Joanna Klukowska
joannakl@cs.nyu.edu

Lecture 7: Searching and Sorting Algorithms

Reading materials

Dale, Joyce, Weems:Dale, Joyce, Weems: 10.1-10.5
OpenDSA: 11 (Sorting) and 13 (Searching)
Liang (10): 7 (for searching and quadratic sorts), 25 (comprehensive edition only)

Contents

1 Comparable Interface

When we search for elements in ”collections of elements” or try to sort such ”collections of elements” we need some way of comparing items. The primitive types in Java all come with some natural way of comparing: the numerical types are compared according to their values and the characters are compared according to their UTF-8 codes (which are just numbers). But reference types have to provide a definition of what does smaller and larger mean.

Java provides the Comparable interface that any class can implement. The documentation of that interface is at http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html.

If we want to be able to use Arrays.sort(...) or Collections.sort(...) on an array or an ArrayList, the class has to implement the Comparable interface.

The only requirement of this interface is that the class provides the compareTo(...) method. You should read the entire specification of that method on the Javadoc page, but the most important part of that specification is:

> Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

In general, if your class C implements the Comparable interface, its definition should look something like this:

```java
public class C implements Comparable<C> {

  //definition of class C

  int compareTo ( C obj ) {
    //if this is smaller than obj by some criterion
      return -1;
    //if this is larger than obj by some criterion
      return 1;
    //otherwise they are the same by some criterion
      return 0;
    }

}
```

Source code: See the definition of `Point` class in `Point.java`.

2 Implementing .equals() Method

Every class inherits the .equals() method from the Object class. The Javadoc for the Object class provides a very detailed description of that method and its requirements: http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-. When we implement our own classes, they should always override the naive implementation provided by the Object class (it simply uses the == comparison operator).

In most cases, the result of the .equals() method should be the same as when .compareTo() returns zero. But there are cases in which this is not appropriate (see the definition of the Point class for an example).

You can generate the .equals() method in Eclipse automatically by going to Source menu and selecting Generate hashCode() and equals() option. We will talk about hashCode() method towards the end of the semester. For now, you can ignore it. The method provided by Eclipse is just a template - you may need to modify it depending on your own class definition.

Source code: See the definition of `Point` class in `Point.java`.

3 Searching

We will consider searching for an element in an unsorted and in a sorted array.

3.1 Searching in an Unsorted Array

When we do not know anything about organization of the data in the array, it is hard to predict where we should start the search in order to find the elements as fast as possible. Moreover, if the element is not in the array, we cannot decide that until we check every single potential location.

The algorithm that sequentially checks every location in the array is called **linear search**: it starts at the beginning of the array and proceeds through the the array one element at a time until either it finds what we are looking for, or reaches the end of the array.

3.1.1 Linear Search in Arrays of Primitive Types

Here is the Java source code for linear search algorithm in an array of type int:

```java
/** Find the position in A that holds value K, if any does.
 *  Note, the location of the first occurance of K is returned.
 *  @param A  an array of integers
 *  @param K  value to look for
 *  @return the index at which K is located in array A, or
 *     -1 if K is not found
 */
int linSearch(int[] A, int K) {
  for (int i=0; i<A.length; i++)
    if (A[i] == K)           // if we found it return this position
        return i;
  return -1;                 // otherwise, return -1
}
```

The search in an array of type double would look practically identical, except for the names of the types in the signature of the above method.

3.1.2 Linear Search in Arrays of Objects

If we are searching for an element in an unsorted array of objects (not primitive type variables), then we can no longer use == operator for comparing elements. The type either has to provide the definition of the .equals() method, or the definition of the .compareTo() method. Depending on those definitions the results of searching might be different.

Source code: See the definition of class `Point.java`, and the program `SearchingForPoints.java`.

3.2 Searching in a Sorted Array

If we know that the array is sorted, we can take advantage of it in the search for an element. There is a particular location where the any item is expected to be. If it is not there, then it cannot be anywhere else. Having a sorted array, allows us to find the item much faster if it is located in the array, and to quickly declare that it is not there, if that's the case. (Well, assuming that we use an algorithm that takes advantage of that fact.)

You most likely have seen an algorithm called **binary search**. At each step it eliminates half of the remaining items in the array (unlike linear search which eliminated a single item in each step). We will look at the iterative and recursive implementation of the binary search algorithm.

The general outline of the binary search algorithm follows. Assume that key is the element that we are searching for.

1. If the array is not empty, pick an element in the middle of the current array. Otherwise, go to step 5.
2. If that element is smaller than the key, discard all elements in the left half of the array and go back to step 1.
3. If that element is greater than the key, discard all elements in the right half of the array and go back to step 1.
4. If that element is greater than the key, return its index.
5. If the array is empty, the key is not there.

3.2.1 Iterative Implementation of Binary Search

The following code provides iterative solution to the binary search algorithm. It is designed for the Point class.

```
int binSearchIterative( Point[] points, Point p) {
  int left = 0;
  int right = points.length - 1;
  int mid;

  while ( left != right ) { //array is not empty
    mid = (left+right) / 2;
    if (0 > points[mid].compareTo(p) )       //mid element is smaller than p
      left = mid;                            //discard left half
    else if ( 0 < points[mid].compareTo(p) ) //mid element is larger than p
      right = mid;                           //discard right half
    else return mid;                         //found it, return the index
    }

  return -1;          //did not find it, return -1
}
```

Source code: See the definition of class `Point.java`, and the program `BinarySearchingForPoints.java`.

3.2.2 Recursive Implementation of Binary Search

Something to Think About:
Write a recursive implementation of a binary search.
HINT: this method will require you to provide a public wrapper method and the private recursive method that performs the search.

3.3 Performance of Search Algorithms

If the array has `n` items how many locations will be accessed by ...	linear search	binary search
in the **best** case?	1	1
in the **worst** case?	n	$\log_2 n$
on **average***?	$< n$	$< \log_2 n$

* the average values depend on the actual data distribution

In computer science, when we analyze problems in this way, we do not like to need to worry about the constant multiples. The reason for it is that for really really large values of n multiplying it by $^1/_2$ does not change the value that much. We use an order notation that only indicates the highest power of n in the number of operations performed. The notation is ocalled Big-O notation, and we write $\mathrm{O}(n)$ to indicate that the number of operations performed is proportional to n (that may mean exactly n, or $n/2$, or $10n$, or $2n + 15$ - it does not matter which, because for large values of n, those are all similar.

Using that notation the table above would look as follows:

If the array has `n` items how many locations will be accessed by ...	linear search	binary search
in the **best** case?	$O(1)$	$O(1)$
in the **worst** case?	$O(n)$	$O(\log_2 n)$
on **average**?	$O(n)$	$O(\log_2 n)$

4 Sorting

In the previous course you should have seen at least one sorting algorithm. We will quickly review three most popular quadratic sorts, and then move on to more efficient sort techniques.

4.1 Quadratic (or Slow) Sorts

4.1.1 Bubble Sort

Bubble sort is one of the first algorithms for sorting that people come up with. If we want to sort the array from smallest to largest, we can simply go through it, look at the elements pairwise and if the first in the pair is smaller than the second one, swap them. We may need to repeat the passes through the array up to n times (assuming there are n elements in the array) before the whole array is in sorted order.

The following code implements Bubble Sort for an array of type double.

```java
public static void bubbleSort(double[] list) {
  double tmp;
  //keep going through the entire array
  for (int i = 0; i < (list.length - 1); i++) {
    for (int j = 0; j < list.length - i - 1; j++) {
      //for each pair of elements,
      //swap them, if they are out of order
      if (list[j] > list[j + 1])
      {
        tmp = list[j];
        list[j] = list[j + 1];
        list[j + 1] = tmp;
      }
    }
  }
}
```

Bubble sort happens to be the slowest of the sorting algorithms discussed here. It is hardly ever used in practice.

4.1.2 Selection Sort

Assume again that we want to sort a list from smallest to largest. Selection sort is a sorting algorithm that starts by finding the smallest item on the list and then swaps it with the first element of the list. Then it finds the smallest element in the remaining list (ignoring the first one) and swaps it with the second element on the list. It continues until the remaining unsorted part of the list is empty.

The following code implements Selection Sort for an array of type double.

```java
public static void selectionSort(double[] list) {

  for (int i = 0; i < list.length - 1; i++) {

    // Find the minimum in the list[i...list.length-1]
    double currentMin = list[i];
    int currentMinIndex = i;
    for (int j = i + 1; j < list.length; j++) {
      if (currentMin > list[j]) {
        currentMin = list[j];
        currentMinIndex = j;
      }
    }

    // Swap list[i] with list[currentMinIndex] if necessary;
    if (currentMinIndex != i) {
      list[currentMinIndex] = list[i];
      list[i] = currentMin;
    }
  }
}
```

Something to Think About:
For the purpose of practicing your newly acquired expertise on recursion, rewrite the above method using recursion.

4.1.3 Insertion Sort

Assume again that you want to sort a list from smallest to largest. Insertion sort algorithm divides the array (just in theory) into a sorted part and unsorted part. Initially, the sorted part consists only of the first item in the array (well, if we have only one thing, it has to be in a sorted order). The unsorted part starts as all the remaining elements. The algorithm works by repeatedly picking the first item from the unsorted part and inserting it into the sorted part. Once the unsorted part is empty, the entire array is sorted.

The following code implements Insertion Sort for an array of type double.

```
public static void insertionSort(double[] list) {
  for (int i = 1; i < list.length; i++) {
    /*
      * insert list[i] into a sorted sublist list[0..i-1] so that
      * list[0..i] is sorted.
      */
    double currentElement = list[i];
    int k;
    for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
      list[k + 1] = list[k];
    }

    // Insert the current element into list[k+1]
    list[k + 1] = currentElement;
  }
}
```

4.1.4 Performance of Quadratic Sorts

How many operations does it take to sort an array using one of the methods above?

The title of this section (”Quadratic Sorts”) really gives it away. But how can we tell that the performance (on average and in the worst case) is O $\left(n^2\right)$? Each of those methods has two nested for loops. At least one of those for loops repeats n times (or $n - 1$ times). The inner for loops repeat fewer than n times, but on average around $n/2$ times. This gives us number of operations that is proportional to n^2, hence the name of this section.

Source code: See `QuadraticSortsForDoubles.java` for the example of using the above sort methods.

4.2 Merge Sort

The merge sort algorithm is based on a very simple observation:

- If we have n elements in the array, it takes approximately n^2 operations to sort that array.
- If we divide the original array into two halves and then sort the two halves separately, it takes approximately $2 \times \left(\frac{n}{2}\right)^2$ or $\frac{n^2}{2}$ operations to sort each half. Merging two halves can be done in time proportional to n. The whole process ends with fewer operations than sorting the whole array.

It turns out that if we keep subdividing the array into smaller and smaller parts, we can complete the sorting in time proportional to $n \log_2 n$, or $\mathrm{O}\left(n \log_2 n\right)$ - this is much faster than $\mathrm{O}\left(n^2\right)$.

The **algorithm for merge sort** is as follows:

1. split the array in half
2. sort the left half
3. sort the right half
4. merge two sorted half-arrays into a single sorted array

What do you think will be used for the second and third steps?

The merge sort algorithm is a **divide-and-conquer algorithm**. It takes a large problem and subdivides it into smaller problems. If they are not small enough for immediate solution, it subdivides the smaller problems into even smaller problems, etc.

4.2.1 Merging Two Sorted Arrays

It turns out that the ”most complicated” part of the implementation is in the merging of two sub-arrays. They should be approximately the same size, but may be off by 1 or so.

The first version of the code that we write will need some revisions, but for now, let’s just assume that we are writing a method that takes two sorted arrays and returns another array that contains all the elements of the two parameter arrays in sorted order.

Here is the pseudocode:

```
merge ( array1, array2 )
  create arrayMerged whose size equals (size of array1 + size of array2)
  set index1 to zero (first index in array1)
  set index2 to zero (first index in array2)
  set indexMerged to zero
  while index1 < size of array1   AND  index2 < size of array2
    if array1[index1] < array2[index2]
      set arrayMerged[indexMerged] to array1[index1]
      increment index1
    else
      set arrayMerged[indexMerged] to array2[index2]
      increment index2
    increment indexMerged
  copy remaining elements from array1 to arrayMerged
  copy remaining elements from array2 to arrayMerged
  return arrayMerged
```

This merges two arrays into a single large array. It does not modify the original arrays and allocates new space for the merged array. For the purpose of merge step used in merge sort, we want to avoid using new memory to store and return the merged array. We do not need the original arrays once we merge them, so it would be nice to be able to reuse all that space.

The version of merge method that we use in the merge sort does not actually take two separate arrays as parameters. It operates on a single array and the two sorted parts are just specific ranges of indexes in that array. In the following array, indexes 0 - 4 contain ”one sorted array” and the indexes 5 - 9 contain another sorted array. We would like to merge these two sub-arrays so that all the data stays in the same array.

0	1	2	3	4	5	6	7	8	9
2	4	6	8	10	1	3	5	7	9

Here is the pseudocode that does it (it assumes that the two ranges (leftFirst - leftLast) and (rightFirst - rightLast) are adjacent):

```
merge ( array, leftFirst, leftLast, rightFirst, rightLast )
  create tmp array whose size equals (rightLast - leftFirst + 1 )
  set indexLeft to leftFirst
  set indexRight to rightFirst
  set index to zero
  while indexLeft <= leftLast   AND  indexRight <= rightLast
    if array[indexLeft] < array[indexRight]
      set tmp[index] to array[indexLeft]
      increment indexLeft
    else
      set tmp[index] to array[indexRight]
      increment indexRight
    increment index
  copy elements in range indexLeft-leftLast to tmp
  copy elements in range indexRight-rightLast to tmp
  copy tmp to array starting at leftFirst
```

We used tmp array to store the sorted array, but once the method terminates the array is no longer referenced and will be removed by Java garbage collector (or some other memory management system).

4.2.2 Splitting the Array in Half

The second version of merge method operates on a single array to save space. This suggests that when we are ”splitting” the array, we should not really create duplicate arrays in memory, but rather decide what the ranges of each half-array should be.

4.2.3 Recursion in Merge Sort

So far we discussed the steps 4 and 1 from the merge sort algorithm. Steps 2 and 3 are recursively calling the merge sort on smaller sub-arrays. This guarantees that the recursion will eventually end, but when? What should be the base case for the recursion? Once the sub-arrays on which recursive call is made have only one element remaining, there is nothing to split and sort recursively. This gives the base case for recursive calls: we keep splitting the array into smaller and smaller pieces until we cannot split any more.

4.2.4 Merge Sort Pseudocode

The final step is to specify pseudocode for merge sort based on the algorithm mentioned before. We want to take into account all the details discussed in the previous subsections. As with many recursive problems we will make use of a helper method that calls a recursive method.

Pseudocode for merge sort:

```
mergeSort( array )
  mergeSortRec (array, 0, array.length-1)

mergeSortRec ( array, firstIndex, lastIndex )
  //base case
  if (firstIndex == lastIndex)
    return
  //split the array
  mid = (firstIndex+lastIndex)/2
  mergeSortRec(array, firstIndex, mid)
  mergeSortRec(array, mid+1, lastIndex)
  merge( array, firstIndex, mid, mid+1, lastInd
```

4.2.5 Merge Sort Visualization

OpenDSA website has a very good visualization of step by step application of the merge sort algorithm to a numerical array. You can find it at http://algoviz.org/OpenDSA/Books/Everything/html/Mergesort.html.

4.2.6 Merge Sort Performance

The actual work in mergesort algorithm is done by merge method. The merge method itself takes time proportional to the number of elements in the two sub-arrays that it is merging.

When we split the original array of n elements into two sub-arrays and then need to merge the sorted sub-arrays, it the merge takes time proportional to n, or in Big-O notation $\mathrm{O}(n)$.

How about all the smaller parts?

Each of the recursive calls (that is applied to the first half and the second half of the original array) merges sub-arrays of total length approximately $^n/_2$ and since there are 2 such calls, the time it takes is proportional to $2 \times {}^n/_2$ or n.

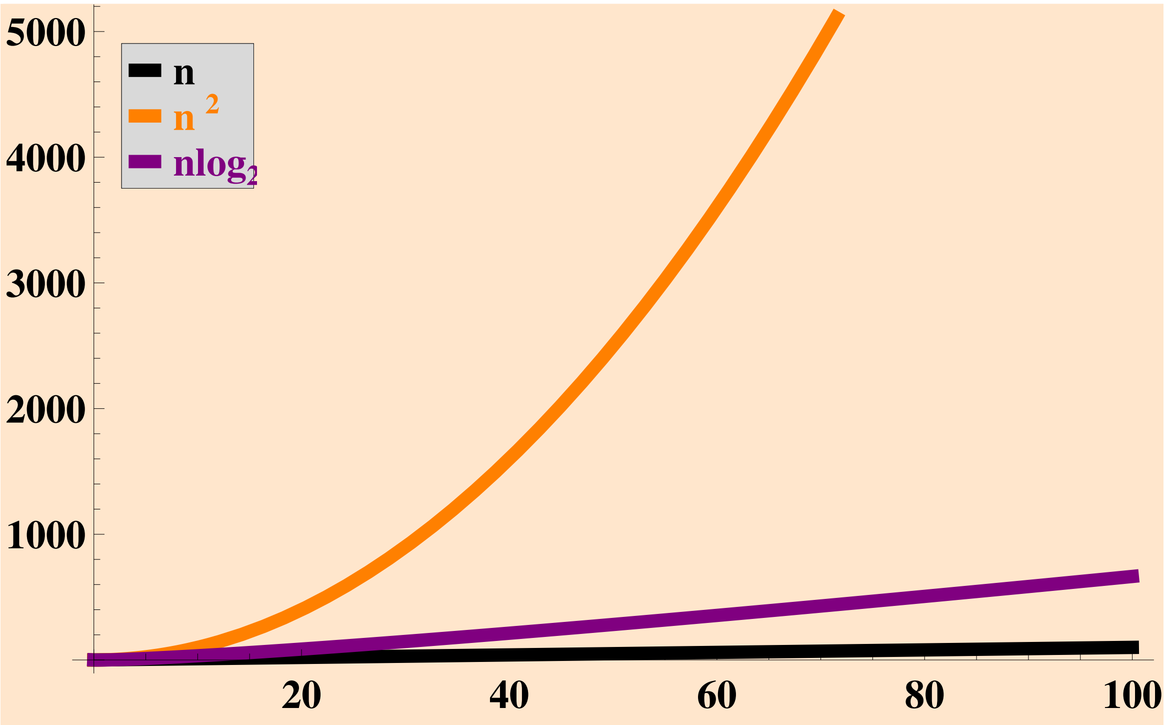
One level deeper, we have four recursive calls. Each of them merges sub-arrays of total length of approximately $^n/_4$, so the time it takes is proportional to $4 \times {}^n/_4$ or n.

Do you see the pattern?

Now, the question is how many times will this process repeat? How many times can we divide n elements in half, and then divide each of those halves in half, etc, before we end up with one element ”halves”? The answer to that is $\log_2 n$.

At each level of recursion we perform $\mathrm{O}(n)$ operations. There are $\log_2 n$ levels of recursion. The entire merge sort takes $\mathrm{O}(n \log_2 n)$ steps. This is the best, average and worst case time performance of merge sort.

So is this any better than $\mathrm{O}\left(n^2\right)$ performance of the quadratic sorts? It is much better. And it gets better and better as the n increases. The table below shows values of n, n^2 and $n \log n$ for values of n from 0 to 100 to give you some sense how the shapes of these curves compare.



4.3 Quick Sort

Quick sort is another algorithm that uses the divide-and-conquer approach. But the way in which the array is divided is different. The dividing step of quick sort ends up performing more operations than the one of merge sort, but it allows us to skip the merging step at the end.

The idea behind the quick sort is the following. Pick a value, called **pivot**, from the array and **partition** the array by moving all the values smaller than the pivot to the left and all the values larger than the pivot to the right. Then place the pivot between the two parts. In the ideal situation, we pick the pivot to be as close to the median of all the values in the array as possible. Once we have the two parts, we use quick sort again to sort the two parts. Notice that after the partitioning step, the pivot is in its final location.

The **algorithm for quick sort** is as follows:

1. pick a pivot
2. partition the array into two parts:
   - left - containing all values less than the pivot
   - right - containing all values grater than the pivot
3. apply quick sort to the left part
4. apply quick sort to the right part

4.3.1 Picking a Pivot

Selecting a good pivot is necessary for guaranteeing a good performance. Imagine that we always pick the smallest or the largest element to be the pivot. In this scenario we end up with one of the parts containing all but one element from the original array and the other part empty. You may guess that this is not very desirable.

Picking the pivot also has to be quick.

There are several ways pivots are picked. All of them have their advantages and disadvanteges. Here are just a few ideas of how we can select a pivot.

- always pick the first element from the given array (or sub-array)
- pick a middle element in the array
- pick a random element from the array
- select three elements from the array and use their median as the pivot; the three elements could be
    - first three elements in the array
    - first, middle and last elements in the array
    - random three elements in the array

4.3.2 Partitioning the Array

Once we decide on the pivot, we need to partition the array into two parts. The goal is to do it using as few operations as possible. We do not care about the order of elements in each part, just so that the left part has elements smaller than the pivot, and the right part has elements larger than the pivot.

The following pseudocode is one way of performing partitioning. We want to partition the `array` given the beginning index `left` and the end index `right`, and the pivot.

```
//move the pivot to the rightmost position
swap array[right] and array[pivot]
set pivot to right
set right to right - 1
while left <= right
  while array[left] < array[pivot]
    left ++
  while right >= left AND array[right] >= array[pivot]
    right --
  if right > left
    swap array[left] with array[right]
//move the pivot to its final location
swap array[left] with array[pivot]
return left
```

4.3.3 Recursion in Quick Sort

The recursive calls to quick sort continue as long as we have parts with more than one element. The complete pseudocode for the quick sort follows.

```
quickSort ( array )
  quickSortRec( array, 0, array.size - 1 )

quickSortRec (array, left, right)
  pivotIndex = findpivot(array, left, right)  //use any method
  newPivotIndex = partition ( array, left, right, pivotIndex )
  if ( newPivotIndex - left > 1 )
    quickSortRec( array, left, newPivotIndex - 1 )
  if ( right - newPivotIndex > 1 )
    quickSortRec( array, newPivotIndex + 1, right )
```

4.3.4 Quick Sort Visualization

OpenDSA website has a very good visualization of step by step application of the quick sort algorithm to a numerical array. You can find it at `http://algoviz.org/OpenDSA/Books/Everything/html/Quicksort.html`.

4.3.5 Quick Sort Performance

If we could always pick a pivot so that the two parts are approximately equal, then the quick sort would perform $\mathrm{O}\left(n \log_2 n\right)$ operations (best and average case). But in extreme cases when we always pick a bad pivot, the quick sort may deteriorate to $\mathrm{O}\left(n^2\right)$ operations (worst case).