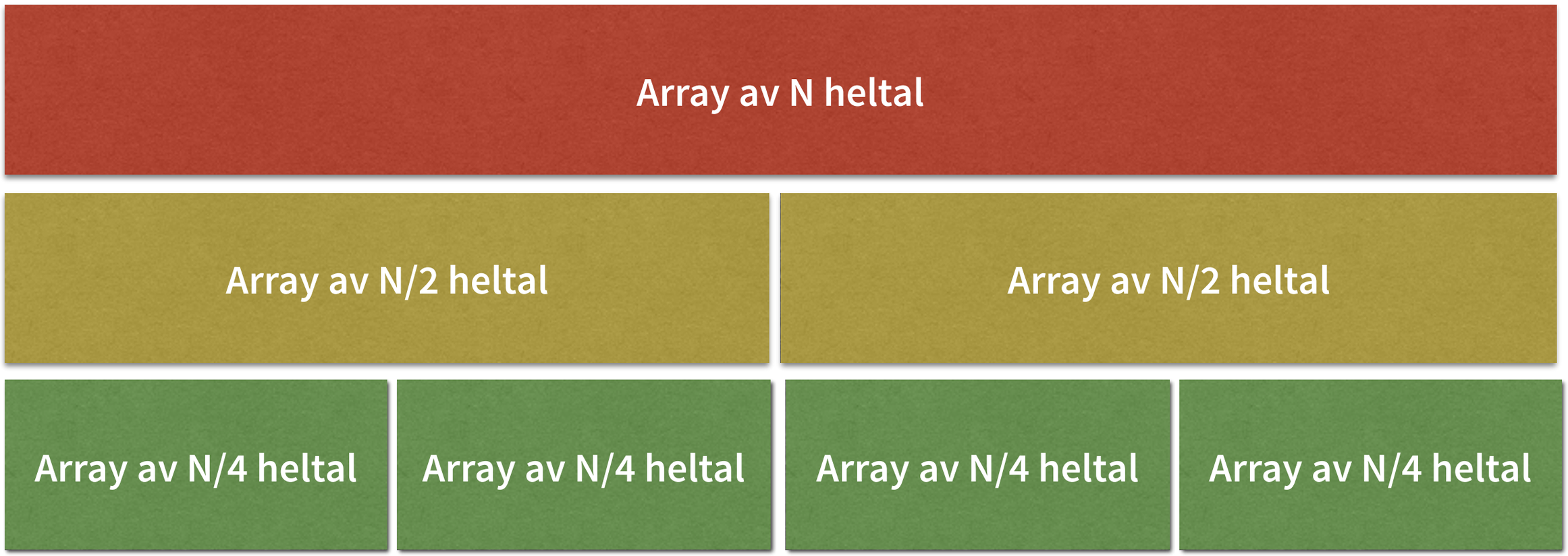
Lektion 4

Stephan Brandauer

Parallellisera ett existerande Javaprogram med fork/join

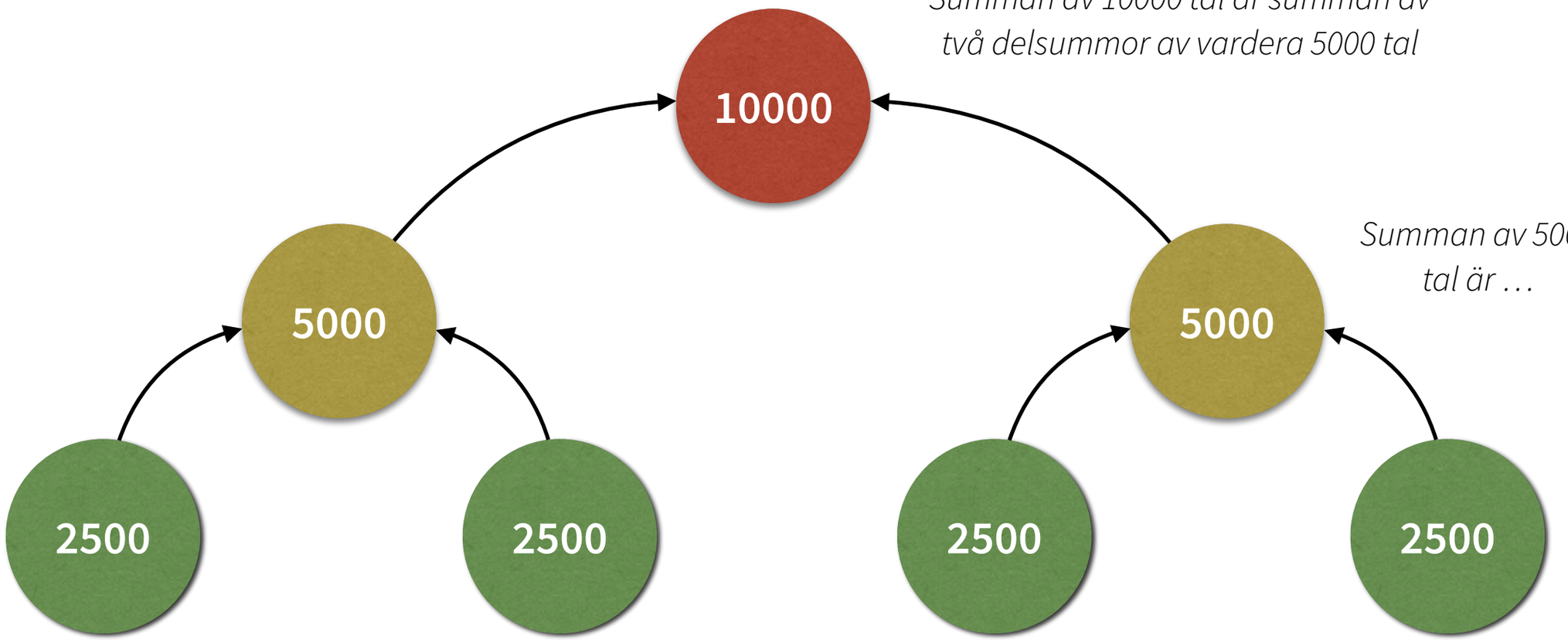
Summera med divide-and-conquer [recap F22]



. . .

Bryt ned tills vi har ett lagom antal tasks! (Vad är lagom?)

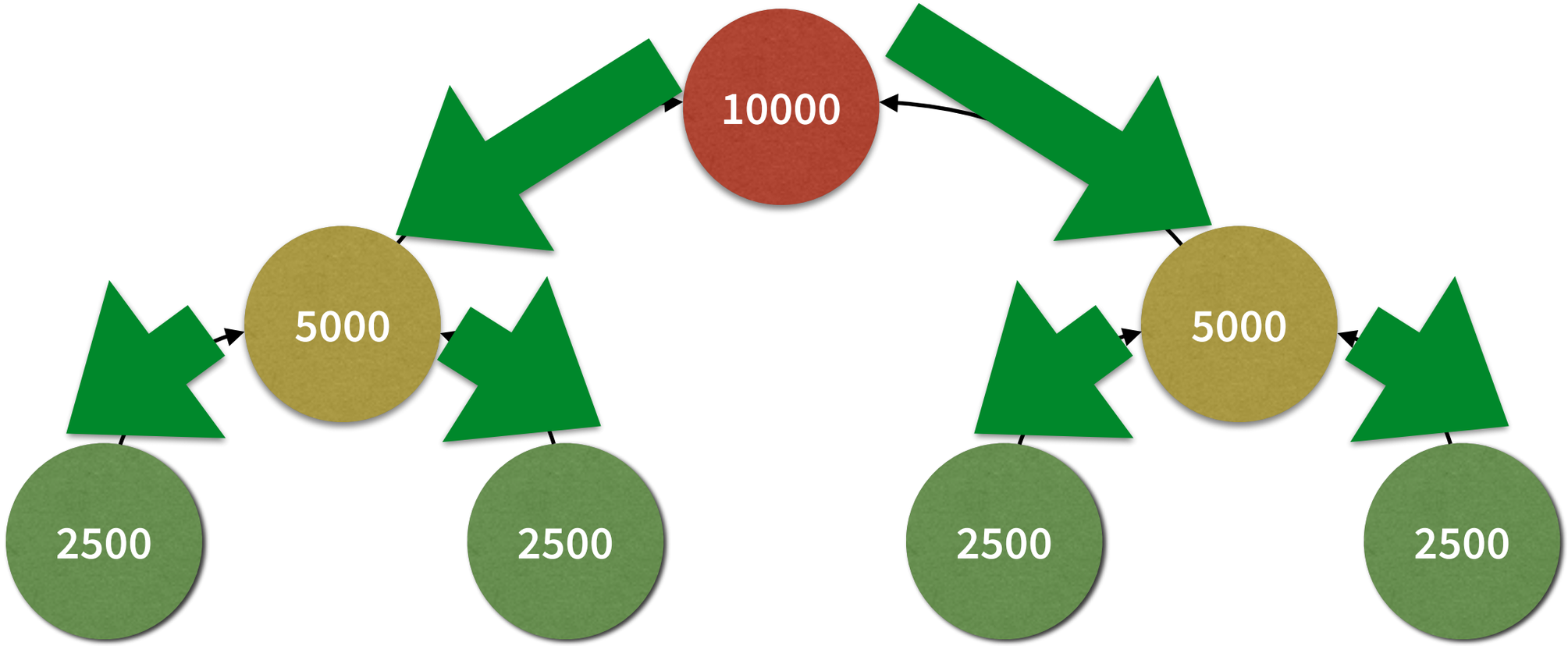
Acyklisk graf av beroenden



. . .

Bryt ned tills vi har ett lagom antal tasks! (Vad är lagom?)

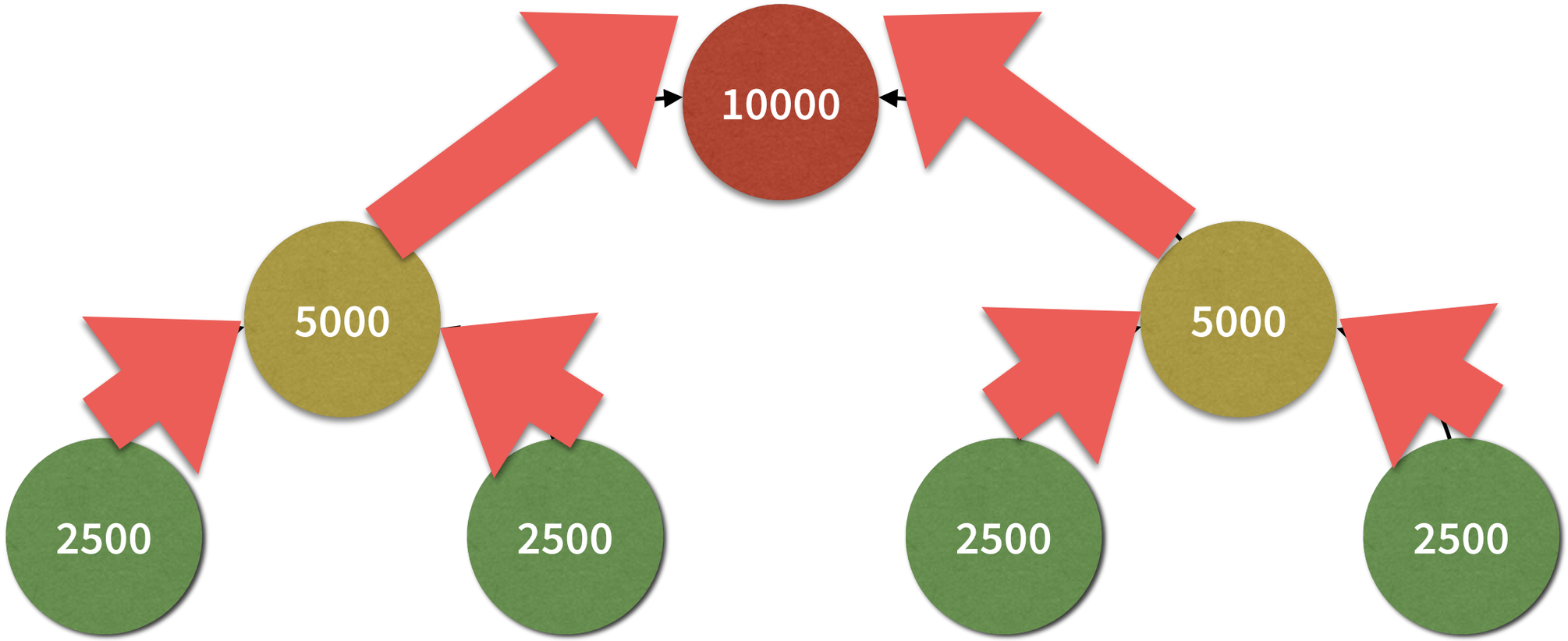
Divide and Conquer — Fork/Join



...

Fork until we have a suitable number of tasks, perform them and join to ”unblock” waiting tasks

Divide and Conquer — Fork/Join



. . .

Fork until we have a suitable number of tasks, perform them and join to ”unblock” waiting tasks

Fork/Join i Java

- Paketet java.util.concurrent

```
import java.util.concurrent.*;
```

- Vi kommer specifikt under denna lektion att använda klasserna

  `java.util.concurrent.ForkJoinPool` — ett stall av trådar för att utföra uppgifter

  `java.util.concurrent.RecursiveAction` — en uppgift som inte returnerar något

RecursiveAction

Constructor Summary

Constructors

Constructor and Description
`RecursiveAction()`

Method Summary

All Methods | **Instance Methods** | **Abstract Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`protected abstract void`	**`compute`**`()` The main computation performed by this task.
`protected boolean`	**`exec`**`()` Implements execution conventions for RecursiveActions.
`Void`	**`getRawResult`**`()` Always returns `null`.
`protected void`	**`setRawResult`**`(`**`Void`** `mustBeNull)` Requires null completion value.

Den viktiga metoden för oss

Från operation till ”task” [sekventiell]

```
public long summarise(int[] values, int start, int end) {
    long sum = 0;
    for (int i = start; i < end; ++i) {
        sum += values[i];
    }
    return sum;
}
```

```
summarise(arr, 0, arr.length);
```

```
public class SeqSumTask {
    SeqSumTask(int[] values, int start, int end) {
        this.values = values;
        this.start = start;
        this.end = end;
    }

    public long summarise() {
        long sum = 0;
        for (int i = this.start; i < this.end; ++i) {
            sum += this.values[i];
        }
        return sum;
    }
}
```

```
t = new SeqSumTask(arr, 0, arr.length);
t.summarise();
```

Från sekventiell till parallell ”task”

```
public class SeqSumTask {
    public SeqSumTask(int[] values,
                      int start,
                      int end) {
        this.values = values;
        this.start = start;
        this.end = end;
    }

    public long summarise() {
        long sum = 0;
        for (int i = this.start;
             i < this.end;
             ++i) {
            sum += this.values[i];
        }
        return sum;
    }
}
```

```
public class ParSumTask extends RecursiveTask<Long> {
    public SeqSumTask(int[] values, int start, int end) {
        this.values = values;
        this.start = start;
        this.end = end;
    }

    public long compute() {
        if (end - start < SEQUENTIAL_THRESHOLD) {
            long sum = 0;
            for (int i = this.start; i < this.end; ++i) {
                sum += this.values[i];
            }
            return sum;
        } else {
            int mid = (end - start) / 2;
            ParSumTask t1 = new ParSumTask(this.values, this.start, mid);
            t1.fork();
            ParSumTask t2 = new ParSumTask(this.values, mid, this.end);
            return t2.compute() + t1.join();
        }
    }
}
```

```
t = new SeqSumTask(arr, 0, arr.length);
p = new ForkJoinPool();
t.invoke(t);
```

RecursiveTask vs. RecursiveAction

- En recursive task har ett returvärde `RecursiveTask<Foo>` betyder att `compute()` skall ha returtypen Foo
- En recursive action är som en `RecursiveTask<Void>`

  Helt enkelt: enbart sidoeffekter, inget resultat