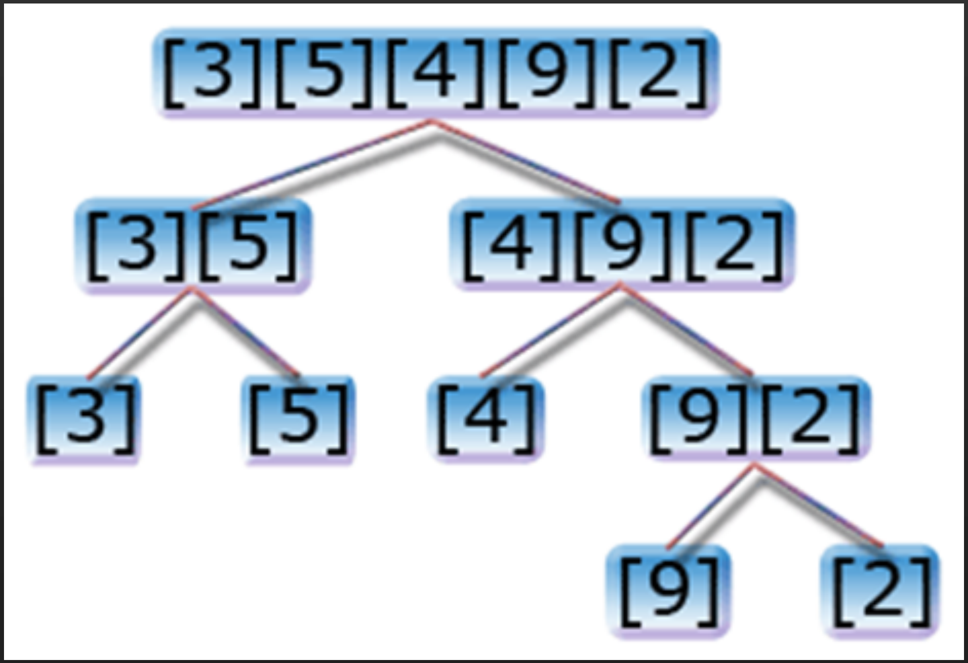
Merge Sort & Fork-Join Parallelism

Edna Jones & Tayler Burns
CSSE 221 Section 2
Fundamentals of Software Development Honors
Rose-Hulman Institute of Technology

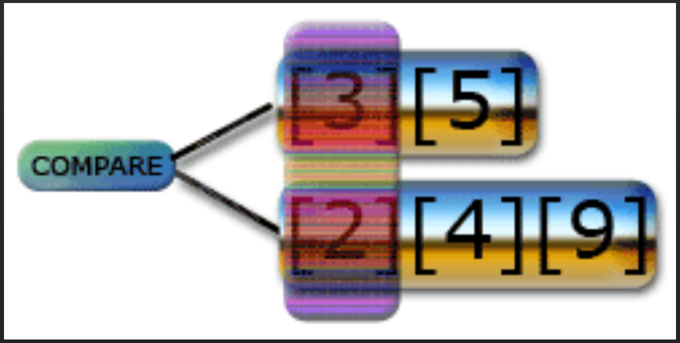
Merge Sort

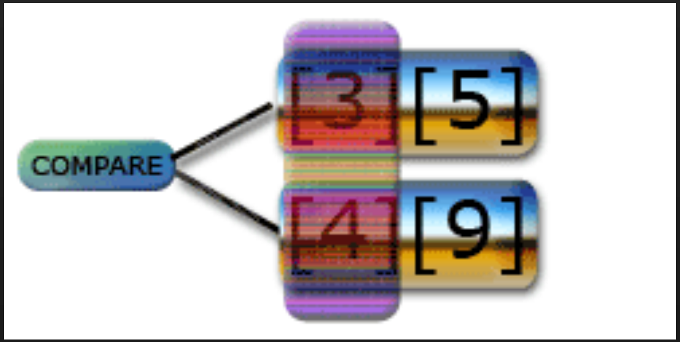
- Sorts usually from smallest to largest
- Why sort a list? Sorted lists can use certain search algorithms.
- Divides array in half recursively
- Call merge sort on each half of the array
- Then merges the sorted halves into a sorted array
- Has O(n log n) efficiency

Dividing the Array



Comparing During Merging





ROSE-HULMAN INSTITUTE OF TECHNOLOGY

Video

- http://www.youtube.com/watch?v=2sLHJpMyNXA&feature=player_detailpage

ROSE-HULMAN INSTITUTE OF TECHNOLOGY

Coding Merge Sort

Classes Used in Fork-Join Parallelism

- **ForkJoinPool**: use exactly one of these to run all your fork-join tasks in the whole program
  - It is the job of the pool to take all the tasks that can be done in parallel and actually use the available processors effectively.
- **RecursiveTask<V>**: you run an object of type a subclass of this in a pool and have it return a result
- **RecursiveAction**: just like RecursiveTask except it does not return a result
- **ForkJoinTask<V>**: superclass of RecursiveTask<V> and RecursiveAction. fork() and join() are methods defined in this class. It is the class with most of the useful javadoc documentation, in case you want to learn about additional methods.

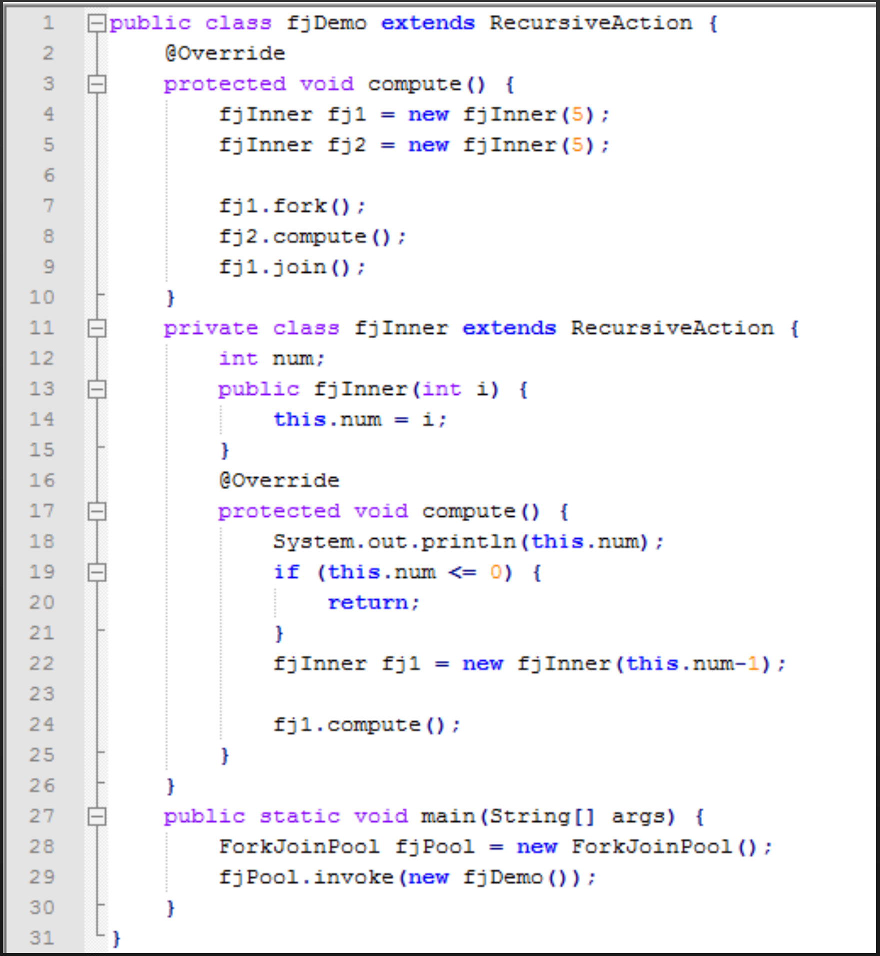
Using Fork-Join Parallelism

- Create a ForkJoinPool
  - ForkJoinPool fjPool = new ForkJoinPool();
- Call the invoke() method on the ForkJoinPool passing an object of type RecursiveTask<V>
  - fjPool.invoke(new RecursiveTask<V>(Object o));
  - This causes the ForkJoinPool to call the compute() method of the RecursiveTask.
- The compute() method

Using Fork-Join Parallelism

- Call fork() method on a RecursiveTask. This starts parallel computation – fork() itself returns quickly, but more computation is now going on.
- When you need the answer, you call the join() method on the object you called fork() on. The join method will get you the answer from compute() that was figured out by fork(). If it is not ready yet, then join will *block* (i.e., not return) until it is ready.

Example



```java
public class fjDemo extends RecursiveAction {
    @Override
    protected void compute() {
        fjInner fj1 = new fjInner(5);
        fjInner fj2 = new fjInner(5);

        fj1.fork();
        fj2.compute();
        fj1.join();
    }
    private class fjInner extends RecursiveAction {
        int num;
        public fjInner(int i) {
            this.num = i;
        }
        @Override
        protected void compute() {
            System.out.println(this.num);
            if (this.num <= 0) {
                return;
            }
            fjInner fj1 = new fjInner(this.num-1);

            fj1.compute();
        }
    }
    public static void main(String[] args) {
        ForkJoinPool fjPool = new ForkJoinPool();
        fjPool.invoke(new fjDemo());
    }
}
```

Things to Watch Out for

- Don’t call fork twice for only two subproblems and then call join twice. This is much less efficient than just calling compute() and has no benefit since you are creating more parallel tasks than is helpful.
- The order in which you call fork(), compute(), and join() matters
  - If you call the methods in the order fork(), join(), compute() or compute(), fork(), join(), the code won’t run in parallel.

Sources

- Horstmann, Cay. *Big Java*.
- "Beginner's Introduction to Java's ForkJoin Framework." <http://www.cs.washington.edu/homes/djg/teachingMaterials/grossmanSPAC_forkJoinFramework.html>
- http://www.mycstutorials.com/articles/sorting/mergesort