1- Num um novo projeto, implemente uma classe **Primos**, com:
- duas variáveis de instância inteiras: start e end
- método para contar e devolver o nº de primos entre start e end
- método go(), para invocar o anterior e imprimir o resultado
- no método main(), deve **criar 4 objetos desta classe** (a cada 50000, desde 0 até 200000), e, sequencialmente, invocar o método go() sobre cada um destes objetos.

Deverá observar o seguinte:

```
$ java Primos
 found 5133
 found 4459
 found 4256
 found 4136
```

ajuda

2- Implemente uma solução para execução em paralelo, com Threads Java, numa nova classe PrimosThread.
A intenção é dedicar **explicitamente** uma thread a cada intervalo de 50000, para já. Basta associar cada thread a um dos objetos que criou no exercício anterior.

Verifique a diferença no tempo e na carga sobre os cores do seu sistema, relativamente à solução anterior.

ajuda

A associação do trabalho a realizar aos objetos que o executam (threads) pode ser simplificada (como fizemos antes, diretamente no método run() de um objeto Thread ou Runable) para pequenas aplicações... mas em problemas de maior dimensão, onde se esperam soluções mais robustas, devemos separar a **descrição do problema** da gestão dos **objetos computacionais** que o executarão.

Estude a descrição dos objetos **Executors**.

Executors

- Thread Pools
- Fork/Join
  - muito conveniente por facilitar a divisão, em tempo de execução, de tarefas em subtarefas mais pequenas, de modo recursivo

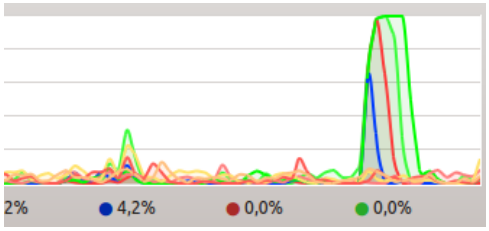
3.a- Reimplemente uma solução (PrimosThreadPool) com Threads, mas simplificando o controlo das Threads e o trabalho que cada uma fará...
Recorra a um serviço Executor, com uma pool de 4 threads.
Distribua trabalho para essa pool, na forma tarefas em objetos Runnable (que podem ser as 4 Threads anteriores), que serão automaticamente processados.

Compare esta solução com a anterior, em termos de **desempenho** e de **complexidade em fase de programação**.

Ver descrição e API, aqui:
- java/util/concurrent/ThreadPoolExecutor.html
- java/util/concurrent/ExecutorService.html

[ajuda ou também esta alternativa]

3.b- Analisando a solução do ponto 2 no gráfico de carga sobre cada unidade de processamento ("core") usada:
imagem



parece existir a desejada concorrência, mas há diferença entre a duração de cada tarefa. Porquê?

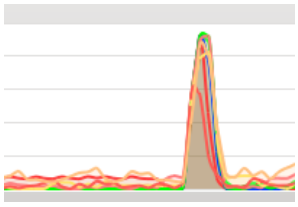
O que poderia fazer para optimizar a distribuição de carga computacional? (uma vez que o core a azul ficou livre muito cedo, podendo talvez ajudar no trabalho da tarefa mais longa)

Escalonamento dinâmico de trabalho
3.c- Tente uma implementação do tipo **Fork/Join**.
A intenção agora é representar o trabalho numa RecursiveAction, deixando que automaticamente se subdivida em tarefas mais pequenas, agregando no final um determinado resultado (o nº de primos entre 0 e 4*50000, como antes).

Terá que redefinir o método compute(), onde decide se executa diretamente, quando a tarefa tem a granularidade certa... ou se divide em duas tarefas, recursivamente...
(ajuda na divisão recursiva do trabalho)

Relativamente à anterior solução, verifique a diferença, em termos de tempo necessário para completar a tarefa geral, e da distribuição de carga entre (todos) os cores do seu PC.
Procura-se algo como:

imagem



Exemplo da ativação:

```
    public static void main(String[] args) {
       PrimosCountAction contaPrimos = new PrimosCountAction(1,200000);

       ForkJoinPool pool = new ForkJoinPool();
       System.out.println( pool.getParallelism() );

       pool.invoke(contaPrimos);

       System.out.println("RESULTADO: " + contaPrimos.getResult());
    }
```

Experimente também diminuir o intervalo de verificação de primos (dimensão do trabalho) atribuído a cada thread na pool, de 10000 para 2000.

Contactar suporte do site