CS 228: Introduction to Data Structures
Lecture 11
Friday, September 16, 2016

A Hierarchy or Running Times

We can establish a rough hierarchy of function classes based on their relative growth rates.

- ***Constant***, O(1), functions don't grow at all.

- ***Logarithmic***, O(log n), functions are slower growing that linear functions.

- ***Linear***, O(n), functions are slower growing than O(n log n) functions.

- ***Linearithmic***, O(n log n) functions are slower growing than quadratic functions.

- ***Polynomial functions*** — i.e., O(n^k) functions, k constant — grow more slowly than

- ***Exponential functions*** — i.e., α^n functions, $\alpha > 1$.

Where the time complexity of an algorithm falls in this hierarchy can have a big practical impact.

Some Rules of Thumb for Algorithm Analysis

- O(1) denotes ***constant time***. Any operation *not* dependent on the input size falls in this category; e.g., assignments, comparisons, and increments.

- A polynomial is always big-O of its leading term.

- For a O(f) operation followed by an O(g) operation, you can ignore the smaller one. E.g., $O(n^2 + n)$ is $O(n^2)$.

- If a O(f) operation is repeated O(g) times, the total time is $O(f \cdot g)$. E.g., if an $O(n^2)$ operation is performed O(n log n) times, the whole thing is $O(n^3 \log n)$.

- If the problem size n is decreased by a ***constant factor*** at each step, the number of steps is O(log n). *Example:* binary search.

Sorting

To ***sort*** an n-element array `A[0..n−1]` is to rearrange its elements so that

```
A[0] ≤ A[1] ≤ A[2] ≤ . . . ≤ A[n−1]
```

The need to sort numbers, strings, and other records arises frequently. The entries in your phone’s contact list are sorted. Spreadsheets and databases allow you to sort the records according to any field the you desire. Google sorts query results by their “relevance”. Sorting is a preprocessing step in hundreds of algorithms.

Modern languages normally have good sorting routines built into their libraries, so nowadays we rarely have to implement sorting algorithms. However, the study of sorting is still useful.

- Sorting offers great examples of algorithm development and analysis. In fact, sorting is perhaps the simplest fundamental problem that offers a huge variety of algorithms, each with its own inherent advantages and disadvantages.

- The difference between O(n log n) and O(n^2) on a million elements looks like 20 ms versus 15 minutes. This is the difference between a sophisticated and a naive sorting algorithm.

- Quicksort provides good example of using recursion

- Sorting is a good motivation for introducing generic types and methods

- To use library sorting methods you must have a good understanding of certain fundamental concepts; e.g., comparators and stability.

Selection Sort

The idea is simple:

```
for i = 0 to n-1:
    find the smallest element in A[i..n-1]
        and exchange it with A[i].
```

For example,

```
4 3 2 1 -> 1 3 2 4 -> 1 2 3 4 -> 1 2 3 4
```

Here's the pseudocode:

```
SELECTIONSORT(A)
    n = A.length
    for i = 0 to n-1
        min = A[i]
        jmin = i
        for j = i+1 to n-1
            if A[j] < min
                min = A[j]
                jmin = j
        swap A[i] and A[jmin]
```

There are n iterations of the outer loop and the inner loop iterates ≤ n times, doing a constant amount of work per iteration. Thus, the algorithm is $O(n^2)$. More formally, at iteration i of the outer loop there are $n - i - 1$ iterations of the inner loop, each of which does a constant amount of work. Therefore, the number of steps is

$$(n - 1) + (n - 2) + (n - 3) + ... + 2 + 1 = n(n - 1) / 2.$$

(from a formula learned in calculus I). So the running time is indeed $O(n^2)$, verifying our initial rough estimate.

The Java code for selection sort is on the next page.

```
public static void selectionSort(int[] arr)
{
  for (int i = 0; i < arr.length - 1; ++i)
  {
    int minIndex = i;
    for (int j = i+1; j < arr.length; ++j)
    {
      if (arr[j] < arr[minIndex])
      {
        minIndex = j;
      }
    }
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

The `Comparable` Interface

If, instead of `int`s, we wanted to sort `String`s alphabetically, the algorithm would be the same, but the comparisons would be done differently, so that, e.g., "`apple`" would precede "`bear`". The "<" operator will not help us here — it is meant for numbers, not strings — but there is an easy fix: use the `compareTo()` method instead of “<“. We can convert a sorting method for `int` arrays into a sorting method for `String` arrays by

1. replacing “`int`” by “`String`” throughout,

2. using `compareTo()` instead of a numerical comparison operator; e.g., write `arr[j].compareTo(x) < 0` instead of `arr[j] < x`.

```
public static void
selectionSort(String[] arr)
{
  for (int i = 0; i < arr.length - 1; ++i)
  {
    int minIndex = i;
    for (int j = i+1; j < arr.length; ++j)
    {
      if (arr[j].compareTo(arr[minIndex])
            < 0)
      {
        minIndex = j;
      }
    }
    String temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

This works because Java's `String` class implements the `Comparable` interface, which consists of objects that have a ***natural ordering*** — i.e., the alphabetical ordering. `Comparable` objects have a `compareTo()` method that enables us to determine their order relative to any other object of the same type.

The `Comparable` interface has only one method, `compareTo()`.

```
public interface Comparable<T>
{
  int compareTo(T rhs)
}
```

The `Comparable` interface is ***generic***. T is a ***type variable***, which establishes the type of a `Comparable` object. The `compareTo` method allows us to compare an object of type T to another object of type T; its desired behavior is as follows.

```
   x.compareTo(y) < 0      ≈      x < y
   x.compareTo(y) = 0      ≈      x == y
   x.compareTo(y) > 0      ≈      x > y
```

Some familiar classes implement `Comparable`. For instance, `String`, `Integer`, `Float`, `Date`, and `Time` implement `Comparable<String>`, `Comparable<Integer>`, `Comparable<Float>`, `Comparable<Date>`, and `Comparable<Time>`, respectively. In other words, `String`, `Integer`, `Float`, `Date`, and `Time` have a natural ordering.

If we want one of our own classes to be `Comparable`, we have to declare it to implement the `Comparable` interface, and we must provide a `compareTo()` method. This has been done for the `Insect` class: the header has been modified to

```
public abstract class Insect
implements Comparable<Insect>
```

and a `compareTo()` method has been added. See Blackboard.

The `Comparator` Interface

We often need the option of sorting a collection of objects according to multiple criteria. For instance, we might have a collection of objects of the form [name, account number], and we may sometimes need to sort these objects by name, and sometimes by account number. In situations like this, it is convenient to have the option of providing the sorting method with a ***comparator*** that specifies how comparisons between objects should be resolved. In Java, the idea of a comparator is captured by the `Comparator` interface, which has essentially one method[1]:

```
public interface Comparator<T>
{
  int compare(T x, T y);
}
```

If `comp` is a `Comparator` for a type `T`, then we can determine the relative order of two objects x and y of type T by calling `comp.compare(x,y)`. The next table contrasts the desired behaviors of `compareTo` and `compare`.

[1] This is, in fact, only true up to Java 7. Java 8 has a much-expanded `Comparator` interface, with many `default` methods. For instance, there is a `reversed()` method, which returns a comparator that imposes the reverse ordering of this comparator, and a `thenComparing()` method, which returns a lexicographic-order comparator with another comparator. For a complete method listing, see http://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Idea	Using a `Comparable` type	Using a `Comparator` comp
`x < y`	`x.compareTo(y) < 0`	`comp.compare(x, y) < 0`
`x > y`	`x.compareTo(y) > 0`	`comp.compare(x, y) < 0`
`x == y`	`x.compareTo(y) == 0`	`comp.compare(x, y) == 0`

For example, to obtain a sorting sort method that sorts `String`s using a comparator instead of the natural ordering, we just replace `x.compareTo(y)` by `comp.compare(x, y)`, where `comp` is a comparator for type `String`.

```
public static void
  selectionSort(String[] arr,
                Comparator<String> comp)
{
  for (int i = 0; i < arr.length - 1; ++i)
  {
    int minIndex = i;
    for (int j = i+1; j < arr.length; ++j)
    {
      if (comp.compare(arr[j],
                       arr[minIndex]) < 0)
      {
        minIndex = j;
      }
    }
    String temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
  }
}
```

Now, suppose we wish to sort `Strings` by length instead of alphabetically. We first define a comparator that compares `String` objects by length.

```
public class LengthComparator implements
      Comparator<String>
{
  public int compare(String lhs,
                     String rhs)
  {
    return lns.length() – rhs.length();
   }
}
```

Then, we invoke

```
selectionSort(arr, new LengthComparator());
```

Comparators are very useful.   This is why most sorting utilities — as well as other utilities that involve ordering — offer the option to supply an external comparator.