COMPARATORS + ITERATORS

David Kauchak
CS 62 – Spring 2021

Partition

```java
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end)
    int lessThanIndex = start-1;
    for (int i = start; i < end; i++) {
        if (vals[i].compareTo(vals[end]) <= 0) {
            swap(vals, lessThanIndex, i);
        }
        swap(vals, lessThanIndex, end);
    }
    return lessThanIndex+1;
```
vals is called the pivot
Partitions the elements nums[start...end-1] into two sets, those ≤ pivot and those > pivot
Operates in place
Final result:

```
start  pivot  end
≤ pivot  > pivot
```

**Partition running time?**

$O(n)$

```java
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end) {
    int lessThanIndex = start-1;
    for (int i = start; i <= end; i++) {
        if ( vals[i].compareTo(vals[end]) == 0 ) {
            swap(vals, lessThanIndex, i);
            lessThanIndex++;
        }
        swap(vals, lessThanIndex+1, end);
    }
    return lessThanIndex+1;
}
```

**Quicksort**

```java
void quicksortHelper(E[] vals, int start, int end) {
    if ( start < end ) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```

```java
public static <E extends Comparable<E>> int partition(E[] vals, int start, int end) {
    int lessThanIndex = start-1;
    for (int i = start; i <= end; i++) {
        if ( vals[i].compareTo(vals[end]) == 0 ) {
            swap(vals, lessThanIndex, i);
            lessThanIndex++;
        }
        swap(vals, lessThanIndex+1, end);
    }
    return lessThanIndex+1;
}
```

```java
void quicksortHelper(E[] vals, int start, int end) {
    if ( start < end ) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
```
void quicksortHelper(E[] vals, int start, int end){
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}

void quicksortHelper(E[] vals, int start, int end){
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
void quicksortHelper(int[] vals, int start, int end){
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}

void quicksortHelper(int[] vals, int start, int end){
    if (start < end) {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    }
}
What happens here?

```c
void quicksortHelper(int vals[], int start, int end)
{
    if (start < end)
    {
        int partition = partition(vals, start, end);
        quicksortHelper(vals, start, partition-1);
        quicksortHelper(vals, partition+1, end);
    } }
```
Running time of Quicksort?

Worst case?

Each call to Partition splits the array into an empty array and \( n-1 \) array

\[
\begin{align*}
n-1 + n-2 + n-3 + \ldots + 1 &= O(n^2)
\end{align*}
\]

When does this happen?
- sorted
- reverse sorted
- near sorted/reverse sorted
Quicksort best case?

Each call to Partition splits the array into two equal parts

... How much work is done at each “level”, i.e. running time of a level?

$O(n)$

---

Quicksort best case?

Each call to Partition splits the array into two equal parts

... How many levels are there?

Similar to mergesort, each call to Partition will throw away half the data until we’re down to one element: $\log_2 n$ levels

---

Quicksort best case?

Each call to Partition splits the array into two equal parts

... Overall runtime?

$O(n \log n)$

---

Quicksort Average case?

Two intuitions

- As long as the Partition procedure always splits the array into some constant ratio between the left and the right, say $L$-to-$R$, e.g. 9-to-1, then we maintain $O(n \log n)$

- As long as we only have a constant number of “bad” partitions intermixed with a “good partition” then we maintain $O(n \log n)$
How can we avoid the worst case?

Inject randomness into the data

```c
void randomizedPartition(E[] nums, int start, int end){
    int i = randomInt(start, end);
    swap(nums, i, end);
    return partition = partition(nums, start, end);
}
```

Randomized quicksort is average case $O(n \log n)$

What is the worst case running time of randomized Quicksort?

$O(n^2)$

We could still get very unlucky and pick “bad” partitions at every step

Quicksort properties

**Stable?**

- Stable: possible, but not the way we’ve written it (and requires more storage!)

**In-place?**

- In-place: yes!
Sorting summarized

<table>
<thead>
<tr>
<th></th>
<th>in-place?</th>
<th>stable?</th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>X</td>
<td></td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td>in-situ</td>
</tr>
<tr>
<td>Insertion</td>
<td>X</td>
<td>X</td>
<td>O(n)</td>
<td>O(n^2)</td>
<td>O(n^2)</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>X</td>
<td></td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>use for partially ordered guaranteed</td>
</tr>
<tr>
<td>Quick</td>
<td>X</td>
<td></td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>fastest in practice</td>
</tr>
</tbody>
</table>

Comparable interface

https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html

Interface Comparable<T>

int compareTo(T other)

-1: this object is less than other (technically, any negative number)
0: this object is equal to other
1: this object is greater than other (technically, any positive number)

Built-in sorting

Arrays: https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html

- static void sort(Object[] a)
- static void sort(int[] a)
- static void sort(int[] a, int fromIndex, int toIndex)

Collections: https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

- static <T extends Comparable<T>> T sort(List<T> list)
- static <T extends List> T sort(List<T> list, Comparator<T> comparator)
- static <T> List<T> sort(List<T> list, Comparator<T> comparator)
Naturally sorting cards

SortableCard:
- implements Comparable<SortableCard>
- Utilizes String.compareTo and Integer.compare
- Foreach loop!

naturalSort()

Comparator: unnatural sorting

Create a different ordering without having to modify the class!

Interface Comparator<T>

int compare(T o1, T o2)
-1: o1 is less than o2 (technically, any negative number)
0: o1 is equal to o2
1: o1 is greater than o2 (technically, any positive number)

Unnatural sorting

Arrays: [link]

Collections: [link]

Unnaturally sorting cards

- Add 20 to aces
- Reverse the suit ordering
- Reverse the number ordering

bridgeOrderingSort
Iterator

https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html

A way to move through all of the data in a collection

Interface Iterator<E>:
- boolean hasNext()
- E next()

Have we seen this before? How can we iterate through the data?

Iterator example

```java
public static void temp(){
    List<String> list = new ArrayList<String>();
    list.add("banana");
    list.add("taste");
    list.add("good");
    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

What would we see printed?

Iterator example

```java
while (dealer.hasNext()) {
    Card c = dealer.next();
    System.out.println(c);
    cards.addNewSortableCard(c.getNumber(), c.getSuit());
}
```

```java
public static void temp(){
    List<String> list = new ArrayList<String>();
    list.add("banana");
    list.add("taste");
    list.add("good");
    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
    bananas
taste
good
```
Iterator example

```java
public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add("banana");
    list.add("peach");
    Iterator<String> iterator = list.iterator();
    System.out.println(iterator.next());
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

What would we see printed?

```
bananas
bananas
taste
taste
```

Each iterator has its own state!

---

Iterable

```java
public static void main(String[] args) {
    List<String> list = new ArrayList<String>();
    list.add("banana");
    list.add("peach");
    Iterator<String> iterator = list.iterator();
    System.out.println(iterator.next());
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
    System.out.println(iterator.next());
}
```

```
bananas
bananas
taste
taste
```

---

**Iterable**

https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

interface Iterable<E> {
    Iterator<E> iterator()
}

Just a single method that returns an Iterator.

---

**Why Iterable??**

```java
for (SortableCard c : cards) {
    System.out.println(c);
}
```

Any class that implements the Iterable class can be used in a foreach loop!
How to make a class Iterable

- Implement Iterable interface
- Make a private inner class that implements the Iterator interface
- Have the iterator method return an instance of the private inner class

An example

https://github.com/pomonacs622021sp/LectureCode/blob/master/Iterable/IterableArrayList.java

Each instance of the inner class will have its own count instance variable

Iterator vs. Iterable

Iterators are a useful mechanism for iterating over almost any type of data

Iterators are the thing that do most of the work (and require most of the coding!)

Iterable allows us to use it in a foreach loop and is often just creating an instance of an Iterator