UNBC **Laboratory Assignment**
Computer Science 200—Winter 2024

Better Big Integer Parsing

Objectives:

The `java.math.BigInteger(String ...)` constructor is inefficient. The objectives of this laboratory exercise are two-fold:

1. Document the current behavior.
2. Write an algorithm that is both practically and asymptotically faster than the Java 10 library code.

Due Date:

This assignment is due Friday, 2024-03-15 *by the beginning of lecture*.

The Current Behaviour

⇒ Write a program that demonstrates that the current Java library code for `java.math.BigInteger(String ...)` exhibits $\Theta(n^2)$ runtime time. (On my laptop[1], constructing a `BigInteger` from a 10^6-digit string takes about 12 s, and using equally spaced x-values with increment 10^5 plots nicely. You may need to experiment to find the best values for your computational environment.)

⇒ Plot your data.

⇒ Describe a simple algorithm to implement `java.math.BigInteger(String ...)` that takes quadratic time.

Do you expect that this is what your Java provider uses?

Divide and Conquer

We've seen this situation before with sorting algorithms. We have an algorithm with $\Theta(n^2)$ behaviour. If we can split the problem into two sub-pieces, instead of taking cn^2

[1] 2019 Mac Pro with 64 G memory running Mac OS 12.7.3, and using openjdk 19.0.2 2023-01-17.

time, we take $2c(n/2)^2 = cn^2/2$ time, or a saving of 50%. However, we then need to be able to recombine the pieces. *If* we can recombine in $o(n^2)$ time, we have a saving. In that case, when we apply this strategy recursively, we end up with a reduction in complexity (as we saw with mergesort and quicksort).

Let's see what happens in the context of converting a `String` to a `BigInteger`.

Suppose that we have a `String s` and we split it into two pieces `s1` and `s2` of approximately equal size so that `s.equals(s1+s2)`. Then, after

```
    BigInteger d = new BigInteger(s) ;

    BigInteger d1 = new BigInteger(s1) ;
    BigInteger d2 = new BigInteger(s2) ;

    BigInteger d3 = d2.add(
        d1.multiply(
            BigInteger.TEN.pow(s2.length()))) ;
```

`d` and `d3` should be the same number. For instance, if

```
    String s  = "123456" ;
    String s1 = "123" ;
    String s2 = "456" ;
```

then $d = 123456$, $d_1 = 123$, $d_2 = 456$, and $d_3 = 456 + 10^3 \cdot 123$. The time to compute d_3 consists of

1. the time to compute d_1
2. the time to compute d_2
3. the time to compute $B = 10^{|s_2|}$
4. the time to multiply d_1 and B
5. the time to add d_2 and $B \cdot d_1$.

Assuming that s_1 and s_2 are close to the same length, we get from Lab 1 that

4′. the time to multiply d_1 and B is $\Theta(\text{length}(s_1)^{1.6})$.
5′. the time to add d_2 and $B \cdot d_1$ is $\Theta(\text{length}(s_1))$.

Assuming that we can avoid the cost of computing B, we get a running time equation

$$T(n) = 2T(n/2) + c \cdot (n/2)^{1.6}, \quad (1)$$

$$\text{which gives} \quad T(n) = \Theta(n^{1.6}). \quad (2)$$

This is our strategy then. Use "divide-and-conquer" to split our problem into half-size pieces, and then use the fact that `BigInteger` has reasonably fast multiplication to recombine the pieces.

Base cases

Our strategy is recursive. What, then, are our base cases? We know that JAVA can read an ordinary integer quickly, and this suggests that our base case be digit strings with 9 or less digits.

Reducing the cost of computing 10^k

In order to the reduce the number of powers of 10 that we compute, we can use the following principle:

> *Whenever we split a string into two sub-problems, the right-hand substring will have a length* $9 \cdot 2^k$.

This means that we only ever need to compute powers of 10 of the form $10^{9 \cdot 2^k}$. Also note that

$$10^{9 \cdot 2^{k+1}} = 10^{9 \cdot 2 \cdot 2^k} = (10^{9 \cdot 2^k})^2, \tag{3}$$

meaning that all of the powers can be produced by squaring.

Memo-ization

We can also use the following trick: Keep a `java.util.Map` of previously computed powers. The clean way to code this is to provide a method, say `tenToThe9xTwoToThe` that does the actual computation, and have it consult a map that keeps previously computed answers. Sample code is shown in Figure 1.

- Lines 2–3 and 7–8 set up the map. (`Map` is an interface, `TreeMap` is a concrete (but generic) class.)
- Line 29 shows how to get a value from the map. Note that we only ever return values that are stored in the map, so that we know that each value is computed only once.
- Line 27 shows how to store an answer in the map.
- Line 14 shows how to check if the map has an answer for k. If so, we skip to line 29 and return it. Otherwise, we compute and store the answer first.
- Lines 20 and 24–25 do the actual computation. Note that these lines are "pure" formulas. That is, they would work whether or not we remembered the answer.

This pattern is worth learning, as it often works, and is the basis of a technique called *dynamic programming*.

```java
import java.Math.BigInteger ;
import java.util.Map;
import java.util.TreeMap ;

class Helper
{
  private static java.util.Map<Integer,BigInteger> stash =
    new java.util.TreeMap<Integer,BigInteger> ();


  public static BigInteger tenToThe9xTwoToThe(int k)
      {
      if (k<0) throw IllegalArgumentException("Positive powers only") ;
      if (! stash.containsKey(k))
          {
          BigInteger answer ;
          if (k==0)
              {
              // base case
              answer = BigInteger.TEN.pow(9) ;
              }
          else
              {
              BigInteger temp = tenToThe9xTwoToThe(k-1) ;
              answer = temp.multiply(temp) ;
              }
          stash.put(k,answer) ;
          }
      return stash.get(k) ;
      }
}
```

Figure 1: Memoizing the powers of 10

Here, then, is a sketch of an algorithm for creating a `BigInteger` from a `String`.

An Approximate Algorithm

1. Set n to the length of the string to be converted.
2. If $n < 10$, use the builtin `BigInteger` constructor. (base)
   Otherwise:
3. Find a value of k so that $n/3 \leq 9 \cdot 2^k < 2n/3$.
4. Set $n_2 = 9 \cdot 2^k$ and $n_1 = n - n_2$ and $B = 10^{n_2}$.
5. Read the first n_1 digits into a `BigInteger` d_1. (recursive)
6. Read the last n_2 digits into a `BigInteger` d_2. (recursive)
7. Return the value of the whole string: which is $d_2 + B \cdot d_1$.

Implementation notes

- Although it doesn't affect the asymptotic time efficiency, splitting a really big `String` into two pieces using `String.substring` is inefficient in both time and space.

  It is more efficient to have a recursive algorithm with signature

  ```
  public BigInteger parseBig(String, int start, int length) { ... }
  ```

  where `start` and `length` describe the substring to convert.

- For step 3, a simple loop on k suffices. Alternatively, $k = \lceil \log_2 n - \log_2 27 \rceil$.

- For step 4, compute B using the function `tenToThe9xTwoTo`$\tilde{\texttt{T}}$`he`(k).

- For step 6, the recursive calls will use a value of k equal to the current value of k minus one $[\frac{1}{2}(9 \cdot 2^k) = 9 \cdot 2^{k-1}]$. Using yet another method

  ```
  public BigInteger parseBig(String, int start, int length, int k) { ... }
  ```

  will allow you to avoid recomputing k, alhtough this is a tiny improvement.

Testing

To help you test your program, a file `test-data.txt.zip` is supplied with the lab. Unzip this file. `test-data.txt` contains one line that conatain a million digit number.

The number in the file is exactly divisble by 37, multiple times. Report the nhmber of times 37 divides evenly, and the remainder when it doesn't. For intance

		The result shoud be	
For n equal		devides exactly	and then has
	39	0 times	remainder 2
$40 \cdot 37^2 =$	53391	2 times	remainder 3

Coding & Hand In

⇒ Complete the three ⇒ points under "**The Current Behaviour**, namely

- Write a program that demonstrates that the Java library code for `java.math.BigInteger(String ...)` has $\Theta(n^2)$ behaviour.
- Plot your data.
- In code comments, make a guess as what algorithm Oracle uses.

⇒ Hand in Java code for your tests, the numerical output from your tests, and a graphics file with your plot.

⇒ Code, test, and time the algorithm described at the top of page 5.

⇒ Again, hand in Java code for your tests, the numerical output from your tests, and a graphics file with your plot.

⇒ Test the correctness of your program by using the `test-data.txt.zip` file as described on the preceding page.

⇒ In your code comments for the memoization of `tenToThe9xTwoTõThe`, describe how many entries you expect to find in the map. If you want a more precise question, how many entries would you have in the map if you read a billion digit string?