Chapter 16 - Searching, Sorting (and the `vector` Type)

Spring 2022

Objectives

In this chapter, you will:

- Learn about list processing and how to search a list using sequential search
- Explore how to sort an array using the bubble sort and insertion sort algorithms
- Learn how to implement the binary search algorithm
- Become familiar with the `std::vector` type

List Processing

- **List**: a collection of values of the same type
- **Array** is a convenient place to store a list
- Basic list operations:
  - Search the list for a given item
  - Sort the list
  - Insert an item in the list
  - Delete an item from the list
  - Print the list

Searching

- **Sequential search** algorithm:
  - Not very efficient for large lists
  - On average, number of key comparisons is equal to half the size of the list
  - Does not assume that the list is sorted
- If the list is sorted, the search algorithm can be improved

Searching an Array for a Specific Item (1 of 2)

- **Sequential search** (or **linear search**)
  - Searching a list for a given item, starting from the first array element
  - Compare each element in the array with value that is being searched
  - Continue the search until item is found or no more data is left in the list

Binary Search (1 of 2)

- Much faster than a sequential search
- List must be sorted
- "Divide and conquer"
- Compare search item with middle element

- If less than middle: search only upper half of list
- If more than middle: search only lower half of list

Binary Search (2 of 2)

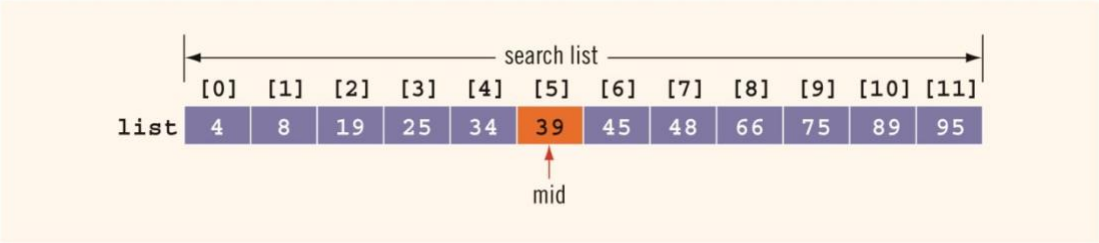


FIGURE 16-14 Search `list`, `list[0]`…`list[11]`

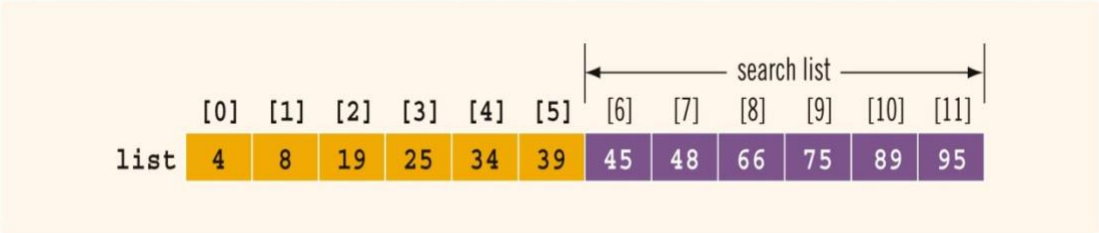


FIGURE 16-15 Search `list`, `list[6]`…`list[11]`

Performance of Binary Search

- If `L` is a sorted list of size 1024
  - Every iteration of the while loop cuts the size of the search list by half
  - At most, 11 iterations to determine whether x is in L
  - Binary search will make 22 comparisons at most
- If `L` has 1,048,576 elements
  - Binary search makes 42 item comparisons at most
- For a sorted list of length `n`:
  - Maximum number comparisons is `2log2n + 2`

Sorting

- **Selection sort**: rearrange the list by selecting an element and moving it to its proper position
- Steps for a selection sort:
  - Find the smallest element in the unsorted portion of the list
  - Move it to the top of the unsorted portion by swapping with the element currently there
  - Start again with the rest of the list

Selection Sort

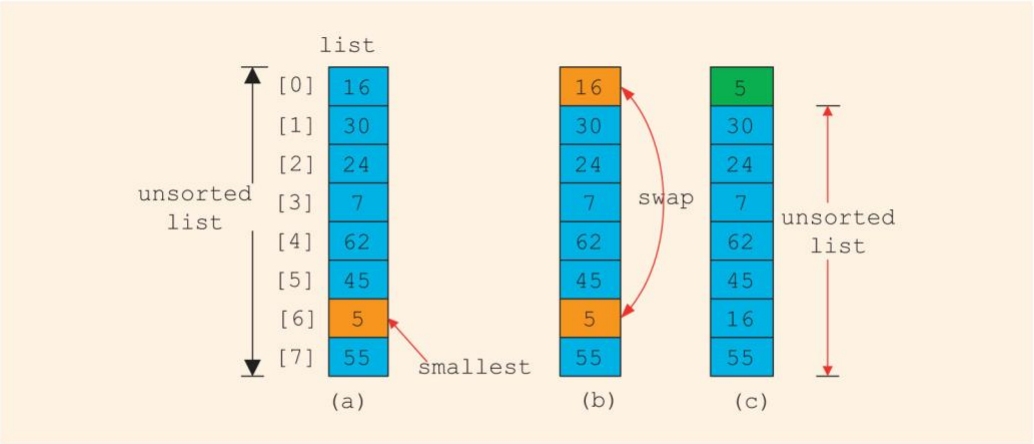


FIGURE 8-10 Elements of `list` during the first iteration

Bubble Sort (1 of 7)

- `list[0]...list[n - 1]`
  - List of n elements, indexed 0 to n – 1
  - Example: a list of five elements (Figure 16-1)

list

list[0]	10
list[1]	7
list[2]	19
list[3]	5
list[4]	16

Bubble Sort (2 of 7)

- Series of `n - 1` iterations
  - Successive elements `list[index]` and `list[index + 1]` of list are compared
  - If `list[index] > list[index + 1]`
    - Swap `list[index]` and `list[index + 1]`
  - Smaller elements move toward the top (beginning of the list)
  - Larger elements move toward the bottom (end of the list)

Bubble Sort (3 of 7)

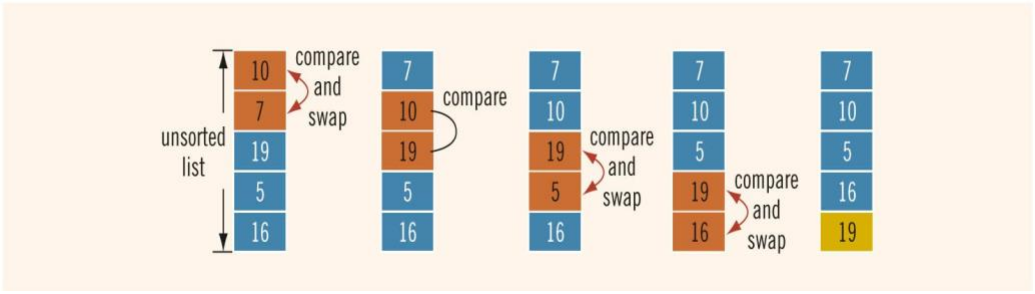


FIGURE 16-2 Elements of `list` during the first iteration

Bubble Sort (4 of 7)

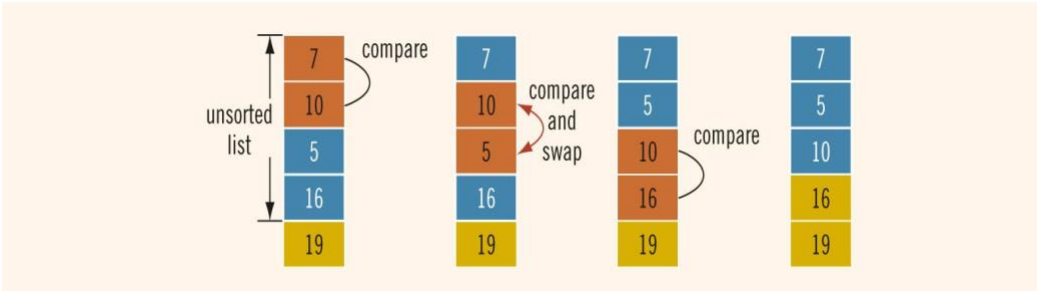


FIGURE 16-3 Elements of `list` during the second iteration

Bubble Sort (5 of 7)

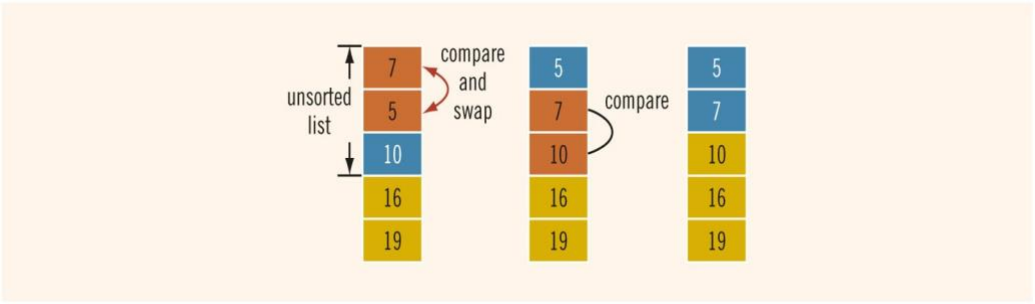


FIGURE 16-4 Elements of `list` during the third iteration

Bubble Sort (6 of 7)

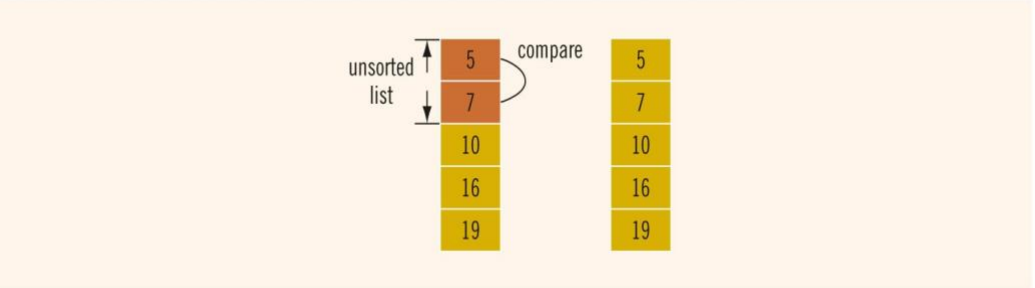


FIGURE 16-5 Elements of list during the fourth iteration

Bubble Sort (7 of 7)

- List of length `n`
  - Exactly `n(n - 1) / 2` key comparisons
  - On average `n(n - 1) / 4` item assignments
- If `n == 1000`
  - 500,000 key comparisons and 250,000 item assignments
- Can improve performance if we stop the sort when no swapping occurs in an iteration

Insertion Sort (1 of 8)

• Sorts the list by moving each element to its proper place

FIGURE 16-6 `List`

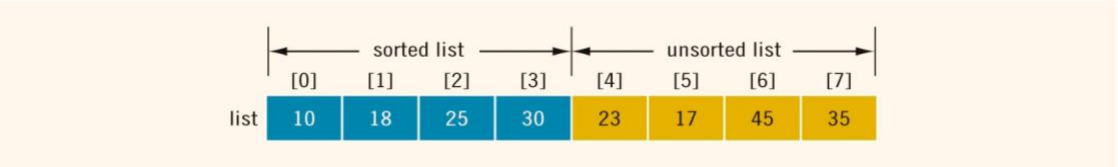


FIGURE 16-7 Sorted and unsorted portion of `List`

Insertion Sort (2 of 8)

- Consider the element `list[4]`
  - First element of unsorted list
  - `list[4] < list[3]`
    - Move `list[4]` to proper location at list[2]

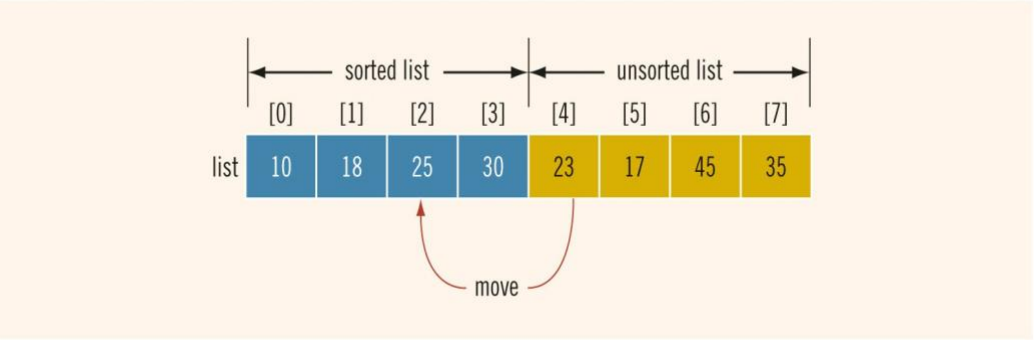


FIGURE 16-8 Move `List[4]` into `List[2]`

Insertion Sort (3 of 8)

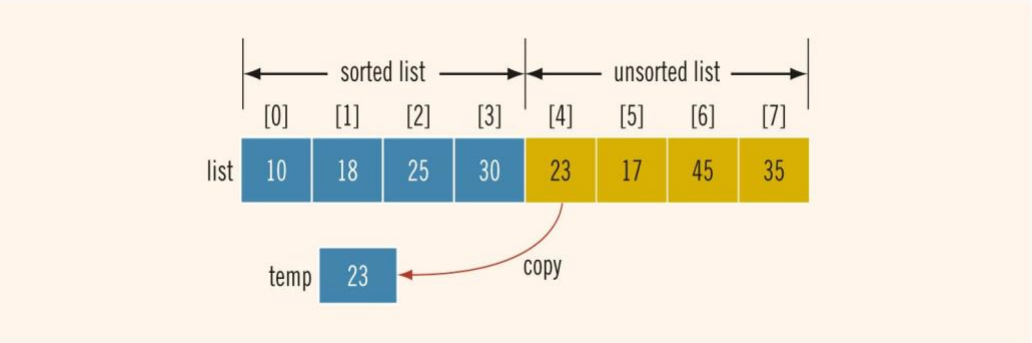


FIGURE 16-9 Copy `list[4]` into `temp`

Insertion Sort (4 of 8)

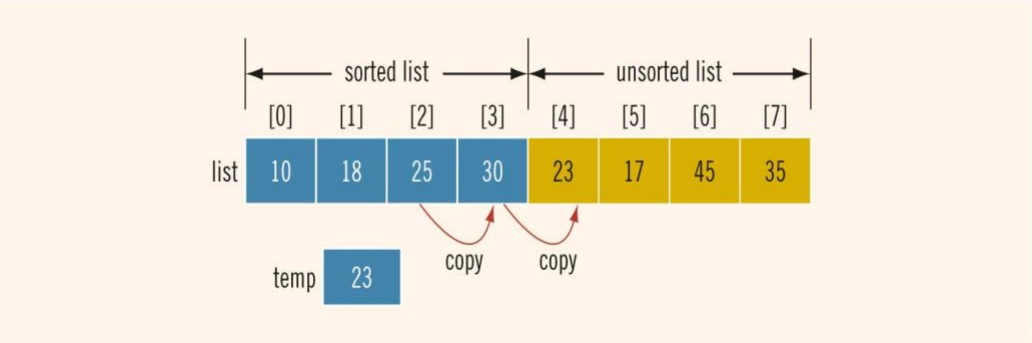


FIGURE 16-10 `list` before copying `list[3]` into `list[4]` and then `list[2]` into `list[3]`

Insertion Sort (5 of 8)

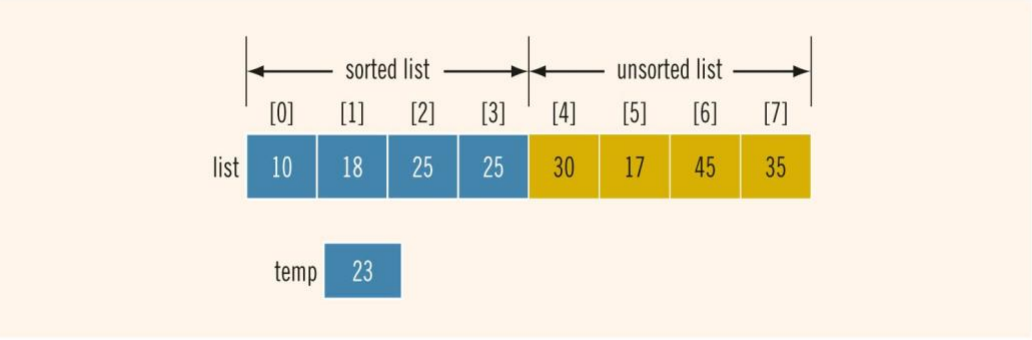


FIGURE 16-11 `list` after copying `list[3]` into `list[4]` and then `list[2]` into `list[3]`

Insertion Sort (6 of 8)

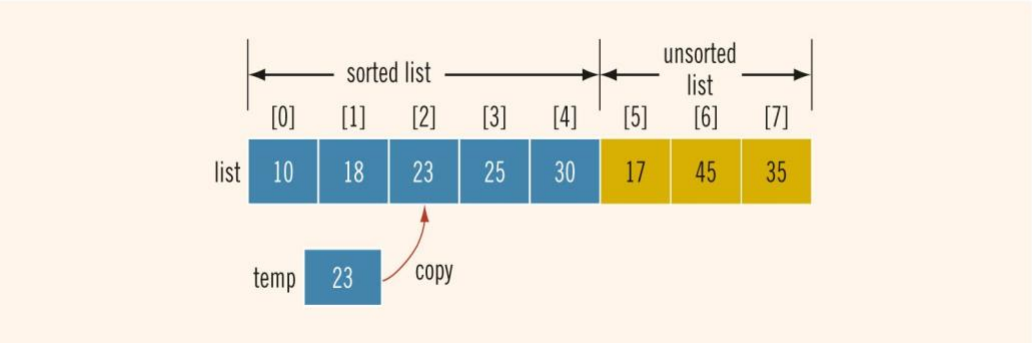


FIGURE 16-12 `List` after copying temp into `List[2]`

Insertion Sort (7 of 8)

- During the sorting phase, the array is divided into two sublists: sorted and unsorted
  - Sorted sublist elements are sorted
  - Elements in the unsorted sublist are to be moved into their proper places in the sorted sublist, one at a time

Insertion Sort (8 of 8)

- List of length `n`
  - About `(n2 + 3n - 4) / 4` key comparisons
  - About `n(n - 1) / 4` item assignments
- If `n == 1000`
  - 250,000 key comparisons
  - 250,000 item assignments

Merge Sort (1 of 4)

- Merge Sort is a Divide and Conquer algorithm.
  1. Divide the unsorted list into n sublists, each containing one element (a list of one element is considered sorted).
  2. Repeatedly merge sublists to produce new sorted sublists until there is only one sublist remaining. This will be the sorted list.
- The merge is a key process that assumes that list[l..m] and list[m+1..r] are sorted and merges the two sorted sublists into one.

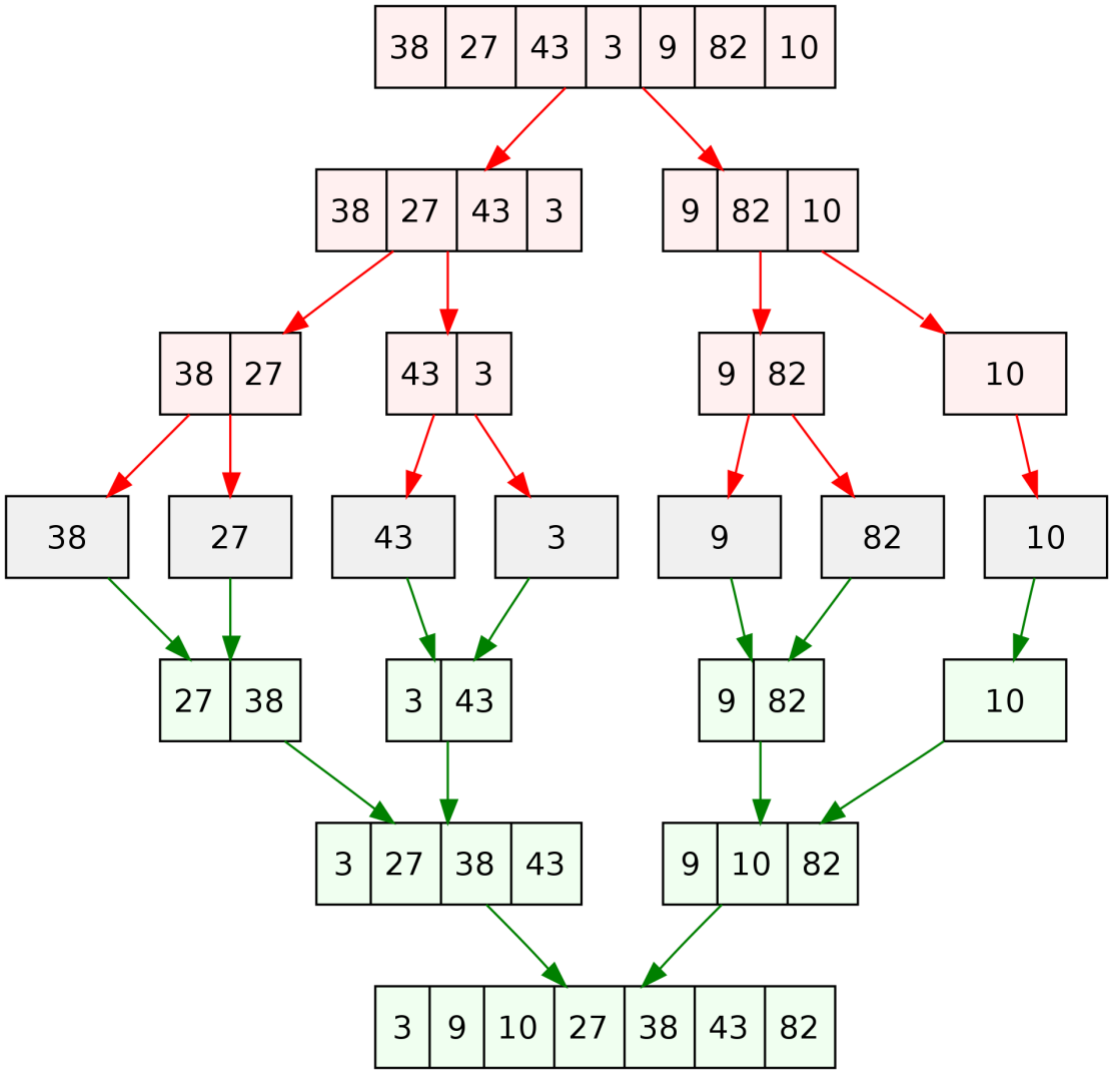
Merge Sort (2 of 4)

- Algorithm

```
MergeSort(arr[], l,  r)
If l < r:
 1. Find the middle point to divide the array into two halves:
    - middle m = (r - l) / 2
 2. Call merge_sort for first half:
    - Call merge_sort(arr, l, m)
 3. Call merge_sort for second half:
```

```
   - Call merge_sort(arr, m + 1, r)
4. Merge the two halves sorted in step 2 and 3:
   - Call merge(arr, l, m, r)
```

Merge Sort (3 of 4)



Top-down merge sort algorithm

Merge sort (4 of 4)

- Merge sort requires $\Theta(n)$ auxiliary space (i.e., twice as much memory).
- Merge sort is a stable sort and is more efficient at handling slow-to-access sequential media.
- Merge sort is often the best choice for sorting a linked list
  - in this situation it is relatively easy to implement a merge sort in such a way that it requires only $\Theta(1)$ extra space, and

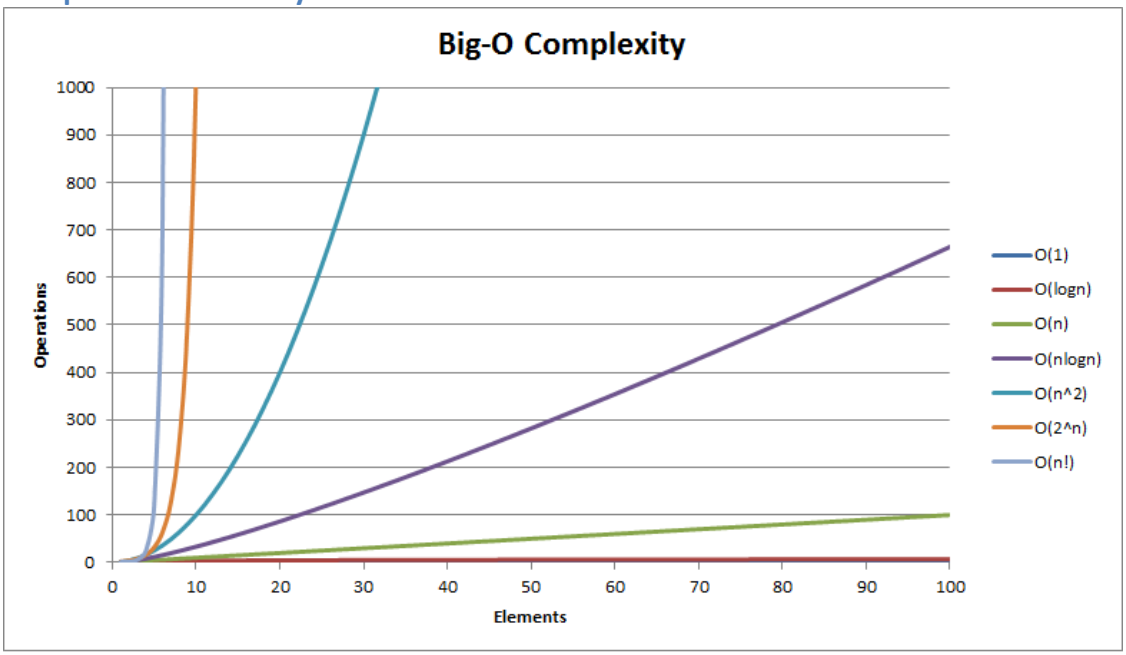
- the slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Comparative Efficiency

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	O(1)	O(n)	O(n)	O(1)
Binary Search	O(1)	O(log n)	O(log n)	O(1)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Quick Sort	O(nlogn)	O(nlogn)	O(n^2)	O(log n)
Heap Sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)
Tim Sort	O(n)	O(nlogn)	O(nlogn)	O(n)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)	O(1)

Efficiency of common searching and sorting algorithms

Comparative Efficiency



Big-O Complexity

`std::vector` type (class) (1 of 4)

- Only a fixed number of elements can be stored in an array
- Inserting and removing elements causes shifting of remaining elements
- `std::vector` type implements a list. Can be referred to by:
  - vector container
  - vector
  - vector object
  - object

`std::vector` type (class) (2 of 4)

Statement	Effect
`vector<elemType> vecList;`	Creates the empty **`vector`** object, **`vecList`**, without any elements.
`vector<elemType> vecList(otherVecList);`	Creates the **`vector`** object, **`vecList`**, and initializes **`vecList`** to the elements of the vector **`otherVecList`**. **`vecList`** and **`otherVecList`** are of the same type.
`vector<elemType> vecList (size);`	Creates the **`vector`** object, **`vecList`**, of size **`size`**. **`vecList`** is initialized using the default values.
`vector<elemType> vecList(n, elem);`	Creates the **`vector`** object, **`vecList`**, of size **`n`**. **`vecList`** is initialized using **`n`** copies of the element **`elem`**.

TABLE 16-1 Various Ways to Declare and Initialize a `std::vector` Object

`std::vector` type (class) (3 of 4)

Expression	Effect
`vecList.at(index)`	Returns the element at the position specified by **`index`**.
`vecList[index]`	Returns the element at the position specified by **`index`**.
`vecList.front()`	Returns the first element. (Does not check whether the object is empty.)
`vecList.back()`	Returns the last element. (Does not check whether the object is empty.)
`vecList.capacity()`	Returns the total number of elements that can be currently added to **`vecList`**.
`vecList.clear()`	Deletes all elements from the object.
`vecList.push_back(elem)`	A copy of **`elem`** is inserted into **`vecList`** at the end.
`vecList.pop_back()`	Deletes the last element of **`vecList`**.

TABLE 16-2 Operations on a vector Object

std::vector type (class) (4 of 4)

Expression	Effect
`vecList.empty()`	Returns `true` if the object **`vecList`** is empty, and `false` otherwise.
`vecList.size()`	Returns the number of elements currently in the object **`vecList`**. The value returned is an `unsigned int` value.
`vecList.max_size()`	Returns the maximum number of elements that can be inserted into the object **`vecList`**.

TABLE 16-2 Operations on a vector Object (cont'd.)

Vectors and Range-Based for loops

- Can use range-based for loop in C++11 Standard to process vector elements

```
for (auto item : list) { // for each item in list
    cout << item << ' ';
}
cout << '\n';
```

- Can initialize a vector object with an `initializer_list`. Examples:

```
std::vector<int> intList = {13, 75, 28, 35};  // canonical
std::vector<int> intList {13, 75, 28, 35};    // modern C++
auto intList = std::vector{13, 75, 28,35};    // using auto
```

Summary (1 of 2)

- List
  - Set of elements of the same type
- Sequential search
  - Searches each element until item is found
- Sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort

Summary (2 of 2)

- Binary search
  - Much faster than sequential search
  - Requires that the list is sorted
- `std::vector` type
  - Implements a list

- Can increase/decrease in size during program execution
- Must specify the type of object the vector stores

References

- https://en.cppreference.com/w/cpp/algorithm/find
- https://en.cppreference.com/w/cpp/algorithm/binary_search
- https://en.cppreference.com/w/cpp/algorithm/sort
- https://en.cppreference.com/w/cpp/container/vector
- https://www.youtube.com/watch?v=kPRA0W1kECg
- https://en.wikipedia.org/wiki/Bubble_sort
- https://en.wikipedia.org/wiki/Insertion_sort
- https://en.wikipedia.org/wiki/Merge_sort
- https://en.wikipedia.org/wiki/Quicksort
- https://en.wikipedia.org/wiki/Selection_sort
- https://en.wikipedia.org/wiki/Time_complexity
- https://www.bigocheatsheet.com/

Questions?