DATA STRUCTURE AND ALGORITHM

Dr. Khine Thin Zar
Professor
Computer Engineering and Information Technology Dept.
Yangon Technological University

Lecture 7

Internal Sorting

Outlines of Class (Lecture 7)

- Introduction
- Three $\Theta(n^2)$ Sorting Algorithms
  - Insertion Sort
  - Bubble Sort
  - Selection Sort
- Shell Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Binsort and Radix Sort

Introduction

- Arranging the data in ascending or descending order.
- Depends on how many objects we have to sort and how hard they are to move around.
- **Sorting** makes searching easier.
- Using divide-and-conquer.
- There are multiple ways to do the dividing:
  - ✓ Mergesort divides a list in half;
  - ✓ Quicksort divides a list into big values and small values;
  - ✓ Radix Sort divides the problem by working on one digit of the key at a time;
- The two main criterias:
  - ✓ Time taken to sort the given data, and
  - ✓ Memory Space required to do so.

Sorting Terminology and Notation

- Input to the sorting algorithms is a collection of records stored in an array.
- Records are compared to one another (comparator class).
- Swap function: can interchange the contents of two records in the array.
- Each record contains a field called the key.
- The Sorting Problem allows input with two or more records that have the same key value.

Sorting Terminology and Notation (Cont.)

- Definition of the sorting problem:
  - ✓Given a set of records $r_1, r_2, \ldots, r_n$ with key values $k_1, k_2, \ldots, k_n$
  - ✓Arrange the records into any order s such that
  - ✓$r_{s1}, r_{s2}, \ldots, r_{sn}$ have keys obeying the property $k_{s1} \leq k_{s2} \ldots \leq k_{sn}$ or $k_{s1} \geq k_{s2} \ldots \geq k_{sn}$
- Duplicate key values may be allowed
  - ✓After sorting, duplicate keys remain in the order in which they occurred in the input
  - ✓This may be desirable, and the property is called stability

Comparing Performance of Sorting Algorithms

- The running time for many sorting algorithms depends on specifics of the input values:
  - ✓ Problem: running time may depend on specifics of input values
  - ✓ Factors: number of records, key size, record size, range of key values, amount by which records are "out of order"
- Analytically, sorts are usually compared using two measures:
  - ✓ The number of comparisons
  - ✓ The number of swaps
- Common Assumptions
  - ✓ Each sort is passed an array containing the elements
  - ✓ **n** is the number of elements to be sorted
  - ✓ Comparators (e.g. $<$, $\leq$, $>$, $\geq$) can be sorted

Internal Sorting

- Two general types of sorting:
  - ✓Internal Sorting: all elements are sorted in main memory
  - ✓External sorting: all elements are sorted on disk or tape
- Sorting Classifications:
  - ✓Exchange sorts: swapping adjacent records that all run in $O(n^2)$
    - ✓Insertion sort, bubble sort, selection sort
  - ✓Shell sorts: an in-the-middle sort that runs in $O(n^{1.5})$ or $O(n^2)$
  - ✓Efficient sorts that run in O(n log n)
    - ✓Heap sort, merge sort, quick sort
  - ✓Special-purpose sorts that run in quicker time:
    - ✓Bin sort or bucket sort, radix sort

Three $\Theta(n^2)$ Sorting Algorithms

- Three simple sorting algorithms:
  - ✓Insertion Sort
  - ✓Bubble Sort
  - ✓Selection Sort
- While easy to understand and implement, they are unacceptably slow when there are many records to sort.
- There are situations where one of these simple algorithms is the best tool for the job.

Insertion Sort

- Array is divided into two parts - sorted one and unsorted one.
- Sorted part contains first element of the array and unsorted one contains the rest.
- Algorithm takes first element in the unsorted part and inserts it to the right place of the sorted one.
- Insertion Sort iterates through a list of records.
- Each record is inserted in turn at the correct position within a sorted list composed of those records already processed.
- When unsorted part becomes empty, algorithm *stops*.

Insertion Sort (Cont.)

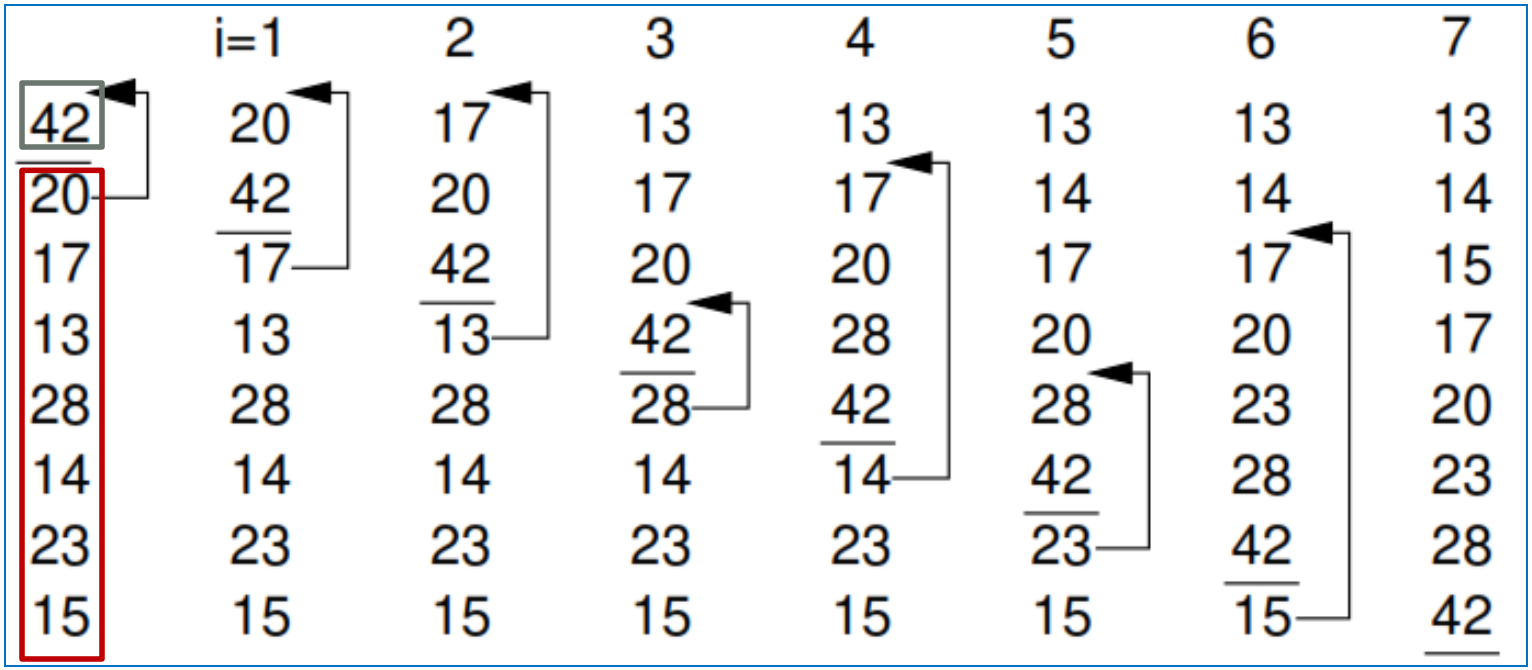


Figure: An illustration of Insertion Sort. Each column shows the array after the iteration with the indicated value of **i** in the outer for loop. Values above the line in each column have been sorted. Arrows indicate the upward motions of records through the array.

C++ Source Code for Insertion Sort (Textbook)

- The input is an array of n records stored in array **A.**

```
template <typename E, typename Comp>
        void inssort(E A[], int n) {   // Insertion Sort
        for (int i=1; i<n; i++)                    // Insert i'th record
            for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
                swap(A, j, j-1);
        }
```

Insertion Sort (Cont.)

- **The outer for loop** is executed n - 1 times.
- **The inner for loop** is harder to analyze because the number of times it executes depends on how many keys in positions **1** to **i – 1** have a value less than that of the key in position i.
  - ✓ When record i is processed, the number of times through the inner for loop depends on how far "out of order" the record is.

- The Best case
  - ✓ This occurs when the keys begin in sorted order from lowest to highest.
  - ✓ In this case, every pass through the inner **for** loop will fail immediately, and no values will be moved.
  - ✓ The total number of comparisons will be n- 1, which is the number of times the outer **for loop** executes.
  - ✓ Thus, the cost for Insertion Sort in the best case is $\Theta(n)$.

Insertion Sort (Cont.)

- The worst case
  - ✓This would occur if the keys are initially arranged from highest to lowest, in the reverse of sorted order.
  - ✓Thus, the total number of comparisons $\sum_{i=2}^{n} i \approx n^2/2 = \Theta(n^2)$.

- The Average Case
  - ✓The average number of inversions will be for the record position i. We expect on average that half of the keys in the first i- 1 array positions will have a value greater than that of the key at position i.
  - ✓Thus, the average case should be about half the cost of the worst case, or around $n^2/4$, which still $\Theta(n^2)$.

Insertion Sort (Cont.)

- Time complexity (number of comparisons)
  - ✓Best Case: $\Theta(n)$
  - ✓Average Case: $\Theta(n^2)$
  - ✓Worst Case : $\Theta(n^2)$

Bubble Sort

- The **bubble sort** makes multiple passes through a list.
- Compares adjacent items and exchanges those that are out of order.
- Each pass through the list places the next largest value in its proper place.
- Each item "bubbles" up to the location where it belongs.
- The inner for loop moves through the record array from bottom to top, comparing adjacent keys.
- If the lower-indexed key's value is greater than its higher-indexed neighbour, then the two values are swapped.
- Once the smallest value is encountered, this process will cause it to "bubble" up to the top of the array.
- The second pass through the array repeats its process.

Bubble Sort (Cont.)

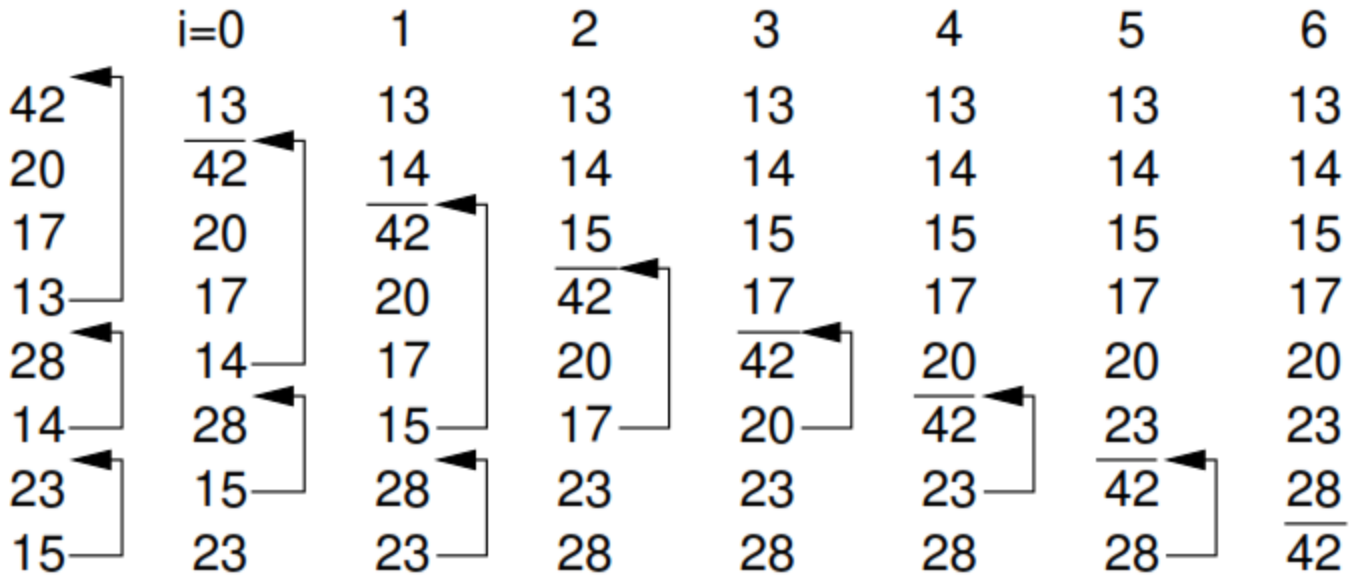


Figure: An illustration of Bubble Sort. Each column shows the array after the iteration with the indicated value of i in the outer for loop. Values above the line in each column have been sorted. Arrows indicate the swaps that take place during a given iteration.

C++ Source Code for Bubble Sort (Textbook)

```
template <typename E, typename Comp>
        void bubsort(E A[], int n) {            // Bubble Sort
        for (int i=0; i<n-1; i++)        // Bubble up i'th record
           for (int j=n-1; j>i; j--)
                 if (Comp::prior(A[j], A[j-1]))
                          swap(A, j, j-1);
                 }
```

Bubble Sort (Cont.)

- Bubble Sort's running time is roughly the same in the best, average, and worst cases.
- Time complexity (number of comparisons)
  - ✓Best Case: $\Theta(n^2)$
  - ✓Average Case: $\Theta(n^2)$
  - ✓Worst Case : $\Theta(n^2)$

Selection Sort

- Divided into two parts - sorted part and unsorted part.
- At the beginning, sorted part is empty, and unsorted part contains whole array.
- Algorithm finds minimal element in the unsorted part and adds it to the end of the sorted part.
- When unsorted part becomes empty, algorithm stops.
- To find the next smallest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record in place.
- The total number of swaps require n-1.
- The number of comparisons is $\Theta(n^2)$
- More efficient than Bubble sort.

Selection Sort (Contd.)

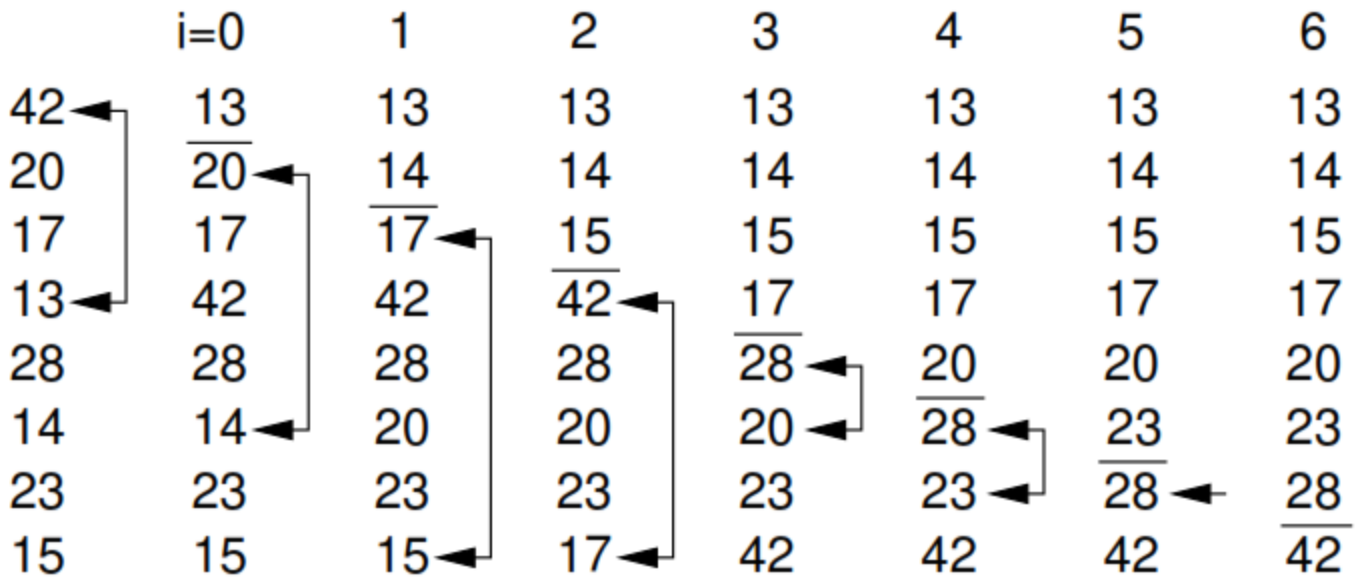


Figure: An example of Selection Sort. Each column shows the array after the iteration with the indicated value of i in the outer for loop. Numbers above the line in each column have been sorted and are in their final positions.

C++ Source Code for Selection Sort (Textbook)

```
template <typename E, typename Comp>
        void selsort(E A[], int n) {       // Selection Sort
        for (int i=0; i<n-1; i++) {        // Select i'th record
            int lowindex = i;               // Remember its index
            for (int j=n-1; j>i; j--)       // Find the least value
                  if (Comp::prior(A[j], A[lowindex]))
                     lowindex = j;                    // Put it in place
                   swap(A, i, lowindex);
}
```

Selection Sort (Cont.)

The number of comparisons is still $\Theta(n^2)$

- Time complexity (number of comparisons)
  - ✓Best Case: $\Theta(n^2)$
  - ✓Average Case: $\Theta(n^2)$
  - ✓Worst Case : $\Theta(n^2)$

The Cost of Exchange Sorting

- Swapping adjacent records is called an exchange.
- These sorts are sometimes referred to as exchange sorts.

	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	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Figure: A comparison of the asymptotic complexities for three simple sorting algorithms.

Swapping Pointers to Records

- There is another approach to keeping the cost of swapping records low that can be used by any sorting algorithm even when the records are large.
- This is to have each element of the array store a pointer to a record rather than store the record itself.
- In this implementation, a swap operation need only exchange the pointer values; the records themselves do not move.
- Additional space is needed to store the pointers, but the return is a faster swap operation.

Swapping Pointers to Records (Cont.)

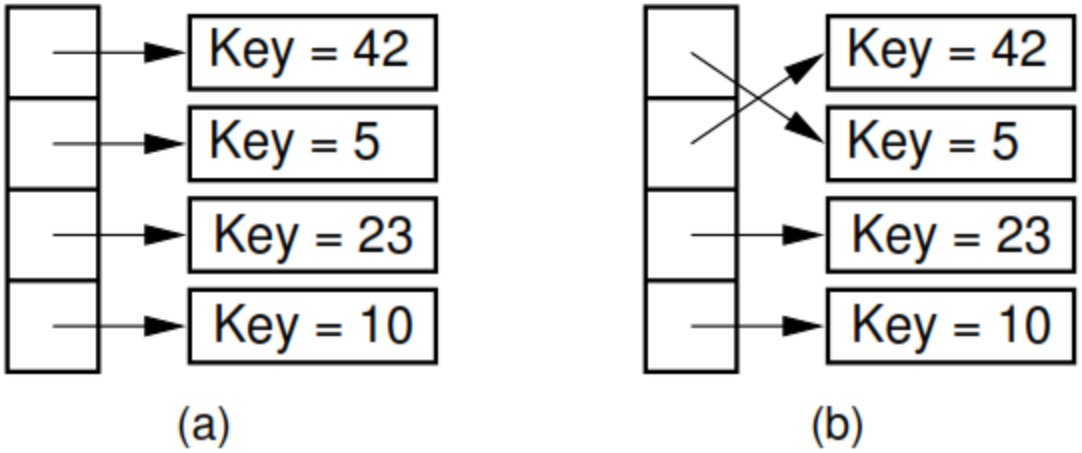
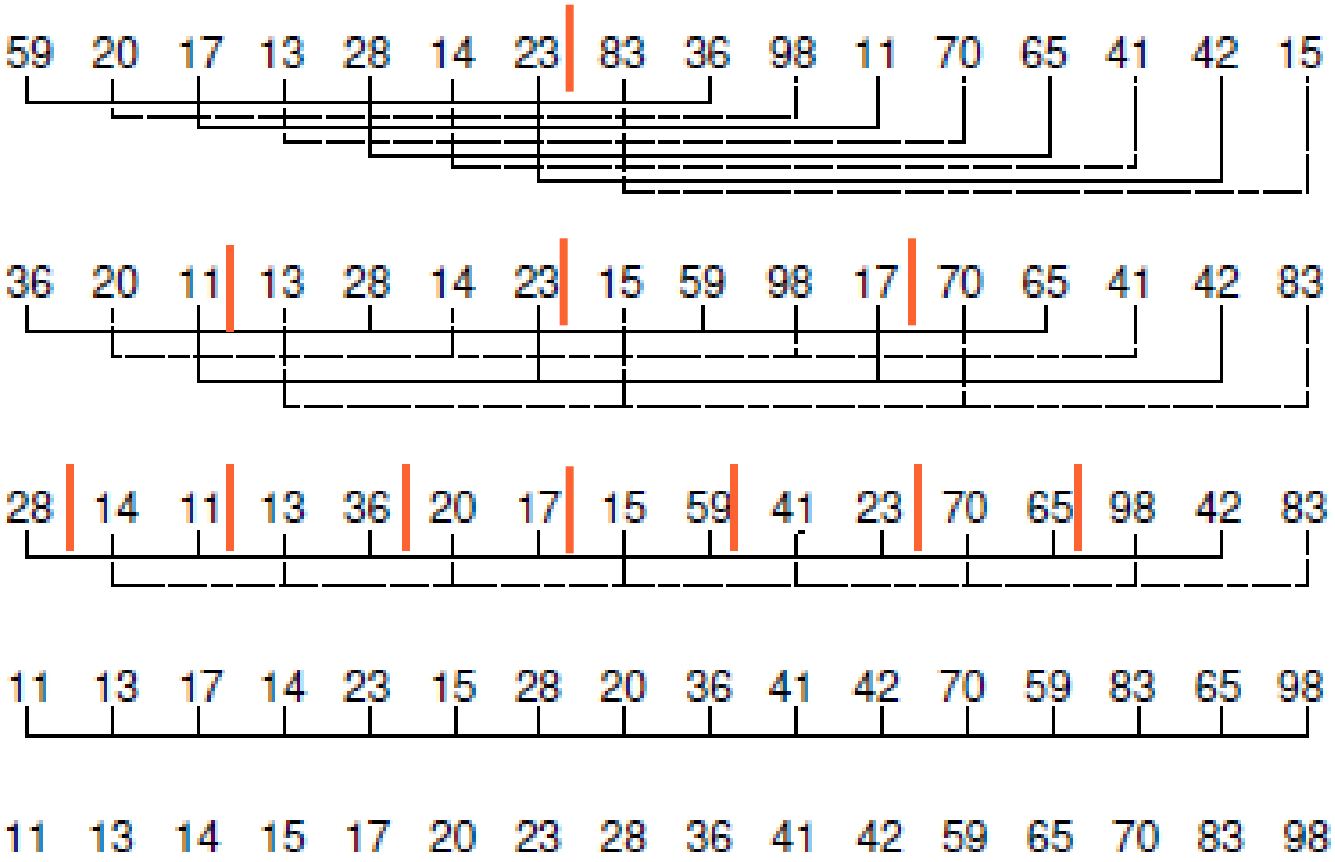


Figure: An example of swapping pointers to records.
(a) A series of four records. The record with key value 42 comes before the record with key value 5.
(b) The four records after the top two pointers have been swapped. Now the record with key value 5 comes before the record with key value 42.

Shell Sort

- Makes comparisons and swaps between non-adjacent elements.
  - ✓Break the list into sublists, sort them, then recombine the sublists.
  - ✓ Shellsort breaks the array of elements into "virtual" sublists.
  - ✓Each sublist is sorted using an Insertion Sort.
  - ✓ Another group of sublists is then chosen and sorted, and so on.
  - ✓Instead of breaking the list into sublists of contiguous items, the shell sort uses an increment i (sometimes called gap=n/2), to create a sublist by choosing all items that are i items apart.

How shell sort works



n = 16,
Gap =16/2
=8

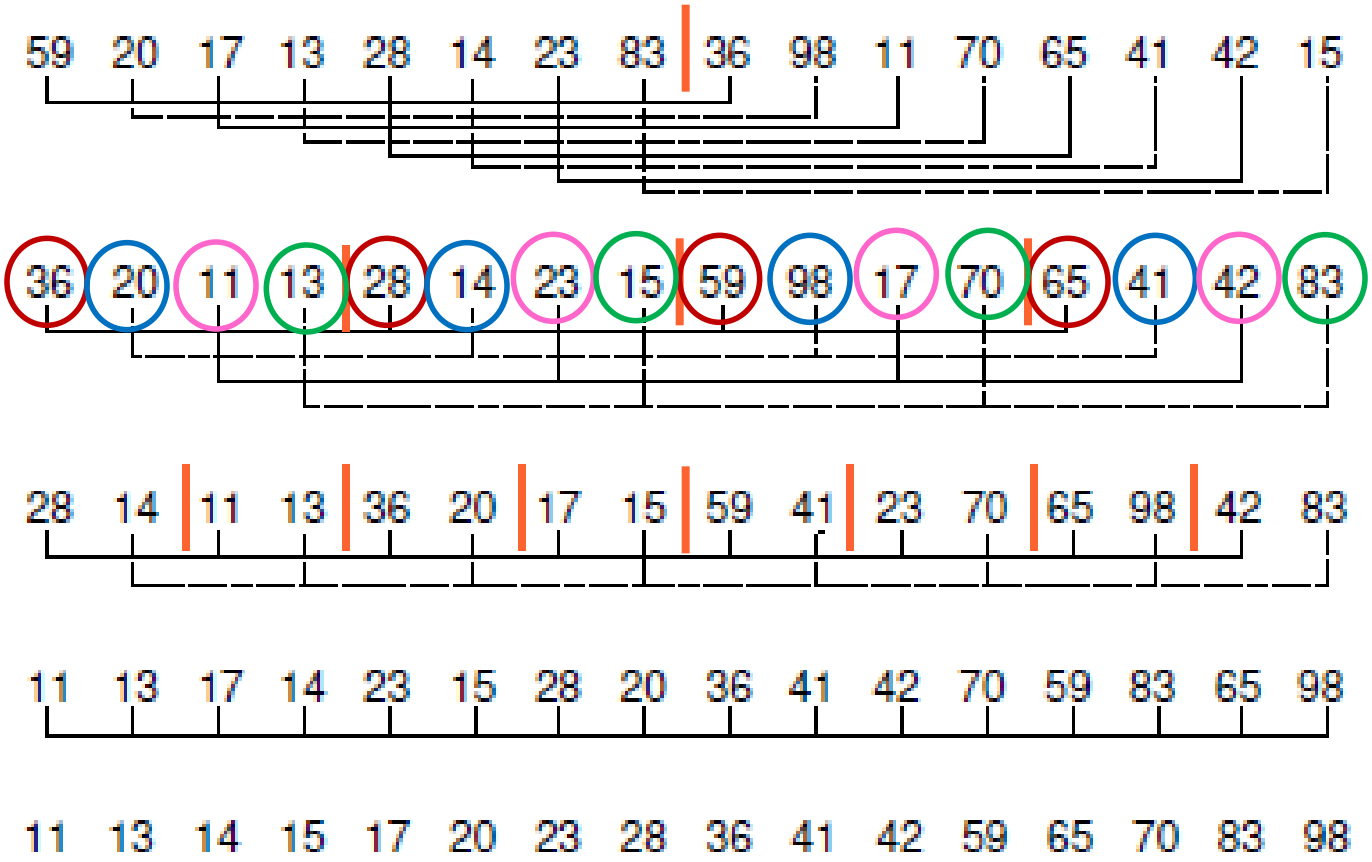
n = 8,
Gap =8/2
=4

n = 4,
Gap =4/2
=2

n = 2,
Gap =2/2
=1

We compare values in each sub-list and swap them (if necessary) in the original array.

How shell sort works



n = 16,
Gap =16/2
=8

n = 8,
Gap =8/2
=4

n = 4,
Gap =4/2
=2

n = 2,
Gap =2/2
=1

We compare values in each sub-list and swap them (if necessary) in the original array.

How Shell Sort Works (Cont.)

- During each iteration, Shellsort breaks the list into disjoint sublists so that each element in a sublist is a fixed number of positions apart.
- Breaking the list into n/2 sublists (n: the number of values to be sorted) of 2 elements each, where the array index of the 2 elements in each sublist differs by n/2.
- If there are 16 elements in the array indexed from 0 to 15, there would initially be 8 sublists of 2 elements each.
- The first sublist would be the elements in positions 0 and 8, the second in positions 1 and 9, 2 and 10, and so on.
- Each list of two elements is sorted using Insertion Sort.

Shell Sort (Cont.)

- The second pass would have n/4 lists of size 4, with the elements in the list being n/4 positions apart.
- Thus, the second pass would have as its first sublist the 4 elements in positions 0, 4, 8, and 12; the second sublist would have elements in positions 1, 5, 9, and 13; and so on.
- Each sublist of four elements would also be sorted using an Insertion Sort.
- Figure illustrates the process for an array of 16 values where the sizes of the increments (the distances between elements on the successive passes) are 8, 4, 2, and 1.

Merge Sort

- Merge-sort is based on an algorithmic design pattern called divide-and-conquer.
- The divide-and-conquer pattern consists of the following three steps:
  - ✓ Divide: If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution obtained. Otherwise, divide the input data into two or more disjoint subsets.
  - ✓ Recur: Recursively solve the subproblems associated with the subsets.
  - ✓ Conquer: Take the solutions to the subproblems and "merge" them into a solution to the original problem.

Merge Sort (Cont.)

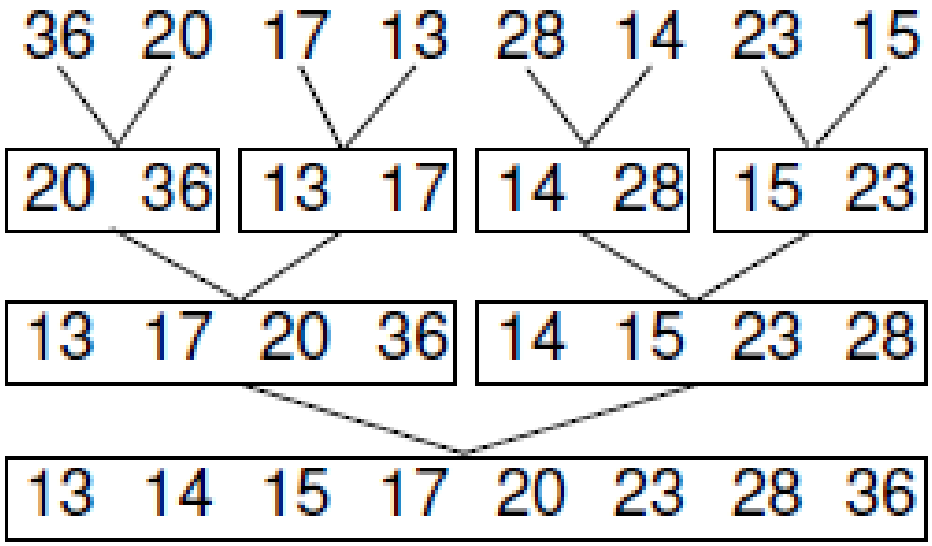


Figure: An illustration of Mergesort. The first row shows eight numbers that are to be sorted. Mergesort will recursively subdivide the list into sublists of one element each, then recombine the sublists. The second row shows the four sublists of size 2 created by the first merging pass. The third row shows the two sublists of size 4 created by the next merging pass on the sublists of row 2. The last row shows the final sorted list created by merging the two sublists of row 3.

Merge Sort (Cont.)

- Mergesort is recursively called until subarrays of size 1 have been created.
- These subarrays are merged into subarrays of size 2, which are in turn merged into subarrays of size 4, and so on.
- Mergesort is a recursive algorithm.
- The first level of recursion can be thought of as working on one array of size n, the next level working on two arrays of size n/2, the next on four arrays of size n/4, and so on.
- The bottom of the recursion has n arrays of size 1.

Merge Sort (Cont.)

- A pseudocode of Mergesort :

```
List mergesort(List inlist) {
  if (inlist.length() <= 1) return inlist;;
  List L1 = half of the items from inlist;
  List L2 = other half of the items from inlist;
  return merge(mergesort(L1), mergesort(L2));
}
```

Running Time of Merge-Sort

- At each level in the binary tree created for Merge Sort, there are n elements, with O(1) time spent at each element
  → O(n) running time for processing one level
- The height of the tree is O(log n)

- Therefore, the time complexity is $O(n \log n)$

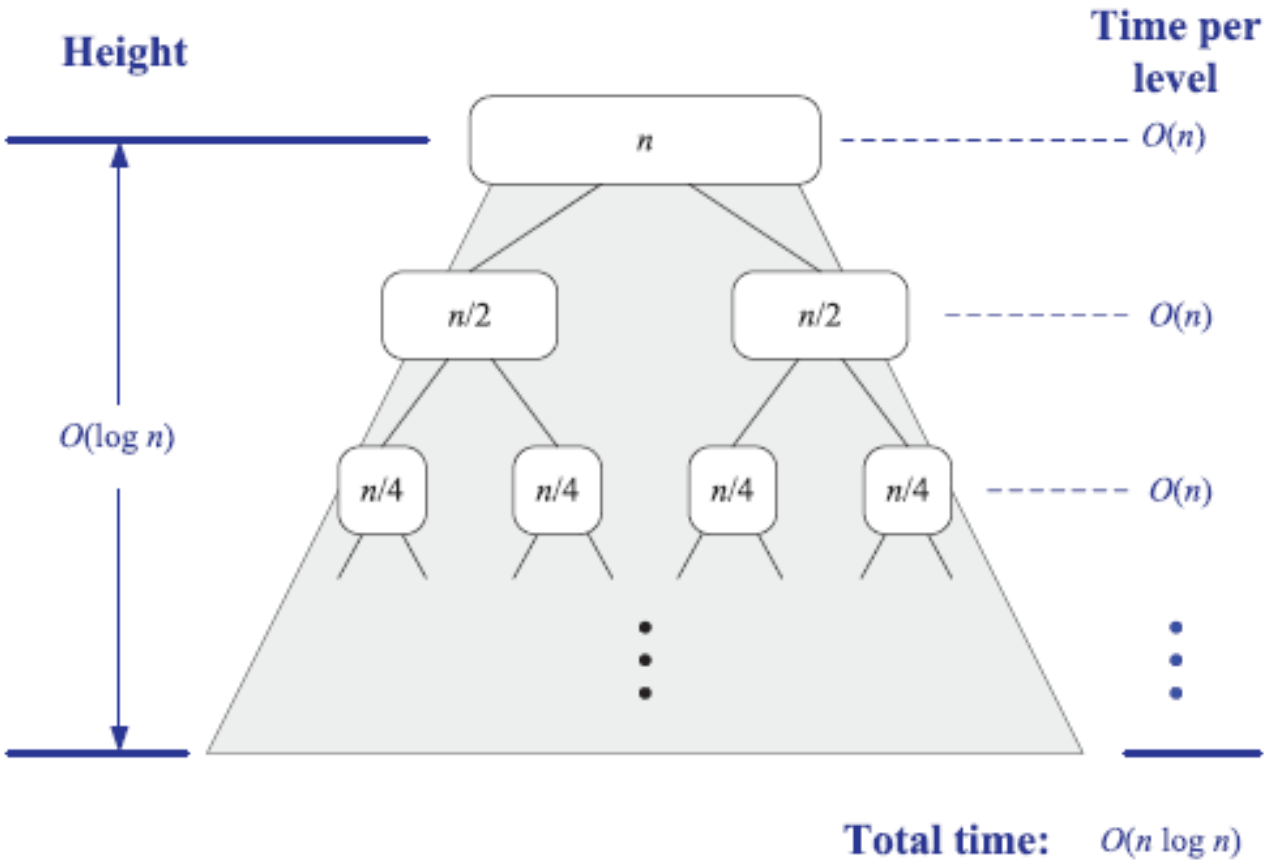


Figure: A visual time analysis of the merge-sort tree *T*. Each node is shown labeled with the size of its subproblem.

Quick Sort

- Approach Method: Divide and Conquer
- An array of n elements (e.g., integers):
- If array only contains one element, return
- Else
  - Selects one element to use as pivot (simplest is to use the first key)
  - Partition elements into two sub-arrays:
    - Elements less than or equal to pivot
    - Elements greater than pivot
  - Quicksort two sub-arrays
  - Return results

Quick Sort (Cont.)

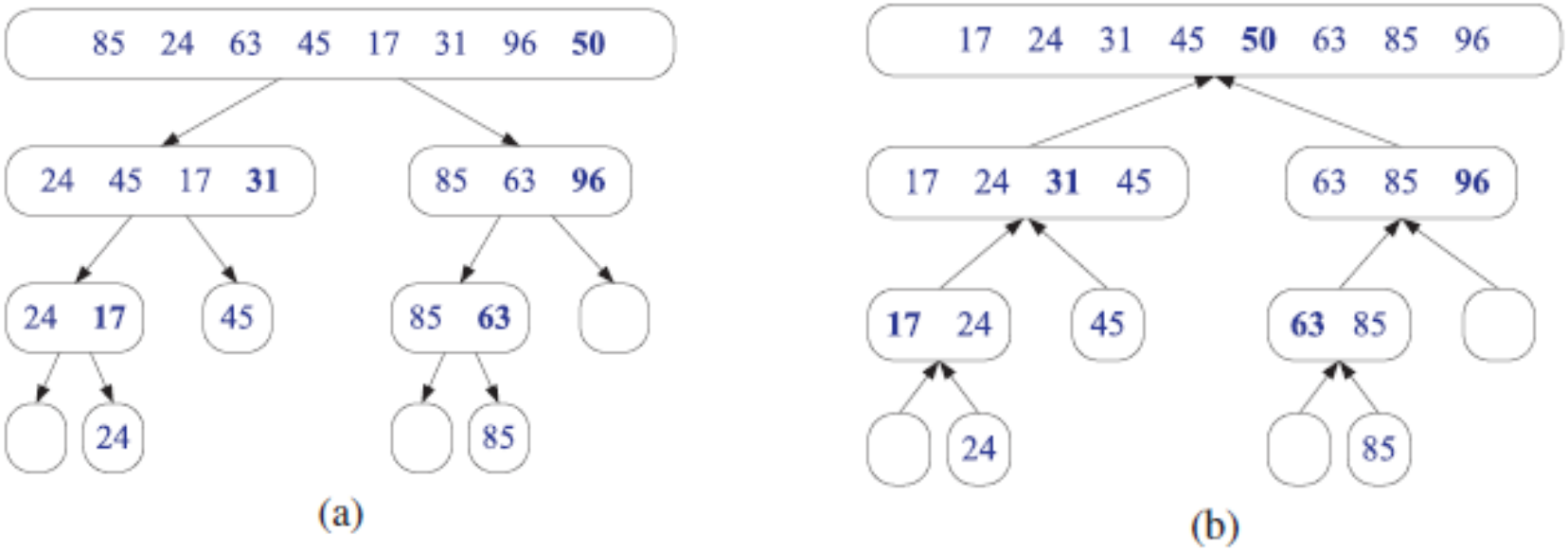


Figure: Quick-sort tree *T* for an execution of the quick-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of *T*; (b) output sequences generated at each node of *T*. The pivot used at each level of the recursion is shown in bold.

Quick Sort Running Time

- General case:
  - ✓Time spent at level i in the tree is O(n)
  - ✓Running time: O(n) * O(height = log n)
- Average case:
  - ✓O(n log n)

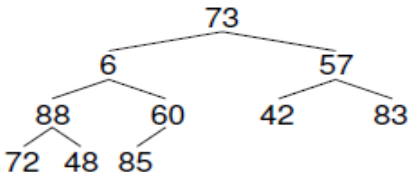
Heap Sort

- Based on max-heaps.
- Convert the array into max-heap order.
- Repeatedly remove the maximum value from the heap, restoring the heap
- Property each time that we do so, until the heap is empty
- Remove the maximum element from the heap, it is placed at the end of the array

Heap Sort (cont.)

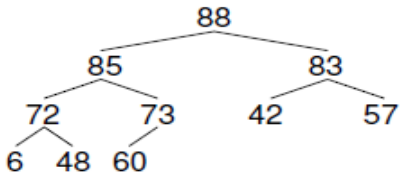
Original Numbers

73	6	57	88	60	42	83	72	48	85



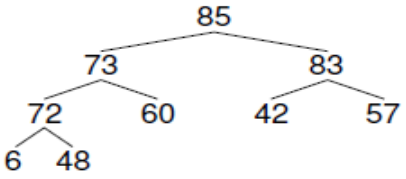
Build Heap

88	85	83	72	73	42	57	6	48	60



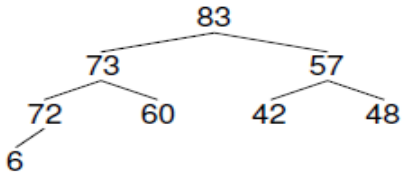
Remove 88

85	73	83	72	60	42	57	6	48	88



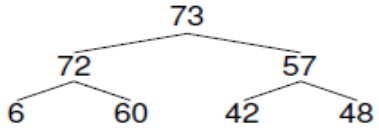
Remove 85

83	73	57	72	60	42	48	6	85	88



Remove 83

73	72	57	6	60	42	48	83	85	88



Sorted Numbers

6	42	48	57	60	72	73	83	85	88

Binsort (Bucket Sort) and Radix Sort

- Non comparative sorting methods
- The elements are put in buckets (or bins) of incremental ranges (e.g. 0-10, 11-20, ... 90-100), depending on the number of digits in the largest number.
- In the next pass,
  - ✓Bucket Sort orders up these 'buckets' and appends them into one array.
  - ✓Radix Sort appends the buckets without further sorting and 're-buckets' it based on the second digit (ten's place) of the numbers.
  - ✓First sort by digit in units place
  - ✓Second sort by digit in tens place
  - ✓Third sort by digit in hundreds place

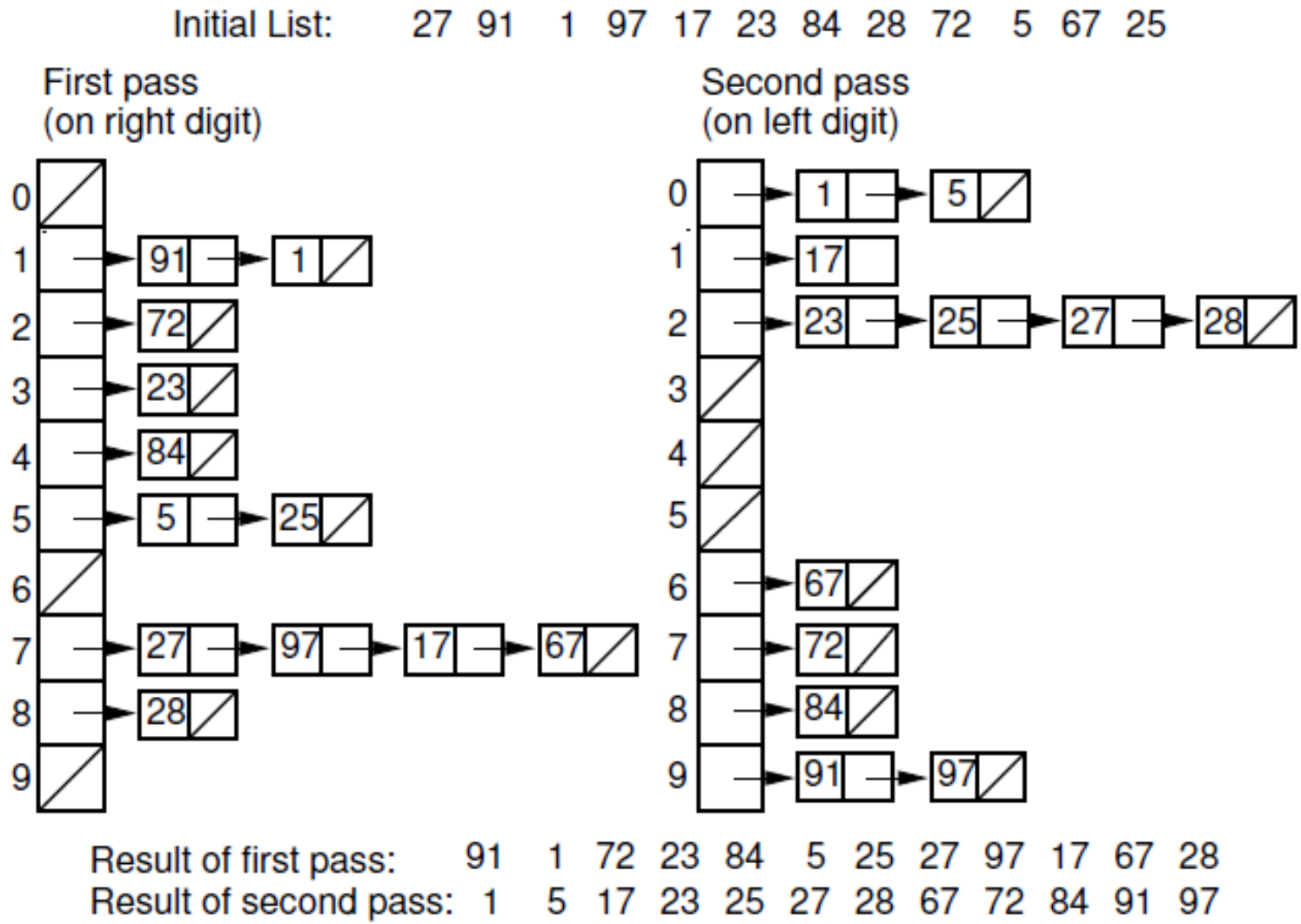


Figure: An example of Radix Sort for twelve two-digit numbers in base ten. Two passes are required to sort the list.

Initial Input: Array A

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

First pass values for Count.
rtoi = 1.

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

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

End of Pass 1: Array A.

91	1	72	23	84	5	25	27	97	17	67	28
0	1	2	3	4	5	6	7	8	9	10	11

Second pass values for Count.
rtoi = 10.

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

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

End of Pass 2: Array A.

1	5	17	23	25	27	28	67	72	84	91	97
0	1	2	3	4	5	6	7	8	9	10	11

Next Week Lecture (Week 8)

Lecture 8: File processing and External Sorting

Thank you!