CS310 – Advanced Algorithms and Data Structures

Class 10 Sorting: Review and JDK Support

Sorting (Chapter 2)

- One of the most fundamental problems in CS.
- Problem definition: Given a sequence of elements with a well-defined order, return a sequence of the elements sorted according to this order.
- Simple (insertion) Sort – runs in quadratic time ("bad sort")
- Shellsort – runs in sub-quadratic time ("pretty good sort")
- Mergesort – runs in \(O(N\log N)\) time ("good sort")
- Quicksort - runs in average \(O(N\log N)\) time ("good sort")

Insertion sort, pg. 251

For each \(i\) from 1 to \(n-1\), exchange \(a[i]\) with entries that are larger than \(a[i]\) in \(a[0]\) through \(a[j-1]\)

- First pass: exchange \(a[1]\) with \(a[0]\) through \(a[0]\)
- Second: exchange \(a[2]\) with \(a[0]\) through \(a[1]\)
- …
- \(n-1\): exchange \(a[n-1]\) (last elt) with \(a[0]\) through \(a[n-2]\)

```
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exchange(a, j, j-1);
    }
    private static void exchange(Object[] a, int i, int j) {
        Object swap = a[i]; a[i] = a[j]; a[j] = swap;
    }
```

Clearly \(O(n^2)\)

Shellsort

- Named for its inventor, Donald Shell, 1959
- Elements separated by a distance of gap in the array are sorted. When gap is 1, the sort is the same as insertion sort.
- How to choose the gaps? One way (Shell’s way):
  - Start gap at \(N/2\)
  - Halving it until reaches 1
- Better than bad sorts, but not as good as good sorts for large arrays

Mergesort – quick reminder

- 3 steps
  - Return if the number of items to sort is 0 or 1
  - Recursively Mergesort the first and second halves separately
  - Merge the two sorted halves into a sorted group
- This approach is called "divide and conquer".
- Mergesort is an \(O(N^2\log N)\) algorithm, as we showed in class 2 as an example of well-behaved double recursion. This is a good sort.

```
public static void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        for (int j = i; j > 0 && less(a[j], a[j-1]); j--)
            exchange(a, j, j-1);
    }
    private static void exchange(Object[] a, int i, int j) {
        Object swap = a[i]; a[i] = a[j]; a[j] = swap;
    }
```

MergeSort performance

Performance:
\[
T(N) = 2T(N/2) + O(N) = 2(2T(N/4) + O(N/2)) + O(N) = 4T(N/4) + O(N) + O(N) = 8T(N/8) + O(N)+O(N) + O(N) = .... = 2^\log N T(1) + O(N) + O(N) + ... + O(N) = N^2 O(1) + O(N) + O(N) + ... + O(N).
\]

The terms are expanded \(\log N\) times, each produces an \(O(N)\).

\(\log N\) terms of \(O(N)\) = \(O(N\log N)\)
Quicksort – another divide-and-conquer sort algorithm

- 4 steps to sort S:
  - Return if the number of elements in S is 0 or 1
  - Pick a “pivot” element v in S
  - Partition S-{v} into 2 disjoint groups:
    - L = {x in S-{v} | x ≤ v}
    - R = {x in S-{v} | x ≥ v}
  - Return the result of Quicksort(L) followed by v followed by Quicksort(R)

Quicksort analysis

- $T(N) = O(N) + T(|L|) + T(|R|)$
  - Partitioning, linear in N
  - Size of L
  - Size of R
- Similar to mergesort analysis, so should be $O(N\log N)$... or is it?
  - The result depends on the size of L and R. If roughly the same – yes. Otherwise – if one partition is $O(1)$ and the other $O(N)$, may be quadratic!

Picking the Pivot

- A wrong way
  - Pick the first element or the larger of the first two elements
  - If the input has been presorted or is reverse order, this is a poor choice
- A safe choice
  - Pick the middle element
- S&W way
  - Randomize the array and then just use the first element
  - Nothing guarantees asymptotic $O(N\log N)$, but it can be shown that mostly this is the case, so considered a good sort

Sorting implementations: JDK methods in class Arrays

- First form: use “natural order” of elements (using compareTo of element)
  - This method can be used with generic-typed elements too, since any object ISA Object.
  - Call this by Arrays.sort(a), where a is an array of elements with compareTo.
- Second form, with Comparator argument: we can supply a Comparator, similar to what we did earlier with PriorityQueue
  - But what is the funny syntax about?

FYI: Point2Dim Comparator

```java
// For example, sort by distance from origin using this Comparator
class Point2DimComparator implements Comparator<Point2Dim> { 
  @Override
  public int compare(Point2Dim lhs, Point2Dim rhs) { 
    double dist1 = lhs.x()**lhs.x() + lhs.y()**lhs.y();
    double dist2 = rhs.x()**rhs.x() + rhs.y()**rhs.y();
    double diff = dist1 - dist2;
    if (diff > 0) return 1;
    else if (diff < 0) return -1;
    else return 0;
  }
} 
```
Here the code uses just Point2Dim interface methods to decide which is larger. This is an example of coding a Comparator class ourselves.
Arrays.sort Example

- Recall Student, with id as identifier field, and gpa as another field
- Suppose we have an array of Students students:
  
  ```
  Student students = [new Student(100, "Joe", 3.2),
                     new Student(130, "Sue", 3.9), …];
  ```

  Arrays.sort(students) sorts by id
  
  Arrays.sort(students, Comparator.comparing(Student::getGpa))
  
  Sorts by GPA using a Java 8 easy comparator

Arrays.sort Examples

- Arrays.sort can sort primitive types too
  ```
  int[] arr = {13, 7, 6, 45, 21, 9, 2, 100};
  Arrays.sort(arr);
  ```

  // Sort subarray from index 1 to 4, i.e.,
  // only sort subarray {7, 6, 45, 21} and
  // keep other elements as it is.
  Arrays.sort(arr, 1, 5);

Sort a Collection: use JDK class Collections

Collections.sort can sort Collection objects
  ```
  ArrayList<Student> arraylist = new ArrayList<Student>();
  arraylist.add(new Student(223, "Chaitanya", 2.6));
  arraylist.add(new Student(245, "Rahul", 2.4));
  arraylist.add(new Student(209, "Ajeet", 3.2));
  Collections.sort(arraylist);
  Collections.sort(arraylist, Comparator.comparing(Student::getGpa));
  ```

FYI: JDK Sorting: how is it done?

- Sort for int[] Javadoc says: Implementation note: The sorting
  algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy,
  Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n))
  performance on many data sets that cause other quicksorts to
degradate to quadratic performance, and is typically faster than
traditional (one-pivot) Quicksort implementations.

- Sort for T[] Javadoc says: Implementation note: This
implementation is a stable, adaptive, iterative
mergesort that
requires far fewer than n log(n) comparisons when the input array
is partially sorted, while offering the performance of a traditional
mergesort when the input array is randomly ordered. If the input
array is nearly sorted, the implementation requires approximately
n comparisons.

Best Performance of sorting

- How well can we really do?
- Is there a sorting method whose worst case runtime is O(n)?
  — obviously we can’t do better than that (why?).
- For the class of algorithms we’ve seen so far the
  answer is no. The lower bound really is n log(n).
- These sorting algorithms are based on
  comparisons and can be modeled as binary
decision trees.

A simple example

An array of 3 elements: {a1, a2, a3}
Sort Perf Theorem

- In a sorting algorithm modeled by a binary decision tree, the worst case running time is $n \log n$.
- Proof: The runtime is bound from below by the depth of the decision tree. The number of permutations is $n!$.
- Math result: $n! \sim \exp(n \log n)$ (roughly, and good as an upper bound, note it’s super-exponential).
- The depth of a binary tree with $L$ leaves is $\log(L)$.
- Therefore the depth of the decision tree is: $\log(n!) = \log(\exp(n \log n)) = n \log n$.

Still, can we ever do better?

- When our sorting is not based on comparisons, we can do better.
- Simple case: We have an array of integers 1..n in some random order, where n is not huge.
- How do we sort this?

Still, can we ever do better?

- Simple case: We have an array of integers 1..n in some random order, where n is not huge.
- How do we sort this?
- Answer: just fill it with 1..n in order.
- OK, stupid question.

Still, can we ever do better?

- Simple case, fixed: We have an array of objects with integer ids 1..n (i.e. object.getId() for each) in some random order, where n is not huge.
- How do we sort this?
- Create a second array 0..n (n+1 spots), put each object from first in its id-position in the second. $T(n) = O(n) \ll O(n \log n)$.

Binary search

- Definition: Search for an element in a sorted array. Implemented, S&W pg. 47 by iteration. Code for recursive implementation is in class 2.
- Idea from the book – start in the middle of the array. If the element is smaller than that, search in the smaller half. Otherwise search in the larger half.
- Class 2 analysis shows $T(n) = \log(n)$: really fast!
- Implemented in JDK as part of the Collections API.
- Return array index where element is found or a negative value if not found.
Binary search in the JDK Arrays class

\[
\text{static int binarySearch(Object[]} a, Object key) \\
\text{static } <T> \text{ int binarySearch(T[] a, T key, Comparator} <T> super T> c)
\]

- The version without the Comparator uses "natural order" of the array elements, i.e., calls compareTo of the element type to compare elements. The array needs to be sorted this way.
- As with sort, a Comparator can be supplied instead.
- Collections also has binarySearch over a Collection, but it may have lower performance (on a LinkedList, for example).

sort and binarySearch Example

- Recall Student, with id as identifier field, and gpa as another field:
  
  Student students = [new Student(100, "Joe", 3.2), new Student(130, "Sue", 3.9), ...];
  Arrays.sort(students); // sorts by id
  int index = Arrays.binarySearch(students, new Student(130, "", 0)); // yields index = 1
  index = Arrays.binarySearch(students, new Student(140, "", 0)); // yields index = -1

Binary search App

- S&W page 47 has an implementation of binarySearch, and also a little app in main there:
- Take a list of numbers (in text) tinyW.txt, and another text file of numbers tinyT.txt, and find all the numbers in tinyT.txt that are not in tinyW.txt.
- We could do this with Set<Integer>, but here it is done by sorting an array with the W numbers, and looking up each T number by binary search.
- Is this better or worse performing than the Set approach?

Recursive Binary search

- We need a method that can search any part of an array for the value we're looking for. This is essentially the same as the method we studied in class 2 as a recursion example.

```java
private int binarySearch(int[]} a, int key, int lo, int hi) 
  if (lo > hi) return -1;
  int mid = lo + (hi - lo)/ 2;
  if (key < a[mid])
    return binarySearch(a, key, lo, mid - 1);
  else if (key > a[mid])
    return binarySearch(a, key, mid + 1, hi);
  else return mid;
```

Recursive Binary search

- To binsearch the whole array, use the middle element to decide which half array to binsearch, the same action on a smaller array.
- So we need a method that can search any part of an array...
- This is slightly different than the original binarySearch method header, that takes a whole array.
- This is often the case: to use recursion, we have to cook up a method to do the recursion steps.

Examples:

```
private int binarySearch(int[]} a, int key, int lo, int hi) 
  if (lo > hi) return -1;
  int mid = lo + (hi - lo)/ 2;
  if (key < a[mid])
    return binarySearch(a, key, lo, mid - 1);
  else if (key > a[mid])
    return binarySearch(a, key, mid + 1, hi);
  else return mid;
```
Recursive Binary search

- We now have a method that can search any part of an array for the value we’re looking for.
  ```java
  private static int binarySearch(int[] a, int key, int lo, int hi) { ... }
  ```
- Finish the job... call binarySearch from the top-level method:
  ```java
  public static int indexOf(int[] a, int key) {
    return binarySearch(a, key, 0, a.length-1);
  }
  ```
- We can call this the recursion helper method
- Note that it’s private because it’s not meant to be called directly, only as part of the implementation of the top-level method indexOf.

Binary search: object elements

- The helper
  ```java
  private static <T> int binarySearch(T[] a, T key, int lo, int hi) { ... }
  ```
- Finish the job... call binarySearch from the top-level method:
  ```java
  public static <T> int binarySearch(T[] a, T key) {
    return binarySearch(a, key, 0, a.length-1);
  }
  ```

JDK Binary search: object elements

- In the JDK sort, we don’t see the helper, it’s private
- To use “natural order” (compareTo of elements), use c null here:
  ```java
  public static <T> int binarySearch(T[] a, T key, Comparator<? super T> c)
  ```

Summary on Divide and Conquer

We have seen three examples of the divide-and-conquer algorithm technique:
- Merge sort an array: sort halves by merge sort, merge them
- Quick sort an array: choose pivot, partition into two parts, above and below pivot, quick-sort each part
- Binary Search in an ordered array: Use middle element to determine which half to look in, do binary search in that half