Projekt zaliczeniowy
Michał Wendt
Wyznaczanie liczb pierwszych

Założenia programu

Głównym celem wykonanego przeze mnie programu jest znalezienie wszystkich liczb pierwszych w podanym przez użytkownika przedziale liczbowym. Przez przedział liczbowy rozumiem ograniczenie górne oraz dolne przedziału domkniętego w zbiorze liczb naturalnych.

Do problemu można podejść wielorako. Istnieje kilka sprawdzonych metod różnych jedynie we wzorze na usuwanie kolejnych liczb nie będących pierwszymi. Kolejne podejścia przedstawiają proces wprowadzania przeze mnie ulepszeń do wcześniejszego pomysłu.

Aplikacja może zostać skompilowana oraz uruchomiona jako standardowa aplikacji java dla bilbioteki PCJ (https://pcj.icm.edu.pl/examples/running-pcj-application) lub poprzez użycie dowolnego IDE (np. Intellij).

Interfejs użytkownika istnieje w postaci tekstowej. Na początku jesteśmy proszeni o podanie ilości wątków na których chcemy uruchomić nasz program.

```
System.out.println("Prosze podac ilosc watkow");
System.out.println("Przyklad: 2");

Scanner scan = new Scanner(System.in);
threads = scan.nextInt();
```

Następnie czytelna instrukcja przedstawia możliwe tryby uruchomienia aplikacji.

```
System.out.println("Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>");
System.out.println("Przyklad: 1 100 1000000");  // limit 4 611 686 014 132 420 609 (int size ^ 2)
System.out.println("Prosze wybrac tryb");
System.out.println("1 - Wypisz wszystkie liczby pierwsze z przedzialu");
System.out.println("2 - Wypisz najwieksza z przedzialu");
System.out.println("3 - Policz ilosc liczb piewszych w przedziale");
System.out.println("0 - Zakoncz dzialanie");
```

Po poprawnym wpisaniu wartości uruchamiana jest odpowiednia metoda i wypisywany jest wynik dla dwóch następnych liczb określających przedział. Oczywiście przy dużych liczbach cierpliwość może być wymagana.

Podejście naiwne

Podejście naiwne polega na przejściu pętli iteracyjnej od liczby 2 do najwyższej podanej wartości i sprawdzeniu każdej kolejnej liczby pod kątem pierwszości. Podejście jest sensowne w problemie sprawdzania pierwszości pojedynczej liczby podanej przez użytkownika, jednak traci jakikolwiek sens w sytuacji przeszukiwania większego zbioru.

Oto dwie implementacje naiwnego szukania liczb pierwszych do podanej wartości.

```
static boolean isPrime(long n) {
    for (long i = 2L; i < n; i++) {
        if (n % i == 0L) {
            return false;
        }
    }
    return true;
}
```

Drugie rozwiązanie jest w pewien sposób zoptymalizowane, jednak nie zmienia to faktu zbyt długiego działania dla ogromnych przedziałów.

```
static boolean isPrime2(long n) {
    if (n == 2L) return true;
    if (n % 2 == 0) return false;   // Unikanie l
    for (int i = 3; (long) i * i <= n; i += 2) {
        if (n % i == 0)
            return false;
    }
    return true;
}
```

Podejście zoptymalizowane

Lepszym podejściem jest wstępne obliczenie liczb pierwszych do maksymalnego limitu za pomocą Sita Eratostenesa, a następnie wydrukowanie wszystkich liczb pierwszych w zakresie.

Sito Eratostenesa jest jednym z najskuteczniejszych sposobów znajdowania wszystkich liczb pierwszych mniejszych niż n dla dużych, ale nie ogromny n (do około 10 milionów).

Poniżej można zobaczyć fragment kodu obliczający liczby pierwsze zwykłym sitem.

```
public static void simpleSieve(int sqrtUpperRange, Vector<Integer> smallPrimes) {
    boolean[] isPrime = new boolean[sqrtUpperRange+1];   // Kolekcja małych liczb
    Arrays.fill(isPrime, val: true);  // Na początku uznaję, że każda może być pierw
    isPrime[0]=false;
    isPrime[1]=false;   // 0 i 1 nie są liczbami pierwszymi

    for(int i = 2; i <= sqrtUpperRange; ++i) {   // Oznaczam wszystkie liczby nie
        if(isPrime[i]) {
            for(int j = i * i; j <= sqrtUpperRange && j > 0; j = j + i) {  // Usuw
                isPrime[j] = false;
            }
        }
    }
    for(int i = 2; i <= sqrtUpperRange; ++i) {   //  Sprawdzam oznaczone liczby i
        if(isPrime[i]) {
            smallPrimes.add(i); // Zapisuje jedynie liczby pierwsze oszczędzając p
        }
    }
}
```

Dalsze udoskonalenia

Powyższe podejście działa bardzo dobrze, ale jedynie dla zbiorów zaczynających się na liczbie 2. W przypadku podania dużego ograniczenia dolnego metoda nadal musiałaby obliczyć wszystkie liczby pierwsze od 2 do tego ograniczenia. Od razu można zauważyć dużą stratę pamięci oraz czasu. Poniżej przedstawiono podejście oparte na sicie segmentowym.

```
for (int k = PCJ.myId(); k < smallPrimes.size(); k += PCJ.threadCount()) {
    int i = smallPrimes.get(k);
    int optimizedLow = (lowRange / i);  // Zwiększanie dolnej granicy w cel
    if (optimizedLow <= 1) {
        optimizedLow = i + i;
    } else if (lowRange % i != 0) {
        optimizedLow = (optimizedLow * i) + i;
    } else {
        optimizedLow = (optimizedLow * i);
    }
    // Odkomentowując kolejną linijkę można zobaczyć, który wątek aktualnie
    //System.out.println("Hello from PCJ Thread " + PCJ.myId() + " out of "
    for (int j = optimizedLow; j <= upRange; j = j + i) {
        bigPrimes.set(j - lowRange, false);
    }
}
```

Założeniem sita segmentowego jest podzielenie zakresu [0..n-1] na różne segmenty i obliczenie liczb pierwszych we wszystkich segmentach jeden po drugim. Ten algorytm najpierw używa prostego sita do znalezienia liczb pierwszych mniejszych lub równych √ (n).

Kolejnym etapem było dostosowanie tego rozwiązania dla bardzo dużych liczb tzn. przekraczających rozmiar typu int. Udało się to wykonać dzięki podzieleniu zbioru na podzbiory wielkości 2 mld czyli liczby minimalnie mniejszej niż ograniczający nas rozmiar Integera.

```
for (int k = PCJ.myId(); k < smallPrimes.size(); k += PCJ.threadCount()) {  // Wątki dostają do przetworzenia
    int i = smallPrimes.get(k);
    long optimizedLow = lowRange / i;   // Zwiększanie dolnej granicy w celu zmniejszenia liczby obliczeń
    if (optimizedLow <= 1) {
        optimizedLow = i + i;
    } else if (lowRange % i != 0) {
        optimizedLow = (optimizedLow * i) + i;
    } else {
        optimizedLow = (optimizedLow * i);
    }
    // Odkomentowując kolejną linijkę można zobaczyć, który wątek aktualnie przetwarza daną małą liczbę pierws
    //System.out.println("Hello from PCJ Thread " + PCJ.myId() + " out of " + PCJ.threadCount() + " with prime
        for (long j = optimizedLow; j <= upRange; j = j + i) {
        longBigPrimes[(int) ((j - lowRange) / 2000000000L)].set((int) ((j - lowRange) % 2000000000L), false);
    }
}
```

Cechy równoległości

Równoległość w kwestii tego zagadnienia można implementować na dwa sposoby. Jednym z nim byłoby podzielenie zbioru na mniejsze podzbiory i przydzielenie każdego z nich do innego wątku. Innym równie dobrym podejściem jest przekazanie każdej z wcześniej obliczonych małych liczb pierwszych kolejnym wątkom i pozwolenie im na szukanie ich wielokrotności na całym zbiorze.
Próbowałem obu z tych podejść, ale końcowo zdecydowałem się na drugie z wyżej wymienionych.
Jak widać na załączonym fragmencie kodu iterujemy po dostępnych wątkach oddając każdemu z nich kolejną z małych liczb pierwszych.

```
for (int k = PCJ.myId(); k < smallPrimes.size(); k += PCJ.threadCount()) {
    int i = smallPrimes.get(k);
```

Gdy wątek dochodzi do najważniejszej części algorytmu, czyli wykreślania wielokrotności wybranej liczby przechodzi on po całej wielkości zbioru, a nie po jego fragmencie.

```
for (int j = lower; j <= high; j = j + i) {
    bigPrimes.set(j - low, false);
}
```

W ten sposób unikam możliwych “najść” jakie występowały podczas dzielenia na podzbiory. W zależności od całkowitości pierwiastka mogło się okazać, że niektóre liczby pierwsze leżące na granicy podzbiorów były zapisywane w dwóch podzbiorach jednocześnie. W tym podejściu nie spotykamy się z duplikatami.

Na koniec wyniki wszystkich wątków są łączone w jeden wynikowy BitSet przez główny wątek.

```
PcjFuture cL[] = new PcjFuture[PCJ.threadCount()];
if (PCJ.myId() == 0) {
    for (int p = 1; p < PCJ.threadCount(); p++) {
        cL[p] = PCJ.asyncGet(p, Shared.bigPrimes);
    }
    for (int p = 1; p < PCJ.threadCount(); p++) {
        bigPrimes = concatenateVectors(bigPrimes, (BitSet) cL[p].get());
    }
}
```

Porównanie czasowe

Badanie zostało przeprowadzone na pojedynczych wywołaniach, w celu jednakowego sprawdzenia wyników na “zimnej” maszynie javy.

Trzeba tutaj także pamiętać, że implementacja dla liczb większych niż 2 mld jest nieco wolniejsza ze względu na tworzenie podgrup i ich tablicowanie.

Dla jednego wątku

Liczby poniżej 2 mld

```
sty 30, 2023 1:04:50 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.3.0-c529b9c (2021-06-28T10:15:49.880+0200)
sty 30, 2023 1:04:50 PM org.pcj.internal.InternalPCJ start
INFO: Starting org.example.Main with 1 thread (on 1 node)...
Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>
Przyklad: 1 100 1000000
Prosze wybrac tryb
1 - Wypisz wszystkie liczby pierwsze z przedzialu
2 - Wypisz najwieksza z przedzialu
3 - Policz ilosc liczb piewszych w przedziale
0 - Zakoncz dzialanie
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0032227s
Czas znalezenia wszystkich liczb pierwszych to 20,0523941s
W podanym przedziale jest 98222287 liczb pierwszych
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0010721s
Czas znalezenia wszystkich liczb pierwszych to 19,7798030s
W podanym przedziale jest 98222287 liczb pierwszych
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0002983s
Czas znalezenia wszystkich liczb pierwszych to 21,1720071s
W podanym przedziale jest 98222287 liczb pierwszych
```

Liczby powyżej 2 mld

```
sty 29, 2023 3:15:32 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.3.0-c529b9c (2021-06-28T10:15:49.880+0200)
sty 29, 2023 3:15:32 PM org.pcj.internal.InternalPCJ start
INFO: Starting org.example.Main with 1 thread (on 1 node)...
Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>
Przyklad: 1 100 1000000
Prosze wybrac tryb
1 - Wypisz wszystkie liczby pierwsze z przedzialu
2 - Wypisz najwieksza z przedzialu
3 - Policz ilosc liczb piewszych w przedziale
0 - Zakoncz dzialanie
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0028667s
Czas znalezenia wszystkich liczb pierwszych to 32,7320434s
W podanym przedziale jest 2091739524 liczb pierwszych
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0006420s
Czas znalezenia wszystkich liczb pierwszych to 32,5208006s
W podanym przedziale jest 2091739524 liczb pierwszych
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0004784s
Czas znalezenia wszystkich liczb pierwszych to 32,3787092s
W podanym przedziale jest 2091739524 liczb pierwszych
```

Dla dwóch wątków

Liczby poniżej 2 mld

```
sty 30, 2023 1:15:11 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.3.0-c529b9c (2021-06-28T10:15:49.880+0200)
sty 30, 2023 1:15:11 PM org.pcj.internal.InternalPCJ start
INFO: Starting org.example.Main with 2 threads (on 1 node)...
Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>
Przyklad: 1 100 1000000
Prosze wybrac tryb
1 - Wypisz wszystkie liczby pierwsze z przedzialu
2 - Wypisz najwieksza z przedzialu
3 - Policz ilosc liczb piewszych w przedziale
0 - Zakoncz dzialanie
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0031189s
Czas znalezenia wszystkich liczb pierwszych to 12,8718776s
W podanym przedziale jest 98222287 liczb pierwszych
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0014605s
Czas znalezenia wszystkich liczb pierwszych to 13,2899360s
W podanym przedziale jest 98222287 liczb pierwszych
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0009612s
Czas znalezenia wszystkich liczb pierwszych to 13,2836100s
W podanym przedziale jest 98222287 liczb pierwszych
```

Liczby powyżej 2 mld

```
sty 30, 2023 1:11:08 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.3.0-c529b9c (2021-06-28T10:15:49.880+0200)
sty 30, 2023 1:11:09 PM org.pcj.internal.InternalPCJ start
INFO: Starting org.example.Main with 2 threads (on 1 node)...
Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>
Przyklad: 1 100 1000000
Prosze wybrac tryb
1 - Wypisz wszystkie liczby pierwsze z przedzialu
2 - Wypisz najwieksza z przedzialu
3 - Policz ilosc liczb piewszych w przedziale
0 - Zakoncz dzialanie
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0027500s
Czas znalezenia wszystkich liczb pierwszych to 21,7068241s
W podanym przedziale jest 2091739524 liczb pierwszych
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0010098s
Czas znalezenia wszystkich liczb pierwszych to 19,7030805s
W podanym przedziale jest 2091739524 liczb pierwszych
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0008586s
Czas znalezenia wszystkich liczb pierwszych to 20,3599537s
W podanym przedziale jest 2091739524 liczb pierwszych
```

Dla czterech wątków

Liczby poniżej 2 mld

```
sty 30, 2023 1:18:03 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.3.0-c529b9c (2021-06-28T10:15:49.880+0200)
sty 30, 2023 1:18:03 PM org.pcj.internal.InternalPCJ start
INFO: Starting org.example.Main with 4 threads (on 1 node)...
Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>
Przyklad: 1 100 1000000
Prosze wybrac tryb
1 - Wypisz wszystkie liczby pierwsze z przedzialu
2 - Wypisz najwieksza z przedzialu
3 - Policz ilosc liczb piewszych w przedziale
0 - Zakoncz dzialanie
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0030750s
Czas znalezenia wszystkich liczb pierwszych to 9,7993909s
W podanym przedziale jest 98222287 liczb pierwszych
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0012918s
Czas znalezenia wszystkich liczb pierwszych to 10,5584844s
W podanym przedziale jest 98222287 liczb pierwszych
3 2 2000000000
Czas znalezenia malych liczb pierwszych to 0,0008155s
Czas znalezenia wszystkich liczb pierwszych to 10,4714192s
W podanym przedziale jest 98222287 liczb pierwszych
```

Liczby powyżej 2 mld

```
sty 30, 2023 1:21:06 PM org.pcj.internal.InternalPCJ start
INFO: PCJ version 5.3.0-c529b9c (2021-06-28T10:15:49.880+0200)
sty 30, 2023 1:21:07 PM org.pcj.internal.InternalPCJ start
INFO: Starting org.example.Main with 4 threads (on 1 node)...
Sposob uzycia: <liczba oznaczajaca tryb> <dwie liczby oznaczajace przedzial liczbowy>
Przyklad: 1 100 1000000
Prosze wybrac tryb
1 - Wypisz wszystkie liczby pierwsze z przedzialu
2 - Wypisz najwieksza z przedzialu
3 - Policz ilosc liczb piewszych w przedziale
0 - Zakoncz dzialanie
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0027897s
Czas znalezenia wszystkich liczb pierwszych to 13,9230026s
W podanym przedziale jest 2091739524 liczb pierwszych
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0009442s
Czas znalezenia wszystkich liczb pierwszych to 13,7627792s
W podanym przedziale jest 2091739524 liczb pierwszych
3 2000000000 4000000000
Czas znalezenia malych liczb pierwszych to 0,0015838s
Czas znalezenia wszystkich liczb pierwszych to 14,7220921s
W podanym przedziale jest 2091739524 liczb pierwszych
```

Podsumowanie

Jak widać wyniki są zadowalające i wyraźnie wskazują na duże przyspieszenie wraz z ilością wątków. Przetwarzanie dla większych liczb oczywiście było wolniejsze co mogliśmy przewidzieć, jednak różnica nie była drastycznie duża.

Wszystkie podane czasy wykonania określają wyliczenie liczb pierwszych w zbiorze i nie uwzględniają czasu na zebranie informacji do wspólnego zbioru, ani ich wypisania.

Poza raportem kod został, także całkowicie omówiony w samej implementacji poprzez komentarze. Mam nadzieję, że dzięki nim wszystkie niepewności zostaną rozwiane.