CS310 – Advanced Algorithms and Data Structures

Class 11 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 $n$, exchange $a[i]$ with entries that are larger than $a[i]$ in $a[0]$ through $a[i-1]$

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

```java
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);
        }
    }
}
```

Gap Insertion Sort (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

Shellsort Example

For 5-sort, arrange inputs every 5 apart in columns and sort the columns:

<table>
<thead>
<tr>
<th>Original</th>
<th>81 94 11 96 12 35 75 95 84 95 96</th>
</tr>
</thead>
<tbody>
<tr>
<td>After 5-sort</td>
<td>15 17 11 28 12</td>
</tr>
<tr>
<td>After 5-sort</td>
<td>29 12 11 15 64</td>
</tr>
<tr>
<td>After 1-sort</td>
<td>11 25 17 28 29 35 41 15 75 95 94 95</td>
</tr>
</tbody>
</table>

Performance of Shellsort

From Weiss, page 359

<table>
<thead>
<tr>
<th>$N$</th>
<th>Insertion Sort</th>
<th>Shell's Increments</th>
<th>Odd Gaps Only</th>
<th>Dividing by 2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>575</td>
<td>10</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>20,000</td>
<td>2,489</td>
<td>23</td>
<td>13</td>
<td>20</td>
</tr>
<tr>
<td>40,000</td>
<td>10,635</td>
<td>51</td>
<td>49</td>
<td>41</td>
</tr>
<tr>
<td>80,000</td>
<td>42,818</td>
<td>114</td>
<td>105</td>
<td>86</td>
</tr>
<tr>
<td>160,000</td>
<td>174,333</td>
<td>270</td>
<td>233</td>
<td>194</td>
</tr>
<tr>
<td>320,000</td>
<td>665</td>
<td>330</td>
<td>451</td>
<td></td>
</tr>
<tr>
<td>640,000</td>
<td>NA</td>
<td>1,593</td>
<td>1,161</td>
<td>859</td>
</tr>
</tbody>
</table>
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 $O(N \log N)$ algorithm

Performance:

\[
T(N) = 2T(N/2) + O(N)
\]

\[
= 2(2T(N/4) + O(N/2)) + O(N)
\]

\[
= 2^2T(N/4) + O(N) + O(N)
\]

\[
= 2^3T(N/8) + O(N) + O(N) + O(N)
\]

\[
= \ldots = 2^\log N T(1) + O(N) + O(N) + \ldots + O(N)
\]

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

Mergesort performance

Quick sort – 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:
    - $L = \{x \in S-v | x < v\}$
    - $R = \{x \in S-v | x \geq v\}$
  - Return the result of Quicksort(L) followed by $v$ followed by Quicksort(R)

Quick sort 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.

Sorting implementations: JDK methods in class Arrays

- static void sort(Object[] a) (elements need compareTo)
- static <T> void sort(T[] a, Comparator<? super T> c)

  - First form: use natural order of elements (using compareTo of element) from small to large. This method can be used with generic-typed elements too.
  - Second form, with Comparator argument: example of a static method with a generic type parameter. Need to put parameter at start of method header.
  - Wildcard type: we can provide a Comparator<? X where X is T or a supertype of T (so a X ISA. T).
  - The type flexibility allows us to sort a family of objects of related types.
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

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));

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.

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);

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 degrade 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.

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 \( n! \).
- 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 do better?

- When our model is not based on comparisons we can do better.
- Simple case: We have a set of integers 1..n in some random order, where \( n \) is not huge.
- How do we sort this?

Binary search

- Definition: Search for an element in a sorted array. Implemented, pg. 47.
- Implemented in JDK as part of the Collections API.
- Return array index where element is found or a negative value if not found.
- 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.

Binary search in the JDK Arrays class

```java
static int binarySearch(Object[] a, Object key)
static <T> int binarySearch(T[] a, T key,
                        Comparator<? 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)

Binary search

- \( T(N) = \frac{T(N)}{2} + O(1) \)
- \( T(N) = O(\log N) \)

Binary search tree from wikipedia
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
  // now can use binary search to find individual entries:
  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?

• S&W page 47 has an implementation of binarySearch, done iteratively, with divide and conquer. It’s called indexOf here.
  • Can we do it recursively?
  • Sure: the basic idea is
  • 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...

Recursive Binary search

• We need a method that can search any part of an array for the value we’re looking for...

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 indexOf(T[] a, T key) {
      return binarySearch(a, key, 0, a.length-1);
  }
  ```
- Or to be more explicit in saying that T needs to have compareTo to work, possibly from a superclass method...

Binary search: object elements with explicitly required compareTo

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

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