Searching and Sorting

24.2 Searching Algorithms

- **Linear Search**
  - Searches each element in an array sequentially
  - Has *O(n)* time
    - The worst case is that every element must be checked to determine whether the search item exists in the array
- **Big O notation**
  - One way to describe the efficiency of a search
    - Measures the worst-case run time for an algorithm
    - *O(1)* is said to have a constant run time
    - *O(n)* is said to have a linear run time
    - *O(n^2)* is said to have a quadratic run time

24.3 Sorting Algorithms

- **Selection Sort**
  - The first iteration selects the smallest element in the array and swaps it with the first element
  - The second iteration selects the second-smallest item and swaps it with the second element
  - Continues until the last iteration selects the second-largest element and swaps it with the second-to-last index
    - Leaves the largest element in the last index
  - After the *i*th iteration, the smallest *i* items of the array will be sorted in increasing order in the first *i* elements of the array
  - Is a simple, but inefficient, sorting algorithm; *O(n^2)*

```
Unsorted array:
86  97  83  45  19  31  86  13  57  61
after pass 1: 13  97  83  45  19  31  86  86* 57  61
              --
after pass 2: 13  19  83  45  97* 31  86  86  57  61
              --  --
after pass 3: 13  19  31  45  97  83* 86  86  57  61
              --  --  --
after pass 4: 13  19  31  45* 97  83  86  86  57  61
              --  --  --  --
after pass 5: 13  19  31  45  57  83  86  86  97* 61
              --  --  --  --  --
after pass 6: 13  19  31  45  57  61  86  86  97  83*
              --  --  --  --  --  --
after pass 7: 13  19  31  45  57  61  83  86  97  86*
              --  --  --  --  --  --  --
after pass 8: 13  19  31  45  57  61  83  86* 97  86
              --  --  --  --  --  --  --  --
after pass 9: 13  19  31  45  57  61  83  86  86  97*
              --  --  --  --  --  --  --  --  --

Sorted array:
13  19  31  45  57  61  83  86  86  97
```

Outline

`SelectionSort`

24.3 Sorting Algorithms (Cont.)

- **Insertion Sort**
  - The first iteration takes the second element in the array and swaps it with the first element if it is less than the first element
  - The second iteration looks at the third element and inserts it in the correct position with respect to the first two elements
    - All three elements will be in order
  - At the *i*th iteration of this algorithm, the first *i* elements in the original array will be sorted
  - Another simple, but inefficient, sorting algorithm; *O(n^2)*


```
Unsorted array:
12  27  36  28  33  92  11  93  59  62
after pass 1: 12  27* 36  28  33  92  11  93  59  62
              --  --
after pass 2: 12  27  36* 28  33  92  11  93  59  62
              --  --  --
after pass 3: 12  27  28* 36  33  92  11  93  59  62
              --  --  --  --
after pass 4: 12  27  28  33* 36  92  11  93  59  62
              --  --  --  --  --
after pass 5: 12  27  28  33  36  92* 11  93  59  62
              --  --  --  --  --  --
after pass 6: 11* 12  27  28  33  36  92  93  59  62
              --  --  --  --  --  --  --
after pass 7: 11  12  27  28  33  36  92  93* 59  62
              --  --  --  --  --  --  --  --
after pass 8: 11  12  27  28  33  36  59* 92  93  62
              --  --  --  --  --  --  --  --  --
after pass 9: 11  12  27  28  33  36  59  62* 92  93
              --  --  --  --  --  --  --  --  --  --

Sorted array:
11  12  27  28  33  36  59  62  92  93
```

Outline

`InsertionSort`

24.3 Sorting Algorithms (Cont.)

- **Merge Sort**
  - Sorts an array by splitting it into two equal-sized subarrays
    - Sort each subarray and merge them into one larger array
  - With an odd number of elements, the algorithm still creates two subarrays
    - One subarray will have one more element than the other
  - Merge sort is an efficient sorting algorithm: ***O(n log n)***
    - Conceptually more complex than selection sort and insertion sort


```
Unsorted:  36 38 81 93 85 72 31 11 33 74
split:    36 38 81 93 85 72 31 11 33 74
          36 38 81 93 85
                         72 31 11 33 74
split:    36 38 81 93 85
          36 38 81
                   93 85
split:    36 38 81
          36 38
                81
split:    36 38
          36
             38

merge:    36
             38
          36 38
merge:    36 38
                81
          36 38 81
split:             93 85
                   93
                      85
merge:             93
                      85
                   85 93
merge:    36 38 81
                   85 93
           36 38 81 85 93
```

(continued)

Outline

MergeSortTest

(2 of 3)


```
split:                   72 31 11 33 74
                         72 31 11
                                  33 74
split:                   72 31 11
                         72 31
                               11
split:                   72 31
                         72
                            31
merge:                   72
                            31
                         31 72
merge:                   31 72
                               11
                         11 31 72
split:                            33 74
                                  33
                                     74
merge:                            33
                                     74
                                  33 74
merge:                   11 31 72
                                  33 74
                         11 31 33 72 74
merge:    36 38 81 85 93
                         11 31 33 72 74
          11 31 33 36 38 72 74 81 85 93
Sorted:  11 31 33 36 38 72 74 81 85 93
```

Outline

MergeSortTest

(3 of 3)

Binary Search

- Requires that the array be sorted
  - ***For this example, assume the array is sorted in ascending order***
- The first iteration of this algorithm tests the middle element
  - ***If this matches the search key, the algorithm ends***
  - ***If the search key is less than the middle element, the algorithm continues with only the first half of the array***
    - ***The search key cannot match any element in the second half of the array***
  - ***If the search key is greater than the middle element, the algorithm continues with only the second half of the array***
    - ***The search key cannot match any element in the first half of the array***
- Each iteration tests the middle value of the remaining portion of the array
  - ***Called a subarray***
- If the search key does not match the element, the algorithm eliminates half of the remaining elements
- The algorithm ends either by finding an element that matches the search key or reducing the subarray to zero size
- Is more efficient than the linear search algorithm, *O(log n)*
  - ***Known as logarithmic run time***


```
12 17 22 25 30 39 40 52 56 72 76 82 84 91 93
Please enter an integer value (-1 to quit): 72
12 17 22 25 30 39 40 52 56 72 76 82 84 91 93
                      *
                        56 72 76 82 84 91 93
                                  *
                        56 72 76
                            *
The integer 72 was found in position 9.
Please enter an integer value (-1 to quit): 13
12 17 22 25 30 39 40 52 56 72 76 82 84 91 93
                      *
12 17 22 25 30 39 40
          *
12 17 22
    *
12
 *
The integer 13 was not found.
Please enter an integer value (-1 to quit): -1
```

Binary search

Algorithm	Location	Big O
Searching Algorithms:		
Linear Search		***O(n)***
Binary Search		***O(log n)***
Recursive Linear Search		***O(n)***
Recursive Binary Search		***O(log n)***
Sorting Algorithms:		
Selection Sort		***$O(n^2)$***
Insertion Sort		***$O(n^2)$***
Merge Sort		***O(n log n)***
Bubble Sort		***$O(n^2)$***

Fig. 24.12 | Searching and sorting algorithms with Big O values.

n =	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
1	0	1	0	1
2	1	2	2	4
3	1	3	3	9
4	1	4	4	16
5	1	5	5	25
10	1	10	10	100
100	2	100	200	10000
1,000	3	1000	3000	10^6
1,000,000	6	1000000	6000000	10^{12}
1,000,000,000	9	1000000000	9000000000	10^{18}

Fig. 24.13 | Number of comparisons for common Big O notations.