In[5]:=
```
twinFraction[low_, high_] :=
 Module[{listOfPrimes, (* List of all prime numbers in the interval. *)
   numberOfPrimes, (* Number of primes in the interval. *)
   numberOfTwins, (* Number of Mersenne primes in the interval. *)
   previousPrime, (* The previous prime at which we looked. *)
   currentPrime, (* The first prime larger than currentPrime. *)
   futurePrime (* We actually have to look at three primes in a row. *)
  },

  If[low < high, (* Basic error checking: low should be < high. *)
   {
    numberOfPrimes = PrimePi[high] - PrimePi[low];
    (* PrimePi gives the number of primes ≤ its argument. *)
    listOfPrimes = Table[0, {i, 1, numberOfPrimes}];
    (* Initialize listOfPrimes to the right size. *)
    previousPrime = NextPrime[low];
    (* Init previousPrime to the lowest prime in the interval. *)
    listOfPrimes[[1]] = previousPrime;
    currentPrime = NextPrime[previousPrime];
    (* Go higher up on the prime ladder. *)
    futurePrime = NextPrime[currentPrime];
    (* Go even higher up on the prime ladder rung. *)
    numberOfTwins = 0;
    (* We will check which of them are twin as we go. If a differs from the next
     one by 2, then it is a twin prime. This algorithm is crafted to allow the 3,
    5,7 chain to increment the counter by 3, as it should. *)
    For[i = 2, currentPrime ≤ high, i++,
     {
      listOfPrimes[[i]] = currentPrime;
      (* Stuff the current prime into the list. Assume we have at least 2. *)
      If[(2 == currentPrime - previousPrime) || (2 == futurePrime - currentPrime),
       numberOfTwins++] ;
      (* Increment the counter. *)
      previousPrime = currentPrime;
      currentPrime = futurePrime;
      futurePrime = NextPrime[currentPrime]
      (* Move on to the next
       prime. futurePrime can go beyond the end of the interval. *)
     }
    ];
   },
   (* else *)
   {
     Print["Please enter the lower endpoint
        of the interval first, then the higher endpoint."];
     numberOfTwins = 0;
     numberOfPrimes = 1; (* In order to return a reasonable answer. *)
```

```
    };
  ];
  N[numberOfTwins / numberOfPrimes, 5]
  (* This fraction yields the desired return value of the function. *)
 ]
```

Testing the first few intervals:

In[14]:=
```
twinFraction[0, 10]
N[3 / PrimePi[10], 5]
```

Out[14]= 0.75000

Out[15]= 0.75000

In[16]:=
```
twinFraction[0, 20]
N[7 / PrimePi[20], 5]
```

Out[16]= 0.87500

Out[17]= 0.87500

In[20]:=
```
twinFraction[0, 100]
N[15 / PrimePi[100], 5]
```

Out[20]= 0.60000

Out[21]= 0.60000

These all seem to work fine. Let' s see how high up we can go with this algorithm.

In[22]:=
```
twinFraction[0, 10 000 000]
```

Out[22]= 0.17749

Let's go for the table now.

In[23]:=
```
di = 10^6;
Timing[Table[{0, i + di, twinFraction[0, i + di]}, {i, 0, 10^7, di}] // MatrixForm]
```

Out[24]=
$$\{109.048,\ \begin{pmatrix} 0 & 1\,000\,000 & 0.20812 \\ 0 & 2\,000\,000 & 0.19969 \\ 0 & 3\,000\,000 & 0.19308 \\ 0 & 4\,000\,000 & 0.18972 \\ 0 & 5\,000\,000 & 0.18629 \\ 0 & 6\,000\,000 & 0.18368 \\ 0 & 7\,000\,000 & 0.18151 \\ 0 & 8\,000\,000 & 0.18014 \\ 0 & 9\,000\,000 & 0.17881 \\ 0 & 10\,000\,000 & 0.17749 \\ 0 & 11\,000\,000 & 0.17629 \end{pmatrix}\}$$

From this table, we can observe a number of things. First of all, the algorithm took a little under two minutes to run on the following hardware/software environment:

Dell Inspiron, 8GB RAM, Intel Core i3-2350M CPU @ 2.30GHz x 2, Linux Mint 17.3 Cinnamon 64-bit, *Mathematica* 10.3.

We could slap additional Timing functions in our code to see where time is being spent.

Second of all, as we widen our intervals, we can see the fraction of twin primes go down, but not terribly quickly. This would seem to indicate that twin primes form a significant percentage of all of the prime numbers.