Searching and Sorting

Mason Vail, Boise State University Computer Science

Searching

Searching is the process of locating a target element in a collection or determining that it is not present by comparing the target to candidate elements in the collection.

Comparisons are typically made using the equals() method, the compareTo() method of objects that implement the *Comparable* interface, or by using an external *Comparator* and its compare() method.

An efficient search performs no unnecessary comparisons.

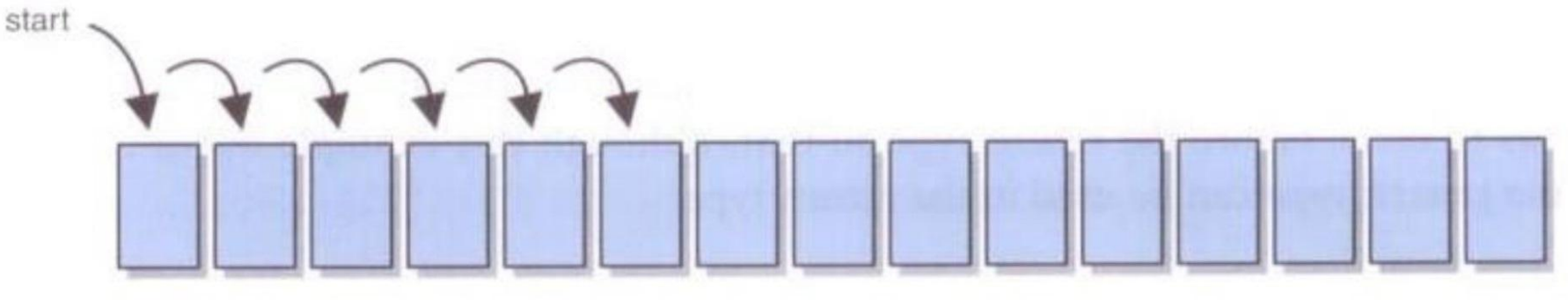
Generic Methods

If a method (such as a general purpose search method) expects a generic type argument, it must declare the expected generic in its method header, along with any specific restrictions on the allowed generic type. For example, this static utility method declares that it expects generic arguments that must be *Comparable*:

```
public static <T extends Comparable<T>> T linearSearch(T[] data, T target) { ... }
```

Linear Search

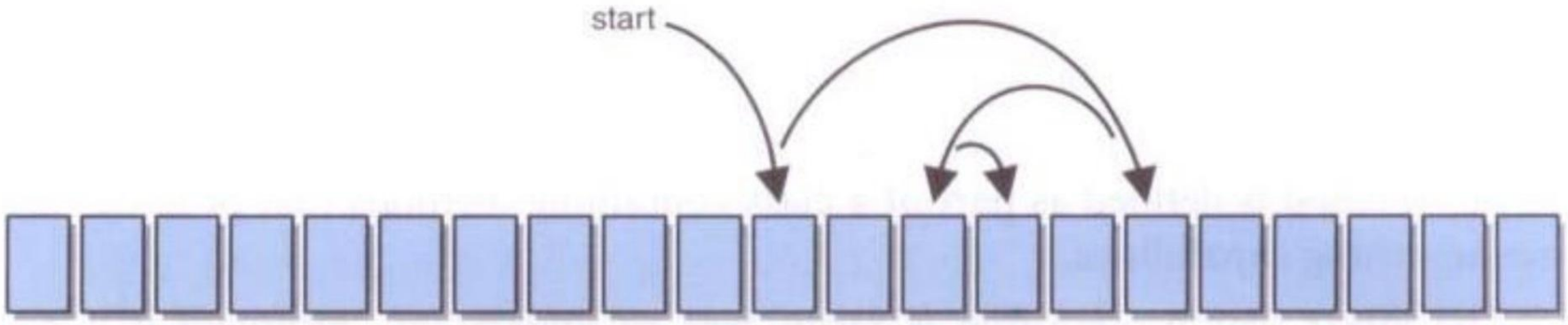
When no assumptions are made about the order of elements within a collection, the only reliable approach to search is to compare every candidate element to the target element, one at a time, until it is found or there are no remaining elements to compare. This is a linear search and its order is O(n).



Java Foundations, 3rd Edition, Lewis/DePasquale/Chase

Binary Search

A little knowledge is a powerful thing. If we know a list is in sorted order, we can be much more efficient than a linear search. In a binary search, we eliminate half of the remaining candidate pool with each comparison, such that the algorithm's order is O(log n). Binary searches are typically implemented recursively.



Java Foundations, 3rd Edition, Lewis/DePasquale/Chase

Sorting

Sorting is the process of arranging elements according to specific ordering criteria. Like searches, sorts typically use comparisons defined in *Comparable* element implementations or stand-alone *Comparator* objects.

Most sorting algorithms for lists fall into one of two types:

- **Sequential** (or quadratic) sorts are characterized by nested loops and may compare all elements to all other elements in their worst cases for $O(n^2)$ runtime. They are simple and have low overhead, but do not scale well.
- **Logarithmic** sorts (more accurately, log-linear) are characterized by recursive divide-and-conquer algorithms and typically have $O(n \log_2 n)$ runtime. As the input size increases, their efficiency overcomes their initial overhead.

Sequential Sort 1 - Selection Sort

Concept: Find the next smallest value from the unsorted region and place it in the next sorted position. Repeat until all positions have been filled.

```
for( int idxToFill = 0; idxToFill < array.length-1; idxToFill++ ) { //for each idx to fill
        int idxOfMin = idxToFill; //until something smaller is found
        for( int idxNext = idxToFill+1; idxNext < array.length; idxNext++ ) {
                if( array[idxNext].compareTo( array[idxOfMin]  < 0 ) { //something smaller is found
                        idxOfMin = idxNext; //remember idx of min value
                }
        }
        swap( array, idxOfMin, idxToFill );
}
```

Sequential Sort 1 - Selection Sort

Selection Sort has only minimal difference between growth functions of best case and worst case scenarios.

The range of indexes explored (number of loop iterations) and the number of comparisons made is unaffected by the original ordering of elements. The only difference between cases is how often a smaller value is found and the index of the minimal value is updated. In an already-sorted array, the condition is never true. In a reverse-sorted array, the condition is always true.

In any case, the order is $O(n^2)$.

Sequential Sort 2 - Insertion Sort

Concept: Take the next unsorted element and find its appropriate insertion position in the sorted region by backing it up until it is greater than or equal to its predecessor or the beginning of the range is reached.

```
for( int idxUnsorted = 1; idxUnsorted < array.length; idxUnsorted++ ) {
        T valToInsert = array[idxUnsorted];
        int idxInsert = idxUnsorted;
        while( idxInsert > 0 && array[idxInsert-1].compareTo( valToInsert ) > 0 ) {
                array[idxInsert] = array[idxInsert-1]; //shift larger elements back
                idxInsert--;
        }
        array[idxInsert] = valToInsert;
}
```

Sequential Sort 2 - Insertion Sort

Unlike Selection Sort, Insertion Sort has dramatically different best case and worst case scenario growth functions. By leveraging the ever-growing sorted region as the algorithm progresses, the total number of comparisons can be greatly reduced.

In the best case scenario of an already-sorted array, the condition check of the inner loop will always fail, resulting in a O(n) runtime for the algorithm. In essence, the algorithm simply confirms the already-sorted order. For the worst case reverse-ordered array, every element must be backed all the way to the beginning and the maximum number of comparisons and shifts occur for $O(n^2)$ runtime.

Due to its best-case efficiency, Insertion Sort is often used when elements are expected to be in nearly-sorted order already.

Sequential Sort 3 - Bubble Sort

Concept: Make pairwise comparisons and swaps of out-of-order elements through the unsorted range, “bubbling” the largest unsorted element to the top. A classic example of naive inefficiency, maximizing comparisons and swaps while always making progress.

```
for( int idxToFill = array.length-1; idxToFill > 0; idxToFill-- ) {
        for( int idxCurrent = 0; idxCurrent < idxToFill; idxCurrent++ ) {
                if( array[ idxCurrent ].compareTo( array[ idxCurrent+1 ] ) > 0 ) {
                        swap( array, idxCurrent, idxCurrent+1 );
                }
        }
}
```

Sequential Sort 3 - Bubble Sort

The total number of comparisons for Bubble Sort is similar to that of Selection Sort and is unaffected by the original ordering of array elements.

In the best case scenario of an already-sorted array, no swaps are made, but the unavoidable pairwise comparisons result in $O(n^2)$ runtime, nonetheless, for a best-case growth function comparable to Selection Sort.

In the worst case scenario of a reverse-ordered array, however, the number of swaps is maximized. In the first pass through the outer loop, for example, n-1 comparisons are made and n-1 swaps take place, with the end result being only that the largest element has been moved to the end of the array. Runtime remains $O(n^2)$, but performance is noticeably slower than other quadratic sorts.

Logarithmic Sort 1 - Quick Sort

Concept: Divide-and-conquer by choosing a single element to serve as a pivot (also called partition) element. All elements smaller than the pivot are placed on its left and all other elements are placed on its right. Recursively quick sort the left side and right side.

- Choose a pivot element.
- Compare all other elements to the pivot. Place smaller elements on the left and other elements on the right.
- Recursively apply the Quick Sort algorithm to the left side and right side.

Logarithmic Sort 1 - Quick Sort

Quick Sort performs exceptionally well when the number of elements partitioned into the left and right sides (partitions) are nearly equal. When this is achieved, Quick Sort has a $O(n \log_2 n)$ runtime and is, in fact, widely regarded as the fastest general-purpose sorting algorithm on average (especially when sorting arrays in-place). The default sorting algorithms in programming language standard libraries are usually Quick Sorts.

However, consistently balanced partitions are *critical* to efficiency and a production-ready implementation must make efforts to ensure good pivot choices likely to produce balanced partitions. This is because highly lopsided partitions can result in $O(n^2)$ worst-case runtime!

Logarithmic Sort 2 - Merge Sort

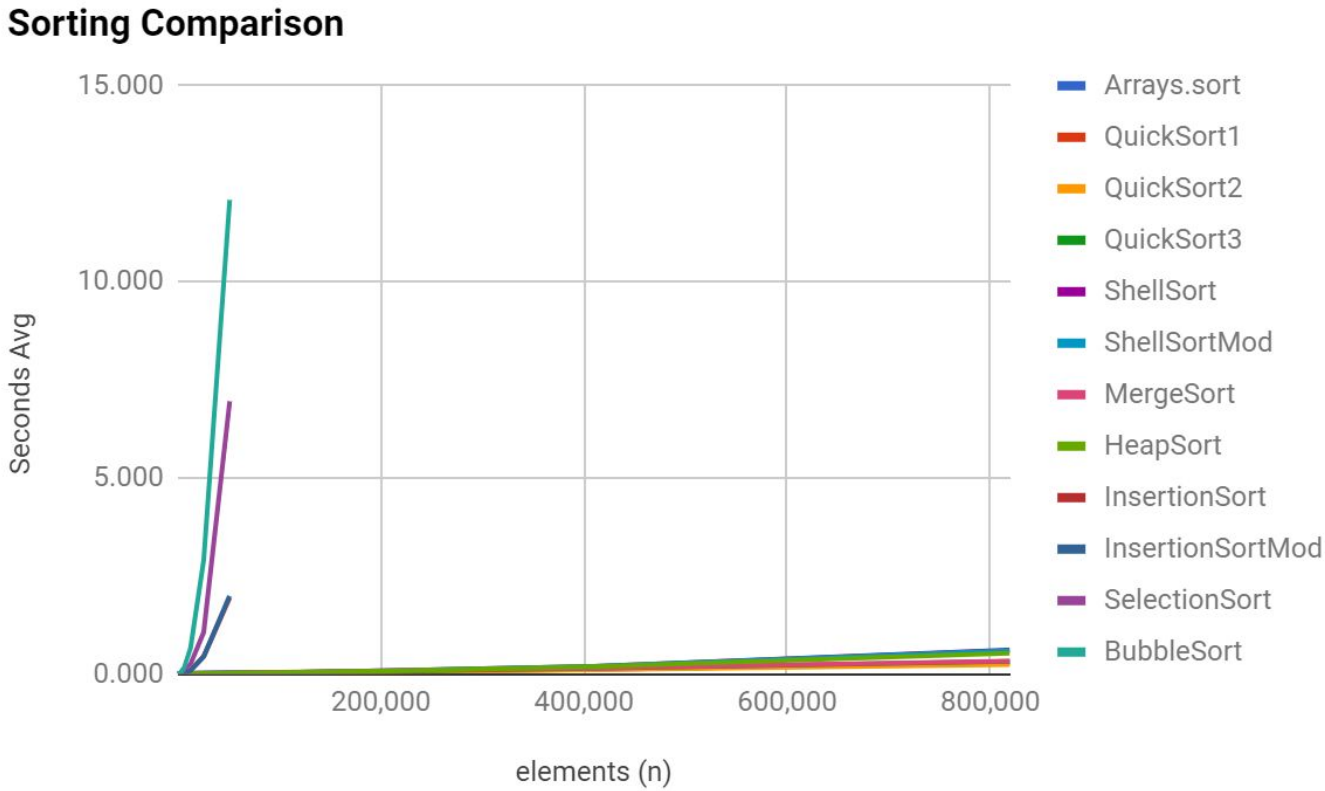
Concept: Divide-and-conquer by guaranteeing even subdivisions and minimizing comparisons. Split elements evenly into a left side and right side. Recursively merge sort each half. When both sides are sorted, merge elements back into the original list by comparing the first element in each side and adding the smallest element to the original list until both sides are exhausted.

- Divide the original list into two halves
- Recursively apply the Merge Sort algorithm to each half
- Merge the sorted half lists back into the original list by repeatedly taking the smaller of the two first elements of the halves and adding it back into the original list until both halves are emptied

Logarithmic Sort 2 - Merge Sort

Merge Sort guarantees even divisions in its divide-and-conquer strategy and minimizes the number of comparisons between elements. Its best and worst cases are both $O(n \log_2 n)$ and it has the further property of maintaining stable ordering of equivalent elements. In practice, its memory overhead (specifically when sorting arrays) can cause it to lag slightly behind Quick Sort in performance. However, most stable sort algorithms in programming language standard libraries are Merge Sorts.

Comparing Sequential Sorts to Logarithmic Sorts



Searching and Sorting

Mason Vail, Boise State University Computer Science