CS2030 Programming Methodology
Semester 1, 2020/2021

21 October 2020
Problem Set #8 Lazy Evaluation
Suggested Guidance

1. Let's explore the difference in computational efficiency between using `ArrayList` (which employs eager computation) and `LazyList` (which employs lazy computation). To do this, we will find the k^{th} prime number in a given interval of integers $[a, b]$ (eg. the 4^{th} prime number in $[2, 100]$ is 7) using the two different types of lists.

   Study the program listed in `Primes.java`; in particular, compare the methods `findKthPrimeLL` and `findKthPrimeArr`. Both use the same functional programming approach: (i) generate a list of integers in the given range, (ii) filter the list using the `isPrime` predicate, and (iii) retrieve the k^{th} element from the filtered list.

   The program is already written and compiled for you; you just need to run it. To do so:

   1. Read the Appendix for instructions on how to setup.
   2. In the `processed/` directory, run the program:

      ```
      java Primes 100000 200000 5 1
      ```

      This will find the 5^{th} prime in the interval $[100000, 200000]$ using `LazyList`. The program will report the prime number found, and also the number of times `isPrime` was called:

      ```
      The 5-th prime is 100057
      isPrime was called 58 times
      ```

   3. Type: `java Primes` to read further instructions on how to run the program.
   4. By choosing between the 2 methods of finding primes, compare how many times `isPrime` is invoked. Try large ranges to see the difference, eg. there are over 8000 primes in the interval $[100000, 200000]$.[1]

   Answer these questions:

   (a) To find the 5^{th} prime in the interval $[100000, 200000]$, why does `findKthPrimeLL` make so many fewer calls to `isPrime` compared to `findKthPrimeArr` ?

   (b) Does the number of calls depend on k, or on the interval $[a, b]$, or both?

   (c) Can `findKthPrimeLL` make more calls to `isPrime` than `findKthPrimeArr`? Epxlain.

   Solution:

[1] The Prime Counting Function, $\pi(x)$, is the number of primes up to and including integer x. Try it here: `https://www.dcode.fr/prime-number-pi-count`

(a) `findKthPrimeLL` makes 58 calls because it only needs to examine $100000, 100001, \ldots, 100057$ to find the 5^{th} prime. The rest of the list is not thawed because there is no need to. On the other hand, `FindKthPrimeArr` makes 100000 calls because it checks *every number* in the interval and keeps (in `primesArrayList`) only those which are prime.

(b) For `findKthPrimeArr`, the number of calls to `isPrime` is always $b - a$, and does not depend on k. For `findKthPrimeLL`, the number of calls depends on k, as well as on the interval $[a, b]$. This is because for smaller numbers a, b, primes occur more often. One does not have to test many integers to find a prime. Whereas for larger a, b, primes are more sparsely distributed, and one needs to check many more integers before finding a prime. For example: `findKthPrimeLL` tests only 10 integers to find the 5^{th} prime in $[2, 100002]$, compared to 58 in the former case.

(c) No, it can't, because `findKthPrimeLL` will test only as many integers as needed, which is at most all the integers in the interval. The worst case is when k equals the number of primes in the interval, and that k^{th} prime is $b - 1$. In such a case, `findKthPrimeLL` will call `isPrime` exaclty $b - a$ times, just like `findKthPrimeArr`. For example, there are exactly 25 primes up to 100, the last one being 97. Thus, for the interval $[2, 98]$, the 25^{th} prime represents the worst case scenario. Try it to see: `java Primes 2 98 25 1`.

2. Two LazyLists may be *concatenated*, ie. a new LazyList is created whose elements are those in the first list, in order, followed by those in the second list, in order. Example:

```
var s1 = LazyList.fromList(1, 2, 3);
var s2 = LazyList.fromList(4, 5);
s1.print();
=> (* 1, 2, 3, *)

s2.print();
=> (* 4, 5, *)

s1.concat(s2).print();
=> (* 1, 2, 3, 4, 5, *)
```

Note that `concat` will not work if the lists are infinite. Here's the code:

```
public LazyList<T> concat(LazyList<T> other) {
    if (this.isEmpty())
        return other;
    else
        return LLmake(this.head(),
                      this.tail().concat(other));
}
```

(a) Using `concat`, add an instance method `reverse()` to the LazyList class. Since LazyLists are immutable, `reverse()` should return a new list, whose elements are the elements of `this` list, but in reverse order.

(b) Try your code on: `LazyList.fromList(1, 2, 3, 4, 5).reverse().print()`

(c) Complete the code below to create a `flatmap` instance method. *Hint:* Use `concat`.

```
/** **************
    Apply the function f onto each element of this list, and
    return a new LazyList containing all the flattened mapped elements.
    Note that f produces a list for each element. But the returned
    list flattens them all, ie. removes nested lists.
    */
    <R> LazyList<R> flatmap(Function<T, LazyList<R>> f) {
        if (this.isEmpty())
            return LazyList.makeEmpty();
        else
            // insert code here
    }
```

Solution:

```
//(a)
public LazyList<T> reverse() {
    if (this.isEmpty())
        return this;
    else
        return this.tail()
            .reverse()
            .concat(LLmake(this.head(),
                           LazyList.makeEmpty()));
}
//(b)
LazyList.fromList(1, 2, 3, 4, 5).reverse().print();
=> (* 5, 4, 3, 2, 1, *)
//(c)
public <R> LazyList<R> flatmap(Function<T, LazyList<R>> f) {
    if (this.isEmpty())
        return LazyList.makeEmpty();
    else
        return f.apply(this.head())
            .concat(this.tail().flatmap(f));
}
```

(d) An $r-$Permutation of a list of n integers is a arrangement r integers, taken without repetition from the list. Example: $(3, 1, 2)$ is a 3-Permutation of $(1, 2, 3, 4)$; a different 3-Permutation is $(2, 3, 1)$. Here's how we may generate all the $r-$Permutations of a list L of n integers.

1. If $r = 1$, then this is a list of singleton lists of each of the elements in L. Example: for $L = (1, 2, 3, 4)$, there are four 1-Permutations: $((1), (2), (3), (4))$.
2. If $r > 1$, take each element x in turn from L, recursively compute the $(r-1)$-Permutation of $L - x$ (ie. L with x removed). This will give a list of $(r-1)-$Permutations. We now insert x to the front of each of these.

In the file `Puzzle.java`, complete the code implement the `permute` function.

```
LazyList<Integer> remove(LazyList<Integer> LL, int n) {
    return LL.filter(x-> x != n);
}

LazyList<LazyList<Integer>> permute(LazyList<Integer> LL, int r) {
    if (r == 1)
        return LL.map(x-> LLmake(x, LazyList.makeEmpty()));
    else
        // Insert code here
}
```

(e) Try it on: `permute(LazyList.intRange(1,5), 3).forEach(LazyList::print);` This should print ${}^4P_3 = 24$ 3-permutations. Note that the `forEach` instance method applies the given `Consumer` function onto each element of the list.

Read the Appendix to see how to compile your code.

Solution:

```
//d
public static LazyList<LazyList<Integer>> permute(LazyList<Integer> LL,
                                                  int r) {
    if (r == 1)
        return LL.map(x-> LLmake(x, LazyList.makeEmpty()));
    else
        return LL.flatmap(x ->
                          permute(remove(LL, x), r - 1)
                          .map(y -> LLmake(x,y)));
}
//e
(* 1, 2, 3, *)
(* 1, 2, 4, *)
(* 1, 3, 2, *)
(* 1, 3, 4, *)
(* 1, 4, 2, *)
(* 1, 4, 3, *)
(* 2, 1, 3, *)
(* 2, 1, 4, *)
(* 2, 3, 1, *)
(* 2, 3, 4, *)
(* 2, 4, 1, *)
(* 2, 4, 3, *)
(* 3, 1, 2, *)
(* 3, 1, 4, *)
(* 3, 2, 1, *)
(* 3, 2, 4, *)
(* 3, 4, 1, *)
(* 3, 4, 2, *)
(* 4, 1, 2, *)
```

```
(* 4, 1, 3, *)
(* 4, 2, 1, *)
(* 4, 2, 3, *)
(* 4, 3, 1, *)
(* 4, 3, 2, *)
```

3. A cryptarithmetic puzzle is shown below. Each letter represents a distinct numeric digit. The problem is to find what each letter represents so that the mathematical statement is true.

$$\begin{array}{cccc} & A & B \\ + & B & C \\ \hline A & X & Y \\ \hline \end{array}$$

In this example: $A = 1, B = 8, C = 4, X = 0, Y = 2$ is a solution to the puzzle. Other solutions are also possible, as you can easily determine.

(a) Let's try to solve the cryptarithmetic problem below by brute force, ie. checking all the possibilities. First, generate all the 6-Permutations of the list of 10 digits (0 to 9). These 6-Permutations correspond to our choice of C, H, M, P, U, Z. Next, check each 6-Permutation to see if it satisfies the given equation.

$$\begin{array}{ccccc} & P & Z & C & Z \\ \times & & & 1 & 5 \\ \hline M & U & C & H & Z \\ \hline \end{array}$$

Complete the definition of `pzczSatisfies` in Puzzle.java to solve the problem. To run it, increase the stack size of the Java Virtual Machine (JVM) to 8Mb as follows: `java -Xss8m Puzzle`. Otherwise, you may run out of stack space.

```
public class Puzzle {

    static boolean pzczSatisfies(LazyList<Integer> term) {
    // term is a list of 6 digits
    int c = term.get(0);
    int h = term.get(1);
    int m = term.get(2);
    int p = term.get(3);
    int u = term.get(4);
    int z = term.get(5);

    //Insert your code here.
    //Return true only when both m,p are not 0,
    //and the given equation is satisfied.
    }

    public static void main(String[] args) {
        permute(LazyList.intRange(0,10), 6)
```

```
            .filter(Puzzle::pzczSatisfies)
            .forEach(LazyList::print);
    }
}
```

Solution:

```
static boolean pzczSatisfies(LazyList<Integer> term) {
    int c = term.get(0);
    int h = term.get(1);
    int m = term.get(2);
    int p = term.get(3);
    int u = term.get(4);
    int z = term.get(5);

    int pzcz = p*1000 + z*100 + c*10 + z;
    int muchz = m*10000 + u*1000 + c*100 + h*10 + z;

    return m!=0 && p!=0 && (pzcz*15 == muchz);
}

//To run it, increase the stack size to 16Mb:
// java -Xss16m Puzzle
//The only solution is:
(* 9, 2, 6, 4, 8, 5, *)

// In other words:
    4595 * 15 = 68925
```

Notice how elegant the solution is! By comparison, the following code is masochistic and bad for health :-(

```
void bruteForceForLoop() {
    List<List<Integer>> allSolutions = new ArrayList<>();

    for (int c=0; c<10; c++)
        for (int h=0; h<10; h++)
            if (h!=c)
                for (int m=1; m<10; m++)
                    if (m!=c && m!=h)
                        for (int p=1; p<10; p++)
                            if (p!=c && p!=h && p!=m)
                                for (int u=0; u<10; u++)
                                    if (u!=c && u!=h && u!=m && u!=p)
                                        for (int z=0; z<10; z++)
                                            if (z!=c && z!=h && z!=m && z!=p && z!=u) {
                                                int pzcz = p*1000 + z*100 + c*10 + z;
                                                int muchz = m*10000 + u*1000 + c*100 + h*10 + z;
                                                if (pzcz*15 == muchz)
                                                    allSolutions.add(Arrays.asList(c,h,m,p,u,z));
                                            }

    System.out.println(allSolutions);
}
```

Actually, the 6-deep for-loop is an overkill. Notice that all the candidates for PZCZ are 4-digit numbers, so you can simply generate such numbers and check if they fit the PZCZ format, and also that PZCZ*15 fit the MUCHZ format. This yields the following code, which has only a single for-loop.

```
void betterForLoop() {
    List<Integer> allSolutions = new ArrayList<>();

    for (int n=1000; n<10000; n++)
        if (isPZCZ(n) && isProductMuchz(n))
            allSolutions.add(n);

    System.out.println(allSolutions);
}

boolean isPZCZ(int n) {
    /* check that: n has 4 digits, its format is PZCZ,
       and P,C,Z are distinct */

    if (!(1000 <= n && n<10000))
        return false;

    String strPZCZ = String.valueOf(n);
    char p = strPZCZ.charAt(0);
    char z = strPZCZ.charAt(1);
    char c = strPZCZ.charAt(2);
    char last = strPZCZ.charAt(3);

    return z==last && p!=z && p!=c && z!=c;
}

boolean isProductMuchz(int pzcz) {
    /* check that: pzcz*15 has 5 digits, it has C,Z in the right positions,
       and all digits are distinct from PZCZ. */

    int muchz = pzcz * 15;
    if (!(10000<= muchz && muchz<100000))
        return false;

    String strPZCZ = String.valueOf(pzcz);
    char p = strPZCZ.charAt(0);
    char z = strPZCZ.charAt(1);
    char c = strPZCZ.charAt(2);

    String strMUCHZ = String.valueOf(muchz);
    char mm = strMUCHZ.charAt(0);
    char uu = strMUCHZ.charAt(1);
    char cc = strMUCHZ.charAt(2);
```

```
    char hh = strMUCHZ.charAt(3);
    char zz = strMUCHZ.charAt(4);

    return z==zz && c==cc && mm!=p && mm!=z && mm!=c &&
        uu!=mm && uu!=p && uu!=z && uu!=c &&
        hh!=mm && hh!=uu && hh!=p && hh!=z && hh!=c;
}
```

But this idea can be used in lazy evaluation too. Just for a change, let's use the Java `IntStream` instead. Again, notice how elegant the code is!

```
void streamSolution() {
    IntStream.range(1000,10000)
        .filter(Puzzle::isPZCZ)
        .filter(Puzzle::isProductMuchz)
        .forEach(System.out::println);
}
```

(b) Using a similar strategy, solve this problem:

$$\begin{array}{cccccc} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Solution: Here, there are 8 letters: D, E, M, N, O, R, S, Y. Thus we need to enumerate all $8-$Permutations of the digits $0-9$, and then filter it with a suitable predicate.

```
static boolean moneySatisfies(LazyList<Integer> term) {
    int d = term.get(0);
    int e = term.get(1);
    int m = term.get(2);
    int n = term.get(3);
    int o = term.get(4);
    int r = term.get(5);
    int s = term.get(6);
    int y = term.get(7);

    int send = s*1000 + e*100 + n*10 + d;
    int more = m*1000 + o*100 + r*10 + e;
    int money = m*10000 + o*1000 + n*100 + e*10 + y;

    return m!=0 && s!=0 && (send + more == money);
}

public static void main(String[] args) {
    permute(LazyList.intRange(0,10), 8)
```

```
        .filter(Puzzle::moneySatisfies)
        .forEach(LazyList::print);
}

// To run this successfully, increase the stack size to 64Mb:
// java -Xss64m Puzzle
// The only solution is:
(* 7, 5, 1, 6, 0, 8, 9, 2, *)

// In other words:
    9567 + 1085 = 10652
```