Sorting

Professor Frank Kreimendahl

School of Computing and Data Science
Wentworth Institute of Technology

November 28, 2022

Sorting Techniques

Sorting

- Problem: Given a set of values, arrange them from smallest to largest
- The sorting problem is popular in computer science for several reasons:
  - Humans like ordered information
  - There are many techniques to solve the problem
  - These solutions provide examples of different algorithm techniques
  - Different solutions give opportunities to study algorithm complexity

Java Sorting

- Java's built-in `Arrays` and `Collections` classes provides a sorting method for arrays and lists
- Sorting arrays with primitive types uses quicksort algorithm
- Sorting lists and arrays with objects uses mergesort algorithm
- Both of these are efficient algorithms – the number of comparison/copy operations is minimized

Java `Arrays` Sorting

Method sort in Class Arrays	Behavior
`public static void sort(int[] items)`	Sorts the array `items` in ascending order.
`public static void sort(int[] items, int fromIndex, int toIndex)`	Sorts array elements `items[fromIndex]` to `items[toIndex]` in ascending order.
`public static void sort(Object[] items)`	Sorts the objects in array `items` in ascending order using their natural ordering (defined by method `compareTo`). All objects in `items` must implement the `Comparable` interface and must be mutually comparable.
`public static void sort(Object[] items, int fromIndex, int toIndex)`	Sorts array elements `items[fromIndex]` to `items[toIndex]` in ascending order using their natural ordering (defined by method `compareTo`). All objects must implement the `Comparable` interface and must be mutually comparable.
`public static <T> void sort(T[] items, Comparator<? super T> comp)`	Sorts the objects in `items` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`.
`public static <T> void sort(T[] items, int fromIndex, int toIndex, Comparator<? super T> comp)`	Sorts the objects in `items[fromIndex]` to `items[toIndex]` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`.

Java `List` Sorting

Method sort in Class Collections	Behavior
`public static <T extends Comparable<T>> void sort(List<T> list)`	Sorts the objects in `list` in ascending order using their natural ordering (defined by method `compareTo`). All objects in `list` must implement the `Comparable` interface and must be mutually comparable.
`public static <T> void sort (List<T> list, Comparator<? super T> comp)`	Sorts the objects in `list` in ascending order as defined by method `comp.compare`. All objects must be mutually comparable.

compareTo Example Class

```
public class Person implements Comparable<Person> {
  // Data Fields
  /* The last name */
  private String lastName;
  /* The first name */
  private String firstName;
  /* Birthday represented by an int from 1 to 366 */
  private int birthDay;

  // Methods
}
```

compareTo Example Method

```
/** Compares two Person objects based on names. The
    result is based on the last names if they are
    different, using first names as a tie-breaker.
    @param obj The other Person
    @return A negative int if this person's name
      precedes the other person's name;
      0 if the names are the same;
      a positive int if this person's name follows
      the other person's name.
*/
@Override
public int compareTo(Person other) {
  // Compare this Person to other using last names.
  int result = lastName.compareTo(other.lastName);
  // Compare first names if last names are the same.
  if (result == 0)
    return firstName.compareTo(other.firstName);
  else
    return result;
}
```

Comparator Example Class

```
import java.util.Comparator;

public class CmpPerson implements Comparator<Person> {
  /** Compare two Person objects based on birth date.
    @param left The left-hand side of the comparison
    @param right The right-hand side of the comparison
    @return A negative int if the left person's birthday
      precedes the right person's birthday;
      0 if the birthdays are the same;
      a positive int if the left person's birthday
      follows the right person's birthday.
  */
  @Override
  public int compare(Person left, Person right) {
   return left.getBirthDay() - right.getBirthDay();
  }
}
```

Selection Sort

Selection Sort

- Selection sort is a simple sorting algorithm
- Given an array, it will reorder the values from smallest to largest
- Look through the entire array for the smallest value, and swap that value to the front
- Repeat this operation with the remaining array
- Stop when there are no remaining values

Algorithm

SELECTIONSORT(A)

```
for fill in 0 to A.length-2 do
    posMin ← fill
    for next in fill to A.length-1 do
        if A[next] < A[posMin] then
            posMin ← next
    swap A[posMin] with A[fill]
```

Insertion Sort

Insertion Sort

- Insertion sort is another simple sorting algorithm
- Given an array, it will reorder the values from smallest to largest
- Select a value in the unsorted array and shift values in the sorted array to make room for it
- Repeat this operation with the remaining values
- Stop when there are no remaining values

Algorithm

INSERTIONSORT(A)

```
for all elements e of A do
    nextPos ← location of e
    while nextPos > 0 and element at nextPos - 1 > e do
        shift element at nextPos - 1 to nextPos
        decrement nextPos
    insert e at nextPos
```

Merge Sort

Merge Sort

- Merge sort is less straighforward to describe
- It runs more efficiently than insertion or selection sort though – fewer steps to sort the same array
- Split the array in half
- Sort the left half
- Sort the right half
- Merge the two halves to give a fully sorted array

Merge Operation

- A *merge* operation is a common data processing operation
- Two lists are given, each of which is sorted on its own
- The merge operation “zips” these lists together
- The result is one large list that is fully in order

Merge Algorithm

MERGE(A, B)

```
while both A and B have values left do
    add smaller of front value of A and B to C
    access next value of A or B, whichever was smaller
copy remaining values in A to C
copy remaining values in B to C
return C
```

Merge Sort

- One single merge operation isn't enough to sort everything
- It assumes the two halves of the array are already sorted
- Use recursion to run mergesort on both halves of an array before merging it

Merge Sort Algorithm

```
MERGESORT(table)
if range > 1 then
    halfSize ← tableSize/2
    leftTable ← table[0..halfSize-1]
    rightTable ← table[halfSize..tableSize]
    MERGESORT(leftTable)
    MERGESORT(rightTable)
    table ← MERGE(leftTable, rightTable)
```

Quicksort

Quicksort

- Quicksort shares some features with merge sort
- It usually runs more efficiently than insertion or selection sort as well
- Partition the array into two pieces based on a pivot value
- Sort the left piece
- Sort the right piece

Partition Algorithm

PARTITION(table, first, last)

pivot $\leftarrow$ table[first]
up $\leftarrow$ first, down $\leftarrow$ last
while up $<$ down **do**
    increment up until table[up] $>$ pivot
    decrement down until table[down] $<$ pivot
    if up $<$ down **then**
        swap table[up] and table[down]
swap table[first] and table[down]
return down

Quicksort

- One single partition operation isn't enough to sort everything
- It puts the pivot in the proper place, but the two partitions might still be scrambled
- Use recursion to run quicksort on both parts of an array after partitioning it
- Note that the pivot is not guaranteed to be in the center

Quicksort Algorithm

QUICKSORT(table, first, last)

```
if first < last then
    pivIndex ← PARTITION(table, first, last)
    QUICKSORT(table, first, pivIndex - 1)
    QUICKSORT(table, pivIndex + 1, last)
```