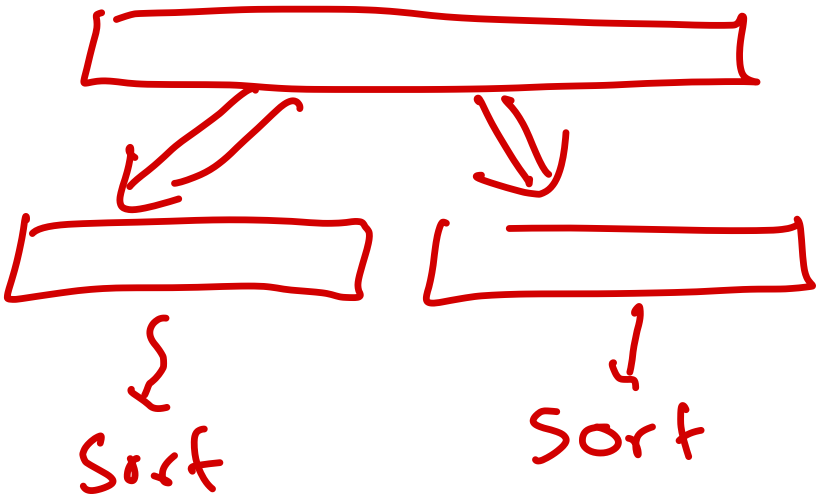
Lecture 29: Fork-Join Pools

COSC 273: Parallel and Distributed Computing

Spring 2023



Last Time

Sorting by Divide-and-Conquer

- To sort an array
  - partition into two (or more) sub-arrays
  - sort the parts
  - combine the sorted parts
- Naturally recursive structure

Today

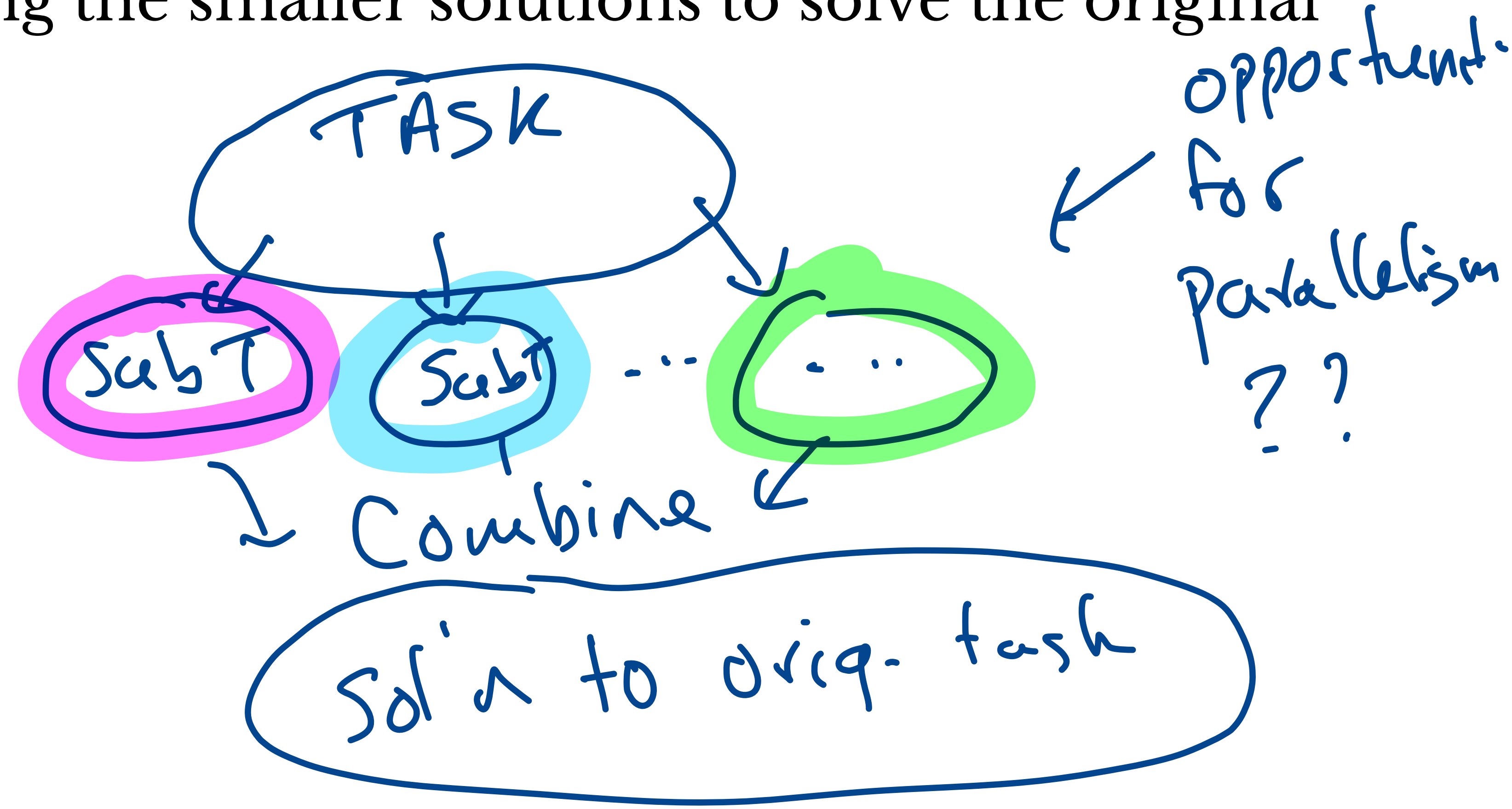
Divide-and-Conquer in Parallel:

- Fork-Join Pools

Divide and Conquer

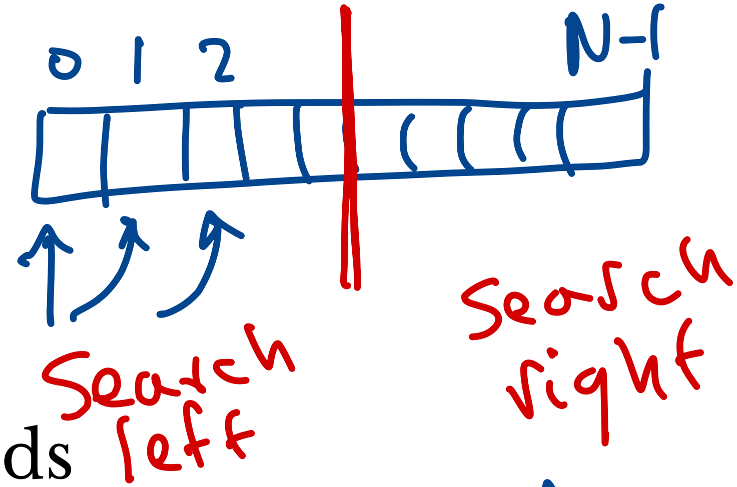
Many computation problems can be solved efficiently by:

1. Breaking an instance into two or more smaller instances
2. Solving the smaller instances (maybe recursively)
3. Combining the smaller solutions to solve the original instance



Example 1: Searching Unsorted Array

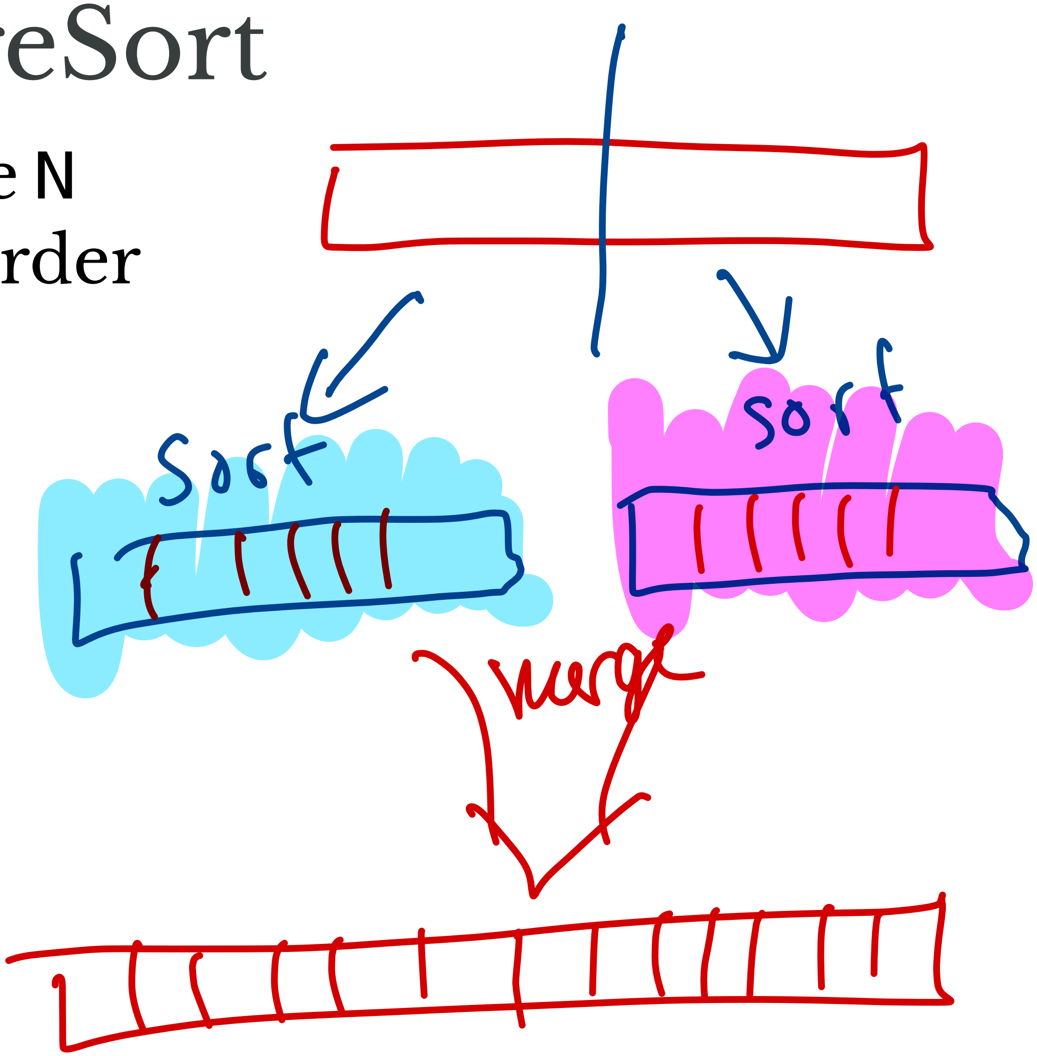
- Given `int[] arr` of size `N`
- Does `arr` contain `1`?
- Idea:
  1. divide `arr` in half
  2. search left half for `1`
  3. serach right half for `1`
  4. return `true` if step 1 or 2 succeeds



success if either thread finds value

Example 2: MergeSort

- Given `int[] arr` of size `N`
- Sort `arr` in increasing order
- Idea:
  1. divide `arr` in half
  2. sort left half
  3. sort right half
  4. merge sorted halves



Observation

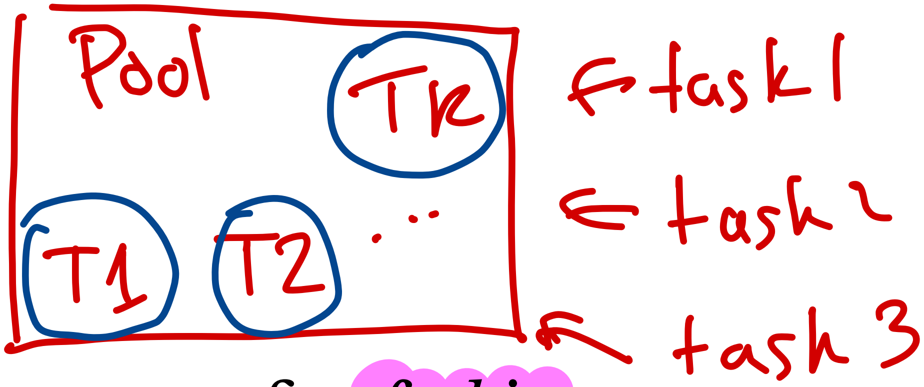
Divide-and-conquer often lends itself well to parallelism:

1. Divide instance into smaller instances
2. Solve smaller instances *in parallel*
3. Combine solutions

harder to employ parallelism?

need sub-tasks to be indep. of one another

Fork-Join Pools

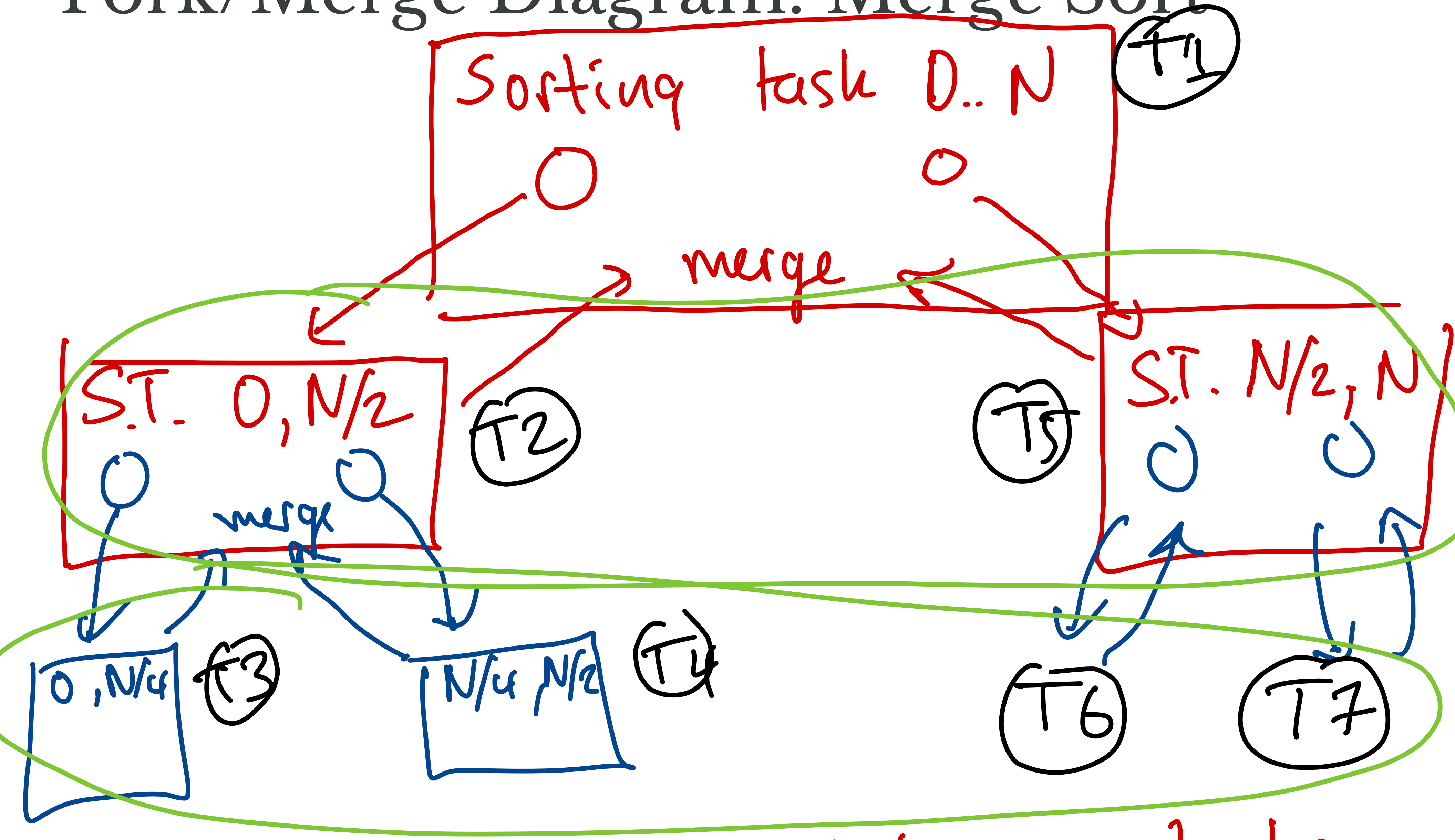


Idea:

- A thread pool with efficient support for *forking*:
  - divide a task into two or more sub-tasks
  - complete sub-tasks
  - combine solutions (if necessary)
- Naturally lends itself to recursion

Fork op: a single task spawns new sub-tasks

Fork/Merge Diagram: Merge Sort

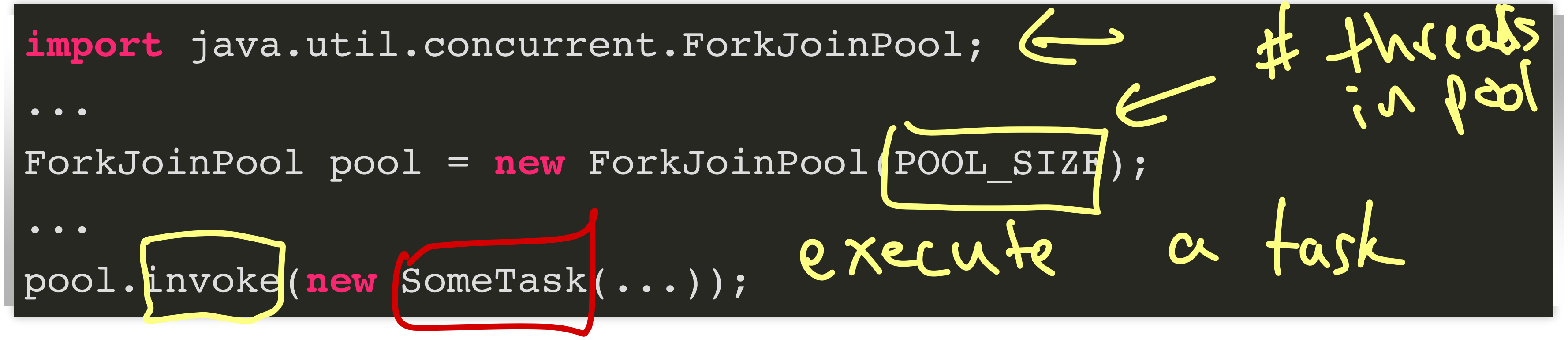


Issue: dependencies between tasks

Creating a Fork-Join Pool

Creating a Fork-Join Pool is easy!

- tasks are **invoked** in FJP



```
import java.util.concurrent.ForkJoinPool;
...
ForkJoinPool pool = new ForkJoinPool(POOL_SIZE);
...
pool.invoke(new SomeTask(...));
```

threads in pool

execute a task

may spawn new tasks to be executed by pool

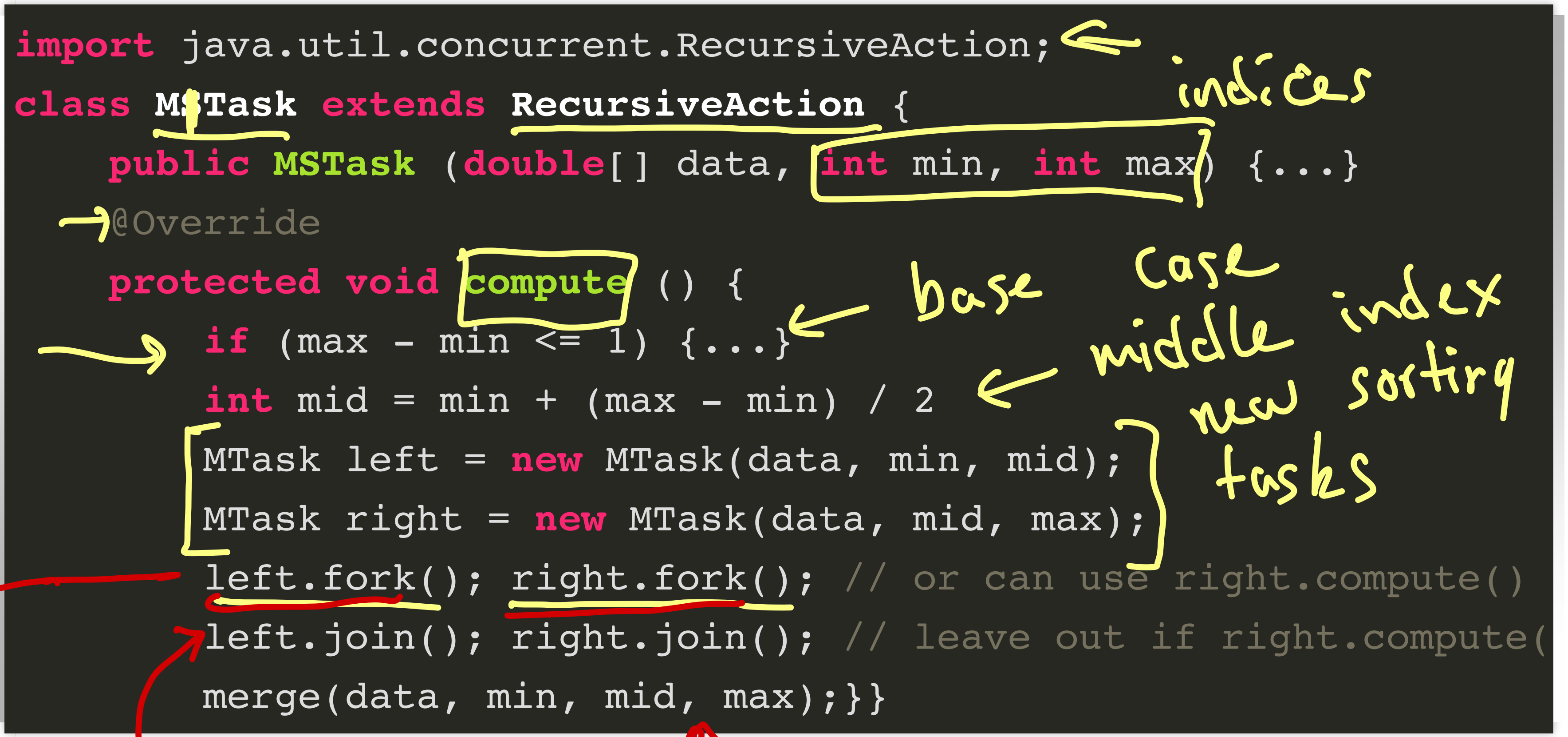
Recursive Actions

Tasks without return values = **recursive action** ← built in

- extend `RecursiveAction` class
- override `compute()` method

defines what task should do

MergeSort as RecursiveAction



```
import java.util.concurrent.RecursiveAction;
class MSTask extends RecursiveAction {
    public MSTask (double[] data, int min, int max) {...}
    @Override
    protected void compute () {
        if (max - min <= 1) {...}
        int mid = min + (max - min) / 2
        MTask left = new MTask(data, min, mid);
        MTask right = new MTask(data, mid, max);
        left.fork(); right.fork(); // or can use right.compute()
        left.join(); right.join(); // leave out if right.compute(
        merge(data, min, mid, max);}}
```

indices

base case

middle index

new sorting tasks

complete sorting by merge!

Invoke with pool.invoke(new MTask(data, 0, data.length))

new task execution

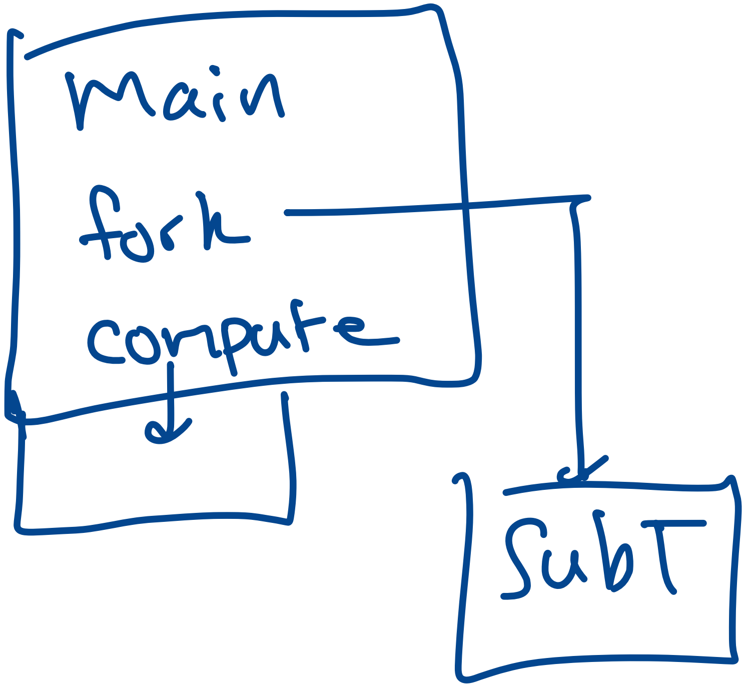
wait for those tasks to complete

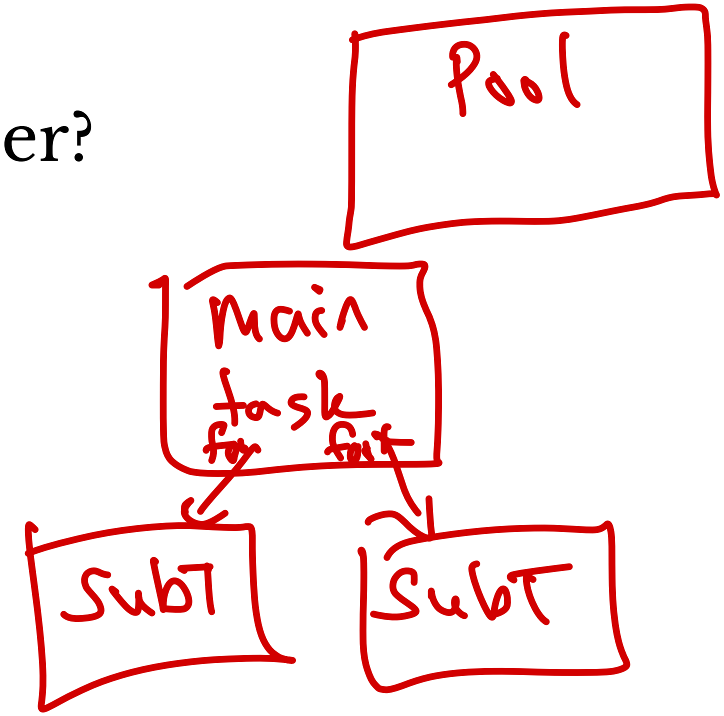
fork versus compute

The difference:

- `fork()` creates new task to be scheduled by the pool
  - must join → less overhead
- `compute()` performs computation as part of this task
  - no join necessary

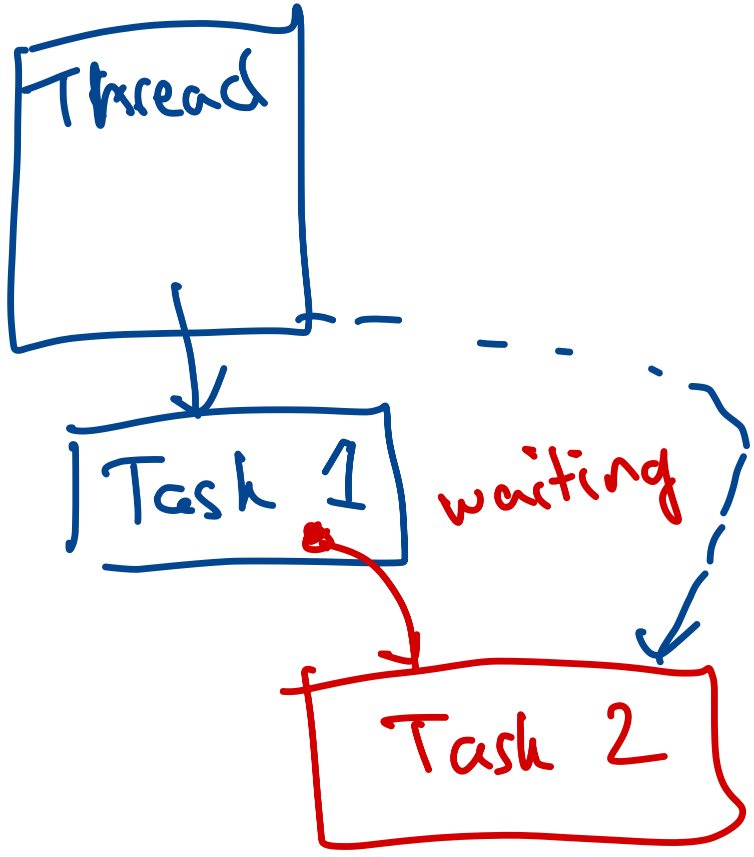
Question. Why use one or the other?





What ForkJoinPool Does

- FJP is a thread pool with a fixed number of threads
- FJP handles scheduling of tasks
- Employs "work-stealing" strategy to minimize time spent waiting for tasks to complete
  - Accounts for dependencies between tasks
  - *AMP* Chapter 16



Efficiency

Often Fork-Join pools are not always as efficient you'd like them to be

To deal with this:

- Use large "base case"
    - don't waste multithreading breaking up small tasks
- Only use on large instances

Still FJPs can lead to elegant solutions, readable code

- Can have better performance if task sizes are irregular

Recursive Task

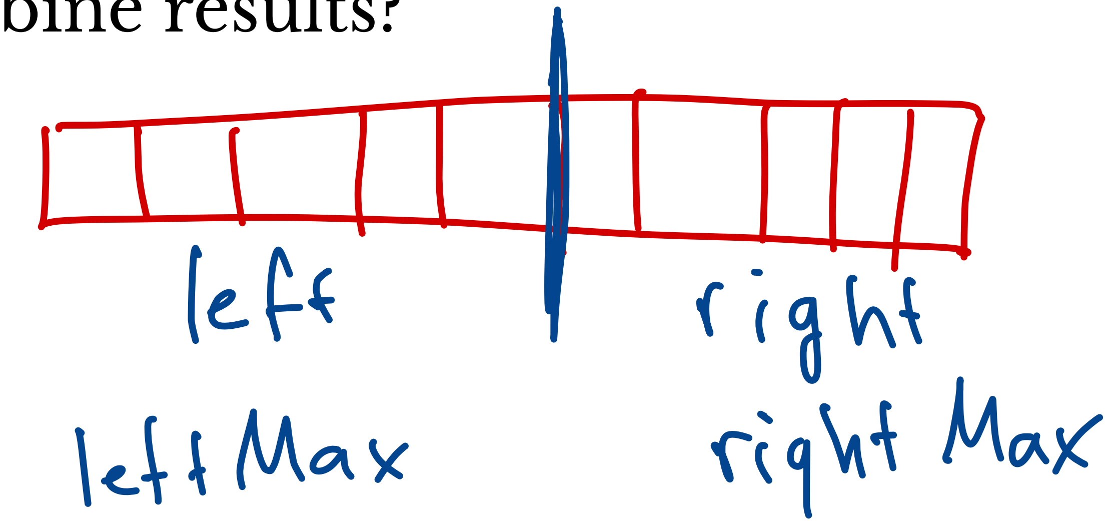
What if we want tasks to return a value?

- Use `RecursiveTask<T>`!
  - task returns a value of type `T`
  - similar to `RecursiveAction` except `compute()` returns a `T`
- `pool.invoke(someRecursiveTask<T>)` now also returns a `T`
- `join()` method also returns a `T`

A Simple Example

Finding the maximum value in an unsorted array

- What is a task?
- How to combine results?



return Max(leftMax, rightMax)

An Activity

Compare the run-times of the two methods!

Download `fork-join-pools.zip`

1. What values of `PARALLEL_LIMIT` give better performance?
2. Is there a performance difference for `fork/compute` compared to `fork/fork`?

Disclaimer:

- everything about Java is optimized to execute code like `findMax` efficiently
- fork-join pools are better suited for more complex tasks…

What Happened?

Next Time

Sorting networks!