell sort
  - THREE fast sorting algorithms ⟵ $\Theta(n \log n)$
    - Merge / quick / heap sorts
  - TWO special cases ⟵ $\Theta(n)$
    - Bin / radix sorts
- External sorting

Homework 3

- See course webpage
- Deadline: midnight before next lecture
- Submit to: cs_scu@foxmail.com
- File name format:
  - CS311_Hw3_yourID_yourLastName.doc (or .pdf)