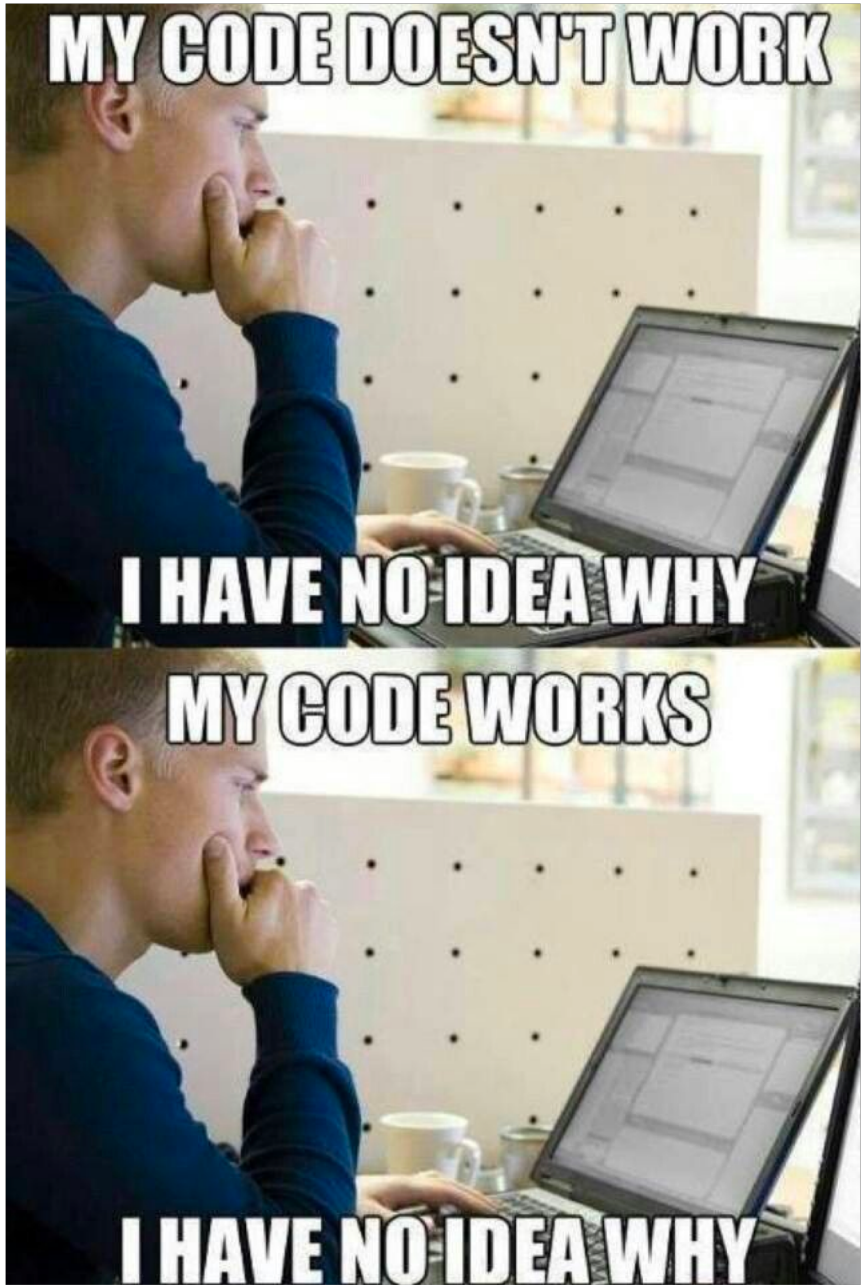
CS2030 AY18/19 SEM 2

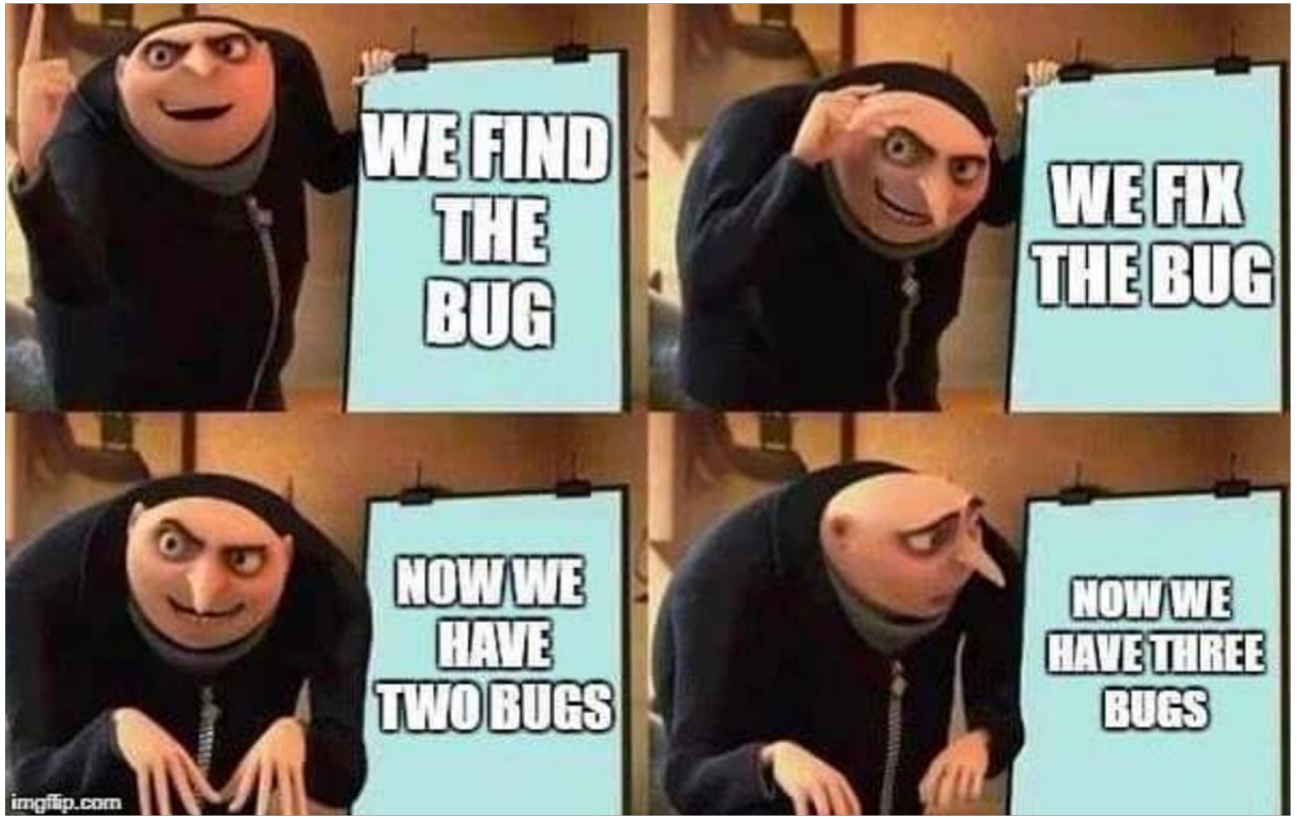
WEEK 12 | 12 APRIL 19

TA GAN CHIN YAO

DISCLAIMER

Slides are made by me, unofficial, optional
Available to download at bit.ly/cs2030_gan
Slides (if any) will be uploaded on
Friday weekly


MY CODE DOESN'T WORK
I HAVE NO IDEA WHY
MY CODE WORKS
I HAVE NO IDEA WHY


WE FIND THE BUG
WE FIX THE BUG
NOW WE HAVE TWO BUGS
NOW WE HAVE THREE BUGS
imgflip.com

Q1.

1. What is the outcome of the following stream pipeline?

```
Stream.of(1, 2, 3, 4)
      .reduce(0, (result, x) -> result * 2 + x);
```

What happens if we parallelize the stream? Explain.

- Running the stream sequentially gives 26 since the pipeline evaluates ((((0 * 2 + 1) * 2 + 2) * 2 + 3) * 2 + 4)

result = 0; x = 1

result = 0 * 2 + 1; x = 2

result = (0 * 2 + 1) * 2 + 2; x = 3

result = ((0 * 2 + 1) * 2 + 2) * 2 + 3; x = 4

Q1.

1. What is the outcome of the following stream pipeline?

```
Stream.of(1, 2, 3, 4)
    .reduce(0, (result, x) -> result * 2 + x);
```

What happens if we parallelize the stream? Explain.

Parallel will give you wrong output

A possible parallel run (with output from each `reduce` operation) gives 18.

```
0 * 2 + 4 = 4 : ForkJoinPool.commonPool-worker-370
0 * 2 + 3 = 3 : main
0 * 2 + 2 = 2 : ForkJoinPool.commonPool-worker-441
0 * 2 + 1 = 1 : ForkJoinPool.commonPool-worker-299
3 * 2 + 4 = 10 : main
1 * 2 + 2 = 4 : ForkJoinPool.commonPool-worker-299
4 * 2 + 10 = 18 : ForkJoinPool.commonPool-worker-299
18
```

Notice that reduction with the identity value 0 is happens for all four stream elements. This is followed by reducing `(1, 2)` to give 4, and reducing `(3, 4)` to give 10. Finally, reducing `(4, 10)` gives 18.

The above stream cannot be parallelized because `2 * result + x` is not associative, i.e. the order of reduction matters.

Note:

- When you parallel() a stream, there is no guarantee which elements in the stream will be executed first.
- All will eventually be executed, the order of execution is not guaranteed
- Not associative because (2 * result) + x is not equal to 2 * (result + x)
  e.g. (2 * 5) + 2 != 2 * (5 + 2)

Q2. i.

2. Consider the following `RecursiveTask` called `BinSearch` that finds an item within a sorted array using binary search.

```
class BinSearch extends RecursiveTask<Boolean> {
    int low;
    int high;
    int toFind;
    int[] array;

    BinSearch(int low, int high, int toFind, int[] array) {
        this.low = low;
        this.high = high;
        this.toFind = toFind;
        this.array = array;
    }

    protected Boolean compute() {
        if (high - low <= 1) {
            return array[low] == toFind;
        } else {
            int middle = (low + high)/2;
            if (array[middle] > toFind) {
                BinSearch left = new BinSearch(low, middle, toFind, array);
                left.fork();
                return left.join();
            } else {
                BinSearch right = new BinSearch(middle, high, toFind, array);
                return right.compute();
            }
        }
    }
}
```

return left.compute();

As an example,

```
int[] array = {1, 2, 3, 4, 6};
new BinSearch(0, array.length, 3, array).compute(); // return true
new BinSearch(0, array.length, 5, array).compute(); // return false
```

Assuming that we have a large number of parallel processors in the system and we never run into stack overflow, comment on how `BinSearch` behaves in the following situations:

What happens if:

i) Replace the statements

left.fork();

return left.join();

with

return left.compute();

Ans:

We could just call left.compute() instead of left.fork() then return left.join(). This reduces the overhead of interacting with the ForkJoinPool and therefore (i) will likely achieve a faster performance.

Note:

- BinSearch should not be parallelized since we always either search the left half or search the right half, but never both at the same time.
- i.e. only call fork() and join() if you have at least one other compute() executing
  e.g. together
  left.fork();
  left.join();
  right.compute();
  In BinSearch, our fork join compute is in an if-else

Q2. ii.

2. Consider the following `RecursiveTask` called `BinSearch` that finds an item within a sorted array using binary search.

```
class BinSearch extends RecursiveTask<Boolean> {
    int low;
    int high;
    int toFind;
    int[] array;

    BinSearch(int low, int high, int toFind, int[] array) {
        this.low = low;
        this.high = high;
        this.toFind = toFind;
        this.array = array;
    }

    protected Boolean compute() {
        if (high - low <= 1) {
            return array[low] == toFind;
        } else {
            int middle = (low + high)/2;
            if (array[middle] > toFind) {
                BinSearch left = new BinSearch(low, middle, toFind, array);
                left.fork();
                return left.join();
            } else {
                BinSearch right = new BinSearch(middle, high, toFind, array);
                return right.compute();
            }
        }
    }
}
```

left.join();
return left.fork();

As an example,

```
int[] array = {1, 2, 3, 4, 6};
new BinSearch(0, array.length, 3, array).compute(); // return true
new BinSearch(0, array.length, 5, array).compute(); // return false
```

Assuming that we have a large number of parallel processors in the system and we never run into stack overflow, comment on how `BinSearch` behaves in the following situations:

What happens if:
ii) Swap the order of fork() and join(), i.e. replace

```
left.fork();
return left.join();
```

with

```
left.join();
return left.fork();
```

Ans:
(ii) is obviously wrong since calling left.join() before left.fork() would cause the task to block.
Anyway, this is a compilation error. fork() returns a RecursiveTask, but compute() returns Boolean.

Note:

- Never call join() without first calling fork(). Otherwise, your programme will block, and will hang forever.

Q2. ii.

2. Consider the following `RecursiveTask` called `BinSearch` that finds an item within a sorted array using binary search.

```
class BinSearch extends RecursiveTask<Boolean> {
    int low;
    int high;
    int toFind;
    int[] array;

    BinSearch(int low, int high, int toFind, int[] array) {
        this.low = low;
        this.high = high;
        this.toFind = toFind;
        this.array = array;
    }

    protected Boolean compute() {
        if (high - low <= 1) {
            return array[low] == toFind;
        } else {
            int middle = (low + high)/2;
            if (array[middle] > toFind) {
                BinSearch left = new BinSearch(low, middle, toFind, array);
                left.fork();
                return left.join();
            } else {
                BinSearch right = new BinSearch(middle, high, toFind, array);
                return right.compute();
            }
        }
    }
}
```

As an example,

```
int[] array = {1, 2, 3, 4, 6};
new BinSearch(0, array.length, 3, array).compute(); // return true
new BinSearch(0, array.length, 5, array).compute(); // return false
```

Assuming that we have a large number of parallel processors in the system and we never run into stack overflow, comment on how `BinSearch` behaves in the following situations:

What happens if:
iii) Searching for the largest element versus searching for the smallest element in the input array.

Ans:
(iii) requires an algorithmic understanding of the code in compute() method. If the item that we are looking for is smaller than the middle element, we search on the left. This means that the array is sorted in increasing order. So, searching for the smallest element would lead to the code to keep going left (i.e. keep forking and joining). On the other hand, searching for the largest element would lead to the code to keep going to right (i.e. keep invoking compute() which is faster). So, searching for the largest element is faster.

Note:

- The basic idea is that everytime you call fork() and join(), resources are allocated (i.e. time is spent).

Q3. a.

3. Given below is the classic recursive method to obtain the n^{th} term of the Fibonacci sequence *without memoization*

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

```
static int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

(a) Parallelize the above implementation by transforming the above to a recursive task and inherit from java.util.concurrent.RecursiveTask

```
import java.util.concurrent.RecursiveTask;

class Fib extends RecursiveTask<Integer> {

    final int n;

    Fib(int n) {
        this.n = n;
    }

    @Override
    protected Integer compute() {
        if (n <= 1) {
            return n;
        }

        Fib f1 = new Fib(n - 1);
        Fib f2 = new Fib(n - 2);

        // try different variants here...
    }
}
```

Q3. b.

(b) Explore different variants and combinations of fork, join and compute invocations.

```
// variant 1
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
f1.fork();
return f2.compute() + f1.join();
```

Accuracy is correct. But speed can be better.

```
// variant 2
Fib f1 = new Fib(n - 1);
f1.fork();
Fib f2 = new Fib(n - 2);
return f2.compute() + f1.join();
```

Accuracy is correct. Faster than variant1.

```
// variant 3
Fib f1 = new Fib(n - 1);
f1.fork();
Fib f2 = new Fib(n - 2);
return f1.join() + f2.compute();
```

Accuracy is correct. But speed is worse than variant1. In Java, subexpressions are evaluated left to right, i.e. for A + B, A is evaluated first before B (by the way this has nothing to do with associativity). So f1.join() needs to wait for f1.fork() to complete before f2.compute() can be evaluated. Compare this with (a) where f2.compute() proceeds while f1.fork() is running.

```
// variant 4
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
return f1.compute() + f2.compute();
```

Accuracy is correct. This is sequentially recursive, i.e. f1.compute() will finish executing before f2.compute() commence. Not much different from variant3, but still slightly faster as there is no overhead involved in forking and joining. Everything is done by the main thread.

Q3. b.

(b) Explore different variants and combinations of fork, join and compute invocations.

```
// variant 5
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
f1.fork();
f2.fork();
return f2.join() + f1.join();
```

Accuracy is correct. Mthread delegates all other work to worker threads in the common pool. Worse performance than variant1. You are asking 2 other threads to compute(), and not asking the main thread to compute(). Therefore, wasting resources to create 1 additional fork() when you have main thread to compute as well, since main thread is always there in the first place. Main thread is the thread that executes your code in the first place.

```
// variant 6
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
f1.fork();
f2.fork();
return f1.join() + f2.join();
```

Accuracy is correct. Looks the same as variant5, but variant5 still preferred as it follows the convention of joins to be returned innermost first. Since a thread forks tasks to the front of its own double-ended queue, the last task forked should be the one that is joined when the thread becomes idle; tasks at the back of the dequeue are stolen by other idle worker threads. This becomes apparent when we set the parallelism level to 0 using java.util.concurrent.ForkJoinPool.common.parallelism=0 The main thread actually blocks waiting for f2.join() is behind f1.join().

Q3. b.

(b) Explore different variants and combinations of fork, join and compute invocations.

```
// wrong
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
return f1.join() + f2.join();
```

Cannot call join() without fork(). It will hang forever

```
// wrong
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
return f1.fork() + f2.fork();
```

fork() returns a RecursiveTask. It does not return a result.
To get a result, call join() after fork()
Calling fork() without join() makes your fork **useless**

```
// wrong
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
return f1.compute() + f2.fork();
```

Without a join(), fork() alone is useless. fork() tells other worker to do for you, join() gives you back the result that the worker has done.

```
// wrong
Fib f1 = new Fib(n - 1);
Fib f2 = new Fib(n - 2);
f2.fork();
return f1.fork() + f2.join();
```

Same reason as before, calling fork() alone without join() is useless.

Q3.

b.

(b) Explore different variants and combinations of fork, join and compute invocations.

Best variant

```
// fastest variant
Fib f1 = new Fib(n - 1);
f1.fork();
Fib f2 = new Fib(n - 2);
return f2.compute() + f1.join();
```

TIPS for ForkJoin:

- Call fork() as early as possible because fork() does not block. Once you call fork(), other worker can take the task and do the job. If you call fork() later, the time spent to execute other code could be parallelly used by other worker thread to compute
- Call compute() first before you call join(). join() will block and wait for a result before proceeding. It is helpful to see compute() as a fixed time that has to be spent no matter where you call it. So if you call compute() first, while the time is spent on executing compute(), the forked worker thread can continue to calculate parallelly, thereby reducing the amount of block time left when you call join()
- Call join() in reverse order for best performance. i.e. f1.fork(); f2.fork(); f3.fork(); f3.join();f2.join(); f1.join();
- Do not ever call join() without first calling fork()
- Do not call fork and not call join(). Otherwise, it is pointless to call fork()

SUMMARY CONCEPTS

- How to use RecursiveTask<V> class
- What is fork(), join(), compute()
- Difference in order of fork(), join() and compute()

Matrix lab next week

If you are stuck, consider:

```
@Override
public Matrix compute() {
  // stop splitting into subtask if the dimension of the matrices
  // are below or equal to the threshold value.
  if (dimension <= THRESHOLD) {
    return Matrix.multiplyNonRecursively(/* param */);
  } else {
    // Matrices are large enough such that the time saved when computing
    // in parallel outweigh the costs of splitting them into smaller subtasks.
    // Therefore we split the matrices into smaller matrices and compute
    // them in parallel.

    // split the matrix into half its size.
    int size = dimension / 2;

    MatrixMultiplication a11b11 = // ...
    a11b11.fork();

    MatrixMultiplication a12b21 = // ...
    a12b21.fork();

    MatrixMultiplication a11b12 = // ...
    a11b12.fork();

    MatrixMultiplication a12b22 = // ...
    a12b22.fork();

    MatrixMultiplication a21b11 = // ...
    a21b11.fork();

    MatrixMultiplication a22b21 = // ...
    a22b21.fork();

    MatrixMultiplication a21b12 = // ...
    a21b12.fork();

    MatrixMultiplication a22b22 = // ...
    // do not fork last one. Why?

    // Create the 2 subMatrices.
    // Add the 2 subMatrices entries and set them into the final position
    // in the result matrix.

    // join them in reverse, and set the matrix value

    // return the result matrix.
    return result;
}
```


Multithreaded programming
Theory
Actual


MAY THE FORK
BE WITH YOU

All the best for your finals