Software Engineering 2 (C++)

CSY2006 (Week 5)

Dr. Suraj Ajit

Introduction to Search Algorithms

- Search: locate an item in a list of information

- Two algorithms we will examine:
  - Linear search
  - Binary search

Linear Search

- Also called the sequential search
- Starting at the first element, this algorithm sequentially steps through an array examining each element until it locates the value it is searching for.

Linear Search - Example

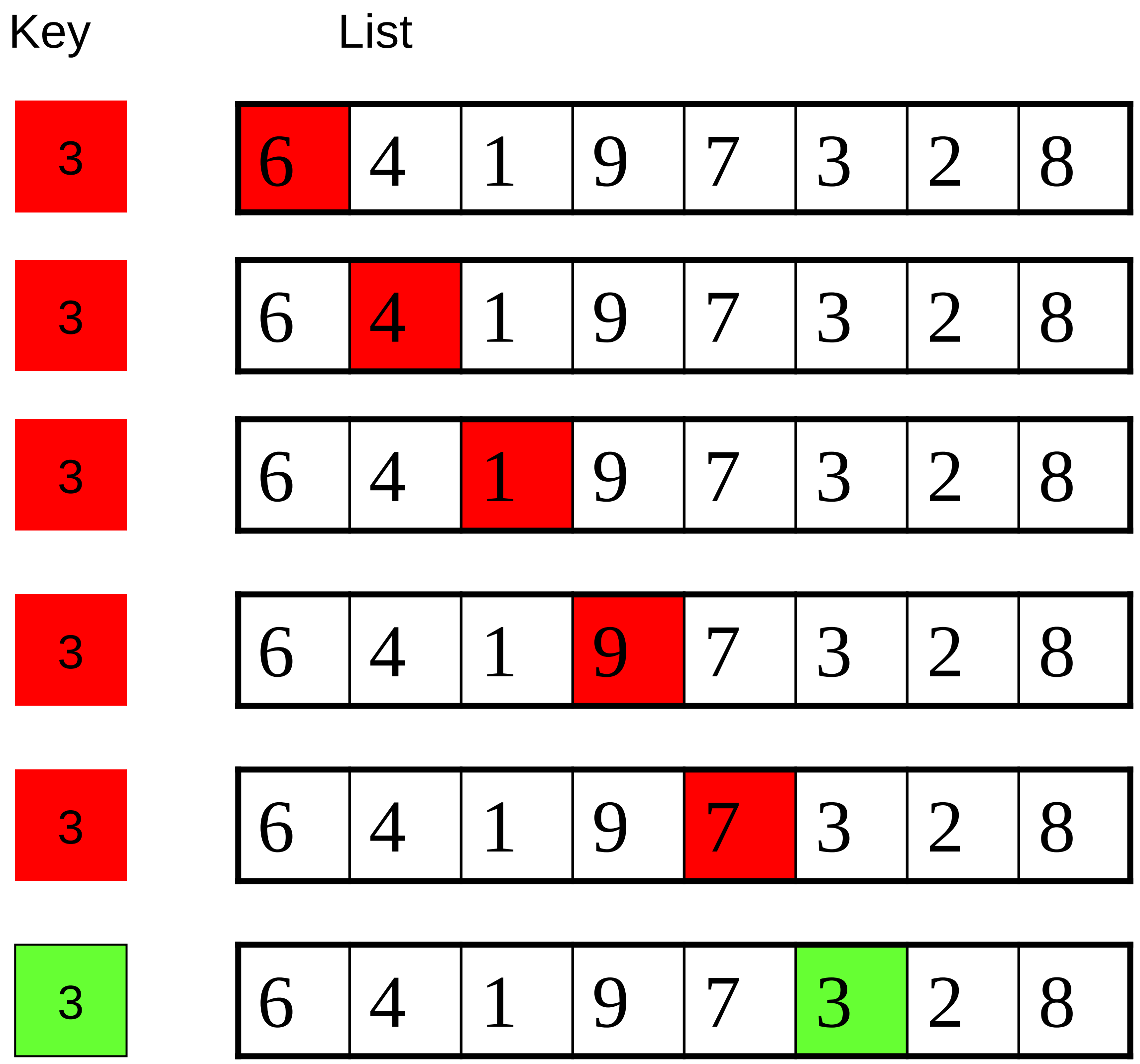
- Array `numlist` contains:

17	23	5	11	2	29	3

- Searching for the the value `11`, linear search examines `17, 23, 5,` and `11`
- Searching for the the value `7`, linear search examines `17, 23, 5, 11, 2, 29,` and `3`

animation

Linear Search Animation



Linear Search

- Algorithm:

```
set found to false; set position to –1; set index to 0
while index < number of elts. and found is false
    if list[index] is equal to search value
            found = true
            position = index
    end if
    add 1 to index
end while
return position
```

A Linear Search Function

```
int searchList(int list[], int numElems, int value)
{
   int index = 0;       // Used as a subscript to search array
   int position = -1;   // To record position of search value
   bool found = false; // Flag to indicate if value was found

   while (index < numElems && !found)
   {
      if (list[index] == value) // If the value is found
      {
         found = true; // Set the flag
         position = index; // Record the value's subscript
      }
      index++; // Go to the next element
   }
return position; // Return the position, or -1
}
```

Linear Search - Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order

- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Binary Search

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

Binary Search - Example

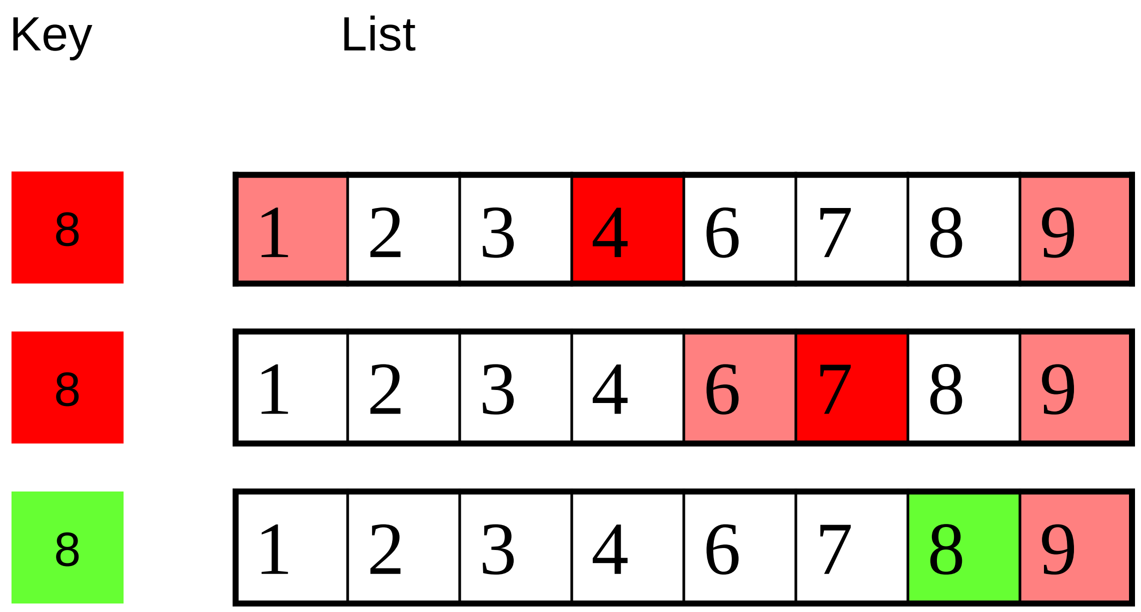
- Array `numlist2` contains:

2	3	5	11	17	23	29

- Searching for the the value `11`, binary search examines `11` and stops
- Searching for the the value `7`, binary search examines `11, 3, 5,` and stops

animation

Binary Search



Binary Search

```
Set first index to 0.
Set last index to the last subscript in the array.
Set found to false.
Set position to -1.
While found is not true and first is less than or equal to last
    Set middle to the subscript half-way between array[first] and array[last].
    If array[middle] equals the desired value
        Set found to true.
        Set position to middle.
    Else If array[middle] is greater than the desired value
        Set last to middle - 1.
    Else
        Set first to middle + 1.
    End If.
End While.
Return position.
```

A Binary Search Function

```
int binarySearch(int array[], int size, int value)
{
   int first = 0,                 // First array element
       last = size - 1,           // Last array element
       middle,                    // Mid point of search
       position = -1;             // Position of search value
   bool found = false;            // Flag

   while (!found && first <= last)
   {
      middle = (first + last) / 2;     // Calculate mid point
      if (array[middle] == value)      // If value is found at mid
      {
         found = true;
         position = middle;
      }
      else if (array[middle] > value)  // If value is in lower half
         last = middle - 1;
      else
         first = middle + 1;           // If value is in upper half
   }
   return position;
}
```

Binary Search - Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of N elements, performs at most $\boldsymbol{\log_2 N}$ comparisons
- Disadvantages:
  - Requires that array elements be sorted

Introduction to Sorting Algorithms

- Sort: arrange values into an order:
  - Alphabetical
  - Ascending numeric
  - Descending numeric

- Two algorithms considered here:
  - Bubble sort
  - Selection sort

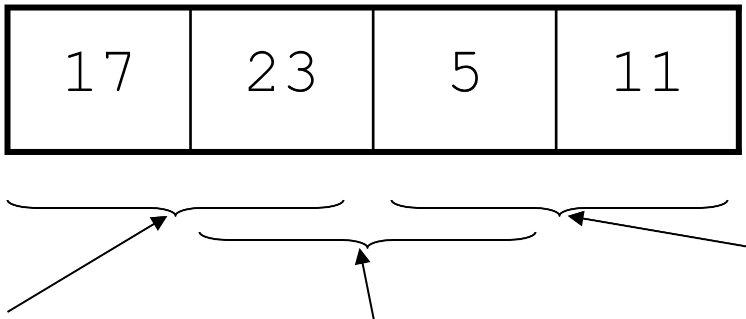
Bubble Sort

Concept:

- Compare 1st two elements
  - If out of order, exchange them to put in order
- Move down one element, compare 2nd and 3rd elements, exchange if necessary. Continue until end of array.
- Pass through array again, exchanging as necessary
- Repeat until pass made with no exchanges

Example – First Pass

Array `numlist3` contains:



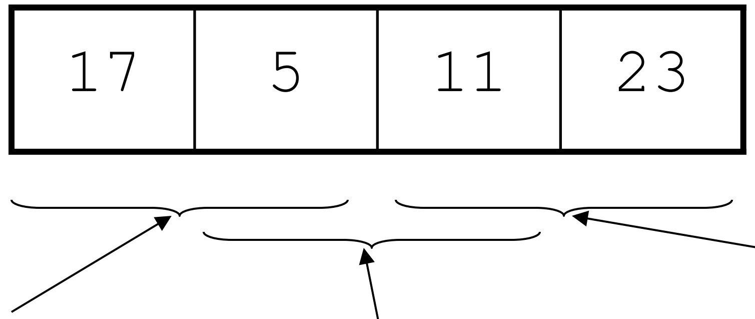
compare values `17` and `23` – in correct order, so no exchange

compare values `23` and `5` – not in correct order, so exchange them

compare values `23` and `11` – not in correct order, so exchange them

Example – Second Pass

After first pass, array `numlist3` contains:



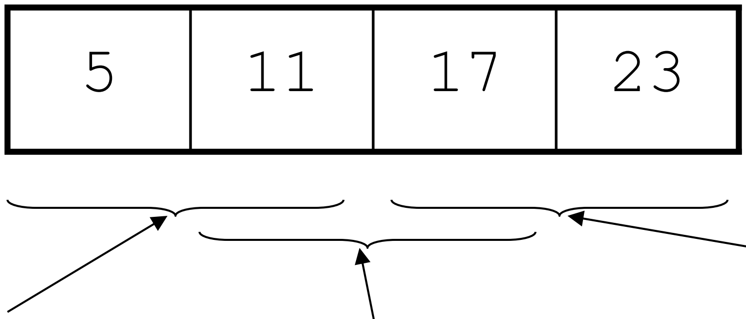
compare values `17` and `5` – not in correct order, so exchange them

compare values `17` and `11` – not in correct order, so exchange them

compare values `17` and `23` – in correct order, so no exchange

Example – Third Pass

After second pass, array `numlist3` contains:



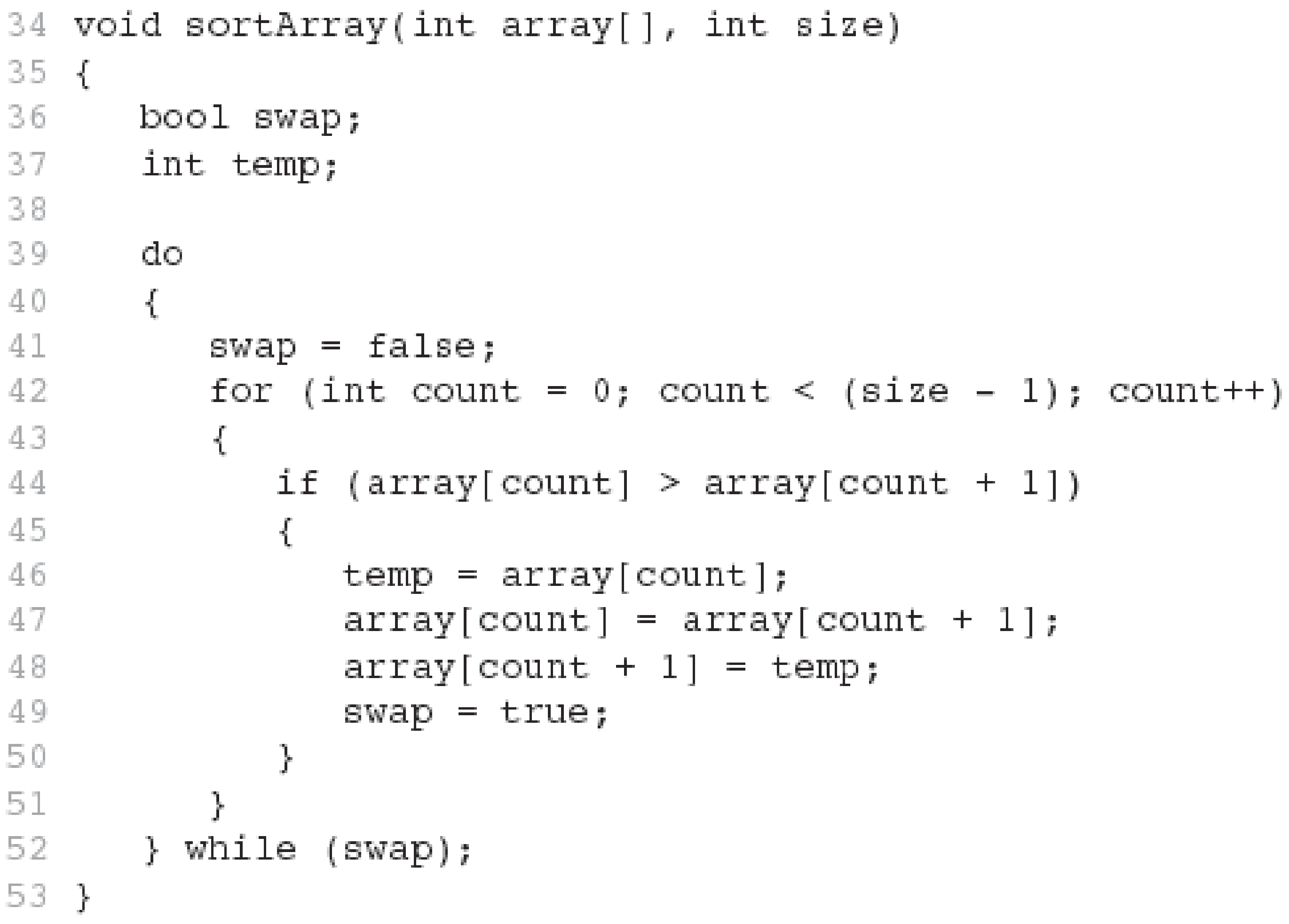
compare values `5` and `11` – in correct order, so no exchange

compare values `11` and `17` – in correct order, so no exchange

compare values `17` and `23` – in correct order, so no exchange

No exchanges, so array is in order

A Bubble Sort Function – From Program 8-4



```
void sortArray(int array[], int size)
{
   bool swap;
   int temp;

   do
   {
      swap = false;
      for (int count = 0; count < (size - 1); count++)
      {
         if (array[count] > array[count + 1])
         {
            temp = array[count];
            array[count] = array[count + 1];
            array[count + 1] = temp;
            swap = true;
         }
      }
   } while (swap);
}
```

Bubble Sort - Tradeoffs

- Benefit:
  - Easy to understand and implement
- Disadvantage:
  - Inefficient: slow for large arrays

Selection Sort

- Concept for sort in ascending order:
  - Locate smallest element in array. Exchange it with element in position 0
  - Locate next smallest element in array. Exchange it with element in position 1.
  - Continue until all elements are arranged in order

Selection Sort - Example

Array `numlist` contains:

11	2	29	3

1. Smallest element is `2`. Exchange `2` with element in 1st position in array:

2	11	29	3

Example (Continued)

2. Next smallest element is 3. Exchange 3 with element in 2nd position in array:

2	3	29	11

3. Next smallest element is 11. Exchange 11 with element in 3rd position in array:

2	3	11	29

A Selection Sort Function – From Program 8-5

```
void selectionSort(int array[], int size)
{
   int startScan, minIndex, minValue;

   for (startScan = 0; startScan < (size - 1); startScan++)
   {
      minIndex = startScan;
      minValue = array[startScan];
      for(int index = startScan + 1; index < size; index++)
      {
         if (array[index] < minValue)
         {
            minValue = array[index];
            minIndex = index;
         }
      }
      array[minIndex] = array[startScan];
      array[startScan] = minValue;
   }
}
```

Selection Sort - Tradeoffs

- Benefit:
  - More efficient than Bubble Sort, since fewer exchanges
- Disadvantage:
  - May not be as easy as Bubble Sort to understand

Insertion Sort

- Note that for any array A[0..N-1], the portion A[0..0] consisting of the single entry *A*[0] is already sorted.
- Insertion Sort works by extending the length of the sorted portion one step at a time:
  - A[0] is sorted
  - A[0..1] is sorted
  - A[0..2] is sorted
  - A[0..3] is sorted, and so on, until *A*[0..N-1] is sorted.

See: InsertionSortDemo.cpp

How Insertion Sort Works

15 10 55 35 30 20	index = 1, Insert A[1] = 10 into A[0..1]:
10 15 55 35 30 20	
10 15 55 35 30 20	index = 2, Insert A[2] = 55 into A[0..2]:
10 15 55 35 30 20	
10 15 55 35 30 20	index = 3, Insert A[3] = 35 into A[0..3]:
10 15 35 55 30 20	
10 15 35 55 30 20	index = 4, Insert A[4] = 30 into A[0..4]:
10 15 30 35 55 20	
10 15 30 35 55 20	index = 5, Insert A[5] = 20 into A[0..5]:
10 15 20 30 35 55	
10 15 20 30 35 55	Array is now sorted

Insertion Sort Function

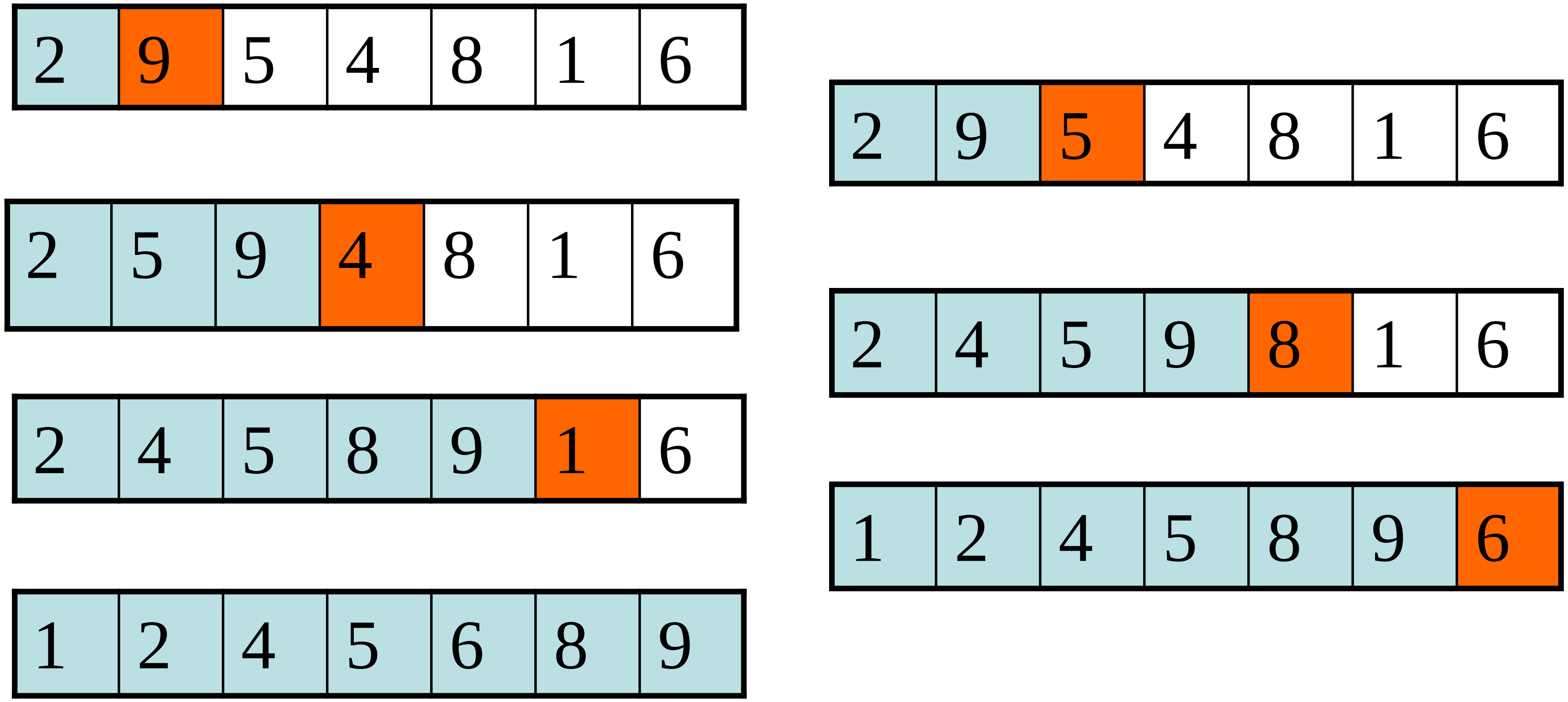
```
void insertionSort(int array[], int size)
 {
        int unsortedValue;  // The first unsorted value
        int scan;           // Used to scan the array

        for (int index = 1; index < size; index++)
        {
          unsortedValue = array[index];
          scan = index;
          while (scan > 0 && array[scan-1] > unsortedValue)
          {
           array[scan] = array[scan - 1];
           scan--;
          }
        array[scan] = unsortedValue;
        }
    }
```

animation

Insertion Sort

int[] myList = {2, 9, 5, 4, 8, 1, 6}; // Unsorted



Executing Time

Suppose two algorithms perform the same task such as search (linear search vs. binary search) and sorting (selection sort vs. insertion sort). Which one is better? One possible approach to answer this question is to implement these algorithms in Java and run the programs to get execution time. But there are two problems for this approach:

- First, there are many tasks running concurrently on a computer. The execution time of a particular program is dependent on the system load.
- Second, the execution time is dependent on specific input. Consider linear search and binary search for example. If an element to be searched happens to be the first in the list, linear search will find the element quicker than binary search.

Space Complexity Functions

- The space complexity function *f*(*N*) of an algorithm is a measure of the amount of memory the algorithm requires to solve a problem of size *N.*
- Space complexity as a measure of efficiency is not often used in theoretical analysis of algorithms, partly because cost of memory keeps declining.

Growth Rate

It is very difficult to compare algorithms by measuring their execution time. To overcome these problems, a theoretical approach was developed to analyze algorithms independent of computers and specific input. This approach approximates the effect of a change on the size of the input. In this way, you can see how fast an algorithm's execution time increases as the input size increases, so you can compare two algorithms by examining their *growth rates*.

Big O Notation

Consider linear search. The linear search algorithm compares the key with the elements in the array sequentially until the key is found or the array is exhausted. If the key is not in the array, it requires *n* comparisons for an array of size *n*. If the key is in the array, it requires *n/2* comparisons on average. The algorithm's execution time is proportional to the size of the array. If you double the size of the array, you will expect the number of comparisons to double. The algorithm grows at a linear rate. The growth rate has an order of magnitude of *n*. Computer scientists use the Big *O* notation to abbreviate for "order of magnitude." Using this notation, the complexity of the linear search algorithm is *O(n)*, pronounced as "*order of n*."

Best, Worst, and Average Cases

For the same input size, an algorithm’s execution time may vary, depending on the input. An input that results in the shortest execution time is called the *best-case* input and an input that results in the longest execution time is called the *worst-case* input. Best-case and worst-case are not representative, but worst-case analysis is very useful. You can show that the algorithm will never be slower than the worst-case. An average-case analysis attempts to determine the average amount of time among all possible input of the same size. Average-case analysis is ideal, but difficult to perform, because it is hard to determine the relative probabilities and distributions of various input instances for many problems. Worst-case analysis is easier to obtain and is thus common. So, the analysis is generally conducted for the worst-case.

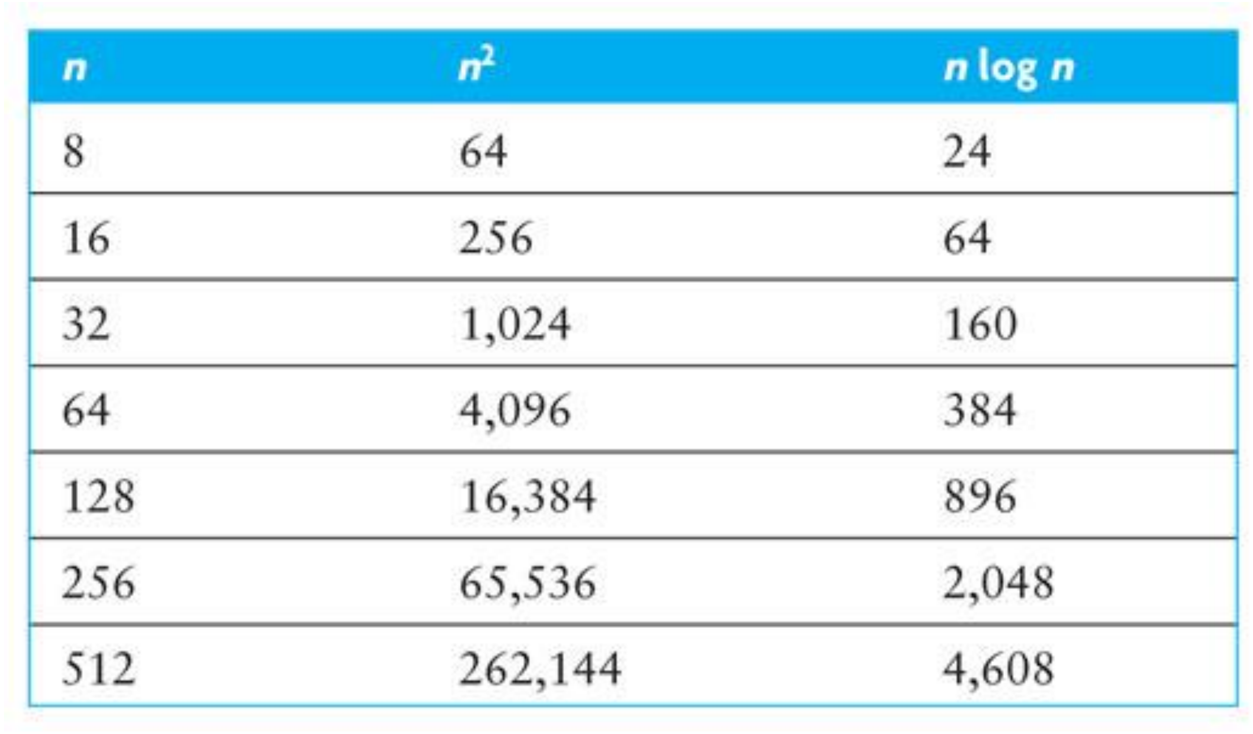
Comparing Common Growth Functions

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$	Constant time
$O(\log n)$	Logarithmic time
$O(n)$	Linear time
$O(n \log n)$	Log-linear time
$O(n^2)$	Quadratic time
$O(n^3)$	Cubic time
$O(2^n)$	Exponential time

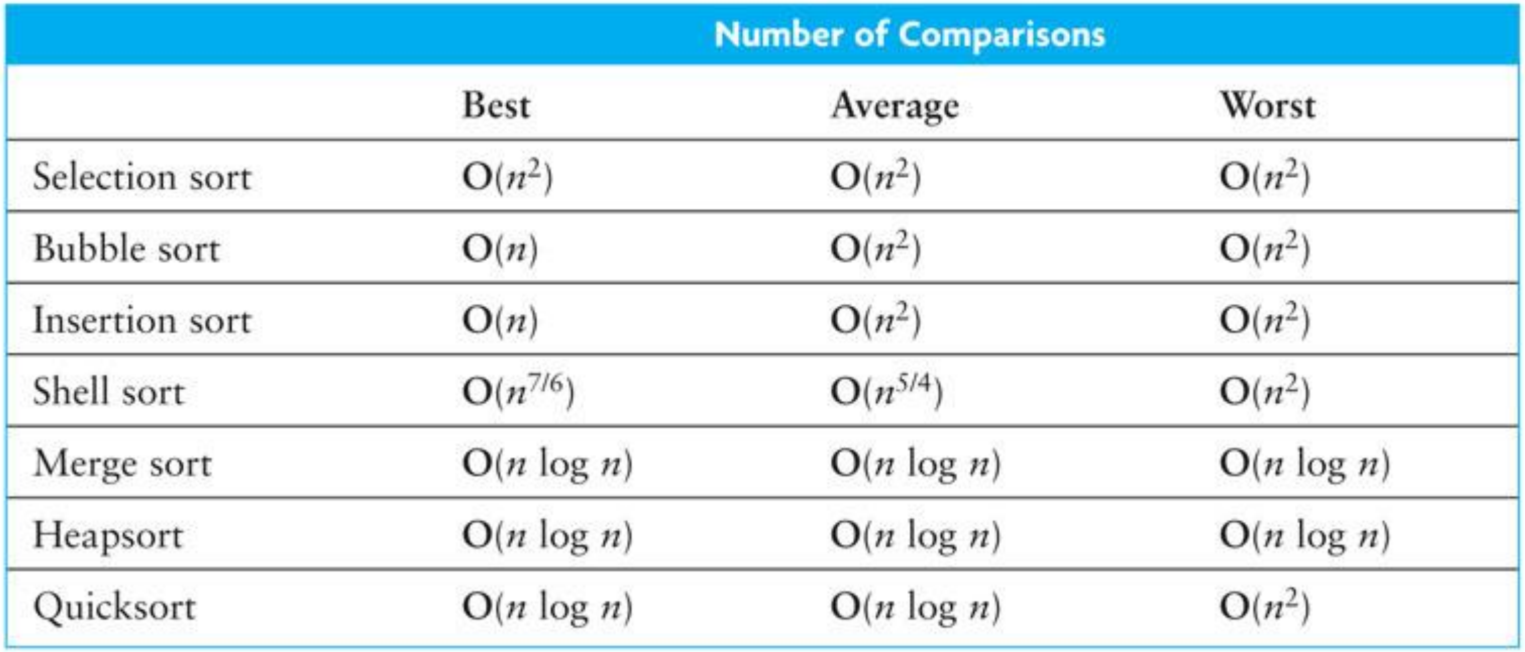
Comparison of Quadratic Sorts

Comparison of growth rates



n	n^2	*n* log *n*
8	64	24
16	256	64
32	1,024	160
64	4,096	384
128	16,384	896
256	65,536	2,048
512	262,144	4,608

Sort Review



	Number of Comparisons		
	Best	Average	Worst
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$
Shell sort	$O(n^{7/6})$	$O(n^{5/4})$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Introduction to the STL vector

Introduction to the STL vector

- A data type defined in the Standard Template Library
- Can hold values of any type:

```
vector<int> scores;
```

- Automatically adds space as more is needed – no need to determine size at definition
- Can use `[]` to access elements

See: examples Pr7-21 to Pr7-26

Declaring Vectors

- You must `#include<vector>`
- Declare a vector to hold `int` element:

```
vector<int> scores;
```

- Declare a vector with initial size 30:

```
vector<int> scores(30);
```

- Declare a vector and initialize all elements to 0:

```
vector<int> scores(30, 0);
```

- Declare a vector initialized to size and contents of another vector:

```
vector<int> finals(scores);
```

Adding Elements to a Vector

- Use `push_back` member function to add element to a full array or to an array that had no defined size:

```
scores.push_back(75);
```

- Use `size` member function to determine size of a vector:

```
howbig = scores.size();
```

Removing Vector Elements

- Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

- To remove all contents of vector, use `clear` member function:

```
scores.clear();
```

- To determine if vector is empty, use `empty` member function:

```
while (!scores.empty()) ...
```

Other Useful Member Functions

Member Function	Description	Example
`at(elt)`	Returns the value of the element at position `elt` in the vector	`cout << vec1.at(i);`
`capacity()`	Returns the maximum number of elements a vector can store without allocating more memory	`maxelts = vec1.capacity();`
`reverse()`	Reverse the order of the elements in a vector	`vec1.reverse();`
`resize (elts,val)`	Add elements to a vector, optionally initializes them	`vec1.resize(5,0);`
`swap(vec2)`	Exchange the contents of two vectors	`vec1.swap(vec2);`