Sorting and Searching

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Simple Sorting

- What does it mean to **sort** our data?
  - Consider an array, `x` of n items:

$$\texttt{x[0]}, \texttt{x[1]}, \texttt{x[2]}, \ldots, \texttt{x[n - 1]}$$

  - `x` is sorted in **ascending** order if

$$\texttt{x[i]} \leq \texttt{x[j]} \text{ for all } i < j$$

  - `x` is sorted in **descending** order if

$$\texttt{x[i]} \geq \texttt{x[j]} \text{ for all } i < j$$

Simple Sorting

- How do we sort?
  - There are many ways of sorting data
    - Sorting has been widely studied in computer science
  - Some algorithms are better than others
    - The most useful measure of "better" here is how long it takes to run
    - The **better algorithms run a lot more quickly** than the poorer ones
  - However, some very simple algorithms are fine if the number of elements to sort is not too large
    - We will look at a simple algorithm here
    - More complicate ones will be CS0445.

Selection Sort

- **Selection Sort** is very intuitive
- Here is the idea behind the Selection Sort
  1. Find the smallest item and swap it into index 0
  2. Find the next smallest item and swap it into index 1
  3. Find the next smallest item and swap it into index 2
  4. ⋮
  5. Find the next smallest item and swap it into index $n-2$
- What about index $n-1$?
- Let's trace it on the board with the following array:

0	1	2	3	4	5	6	7
35	50	20	40	75	10	15	60

An Implementation

- Let's implement one together
- Will do this in a modular way utilizing method
  - One method to find the index of the smallest (next smallest)
  - One method to swap data between two indexes
  - One method to sort given array utilizing the above two methods

Find the Index of the Smallest

- Recall that once we swap the smallest data with index 0, we do not need to include data in the index 0 when we need to find the next smallest
  - Need a method that returns an index of the smallest between two given indexes:

```
public static int findSmallest(int[] array, int startIndex,
                                            int endIndex)
{
    int smallest = array[startIndex];
    int index = startIndex;

    for(int i = startIndex + 1; i <= endIndex; i++)
    {
        if(array[i] < smallest)
        {
            smallest = array[i];
            index = i;
        }
    }

    return index;
}
```

Swap Data between Two Indexes

- Obviously we need to indexes

```
public static void swap(int[] array, int first, int second)
{
    int temp = array[first];
    array[first] = array[second];
    array[second] = temp;
}
```

- With the `swap()` and `findSmallest()` methods, the Selection sort algorithm can be implement with ease

```
public static void selectionSort(int[] array)
{
    int size = array.length;

    for(int i = 0; i < size - 2; i++)
    {
        int indexOfSmallest = findSmallest(array, i, size - 1);
        swap(i, indexOfSmallest);
    }
}
```

Sort Other Types

- From the previous `selectionSort()` method, we can only sort arrays of integers
- What if we want to sort arrays of different types
  - We could write a version of Selection sort for each type
  - Lots of typing, lots of cut and paste
- Note that the only difference in the sorts of different types is the data values and how they are compared
  - If there is a standard way to compare two elements of any comparable types, then we may need only one implementation
    - **Java Generic**
    - We will discuss this later after we discuss polymorphism and interfaces

Recall the Sequential Search

- We stop the search as soon as we find the item
- However, there is a worst case
  - You have to look at every item on the array
    - The last item is the one we are looking for
    - The item we are looking for is not on the array
- We say this is a **linear run-time** or time proportional to n (the number of items in the array)
- Can we do better?
  - If the data is unsorted, NO
    - The item we are looking for can be anywhere in the array
  - What if the data is sorted?

Binary Search

- The idea of **Binary Search** is as follows:
  - Searching for a giving key, K
  - Guess middle item, `a[mid]` in the array
    - If `a[mid] == K`, we found it and are done
    - If `a[mid] < K`, then K must be on the right side of the array
    - If `a[mid] > K`, then K must be on the left side of the array
  - Note that we eliminate half of the array with just one guess
- Let's search for an item with `leftIndex` and `rightIndex`

0	1	2	3	4	5	6	7
10	15	20	35	40	50	60	75

Binary Search

- What if the item is not in the array?
  - We need a stopping condition
- What is happening with each test (guess)
  - Either we move `leftIndex` to the right or
  - We move `rightIndex` to the left
  - Eventually they will **cross**
    - Nothing more to guess
    - Item is not found
- Is this difficult to implement?
  - It is quite simple
  - See author's code: `BinarySearchDemo.java`

Binary Search

- Notes:
  - To use the Binary Search, data must be sorted
    - We can only search an array of integers (for now)
  - We can use Selection Sort to sort an array before searching
    - Again, we can only sort an array of integers (for now)
- Even better if we can have one implementation (method) that can sort various kinds of arrays
- Similarly, one implementation to search various kinds of items
  - Need Java **generic**
  - We will discuss this later once we learn about **inheritance** and **interfaces**

Binary Search

- Is **Binary Search** really an improvement over **Sequential Search**?
  - In Binary Search, each guess remove half of the remaining items
    - In Sequential Search, each guess (not really a guess) only remove one items
  - The total number of guesses cannot exceed the number of times we can cut the array in half until we reach 0 items
    - $32 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ (6 times at most)
    - Generally speaking it is approximately $\log_2 n$ where n is the number of items in an array
  - Compare to sequential search which can be n
  - We will discuss this more in CS 0445 and CS 1501