problem 1. Compute the number of 'prime numbers' between 1 and 200000.

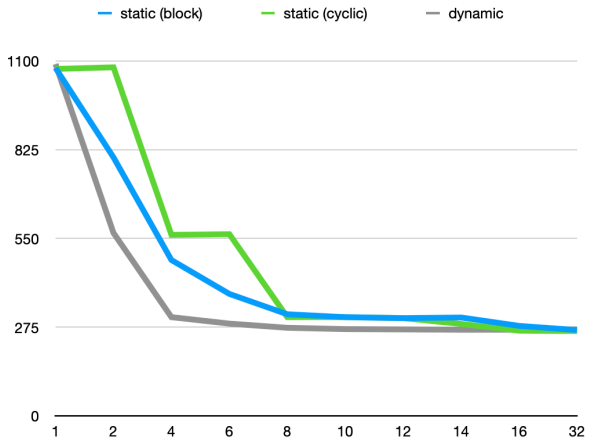
(a) experiment environment

- CPU type : Apple M1
- number of cores : 8 core CPU
- memory size : 16 GB
- OS : macOS Monterey

(b) execution time (ms) per the number of entire threads = {1,2,4,6,8,10,12,14,16,32}.

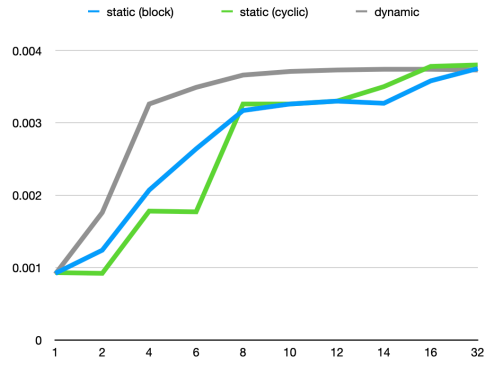
Execution Time

exec time (ms)	1	2	4	6	8	10	12	14	16	32
static (block)	1078	801	483	378	315	306	303	305	279	266
static (cyclic)	1075	1080	561	563	306	306	303	285	264	263
dynamic	1090	567	306	286	273	269	268	267	267	268



performace (1/exec time)

performace (1/exec time)	1	2	4	6	8	10	12	14	16	32
static (block)	0.00092	0.00124	0.00207	0.00264	0.00317	0.00326	0.00330	0.00327	0.00358	0.00375
static (cyclic)	0.00093	0.00092	0.00178	0.00177	0.00326	0.00326	0.00330	0.00350	0.00378	0.00380
dynamic	0.00091	0.00176	0.00326	0.00349	0.00366	0.00371	0.00373	0.00374	0.00374	0.00373



Testing programs were done at 8-cores cpu environment. So even thought you increase the number of threads more than 8, there's no more performance improvement.

But until 8-threads, the more threads you add, the higher performance it shows

Dynamic load balancing shows better performance than the others. The reason why it shows better load balancing is explained below.

(c) Analysis

(1) pc_static_block.java

```java
class PrimeThread extends Thread {
      int to;
      int from;
      int count = 0;
      long timeDiff = 0;

      PrimeThread(int _from, int _to) {
        from = _from;
        to = _to;
      }

      public void run() { // overriding, must have this type
        long s = System.currentTimeMillis();
        for(int i = from; i < to; i++)
            if (isPrime(i))
                count++;
        long e = System.currentTimeMillis();
        timeDiff = e - s;
      }
      long getTimeDiff() {
          return timeDiff;
      }
      int getCount() {
          return count;
      }
      int getFrom() {
          return from;
      }
      int getTo() {
          return to;
      }

      private boolean isPrime(int x) {
            int i;
            if (x<=1) return false;

            for (i=2;i<x;i++)
                if (x%i == 0)
                    return false;
            return true;
        }
}
```

```java
public class pc_static_block {
    private static int NUM_END = 200000; // default input
    private static int NUM_THREADS = 32; // default number of threads

    public static void main (String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }

        PrimeThread[] pt = new PrimeThread[NUM_THREADS];

        int counter=0;

        // find prime numbers from 2 to 200000
        long startTime = System.currentTimeMillis();

        int d = NUM_END/NUM_THREADS;

        for (int i = 0; i < NUM_THREADS; i++) {
            if (i == NUM_THREADS - 1) {
                pt[i] = new PrimeThread(i*d, NUM_END);

            } else {
                pt[i] = new PrimeThread(i*d + 1, (i+1)*d);
            }
            pt[i].start();
        }

        // wait the other threads
        try {
            for (int i = 0 ;i < NUM_THREADS; i++) {
                pt[i].join();
                counter += pt[i].getCount();
            }
        } catch (InterruptedException e) { System.out.println("Error occurred!");}
        long endTime = System.currentTimeMillis();


        // calculate and print the result and the time taken
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time: " + timeDiff +"ms");
        System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter + "\n");

        for (int i = 0 ;i < NUM_THREADS; i++) {
            System.out.println("#" + i + " thread : range " + pt[i].getFrom() + " ... " + pt[i].getTo());
            System.out.println("   execution time : " + pt[i].getTimeDiff() +"ms");
        }
    }

}
```

N number of threads find the number of prime numbers at different range and gather them into one variable at last.

Main thread divides the whole range, 0…200000, into n sub range blocks and allocate them to n threads separately.

```
Program Execution Time: 483ms
1...199999 prime# counter=17984

#0 thread : range 1 ... 50000
   execution time : 86ms
#1 thread : range 50001 ... 100000
   execution time : 234ms
#2 thread : range 100001 ... 150000
   execution time : 357ms
#3 thread : range 150000 ... 200000
   execution time : 483ms
```

If an integer gets bigger, it takes more time to check whether it's prime number or not. So Thread#3(blue) takes more time to count the number of prime numbers at its range than other Threads(red, yellow, green).

Therefore, the loads allocated to each thread are not equal.

4 threads

BLOCK

(2) pc_static_cyclic.java

```java
class PrimeThread extends Thread {
    int r;
    int n;
    int end;
    int count = 0;
    long timeDiff = 0;

    PrimeThread(int _n, int _end, int _r) {
        n = _n;
        r = _r;
        end = _end;
    }

    public void run() { // overriding, must have this type
        long s = System.currentTimeMillis();
        int x = 0;
        while (n * x + r <= end) {
            if (isPrime(n * x + r))
                count++;
            x++;
        }
        long e = System.currentTimeMillis();
        timeDiff = e - s;
    }
    long getTimeDiff() {
        return timeDiff;
    }
    int getR() {
        return r;
    }
    int getCount() {
        return count;
    }

    private boolean isPrime(int x) {
        int i;
        if (x<=1) return false;

        for (i=2;i<x;i++)
            if (x%i == 0)
                return false;
            return true;
        }
    }
```

```java
public class pc_static_cyclic {
    private static int NUM_END = 200000; // default input
    private static int NUM_THREADS = 32; // default number of threads

    public static void main (String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }

        PrimeThread[] pt = new PrimeThread[NUM_THREADS];

        int counter=0;

        // find prime numbers from 2 to 200000
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < NUM_THREADS; i++) {
            pt[i] = new PrimeThread(NUM_THREADS, NUM_END, i);
            pt[i].start();
        }

        // wait the other threads
        try {
            for (int i = 0 ;i < NUM_THREADS; i++) {
                pt[i].join();
                counter += pt[i].getCount();
            }
        } catch (InterruptedException e) { System.out.println("Error occurred!");}
        long endTime = System.currentTimeMillis();


        // calculate and print the result and the time taken
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time: " + timeDiff +"ms");
        System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter + "\n");

        for (int i = 0 ;i < NUM_THREADS; i++) {
            System.out.println("#" + i + " thread : range " + NUM_THREADS + "k + " + pt[i].getR());
            System.out.println("   execution time : " + pt[i].getTimeDiff() +"ms");
        }
    }

}
```

```
Program Execution Time: 308ms
1...199999 prime# counter=17984

#0 thread : range 8k + 0
   execution time : 3ms
#1 thread : range 8k + 1
   execution time : 298ms
#2 thread : range 8k + 2
   execution time : 4ms
#3 thread : range 8k + 3
   execution time : 299ms
#4 thread : range 8k + 4
   execution time : 5ms
#5 thread : range 8k + 5
   execution time : 298ms
#6 thread : range 8k + 6
   execution time : 2ms
#7 thread : range 8k + 7
   execution time : 304ms
```

Let's look thread#0, To define whether 8k (1 <= k < 25000) is prime number or not, dividing 8k with 2 is enough. So even though thread#1 checks the same length of range with other threads but it works only 3ms and rest.

In contrast, thread#7 takes way more times to count because it should try to divide more times per integer than thread#0.

Therefore, the loads allocated to each thread are not equal.

8 threads

(3) pc_dynamic.java

```java
import java.util.concurrent.ArrayBlockingQueue;

class PrimeThread extends Thread {
      ArrayBlockingQueue<Integer> q;
      int count = 0;
      long timeDiff = 0;

      PrimeThread(ArrayBlockingQueue<Integer> queue) {
          q = queue;
      }

      public void run() { // overriding, must have this type
          long s = System.currentTimeMillis();
          while (!q.isEmpty()) {
              try {
                  if (isPrime(q.take()))
                      count++;
              } catch (InterruptedException e) {
                  System.out.println("error occurred while taking element out of queue ");
              }
          }
          long e = System.currentTimeMillis();
          timeDiff = e - s;
      }
      long getTimeDiff() {
          return timeDiff;
      }
      int getCount() {
          return count;
      }

      private boolean isPrime(int x) {
            int i;
            if (x<=1) return false;

            for (i=2;i<x;i++)
                if (x%i == 0)
                    return false;
            return true;
        }
    }
```

```java
public class pc_dynamic {
    private static int NUM_END = 200000; // default input
    private static int NUM_THREADS = 1; // default number of threads

    public static void main (String[] args) {
        if (args.length == 2) {
            NUM_THREADS = Integer.parseInt(args[0]);
            NUM_END = Integer.parseInt(args[1]);
        }

        int[] numbers = new int[NUM_END+1];
        numbers[NUM_END] = NUM_END;
        ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<Integer>(NUM_END);
        for (int i = 0; i<NUM_END; i++) {
            numbers[i] = i;
            queue.add(i+1);
        }
        PrimeThread[] pt = new PrimeThread[NUM_THREADS];

        int counter=0;

        // find prime numbers from 2 to 200000
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < NUM_THREADS; i++) {
            pt[i] = new PrimeThread(queue);
            pt[i].start();
        }

        // wait the other threads
        try {
            for (int i = 0 ;i < NUM_THREADS; i++) {
                pt[i].join();
                counter += pt[i].getCount();
            }
        } catch (InterruptedException e) { System.out.println("Error occurred!");}
        long endTime = System.currentTimeMillis();


        // calculate and print the result and the time taken
        long timeDiff = endTime - startTime;
        System.out.println("Program Execution Time: " + timeDiff +"ms");
        System.out.println("1..." + (NUM_END-1) + " prime# counter=" + counter + "\n");

        for (int i = 0 ;i < NUM_THREADS; i++) {
            System.out.println("#" + i + " thread execution time : " + pt[i].getTimeDiff() +"ms");

        }
    }
}
```

```
Program Execution Time: 302ms
1...199999 prime# counter=17984

#0 thread execution time : 302ms
#1 thread execution time : 302ms
#2 thread execution time : 302ms
#3 thread execution time : 302ms
```

4 threads

```
Program Execution Time: 268ms
1...199999 prime# counter=17984

#0 thread execution time : 268ms
#1 thread execution time : 268ms
#2 thread execution time : 268ms
#3 thread execution time : 268ms
#4 thread execution time : 268ms
#5 thread execution time : 268ms
#6 thread execution time : 267ms
#7 thread execution time : 267ms
```

8 threads

In this case, there's only one queue which all the threads can get jobs from. They share the queue together. So loads are distributed to the threads randomly and as equal amount. Therefore all thread work equally.

(d) How to compile and execute the source code

At terminal,
(1) javac pc_static_bock.java
(2) javac pc_static_cyclic.java
(3) javac pc_dynamic.java

After compilation execute the program.
<java "filename" "numberOfThreads" "numRange">
(1) java pc_static_block 4 200000
(2) java pc_static_cyclic 6 12000
(3) java pc_dynamic 8 200000