BIRZEIT UNIVERSITY
Faculty of Engineering and Technology كليـة الهنـدسـة والتكنـولوجيـا
Computer Science Department دائـرة علـم الحـاسـوب

Sorting Algorithms

Compiled and Prepared by:
Dr. Majdi M. Mafarja

Sorting

- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.
- An ***internal sort*** requires that the collection of data fit entirely in the computer's main memory.
- We can use an ***external sort*** when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
- We will analyze only internal sorting algorithms.
- Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.
- Sorting also has indirect uses. An initial sort of the data can significantly enhance the performance of an algorithm.
- Majority of programming projects use a sort somewhere, and in many cases, the sorting cost determines the running time.
- A comparison-based sorting algorithm makes ordering decisions only on the basis of comparisons.

Sorting Algorithms

- There are many sorting algorithms, such as:
  - – Selection Sort
  - – Insertion Sort
  - – Bubble Sort
  - – Merge Sort
  - – Quick Sort
  - – Shell Srot

3

Selection Sort

- The list is divided into two sublists, *sorted* and *unsorted*, which are divided by an imaginary wall.
- We find the smallest element from the unsorted sublist and swap it with the element at the beginning of the unsorted data.
- After each selection and swapping, the imaginary wall between the two sublists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time we move one element from the unsorted sublist to the sorted sublist, we say that we have completed a sort pass.
- A list of *n* elements requires *n-1* passes to completely rearrange the data.

4

Sorted **Unsorted**

23	78	45	8	32	56	Original List
8	78	45	23	32	56	After pass 1
8	23	45	78	32	56	After pass 2
8	23	32	78	45	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

5

Example

8	4	6	9	2	3	1
1	4	6	9	2	3	8
1	2	6	9	4	3	8
1	2	3	9	4	6	8

1	2	3	4	9	6	8
1	2	3	4	6	9	8
1	2	3	4	6	8	9
1	2	3	4	6	8	9

6

Selection Sort (cont.)

```
template <class Item>
void selectionSort( Item a[], int n) {
  for (int i = 0; i < n-1; i++) {
    int min = i;
    for (int j = i+1; j < n; j++)
       if (a[j] < a[min]) min = j;
    swap(a[i], a[min]);
  }
}

template < class Object>
void swap( Object &lhs, Object &rhs )
{
  Object tmp = lhs;
  lhs = rhs;
  rhs = tmp;
}
```

7

Selection Sort -- Analysis

- In general, we compare keys and move items (or exchange items) in a sorting algorithm (which uses key comparisons).

  ➔ **So, to analyze a sorting algorithm we should count the number of key comparisons and the number of moves.**
  - Ignoring other operations does not affect our final result.

- In selectionSort function, the outer for loop executes n-1 times.
- We invoke swap function once at each iteration.

  ➔ Total Swaps: n-1

  ➔ Total Moves: 3*(n-1) (Each swap has three moves)

8

Selection Sort – Analysis (cont.)

- The inner for loop executes the size of the unsorted part minus 1 (from 1 to n-1), and in each iteration we make one key comparison.
  - ➔ # of key comparisons = 1+2+...+n-1 = n*(n-1)/2
  - ➔ **So, Selection sort is $O(n^2)$**
- The best case, the worst case, and the average case of the selection sort algorithm are same. ➔ all of them are **$O(n^2)$**
  - This means that the behavior of the selection sort algorithm does not depend on the initial organization of data.
  - Since $O(n^2)$ grows so rapidly, the selection sort algorithm is appropriate only for small n.
  - Although the selection sort algorithm requires $O(n^2)$ key comparisons, it only requires O(n) moves.
  - A selection sort could be a good choice if data moves are costly but key comparisons are not costly (short keys, long records).

9

Comparison of *N*, *logN* and *N²*

N	O(LogN)	$O(N^2)$
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

10

Insertion Sort

- Insertion sort is a simple sorting algorithm that is appropriate for small inputs.
  - Most common sorting technique used by card players.
- The list is divided into two parts: sorted and unsorted.
- In each pass, the first element of the unsorted part is picked up, transferred to the sorted sublist, and inserted at the appropriate place.
- A list of *n* elements will take at most *n-1* passes to sort the data.

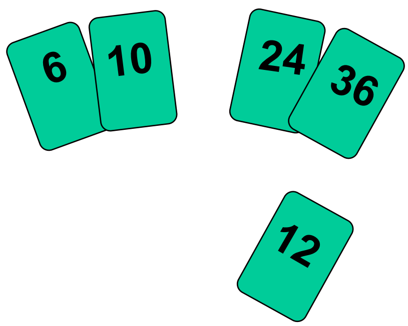
11

Sorted **Unsorted**

23	78	45	8	32	56	Original List
23	78	45	8	32	56	After pass 1
23	45	78	8	32	56	After pass 2
8	23	45	78	32	56	After pass 3
8	23	32	45	78	56	After pass 4
8	23	32	45	56	78	After pass 5

12

Insertion Sort … Example



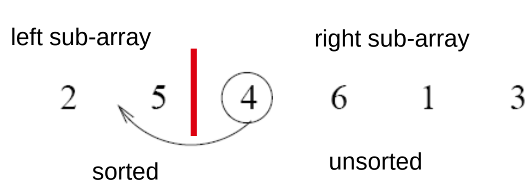
13

Insertion Sort

input array

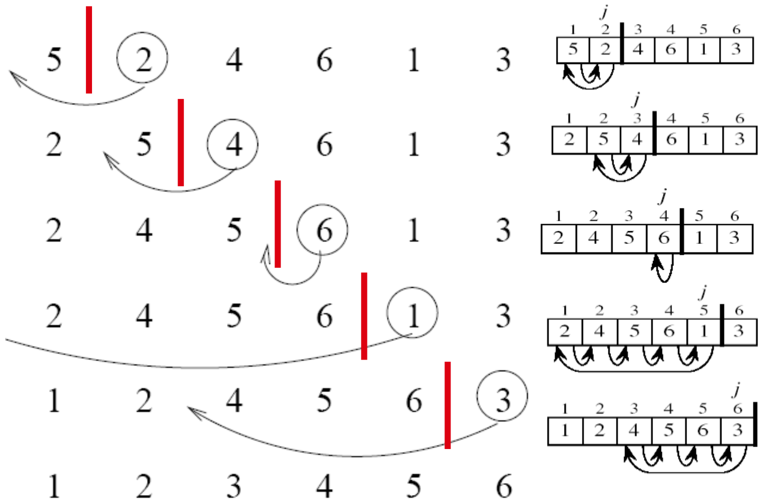
5 2 4 6 1 3

at each iteration, the array is divided in two sub-arrays:



14

Insertion Sort… Example



15

Insertion Sort Algorithm

```
void insertionSort(Item a[], int n)
{
   for (int i = 1; i < n; i++)
   {
      Item tmp = a[i];

      for (int j=i; j>0 && tmp < a[j-1]; j--)
         a[j] = a[j-1];
      a[j] = tmp;
   }
}
```

16

Insertion Sort – Analysis

- Running time depends on not only the size of the array but also the contents of the array.
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - Inner loop will not be executed.
  - The number of moves: 2*(n-1) ➔ O(n)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **$O(n^2)$**
  - Array is in reverse order:
  - Inner loop is executed i-1 times, for i = 2,3, …, n
  - The number of moves: 2*(n-1)+(1+2+...+n-1)= 2*(n-1)+ n*(n-1)/2 ➔ $O(n^2)$
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ $O(n^2)$
- Average-case: ➔ **$O(n^2)$**
  - We have to look at all possible initial data organizations.
- **So, Insertion Sort is $O(n^2)$**

17

Analysis of insertion sort

- Which running time will be used to characterize this algorithm?
  - Best, worst or average?
- Worst:
  - Longest running time (this is the upper limit for the algorithm)
  - It is guaranteed that the algorithm will not be worse than this.
- Sometimes we are interested in average case. But there are some problems with the average case.
  - It is difficult to figure out the average case. i.e. what is average input?
  - Are we going to assume all possible inputs are equally likely?
  - In fact for most algorithms average case is same as the worst case.

18

Bubble Sort

- The list is divided into two sublists: sorted and unsorted.
- The smallest element is bubbled from the unsorted list and moved to the sorted sublist.
- After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
- Given a list of n elements, bubble sort requires up to n-1 passes to sort the data.

19

Bubble Sort

23	78	45	8	32	56	Original List
8	23	78	45	32	56	After pass 1
8	23	32	78	45	56	After pass 2
8	23	32	45	78	56	After pass 3
8	23	32	45	56	78	After pass 4

20

Bubble Sort Algorithm

```
template <class Item>
void bubleSort(Item a[], int n)
{
   bool sorted = false;
   int last = n-1;

   for (int i = 0; (i < last) && !sorted; i++){
      sorted = true;
      for (int j=last; j > i; j--)
         if (a[j-1] > a[j]{
            swap(a[j],a[j-1]);
            sorted = false; // signal exchange
         }
    }
}
```

21

Bubble Sort – Analysis

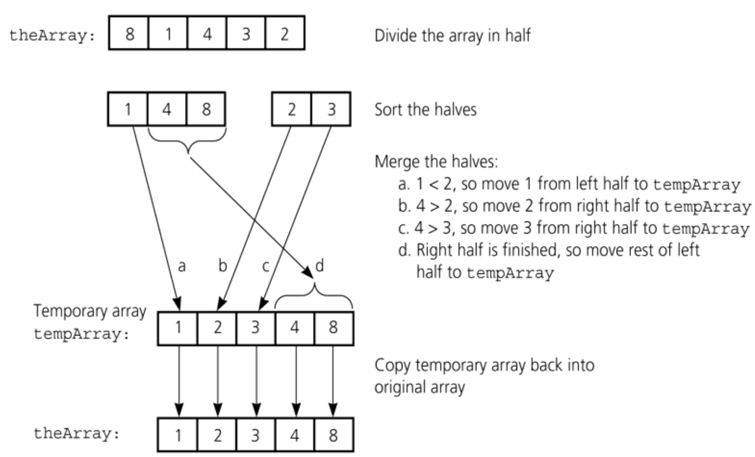
- ***Best-case:*** ➔ **O(n)**
  - Array is already sorted in ascending order.
  - The number of moves: 0 ➔ O(1)
  - The number of key comparisons: (n-1) ➔ O(n)
- ***Worst-case:*** ➔ **O(n^2)**
  - Array is in reverse order:
  - Outer loop is executed n-1 times,
  - The number of moves: 3*(1+2+...+n-1) = 3 * n*(n-1)/2 ➔ O(n^2)
  - The number of key comparisons: (1+2+...+n-1)= n*(n-1)/2 ➔ O(n^2)
- Average-case: ➔ **O(n^2)**
  - We have to look at all possible initial data organizations.
- **So, Bubble Sort is O(n^2)**

22

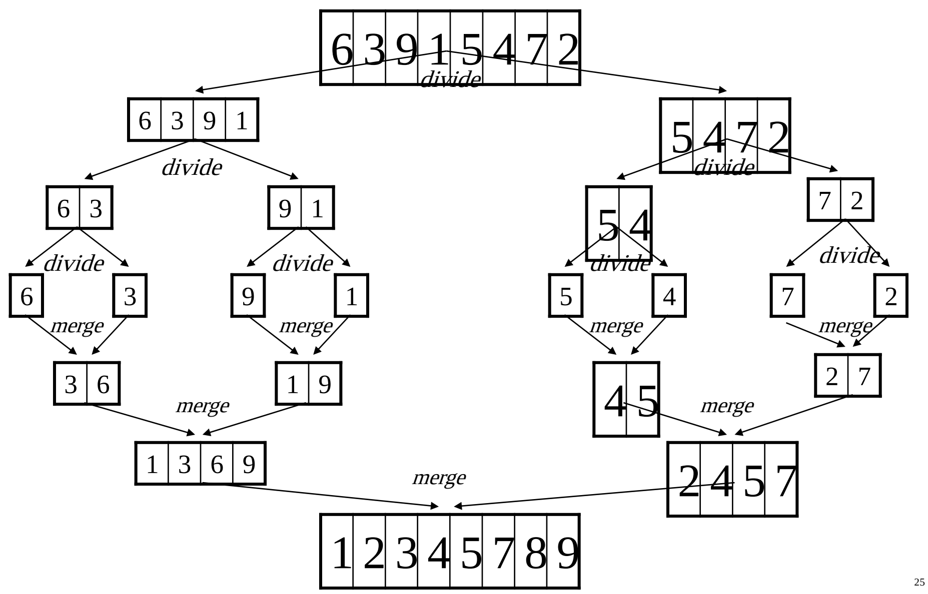
Mergesort

- Mergesort algorithm is one of two important divide-and-conquer sorting algorithms (the other one is quicksort).
- It is a recursive algorithm.
  - Divides the list into halves,
  - Sort each halve separately, and
  - Then merge the sorted halves into one sorted array.

Mergesort - Example



Mergesort - Example



25

Merge

```
const int MAX_SIZE = maximum-number-of-items-in-array;
void merge(DataType theArray[], int first, int mid, int last)
  {
   DataType tempArray[MAX_SIZE];  // temporary array
   int first1 = first;     // beginning of first subarray
   int last1 = mid;        // end of first subarray
   int first2 = mid + 1;   // beginning of second subarray
   int last2 = last;       // end of second subarray
   int index = first1; // next available location in tempArray
   for ( ; (first1 <= last1) && (first2 <= last2); ++index) {
      if (theArray[first1] < theArray[first2]) {
         tempArray[index] = theArray[first1];
         ++first1;
      }
      else {
          tempArray[index] = theArray[first2];
          ++first2;
      } }
```

26

Merge (cont.)

```
   // finish off the first subarray, if necessary
   for (; first1 <= last1; ++first1, ++index)
      tempArray[index] = theArray[first1];

   // finish off the second subarray, if necessary
   for (; first2 <= last2; ++first2, ++index)
      tempArray[index] = theArray[first2];

   // copy the result back into the original array
   for (index = first; index <= last; ++index)
      theArray[index] = tempArray[index];
}  // end merge
```

27

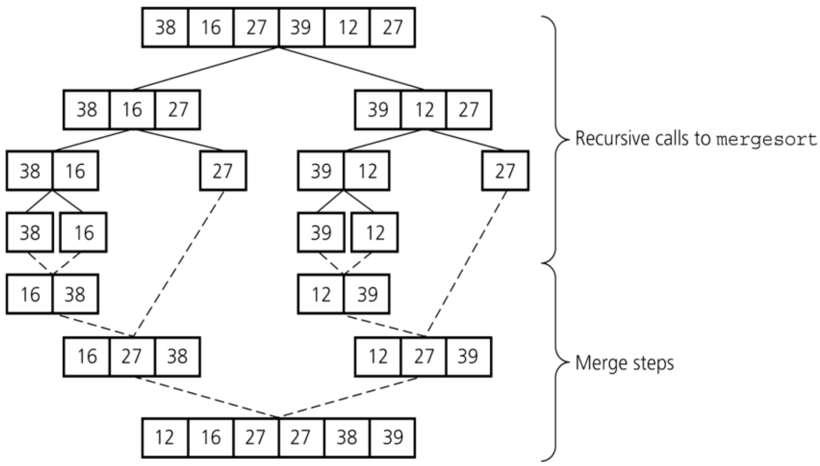
Mergesort

```
void mergesort(DataType theArray[], int first, int last) {
   if (first < last) {
      int mid = (first + last)/2;       // index of midpoint
      mergesort(theArray, first, mid);
      mergesort(theArray, mid+1, last);

      // merge the two halves
      merge(theArray, first, mid, last);
   }
}  // end mergesort
```

28

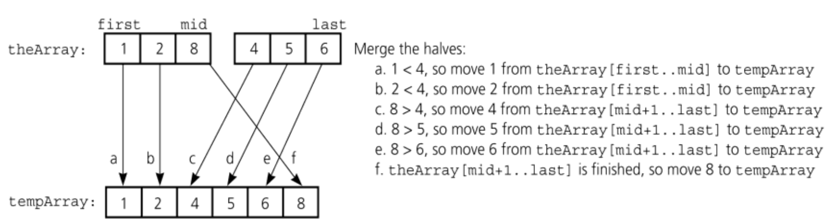
Mergesort – Example2



29

Mergesort – Analysis of Merge

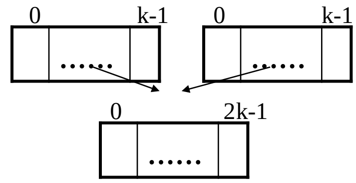
A worst-case instance of the merge step in *mergesort*



30

Mergesort – Analysis of Merge (cont.)

Merging two sorted arrays of size ***k***

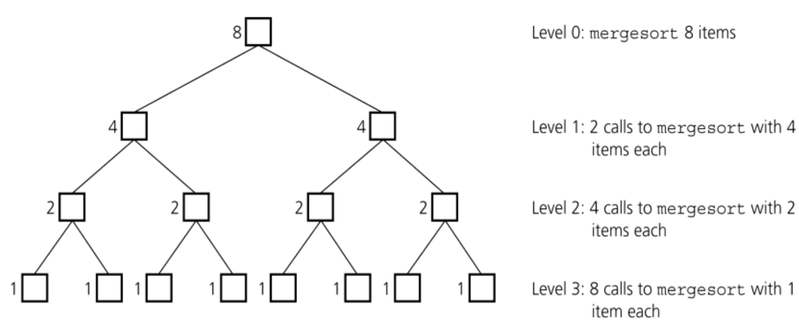


- ***Best-case:***
  - All the elements in the first array are smaller (or larger) than all the elements in the second array.
  - The number of moves: 2k + 2k
  - The number of key comparisons: k
- ***Worst-case:***
  - The number of moves: 2k + 2k
  - The number of key comparisons: 2k-1

31

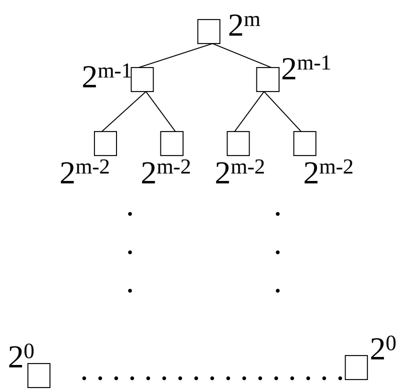
Mergesort - Analysis

Levels of recursive calls to *mergesort*, given an array of eight items



32

Mergesort - Analysis



level 0 : 1 merge (size 2^{m-1})

level 1 : 2 merges (size 2^{m-2})

level 2 : 4 merges (size 2^{m-3})

level m-1 : 2^{m-1} merges (size 2^0)

level m

33

Mergesort - Analysis

- ***Worst-case*** **–**

The number of key comparisons:

$= 2^0*(2*2^{m-1}-1) + 2^1*(2*2^{m-2}-1) + ... + 2^{m-1}*(2*2^0-1)$

$= (2^m - 1) + (2^m - 2) + ... + (2^m – 2^{m-1})$ (m terms)

$= m*2^m – \sum_{i=0}^{m-1} 2^i$

$= m*2^m – 2^m – 1$

Using m = log n

$\mathbf{= n * log_2 n – n – 1}$

➔ $\mathbf{O (n * log_2 n)}$

34

Mergesort – Analysis

- Mergesort is extremely efficient algorithm with respect to time.
  - Both worst case and average cases are **O (n * $\log_2$n)**
- But, mergesort requires an extra array whose size equals to the size of the original array.
- If we use a linked list, we do not need an extra array
  - But, we need space for the links
  - And, it will be difficult to divide the list into half (O(n))

35

Shellsort Examples

Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

*** floor(8/2) ➔ floor(4) = 4**

increment 4: **1** **2** **3** **4** (visualize underlining)

18 32 12 5 38 33 16 2

Step **1**) Only look at **18** and **38** and sort in order ;
18 and **38** stays at its current position because they are in order.

Step **2**) Only look at **32** and **33** and sort in order ;
32 and **33** stays at its current position because they are in order.

36

Shellsort Examples

Sort: 18 32 12 5 38 33 16 2

8 Numbers to be sorted, Shell's increment will be floor(n/2)

*** floor(8/2) ➔ floor(4) = 4**

increment 4: **1** **2** **3** **4** (visualize underlining)

18 32 12 5 38 33 16 2

Step **3**) Only look at **12** and **16** and sort in order ;
12 and **16** stays at its current position because they are in order.

Step **4**) Only look at **5** and **2** and sort in order ;
2 and **5** need to be switched to be in order.

37

Shellsort Examples (con't)

Sort: 18 32 12 5 38 33 16 2

Resulting numbers after increment 4 pass:

18 32 12 *2* 38 33 16 *5*

*** floor(4/2) ➔ floor(2) = 2**

increment 2: **1** **2**

18 32 12 2 38 33 16 5

Step **1**) Look at **18**, **12**, **38**, **16** and sort them in their appropriate location:

12 32 16 2 18 33 38 5

Step **2**) Look at **32**, **2**, **33**, **5** and sort them in their appropriate location:

12 2 16 5 18 32 38 33

38

Shellsort Examples (con't)

Sort: 18 32 12 5 38 33 16 2

* floor(2/2) ➔ floor(1) = 1
increment 1: 1

12	2	16	5	18	32	38	33
2	5	12	16	18	32	33	38

The last increment or phase of Shellsort is basically an Insertion Sort algorithm.

39

Shell Sort Code

```
int j, p, gap;           comparable tmp;
for (gap = N/2; gap > 0; gap = gap/2)
for ( p = gap; p < N ; p++)
  {
   tmp = a[p];
   for ( j = p; j>=gap && tmp<a[j-gap]; j=j-gap)
        a[ j ] = a[ j - gap ];
   a[j] = tmp;
  }
```

40

Increment sequences (How to calculate The Gap)

1. Shell's original sequence:

N/2 , N/4 , ..., 1 (repeatedly divide by 2).

2. Hibbard's increments:

1, 3, 7, ..., 2^k - 1 ; k = 1, 2, …

3. Knuth's increments:

1, 4, 13, ..., (3^k - 1) / 2 ; k = 1, 2, …

4. Sedgewick's increments:

1, 5, 19, 41, 109, ... k = 0, 1, 2, …

Interleaving 9 ($4^k – 2^k$) + 1 and 2^{k+2} (2^{k+2} – 3) + 1.

41

Shell Sort Analysis

Shellsort's **worst-case** performance using Hibbard's increments is **$\Theta(n^{3/2})$.**

The **average** performance is thought to be about **$O(n^{5/4})$**

The exact complexity of this algorithm is still being debated

for **mid-sized** data : nearly as well if not better than the faster **(*n log n*)** sorts.

Animations:
http://www.sorting-algorithms.com/shell-sort
http://www.cs.pitt.edu/~kirk/cs1501/animations/Sort2.html

42

Comparison of Sorting Algorithms

	Worst case	Average case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Mergesort	n * log n	n * log n
Quicksort	n^2	n * log n
Radix sort	n	n
Treesort	n^2	n * log n
Heapsort	n * log n	n * log n

43