Midterm 2023 Meno a priezvisko: ____________________ 1-ID:

test obsahuje 5 úloh za 7+6+6+4+4 =27 bodov.

Čas: 2h min, začiatok 18:10, koniec 20:10.

1. Rekurzia - M1 – 7 bodov

Rekurzívna funkcia **foo** na výpočet hodnoty **foo(a,b,p)** potrebuje dve hodnoty **foo(a,b-1,p)** a **foo(a-1,b,p)**, ktoré spolu sčíta a výsledkom je potom zvyšok súčtu po delení prvočíslom **p** (argument **p** sa pri volaniach nemení). Je ale napísaná tak neefektívne, že už nevypočíta ani hodnotu pre a=50, b=25. A tú hodnotu nevypočíta len preto, že výsledok pre dávno **vypočítané hodnoty pre menšie dvojice (i,j) ako (a,b) si nepamätá**, a teda ich počíta mnohokrát a opakovane. Vašou úlohou je ju prepísať tak, aby počítala hodnoty funkcie **foo** efektívnejšie. Tu je originálna funkcia určená na modifikácie:

```
public static int foo(int a, int b, int p) {
    if (a == 0 || b == 0)
                return 1;
    else
        return (foo(a-1, b, p)+foo(a, b-1, p)) % p;
}
```

Úlohy: V triede **`Rekurzia`**

- **[1.5 bodu]** použite pole (možno dvojrozmerné) či inú dátovú štruktúru na memoizáciu, resp. dynamické programovanie, resp. na pamätanie si výsledkov **foo**, ktoré ste už vypočítali. Prepíšte funkciu **foo** na novú, efektívnejšiu verziu **public static long foo1(int a, int b, int p)** tak, aby vďaka memoizácii/dynamike vypočítala hodnoty aj pre a,b**<=1000**.

- **[1.5 bodu]** prepíšte funkciu **foo** na novú **public static long foo2(int a, int b, int p)** tak, aby táto **nebola rekurzívna** – funkcia nesmie volať sama seba. **Hint**: Na implementáciu rekurzie sa používa zásobník, príp. cyklus, preto očakávané riešenie bez rekurzie bude asi používať zásobník alebo cyklus.

[2 body] Všimnite si, že hodnota **foo(a,b,p)** vznikne sčítaním **(foo(a-1,b,p)+foo(a,b-1,p)) modulo p**. Podobnú rekurzívnu vlastnosť predsa mali kombinačné čísla, a to, že (n nad k)=(n-1 nad k-1)+(n-1 nad k). Chýba len modulo prvočíslo p. Preštudujte si obrázky pre prvočísla p=2,3,5 na nasledujúcej strane, a nájdite v ňom Pascalov trojuholník modulo **p** s vrcholom v ľavom hornom rožku. Ak sa zamyslíte (!...!), tak **foo(a,b,p)** je kombinačné číslo **(a+b nad a) modulo p**. Veríme, že nejaký spôsob na výpočet kombinačných čísel poznáte, napr. ako podiel faktoriálov, resp. niečo podobné. Vyberte si najvhodnejší spôsob na programovanie novej metódy **public static long foo3(int a, int b, int p)**, ktorá vypočíta **foo(a,b,p)** pomocou práve objaveného faktu, že je to kombinačné číslo **(a+b nad a) modulo p**. Keďže metóda má fungovať pre a,b**<=1000**, a kombinačné čísla s takýmito parametrami sú už určite mimo typu **long**, očakáva sa riešenie pomocou triedy **BigInteger**. **Pomôcky:** tento riadok obsahuje asi všetko, čo potrebujete vedieť o triede **BigInteger** a jej metódach. Ak nie, pýtajte sa...

$$\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!} & \text{pre } n \geq k \geq 0; \\ 0 & \text{inak} \end{cases}$$

```
BigInteger.valueOf(2023).divide(BigInteger.valueOf(18))
.multiply(BigInteger.ONE).mod(BigInteger.TWO).intValue()
```

[2 body] Na výpočet kombinačného čísla modulo **p**-prvočíslo existuje **Lucasove** tvrdenie - nie nášho Lukáša 😉, ktoré by ste rýchlo vygooglili, keby to bol príklad na domácu úlohu. Úlohou je to pomocou neho opäť preprogramovať. Tu je:

Lucas' theorem states that for non-negative integers m and n, and a prime p,

$$\binom{m}{n} \equiv \prod_{i=0}^{k} \binom{m_i}{n_i} \pmod{p},$$

where $m = m_k p^k + m_{k-1} p^{k-1} + \cdots + m_1 p + m_0$ and $n = n_k p^k + n_{k-1} p^{k-1} + \cdots + n_1 p + n_0$ are the base p expansions of m and n, respectively. This uses the convention that $\binom{m}{n}$=0 when $m < n$.

Ak teda počítate kombinačné číslo **(m nad n) modulo p**, tak **m** aj **n** si prevediete do číselnej sústavy so základom **p**, v nej bude **m** mať cifry m_i, **n** cifry n_i. Potom vypočítate uvedený súčin kombinačných čísel **(m_i nad n_i)** a po každom násobení urobíte modulo **p**. Cifry m_i a n_i v číselnej sústave so základom **p** sú z intervalu 0..(p-1), takže pomerne malé čísla. Kombinačné čísla ste si už naprogramovali v predošlej časti, dokonca v rozsahu **BigInteger** – preto ich použite. Na získanie cifier v sústave so základom **p** asi stačí číslo deliť **p**, pozerať zvyšky modulo **p**, a to pôjde aj cyklom, ba aj rekurzívne... Alebo, pre milovníkov Stringov, Integer.toString(n,**p**). Definujte novú metódu **public static long foo4(int a, int b, int p),** ktorá výsledok vypočíta pomocou Lucasovej formuly pre a,b**<=1000**, a očakáva sa, že spočítate **(a+b nad a) modulo p**.

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO, NASLEDUJÚCI LIST

1. Rekurzia - M1 – 7 bodov Meno a priezvisko: ___________________ 2-ID:

Vysvetľujúci príklad: Počítajme **foo** pre (a, b, p) = (155, 78, 5), tak **foo4** je kombinačné číslo (155+78 nad 155) modulo 5. Pomocou Lucasa, vo vzorci je m=155+78=233, n = 155. V číselnej sústave so základom 5 je to $m=1413_5$, $n = 1110_5$, preto výsledkom je súčin kombinačných čísel, kam dosadíme zodpovedajúce cifry zo zápisu čísla **m** a **n** v číselnej sústave so základom **p**, teda (1 nad 1)*(4 nad 1)*(1,1)*(3,0) mod 5 = 1*4*1*1 mod 5 = 4. Ak by zápis jedného čísla v číselnej sústave so základom **p** bol kratší, sú tam predsa úvodné nuly... Presne toto máte naprogramovať.

foo p=2	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	1	1	1	1	1	1	1
1	1	0	1	0	1	0	1
2	1	1	0	0	1	1	0
3	1	0	0	0	1	0	0
4	1	1	1	1	0	0	0
5	1	0	1	0	0	0	0
6	1	1	0	0	0	0	0

foo p=3	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	1	1	1	1	1	1	1
1	1	2	0	1	2	0	1
2	1	0	0	1	0	0	1
3	1	1	1	2	2	2	0
4	1	2	0	2	1	0	0
5	1	0	0	2	0	0	0
6	1	1	1	0	0	0	0

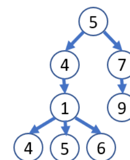
foo p=5	**0**	**1**	**2**	**3**	**4**	**5**	**6**
0	1	1	1	1	1	1	1
1	1	2	3	4	0	1	2
2	1	3	1	0	0	1	3
3	1	4	0	0	0	1	4
4	1	0	0	0	0	1	0
5	1	1	1	1	1	2	2
6	1	2	3	4	0	2	4

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST

2. Stromy – M2 - 6 bodov Meno a priezvisko: ____________________ 3-ID:

Generický strom, v ktorom vrchol môže obsahovať ľubovoľný počet synov, je definovaný ako data-class, teda record v Jave17 takto: `public record Node<E>(E value, List<Node<E>> sons) { … }` Vo vrchole je teda hodnota typu `E`, a zoznam synov, teda vrcholov `Node<E>`. Príklad konštanty typu `Node<Integer>`, ktorú neskôr použijeme v príkladoch, je:

```
Node<Integer> root =
        new Node<>(5, List.of(
            new Node<>(4, List.of(
                new Node<>(1, List.of(
                    new Node<>(4, List.of()),
                    new Node<>(5, null),
                    new Node<>(6, List.of())))))),
            new Node<>(7, List.of(
                new Node<>(9, null)))));
```



Strom uvedený v príklade má na úrovni 0 jeden vrchol 5, ktorý má dvoch synov, preto na úrovni 1 sú dva vrcholy 4 a 7, na úrovni 2 má dva vrcholy 1 a 9, a na úrovni 3 sú traja synovia 4, 5 a 6 vrcholu 1. Niekde je ale prázdny zoznam synov reprezentovaný hodnotou `null`, inokedy je to skutočne prázdny zoznam `List.of()`. Tento fakt veľmi komplikuje prácu so stromom. Kým v strome nenahradíme všetky `null` za `List.of()`, tak práca so stromom musí byť veľmi ostražitá, aby sme nedostali Null Pointer Exception. Preto hneď v prvej úlohe nahradíme `null` za `List.of()`, čo zjednoduší ďalšie metódy.

Dobrá rada: Ak v tomto príklade použijete StreamAPI, tak vaše riešenia budú prekvapivo jednoduché, elegantné a krátke. Ale nie je to podmienkou správneho riešenia. Cyklistické riešenia budú hodnotené tiež, ak budú dobré.

Úlohy: V generickej triede `Node<E>` definujte **triedne metódy** (aplikujete ich teda na v Jave skrytý argument this):

• **[1 bod]** `public Node<E> removeNulls()` – rekurzívne prejde celý strom a **vytvorí identický strom** len miesto synov sons s hodnotou `null` budú prázdne zoznamy `List.of()`.

V ďalších úlohách potom môžete predpokladať, že pracujete so stromom, ktorý už neobsahuje žiadne `nulls`, a prázdny zoznam synov je reprezentovaný inštanciou prázdneho zoznamu.

• **[1 bod]** `public String toString()` – pričom vrchol bez synov prezentujeme len ako hodnotu `value`, a vrchol s existujúcimi synmi má formát ako **(**`value`**,**`sons`**)**. Pre horeuvedený príklad musíte dostať `root.removeNulls().toString()` "(5,[(4,[(1,[4, 5, 6])]), (7,[9])])".

• **[1 bod]** `public <T> Node<T> map(Function<E,T> f)` – na každú hodnotu value typu E v celom strome typu Node<E> aplikuje funkciu Function<E,T> f, ktorá z hodnoty typu E spraví hodnotu typu T. Výsledkom bude **nový strom iného typu** Node<T> (pôvodný bol Node<E> a ten zostane nezmenený). Vlastne máte urobiť rekurzívnu kópiu stromu s tým, že hodnoty value premapujete funkciou f. **Hint**: pripomíname, že aplikácia funkcie f typu Function<E,T> nevyzerá matematicky f(value), ale v Jave je to f.apply(value).

Príklad, ktorý vám má zbehnúť `root.removeNulls().map("*"::repeat)`, a tento výsledný strom má tvar (*****,[(****,[(*,[****, *****, ******])]), (*******,[*********])]). Mapovaná funkcia `x -> "*".repeat(x)` je v skratke zapísaná ako `"*"::repeat`. Keď si uvedomíte je to funkcia typu `Function<Integer, String>`, preto mapovanie takejto funkcie zmení strom `Node<Integer>` na `Node<String>`, ktorého hodnoty value už nie sú `Integer`, ale sú `String`.

• **[1 bod]** `public Node<E> filter(Predicate<E> p)` – **vráti nový strom rovnakého typu** Node<E>, ktorý sa od pôvodného líši tak, že boli odrezané, resp. neboli prekopírované, všetky podstromy, ktorých hodnota vo vrchole E value **nespĺňa predikát p**, teda p.test(value) == false. To znamená, že ak vrchol nespĺňa podmienku p, celý jeho podstrom synov aj s ním sa vo výsledku nenachádza. A to aj keď niekde v ňom môžu byť vrcholy, ktoré podmienku spĺňajú. V uvedenom príklade `root.removeNulls().filter(x -> x % 2 > 0)` vráti `(5,[(7,[9])])` – zmizol celý podstrom párneho vrcholu 4. Ale `root.filter(x -> x % 2 == 0)` vráti null, lebo v koreni je nepárne číslo 5. **Hint**: Pripomíname, že aplikácia predikátu p nevyzerá matematicky ako p(value), ale p.test(value).

• **[2 body]** `public int width()` vráti najväčší počet vrcholov, ktorý sa nachádza na akejkoľvek úrovni stromu. V príklade stromu jednotlivé úrovne obsahujú 1,2,2,3 vrcholy, maximum je 3 a `root.removeNulls().width()` vráti `3`.

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO, NASLEDUJÚCI LIST

2. Stromy – M2 - 6 bodov Meno a priezvisko: ___________________ 4-ID:

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST

3. Oprav ma – M3 – 6 bodov Meno a priezvisko: ___________________ 5-ID:

Úloha 1: **[1 bod]** V nasledujúcom kóde prečiarknite riadky, ktoré nie sú syntakticky správne. Potom napíšte, čo sa vypíše na konzolu.

```
class Y { int d; }
public class OpravMa1 {
    record X(int c) {}
    static int a = 19;
    static Integer b = 84;
    private static void valueVsReference(int a, Integer b,  X x, Y y) {
        a++;
        b++;
        x.c++;
        x = new X(x.c+1);
        y.d++;
    }
    public static void main(String[] args) {
        X x = new X(1984);
        Y y = new Y(); y.d = 2023;
        valueVsReference(a, b, x, y);
        System.out.println(a + "," + b + "," + x + "," + y.d);  // ??, ??, ??, ??
    }
```

Úloha 2: **[1 bod]** V programe sú definované dve triedy, `Psicek` a `Macicka`. V hlavom programe, ktorý nemeňte, je vytvorený zoznam z štyroch inštancií týchto tried, vašou úlohou je dodefinovať všetko potrebné okolo tried , aby ste dostali požadovaný výpis 1324.

```
class Psicek {}
class Macicka {}
public class OpravMa2 {
    public static void main(String[] args) {
        var p1 = new Psicek();   // ID 1
        var m1 = new Macicka(); // ID 2
        var p3 = new Psicek();   // ID 3
        var m2 = new Macicka(); // ID 4
        var l = List.of(p1,p3,m1,m2);
        l.forEach( System.out::println);  // 1324
    }
}
```

Úloha 3: **[1.5 bodu]** V tom istom hlavom programe pribudli ďalšie riadky kódu. Vašou úlohou je dodefinovať komparátor **???** a všetko potrebné okolo tried tak, aby ste dostali požadované výpisy **4321**, **1234** a **2**.

```
        var x = new ArrayList<>(l);
        x.sort(???);
        x.forEach(System.out::print);  // 4 3 2 1

        var y = new TreeSet<>(l);
        y.forEach(System.out::print); // 1 2 3 4

        var z = new HashSet<>(l);
        System.out.println(z.size());  // 2
```

Úloha 4: **[1 bod]** Niekto chcel napísať bubble sort, ale jeho kód netriedi. Opravte to na funkčný Bubble Sort.

```
public static void main(String[] args)  {
    int[] a = {4,5,2,12,1,2,3,0,3,4,1,2,3,6,4,5,5,2,12,1,2,3,55,2,2,4,8,9,3,54,2,3};
    for (int i = 0; i < a.length ; i++)  {  // cyklus for-to-do
        for (i = a.length-1; i>0 ; i--)  {  // cyklus for-downto-do
            if (a[i-1] > a[i]) {
                int temp = a[i];
                a[i] = a[i-1];
                a[i-1] = temp;
            } // if
        } // for
    } // for
    for (int elem:a) System.out.println(elem);
}
```

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST

3. Oprav ma – M3 – 6 bodov Meno a priezvisko: ___________________ 6-ID:

Úloha 5: **[0.5 bodu]** Program okrem inicializácie obsahuje týchto 6 riadkov kódu, ktorým bol odstopovaný čas výpočtu v milisekundách na mojom notebooku. Namerané výsledky utriedené sú: **17, 21, 26, 32, 51, 78** ms. Zamyslite sa a priraďte namerané časy jednotlivým riadkom podľa toho, čo si myslíte, že je časovo náročnejšie.

```
final int MAX = 1_000_000;
var l = new ArrayList<Integer>();

   1. for(var i = 0; i < MAX; i++) l.add(i);
   2. l.stream().toList();
   3. l.stream().map(x -> x+1).toList();
   4. l.stream().map(x -> x+1).filter(x -> x < MAX/2).toList();
   5. l.stream().filter(x -> x < MAX/2).map(x -> x+1).toList();
   6. l.sort(Comparator.reverseOrder());
```

Úloha 6: **[1 bod]** Nasledujúci program

```
for(int i = 0; i < s.length; i++) {
    for(int j = 0; j < s[i].length; j++) {
        System.out.print(s[i][j]);
    }
}
```

skončí výpisom reťazca 123456789 a chybou, presne takto: 123456789Exception NullPointerException

Pričom Modré je výstup cez System.out.print do out, červené je chyba/výnimka do System.err.

Definujte premennú **s** a **inicializujte** tak, aby k popísanej chybe došlo presne podľa výstupu.

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST

4. Zobrazenia – M4 – 4 body Meno a priezvisko: ____________________ 7-ID:

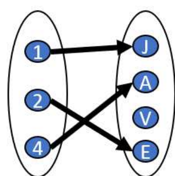
V tomto príklade sa vyskytujú pojmy, ktoré ste počuli na Diskrétke, všetko znova vysvetlíme, pochopíte a naprogramujete. Nepreskakujte zadanie, len pre pár cudzích pojmov.

Zobrazenie je špeciálnym prípadom relácie, ak platí, že môžeme povedať, že jeden vzor sa zobrazuje na najviac jeden obraz. V Jave na to slúži typ **Map<K, V>,** hodnotou ktorého je zobrazenie z typu **K** – kľúče, do hodnôt typu **V**. Zjednodušíme si vyjadrovanie, ak povieme, že definičný obor zobrazenia **Map<K, V> f** je **f.keySet()** a obor hodnôt je **f.values()** – tieto dve metódy určite poznáte. *Disclaimer: toto „zjednodušenie“ oborov na diskrétke nespomínajte* 😉

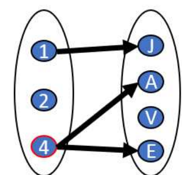
Úlohy [každá 1 bod]: V triede **Zobrazenia** definujte statické metódy:

- public static <K,V> boolean **jeInjektivne**(Map<K,V> z) - je test, ktorý platí, ak zobrazenie je injektívne, teda každý prvok odboru hodnôt je obrazom **najviac jedného prvku** z definičného oboru. Obrázok 2 ilustruje injektívne, aj to, čo nie injektívne.
- public static <K,V> boolean **jeSurjektivne**(Map<K,V> z) - každý prvok oboru hodnôt je obrazom **aspoň nejakého prvku** z definičného oboru. **Hint:** Na pochopenie, pozrite si, ako sme definovali obor hodnôt v druhom odstavci príkladu.
- public static <K,V,W> Map<K,W> **kompozicia**(Map<K,V> z1, Map<V,W> z2) - je zloženie zobrazenia z K do V so zobrazením z V do W, takže v tomto poradí vznikne zobrazenie z K do W.
- public static <K,V> Map<V,K> **inverzne**(Map<K,V> z) - ak je zobrazenie injektívne, existuje k nemu **inverzné zobrazenie**. To znamená, že definičný obor a obor hodnôt sa vymenia, a dvojice sa otočia. Obrázok 3 ilustruje inverziu. Dopisujte do kostry kódu v časti ... if (jeInjektivne(z)) return ... else return null;

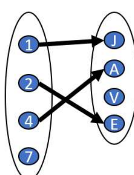
zobrazenie



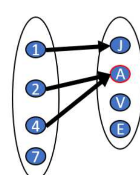
nie je zobrazenie



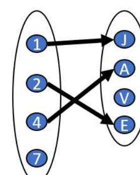
injekcia



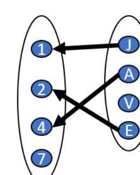
nie je injekcia



injekcia



k nej inverzná



```
Príklad:
var z1 = Map.of(1,2,  2,3,   3,4,  4,1);                   // 1→2,2→3,3→4,4→1
var z2 = Map.of(1,"a",  2,"b",   3,"c",  4,"d",  5,"a");  // 1→"a",2→"b",3→"c",4→"d",5→"a"
System.out.println(jeInjektivne(z1));    // true
System.out.println(jeInjektivne(z2)); // false
System.out.println(jeSurjektivne(z2)); // true
System.out.println(kompozicia(z1,z2)); // {1=b, 2=c, 3=d, 4=a}          // 1→b,2→c,3→d,4→a
System.out.println(inverzne(kompozicia(z1,z2))); // {a=4, b=1, c=2, d=3}// 1←b,2←c,3←d,4←a
System.out.println(inverzne(z1)); // {1=4, 2=1, 3=2, 4=3}               // 1←2,2←3,3←4,4←1
System.out.println(inverzne(z2)); // null
```

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO, NASLEDUJÚCI LIST

4. Zobrazenia – M4 – 4 body Meno a priezvisko: ___________________ 8-ID:

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST

5. Streamy – M5- 4 body Meno a priezvisko: ___________________ 9-ID:

Odporúčame použite StreamAPI, dostanete tak najkompaktnejšie riešenia. Pripomeňme si, že:

- Stream.**iterate**(x, f) = Stream.of(x, f(x), f(f(x)), ….) – iteruje do nekonečna. .**limit**(k) vráti len k-členov.
- Stream.**iterate**(x, p, f) – iteruje f na x, kým platí booleovská podmienka p. Posledným prvkom streamu je posledný člen, kedy ešte podmienka platila. Ak neplatí ani pre počiatočné x, výsledkom je prázdny stream.
- Stream.of(x_1, x_2, … x_n).**reduce**(x, f) = f(…f(f(x, x_1), x_2), …, x_n), napr. range(1,10).reduce(0,(a,b)->a-b)=-45
- LongFunction<Long> je to isté ako IntFunction<Integer>, len pre typ Long, teda Function<Long, Long> je funkcia z Long do Long.
- LongFunction<Long> fooLF a public static Long foo(Long n) {…} **nie je** to isté, ale ak to neviete inak, tak môžete napísať LongFunction<Long> fooLF=n->foo(n), alebo LongFunction<Long> fooLF=n->{…}.

Úlohy: Celý príklad uvažuje cifry a čísla v **desiatkovej sústave**.

- **[1bod]** definujte `static LongFunction<Long> cifSucet`, ktorá vráti **ciferný súčet** čísla v desiatkovej sústave. Príklad, cifsucet.apply(1234) = 1+2+3+4 =10.
- **[1bod]** definujte `static LongFunction<Long> cifRozdiel`, ktorá vráti rozdiel súčtov **nepárne a párne stojacich** cifier čísla. Príklad, cifRozdiel.apply(1234)=1-2+3-4=-2, cifRozdiel.apply(12345)=1-2+3-4+5=3, cifRozdiel.apply(654321)=6-5+4-3+2-1=3.
- **[1bod]** Definujte štyri následujúce `static IntPredicate`
  - ***moreDigits*** – číslo typu Integer nie je jednociferné,
  - ***div3, div9, div11*** – postupne znamenajú, že **jednociferné číslo** (teda 0..9) je deliteľné 3, 9, 11.
- **[1bod]** definujte `public static LongFunction<Long> iterate (LongFunction<Long> f)`, vráti hodnotu typu `LongFunction<Long>`, ktorá na vstupnom číselnom argumente iteruje funkciu **f** až kým výsledok nie je jednociferný (negácia ***moreDigits***). Keď raz dosiahne jednociferný výsledok, neplatí ***moreDigits***, tak ten je výsledkom funkcie.

Pointa: Zrejme viete, že číslo je deliteľné 3, práve vtedy, ak jeho ciferný súčet je deliteľný 3, podobne 9timi, ak jeho ciferný súčet je deliteľný 9timi. S 11 je to podobné, len sa nerobí ciferný súčet ale rozdiel. Číslo je deliteľné 11timi, práve vtedy ak rozdiel súčtov jeho nepárne a párne stojacich cifier (***cifRozdiel***) je deliteľný 11-timi. **Príklad**: Každé rodné číslo je deliteľné 11timi, takže napríklad pre `cifRozdiel.apply(9183979497L)` je `9-1+8-3+9-7+9-4+9-7=22.` Nedostali sme jednociferné číslo, preto opakujeme operáciu `cifRozdiel.apply(22)=0.` Teraz už sme dostali jednociferné číslo a môžeme použiť test ***div11*** a 0 je jednociferné číslo deliteľné 11, preto aj pôvodné bolo deliteľné 11.

Z už definovaných metód implementujte test na deliteľnosť 3,9,11 pomocou spomenutých kritérií deliteľnosti 3, 9, 11. Použijete pri tom funkcie ***cifSucet***, ***cifRozdiel***, ***iterate*** a predikáty ***moreDigits***, ***div3, div9, div11***. Pointa príkladu je, že sa budú veľmi na seba podobať, keďže princíp kritérií deliteľnosti 3,9,11 je takmer rovnaký, len sa opakujú a volajú iné funkcie a predikáty.

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST

5. Streamy – M5- 4 body Meno a priezvisko: ___________________ 10-ID:

RIEŠENIA TOHOTO PRÍKLADU PÍŠTE NA TENTO LIST