HATÉKONY JAVA PROGRAMOZÁS

6. forduló

MSCI

A kategória támogatója: MSCI

Ismertető a feladathoz

Felhasznált idő: 30:00/30:00 **Elért pontszám:** 0/10

1. feladat 0/2 pont

Ha a 47-et megfordítjuk, majd ezt a két számot összeadjuk (tehát 47 + 74 = 121), palindrom számot kapunk. De nem mindegyik szám esetén jutunk el a fordítás / összeadás műveletekkel ilyen gyorsan palindromhoz.

Pl. 349 esetén 3 iteráció kell:

349 + 943 = 1292

1292 + 2921 = 4213

4213 + 3124 = 7337

Bár bizonyítani még nem sikerült, azt sejtjük, hogy bizonyos számok esetén, mint a 196, sosem lesz az eredmény palindrom, ezeket a számokat Lychrel számoknak nevezzük.

Írjunk egy programot Lychrel szám tesztelésére adott szamú maximális iteracióval!

Amennyiben egy számot maximum 50 iterációval tesztelünk, **hány potenciális Lychrel számot találunk 10_000 alatt?**

Válaszok

A helyes válasz:

249

246

Magyarázat

A megoldás működik naív algoritmussal is, érdemes **BigInteger**-t használni, mert a **Long**-ból kifogynánk.

pl: https://rosettacode.org/wiki/Lychrel_numbers#Java

A leíras nem specifikálta külön, hogy egy önmagában palindrom számot elkezdünk-e iterálni vagy sem, 10ezer alatt pedig 3 Lychrel-palindrom létezik, ezért a 246-ot is elfogadjuk megoldásként.

2. feladat 0/2 pont

Amennyiben egy számot maximum 50 iterációval tesztelünk, **melyik lesz a 200. potenciálisan Lychrel szám?**

Válaszok

A helyes válasz:

8958

8979

Magyarázat

-

3. feladat 0/0 pont

A programunkban párhuzamosan szeretnénk feladatokat feldolgozni, és egy **ExecutorService**-t példányosítunk:

```
ExecutorService executorService =
     new ThreadPoolExecutor(5, 10, 10000L, TimeUnit.MILLISECONDS,
     new LinkedBlockingQueue<Runnable>());
executorService.allowCoreThreadTimeout(true);
```

Mely állítások helyesek az alábbiak közül?

Válasz

- Ha az **executorService**-en sosem hívjuk meg a **shutdown** vagy a **shutDownNow** metódust, akkor az alkalmazás sosem tud leállni, akkor sem ha már nincs feldolgozandó feladat, mert a ThreadPoolExecutor szálai nem daemon-szálak.
- Az **executorService** sosem fog 10 szálon párhuzamosan dolgozni, akkor sem ha van 10 beadott feladat.
  Ez a válasz helyes, de nem jelölted meg.
- Ha az **executorService** olyan sok párhuzamos feladatot kap, hogy minden beállított szál foglalt lesz és a LinkedBlockingQueue is betelik, akkor **RejectedException** dobódik.
- Ez az **executorService** sosem dob **RejectedException**-t, de a párhuzamos szálak száma a végteleségig nőhet. (Amíg elfogynak az erőforrások.)
- Ez az **executorService** sosem dob **RejectedException**-t, de a **BlockingQueue** mérete a végteleségig nőhet. (Amíg elfogynak az erőforrások.)

Magyarázat

Kedves Versenyzők!

A feladatot 0 pontosra állítottuk (2022.11.28.), mivel egy elírás van a kiírásban: RejectedExecutionException lenne a helyes.

Köszönjük szépen megértéseteket!

Ha az **ExecutorService**-t **LinkedBlockingQueue**-val példányosítjuk, akkor a szálak száma sosem nő a **corePoolSize** fölé, akkor sem ha van egy nagyobb **maximumPoolSize** megadva. Ilyenkor ha több feladat érkezik a service-hez mint a szálak száma, akkor azok a kapacitáskorlát nélküli queue-ba kerülnek.

Ilyen blockingqueue esetén a threadpool sosem dob kivételt kapacitáskorlát miatt, de ha meghívjuk a shutdown-t, és újabb feladatot akarunk hozzáadni, akkor az vissza lesz utasítva egy kivétellel.

Mivel a core szálakra is érvényes a keepalive, ha nincs újabb feladat, a threadpool minden szálat leállít egy idő után, és akkor is leállhat az alkalmazás, ha nem volt meghívva a shutdown.

4. feladat 0/2 pont

Adott a következő osztály.

```
public class Portfolio {

 private Set<Stock> stocks;
 private long identifier;
 private String ticker;

}
```

Tegyük fel, hogy van helyes és hatékony **equals** és **hashcode** implementációja és rendszerünk 100_000_000 portfóliót tárol. A portfóliót nap elején állítjuk össze, napközben a követett portfóliók nem változnak, csak a részvények ára változik.

Kereskedés közben bármikor lekérhetjük egy portfólió aktuális értékét azonosító (**long identifier**) alapján, illetve kereskedés előtt és tőzsdezárás után kiszámoljuk és **ticker** alapján rendezve továbbítjuk a napi nyitó és záró értékét.

Milyen adatstruktúrában kellene tárolni a portfóliókat az alábbiak közül ,hogy a feldolgozás időkomplexitása a lehető legalacsonyabb legyen?

Válasz

- a)

  LinkedList<Portfolio> felépítés után rendezve ticker szerint és HashMap<Long, Portfolio>.

  Az iterálás ticker alapján a lehető leggyorsabb a linkedlist miatt, a keresés HashMap esetén konstants időben megoldható, ha a hashcode implementáció hatékony.

- b)

  TreeMap<Long, Portfolio> egy Comparator<Portfolio> komparátorral példányosítva, ahol a komparátor a ticker alapján rendez, a kulcs viszont az azonosító.

  A véletlenszerű lekérés long kulcs szerint konstans időben megoldható, de az iterálás is hatékony, mert a fa a ticker alapján van rendezve. Véletlenszerű lekérés ticker alapján nem lenne hatékony.

- c)

  TreeSet<Portfolio> ticker alapján rendezve, és LinkedHashMap<Long, Portfolio>, ahol a map indentifier/azonosító kulcs alapján tárolja portfóliókat és konstants idő alatt keres, a rendezett set a ticker alapján iterálást valósítja meg hatékonyan.

- d)

  ArrayList<Portfolio> felépítés után rendezve ticker szerint és HashMap<Long, Portfolio>.

  Az iterálás ticker alapján a lehető leggyorsabb az arraylist miatt, a keresés hashmapben hatékony, ha a hashcode hatékonyan van megírva.
  Ez a válasz helyes, de nem jelölted meg.

Magyarázat

Rendezés ArrayList-ben valamivel gyorsabb mint LinkedList-ben mert in-place rendezi a backing arrayt-t, míg a LinkedList esetén átmásolja egy újba.

A TreeMap comparatora nem rendezhet érték szerint, csak kulcs szerint.

A LinkedHashMap fölöslegesen tartja fent a LinkedList-jét, mert ezen a gyűjteményen sosem iterálunk.

5. feladat 0/4 pont

Az alábbi program futása után (HotSpot JVM Java 11, semmilyen opciót nem adunk át a JVM-nek futtatáskor) azt látjuk, hogy a rendezett és rendezetlen tömbök között jelentősen más a futási idő. Mivel magyarázható ez?

```
public class SumClass {

  public static void main(String[] args) {
    int data[] = new int[32768];
    Random rnd = new Random(0);

    for (int i = 0; i < data.length; ++i)
      data[i] = rnd.nextInt() % 256;

    foo(data);
    Arrays.sort(data);
    foo(data);
  }

  private static void foo(int[] data) {
    long start = System.nanoTime();
    long sum = 0;

    for (int j = 0; j < 100000; ++j) {
      for (int i = 0; i < data.length; ++i) {
        if (data[i] >= 128) sum += data[i];
      }
    }

    System.out.println("Sum: " + sum);
    System.out.println("Time: " + (System.nanoTime() - start) / 1000000000.0);
  }
}
```

Console out:

Sum: 155184200000

Time: 6.1833708

Sum: 155184200000

Time: 1.0047831

Válasz

- ○ Első futás alatt történt egy full GC. Igazából a futás majdnem ugyanannyi lenne, ha nincs garbage collection. Ha elég nagy heap-pel indítjuk, nem lesz különbség.
- ○ A JIT compiler optimalizálta a programot (loop unrolling), amire a második tesztet végeztük. Mivel itt már nem voltak assembly ugrások, sokkal gyorsabban futott.

Nagyobb hatékonysággal működik a processzor elágazásbecslése (branch prediction)
Ez a válasz helyes, de nem jelölted meg.

A Java Runtime megjegyezte a metódus bemenetét, mivel ugyanazzal a tömbbel hívtuk meg, nem is futott le a for ciklus. Az 1 másodperc a tömbök összehasonlításával telt.

Magyarázat

Nagy heap-pel is látni fogjuk a különbséget a futási időben, tehát nem a GC okozza a különbséget és nincs ebben az esetben olyan loop unrolling ami ekkora különbséget eredményezne.

Olyan optimalizáció ami megjegyzi a bemenetet, hogy később ne hívja meg újra a metódust, nem létezik.

A megoldás a branch prediction.