1806ICT Programming Fundamentals Lecture 13: Searching and Sorting

1

1

Topics

- An array as a list
- Searching
  - Linear search
  - Binary search (sorted list)
- Efficiency of an algorithm

2

2

Arrays as Lists

- An array
  - stores several elements of the same type
  - can be thought of as a list of elements:

13
5
19
10
7
27
17
1

`int a[8]`

13	5	19	10	7	27	17	1

3

3

Linear Search

- Problem: determine if an element is present in an array
- Method:
  - start at one end
  - look at each array element until the sought element is found
- Also called ***sequential search***

4

4

Linear Search: Algorithm and Code

isPresent (array, val, arraySize) { set count to 0 while (not yet processed all array elements) { if (current array element is val) { return true } increment count } return false }	int isPresent (int *arr, int val, int N) { int count; for (count=0;count<N;count++) { if (arr[count]==val) { return 1; } } return 0; }

5

5

Linear Search -- Exercise

- How would you modify the program so that it returns the position of the sought item (i.e., **`findPosition`** rather than **`isPresent`**)?
- How would you indicate "not found"?

6

6

What does Efficiency Mean?

- Algorithm: a set of instructions describing how to do a task
- Program: an implementation of an algorithm
- ***Complexity theory*** describes the time and space used by an algorithm

The time and space requirements of an algorithm enable us to measure how efficient it is

7

7

Types of Computer Resources

- Time: elapsed period from start to finish of the execution of an algorithm
- Space (memory): amount of storage required by an algorithm
- Hardware: physical mechanisms required for executing an algorithm

8

8

How to Measure Efficiency?

- Use your watch? Use the computer clock?
- Not a good idea, because:
  - What if you run your program on different computers?
  - Your program may also wait for I/O or other resources
  - While running a program, a computer performs many other computations
  - Depends on programming/coding skill

9

9

Abstract Notion of Efficiency

- We are interested in the number of steps executed by an algorithm
  - step ≈ execution of an instruction
- The running time of an algorithm is proportional to the number of steps executed by the algorithm
- Running time is given as a function of the size of the input data: "***Big-O Notation***"

10

10

Linear Search Efficiency

- What is the size of the input data?
  - The size of the array being searched is N
- What is the *time complexity* of this algorithm?
  - Each time through the loop we perform
    - 2 comparisons
      - count<N
      - arr[count]==val
    - 1 increment and 1 assignment
      - count++
    - Total: 4 operations
- So we execute approximately f(N)=4*N ops.

11

11

Big-O Notation

- Big-O notation is a function of the size of the input
- Example:
  - Input: N integers
  - Algorithm complexity:
    - Constant O(1)
    - Logarithmic O(log N)
    - Linear O(N)
    - n log(n) O(N log N)
    - Quadratic $O(N^2)$
    - Cubic $O(N^3)$
    - Exponential $O(2^N)$

12

12

Calculating Complexity with the Big-O Notation

- Simplify and choose the highest term
- Examples:
  - ❑ $2 + 3N + 10N + 3N^2 + 100$
    $= 3N^2 + 13N + 102 \approx O(N^2)$
  - ❑ $40N + N^3 \approx O(N^3)$
  - ❑ $25 \approx O(1)$

13

13

Linear Search Efficiency (cont)

- Best case?
  - Wanted item is at the start of the list
    - ❑ 1 (initialization) + 4 operations
- Worst case?
  - Wanted item is not found
    - ❑ $1 + 4N + 2$ (last increment and test) $\approx O(N)$
- Average case?
  - Average of [Wanted item is in position 1, 2, …, N]
    - ❑

$$1 + \frac{(4+4N)\times N}{2N} = 1 + (2+2N) \approx O(n)$$

14

14

Binary Search

- Can we do any better than linear search?
- Example:
  - How do you find a word in the dictionary, or a number in the phone directory?
  - Assume that the array is sorted and use ***bisection***

15

15

Binary Search (cont)

If (value == middle element)
value is found
else if (value < middle element)
search left-half of list with the same method
else
search right-half of list with the same method

16

16

Binary Search -- Example 1

Case 1: **val == a[mid]**

val = 10

low = 0, high = 8

mid = (0 + 8) / 2 = 4

a:

1	5	7	9	10	13	17	19	27
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
low				mid				high

17

17

Binary Search -- Example 2

Case 2: **val > a[mid]**

val = 19

low = 0, high = 8

mid = (0 + 8) / 2 = 4

new low = mid + 1 = 5

a:

1	5	7	9	10	13	17	19	27
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
low				mid	new low			high

18

18

Binary Search -- Example 3

Case 3: **val < a[mid]**

val = 7

low = 0, high = 8

mid = (0 + 8) / 2 = 4

new high = mid - 1 = 3

a:	1	5	7	9	10	13	17	19	27
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
	low			new high	mid				high

19

19

Binary Search -- Example 3 (cont)

val = 7

a:	1	5	7	9	10	13	17	19	27
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

a:	1	5	7	9	10	13	17	19	27
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

a:	1	5	7	9	10	13	17	19	27
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

20

20

Binary Search -- Algorithm and Code

```
isPresent (array,  val,  arraySize)

{
   set low to first array position
   set high to last array position

while ( low < = high )
   {
      set mid to half of low + high
      if (array element in mid is val )
      {
         return true
      }
      else if ( middle value < val )
      {
         set low to mid + 1
       }
      else
      {
         set high to mid - 1
       }
   }
   return false
}
```

```
int isPresent(int *arr, int val,
              int N)
{
   int low = 0;
   int high = N - 1;
   int mid;
   while ( low <= high )
   {
      mid = ( low + high )/2;
      if (arr[mid]==val)
      {
         return 1;
      }
      else if (arr[mid] < val)
      {
         low = mid + 1;
      }
      else
      {
         high = mid - 1;
      }
   }
   return 0;
}
```

21

21

Binary Search: Exercise

- What happens if the sought value is not in the list?
- How would you modify the code so that it returns the position of the sought item (i.e., **`findPosition`** rather than **`isPresent`**)?

22

22

Binary Search Efficiency

- What is the size of the input data?
  - The size of the array being searched is N
- What is the ***time complexity*** of this algorithm?
  - Each time through the loop we perform
    - 3 comparisons
    - 3 arithmetic operations
    - 2 assignments
    - Total: 8 operations

23

23

Binary Search Efficiency (cont)

- Best case?
  - item is in the middle
  - 5 operations ≈ O(1)
- Worst case?
  - item is not found
  - $8 \times \log_2 N$ operations ≈ $O(\log_2 N)$
- Average case?
  - $O(\log_2 N)$

24

24

Calculating the Worst Case Complexity

- After 1 bisection $N/2$ items
- After 2 bisections $N/4 = N/2^2$ items
- . . .
- After i bisections $N/2^i = 1$ item

$i = \log_2 N$

25

25

Exercise

Problem: How would you implement linear search or binary search over an array of structs?

Method: The array must be sorted by ID, name or mark, depending on the search key

26

26

Exercise (cont)

```
struct studentRec
{
  int         IDNumber;
  char        name[NAMELEN];
  float       mark;
};
typedef struct studentRec Student;

struct classRec
{
  int         count;
  Student     student[MAX_STUDENTS];
};
typedef struct classRec ClassType;

ClassType  class;
Student findStudent(ClassType *class, int IDNum)
{
  ...
}
```

27

27

Notes on Searching

- Linear search can be done on any (sorted or unsorted) list, but it is inefficient
- Binary search
  - requires a list to be sorted
  - is more efficient

28

28

Topics

- Sorting lists
  - Selection sort
  - Insertion sort
  - Bubble sort

29

29

Sorting

- Aim:
  - start with an unsorted array
  - end with a sorted array
- How to sort student records?
  - depends on purpose
  - by name, ID number, marks
- Exercise: how to sort words?

30

30

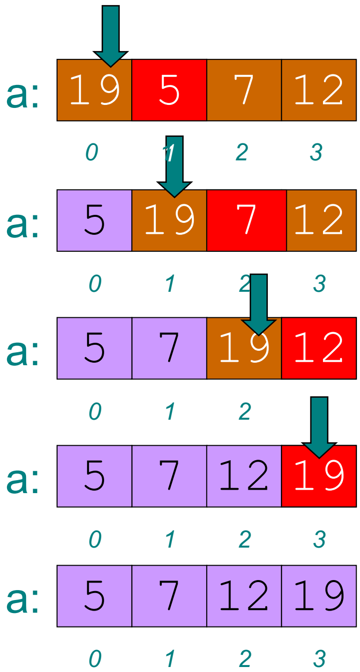
Selection Sort

- Basic idea:
  - find the minimum element
  - exchange it with the first unsorted element of the array
  - repeat for the rest of the array

31

31

Selection Sort -- Example



32

32

Selection Sort: Algorithm and Code

``selectionSort(array, N)``` ```{``` ``` set count to 0``` ``` while (count < N)``` ``` {``` ``` set posmin to index``` ``` of smallest``` ``` element in rest``` ``` of array``` ``` swap item at posmin``` ``` with item at``` ``` count``` ``` add 1 to count``` ``` }``` ```}```	```void selectionSort(int *arr,``` ``` int N)``` ```{``` ``` int posmin;``` ``` int count, tmp;``` ``` for(count=0;count<N;count++)``` ``` {``` ``` posmin=``` ``` findIndexMin(arr,count,N);``` ``` tmp=arr[posmin];``` ``` arr[posmin]=arr[count];``` ``` arr[count]=tmp;``` ``` }``` ```}```

33

33

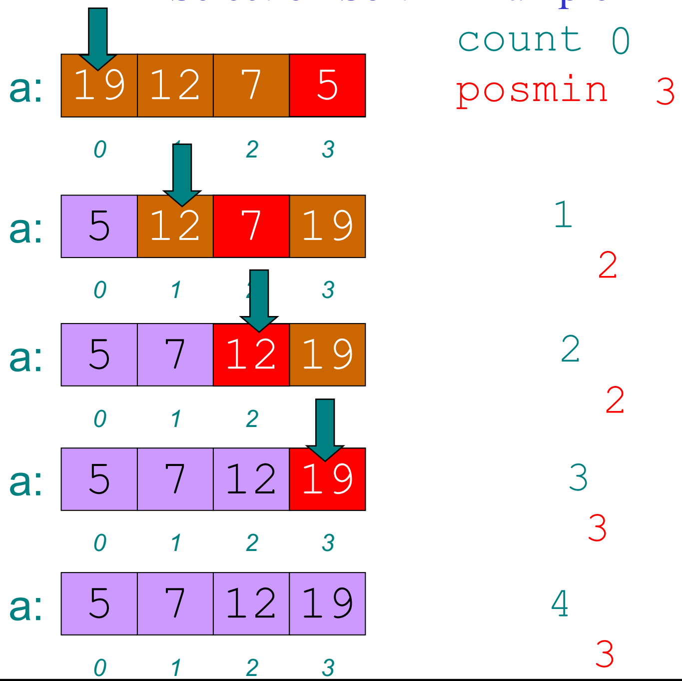
Selection Sort: Algorithm and Code (cont)

``findIndexMin(array,``` ``` start,``` ``` N)``` ```{``` ``` set posmin to start``` ``` set count to start``` ``` while (count < N)``` ``` {``` ``` if(current element <``` ``` element at posmin)``` ``` {``` ``` set posmin to count``` ``` }``` ``` increment count by 1``` ``` }``` ``` return posmin``` ```}```	```int findIndexMin(int *arr,``` ``` int start,``` ``` int N)``` ```{``` ``` int posmin=start;``` ``` int index;``` ``` for(index=start; index<N;``` ``` index++)``` ``` {``` ``` if (arr[index]<arr[posmin])``` ``` {``` ``` posmin=index;``` ``` }``` ``` }``` ``` return posmin;``` ```}```

34

34

Selection Sort -- Example



35

35

Selection Sort Analysis

- What is the time complexity of this algorithm?
- Worst case == Best case == Average case
- Each iteration performs a linear search on the rest of the array
  - first element N +
  - second element N-1 +
  - …
  - penultimate element 2 +
  - last element 1
  - Total $N(N+1)/2 = O(N^2)$

36

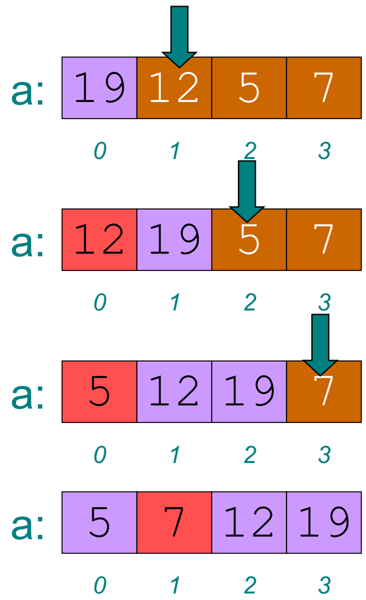
36

Insertion Sort

- Basic idea (*sorting cards*):
  - Take the first unsorted item (assume that the portion of the array in front of this item is sorted)
  - Insert the item in the correct position in the sorted part of the array

37

Insertion Sort -- Example



38

Insertion Sort: Algorithm and Code

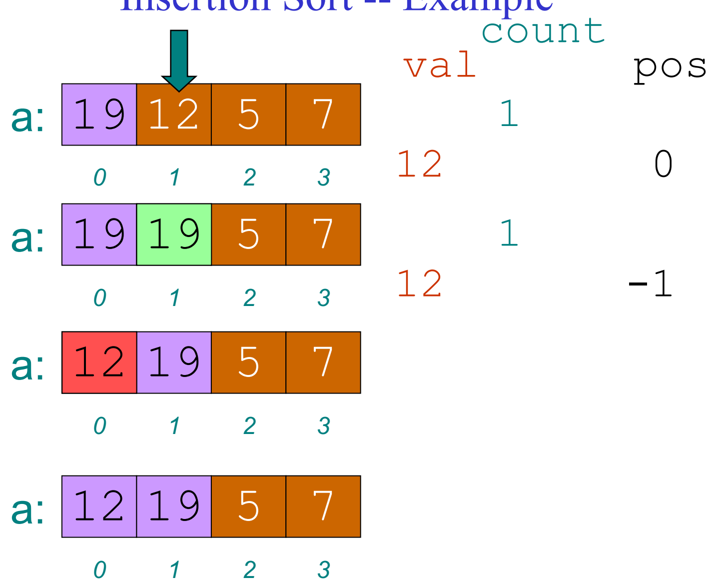
```
insertionSort( array )

{

 set count to 1
 while ( count < N )
 {
  set val to array[count]
  set pos to count-1
  while (pos is in the
         array and val <
         item in pos)
  {
   shuffle item in pos
      one place to right
   decrement pos by 1
  }
  put val in pos+1
  add 1 to count
 }
}
```

```c
void insertionSort(int *arr,
                   int N)
{
 int pos;
 int count, val;
 for(count=1;count<N;count++)
 {
  val = arr[count];

  for(pos=count-1;pos>=0;pos--)
  {
    if (arr[pos]>val)
    {
      arr[pos+1]=arr[pos];
    }
    else { break; }
  }
  arr[pos+1] = val;

 }
}
```

39

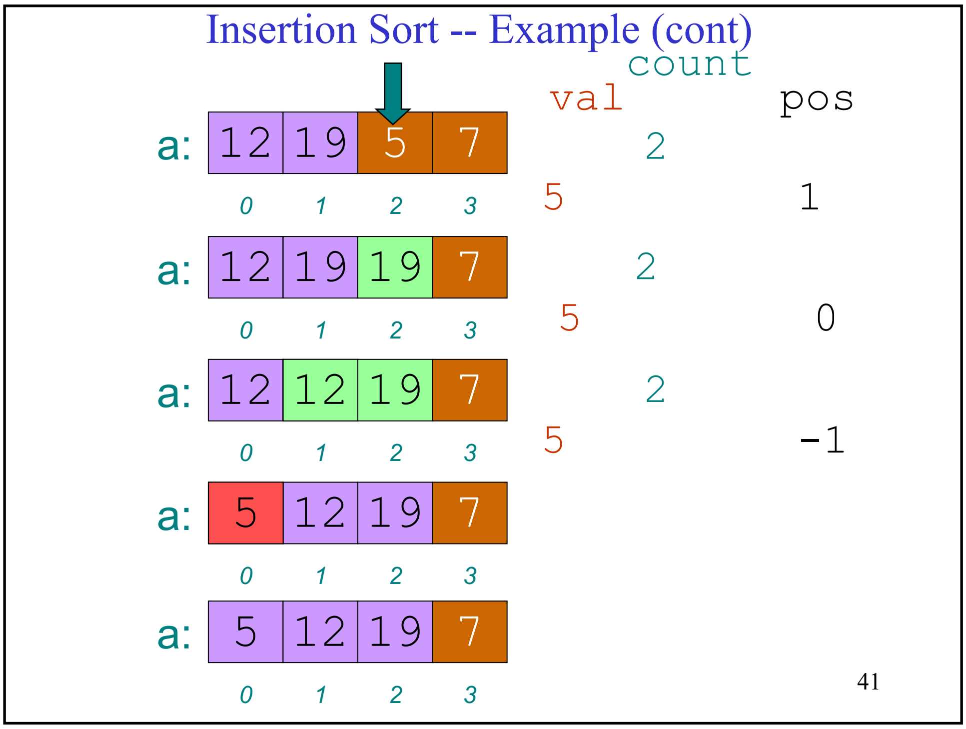
39

Insertion Sort -- Example

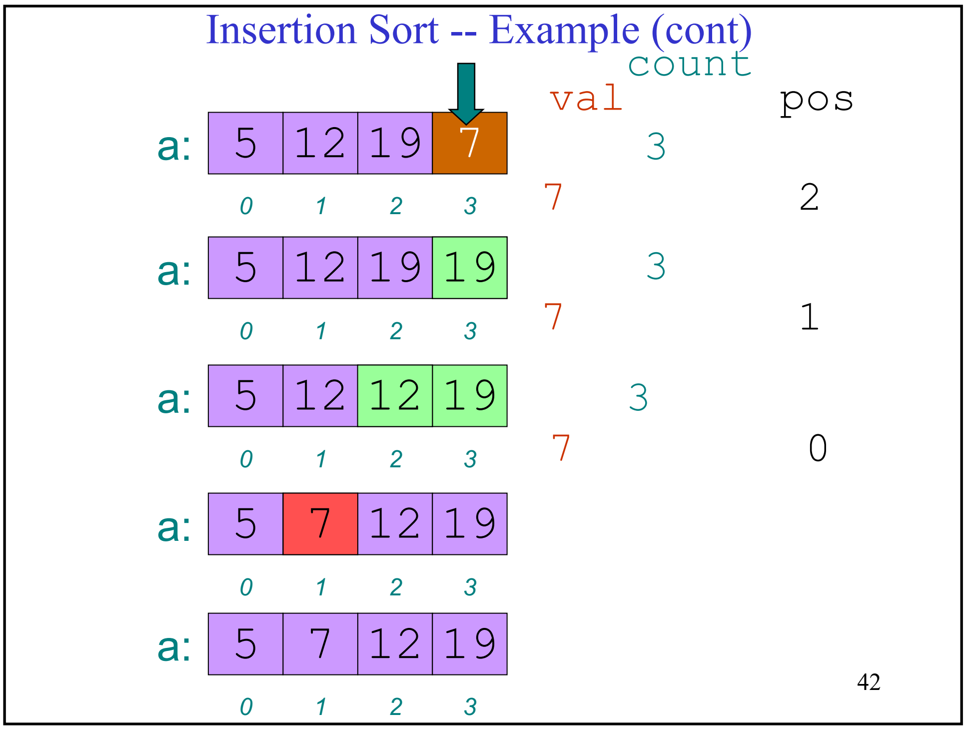


40

40


Insertion Sort -- Example (cont)
count
val
pos
a: 12 19 5 7
0 1 2 3
2
5
1
a: 12 19 19 7
0 1 2 3
2
5
0
a: 12 12 19 7
0 1 2 3
2
5
-1
a: 5 12 19 7
0 1 2 3
a: 5 12 19 7
0 1 2 3
41

41


Insertion Sort -- Example (cont)
count
val
pos
a: 5 12 19 7
0 1 2 3
3
7
2
a: 5 12 19 19
0 1 2 3
3
7
1
a: 5 12 12 19
0 1 2 3
3
7
0
a: 5 7 12 19
0 1 2 3
a: 5 7 12 19
0 1 2 3
42

42

Insertion Sort Analysis

- What is the time complexity of this algorithm?
- Worst case > Average case > Best case
- Each iteration inserts an element at the start of the array, shifting all sorted elements along
  - `second element        2      +`
  - …
  - `penultimate element  N-1   +`
  - `last element          N`

  - `Total              (2+N)(N-1)/2 =` $O(N^2)$

43

43

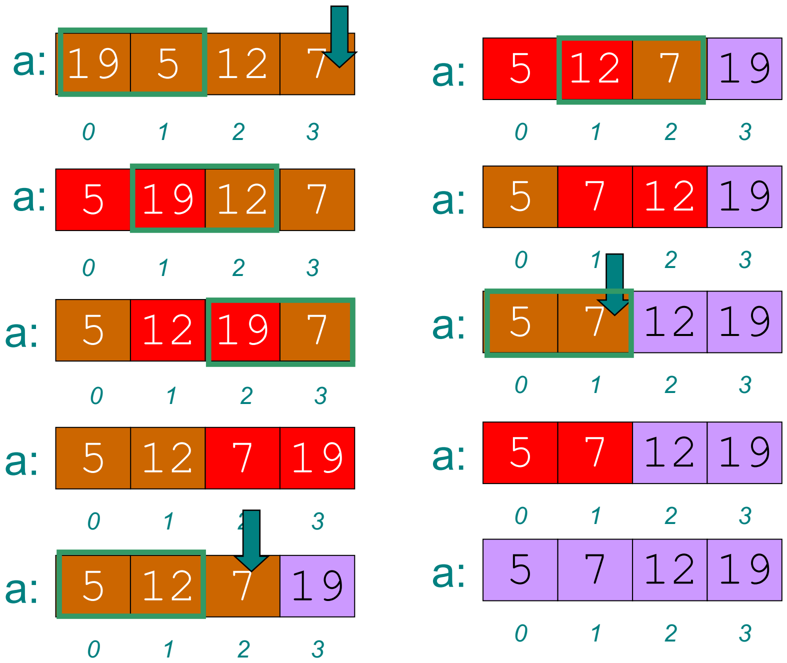
Bubble Sort

- Basic idea (*lighter bubbles rise to the top*):
  - Exchange neighbouring items until the largest item reaches the end of the array
  - Repeat for the rest of the array

44

44

Bubble Sort -- Example



45

45

Bubble Sort: Algorithm and Code

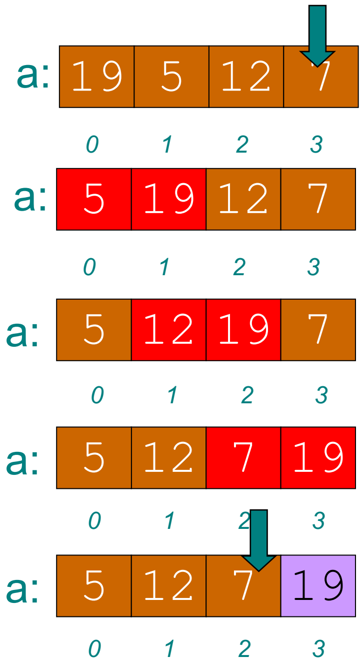
```
bubbleSort( array, N )
{
 set bound to N-1
 set swapped to 1
 set count to 0
 while ( swapped > 0 )
 {
  set swapped to 0
  while ( count < bound )
  {
    if ( array[count] >
         array[count+1] )
    {
     swap array[count] and
          array[count+1]
     set swapped to count
    }
    increment count by 1
  }
  set bound to swapped
 }
}
```

```
void bubbleSort(int *arr,
                int N)
{
 int ct, temp, bound = N-1;
 int swapped = 1;
 while (swapped > 0 )
 {
   swapped = 0;
   for(ct=0;ct<bound;ct++)
   {
     if ( arr[ct] >
          arr[ct+1] )
     { /* swapping items */
       temp = arr[ct];
       arr[ct] = arr[ct+1];
       arr[ct+1] = temp;
       swapped = ct;
     }
   }
   bound=swapped;
 }
}
```

46

46

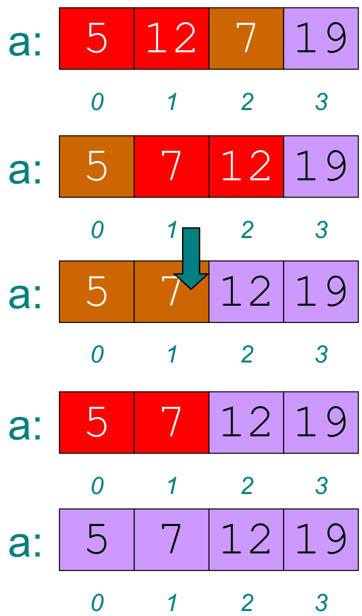
Bubble Sort -- Example



47

47

Bubble Sort -- Example (cont)



48

48

Bubble Sort Analysis

- What is the time complexity of this algorithm?
- Worst case > Average case > Best case
- Each iteration compares all the adjacent elements, swapping them if necessary
  - `first iteration      N      +`
  - `second iteration          N-1   +`
  - …
  - `last iteration       1`
  - `Total           N(1+N)/2 = O(N²)`

49

49

Summary

- Insertion, Selection and Bubble sort:
  - Worst case time complexity is $O(N^2)$

`Best sorting routines are O(N log(N))`

50

50