ZV-01

Java nyelven, írjon olyan függvényt vagy metódust, amely egy természetes számról eldönti, hogy
prímszám-e, vagy sem!

```java
public static boolean isPrime(int num) {
    if (num <= 1) {
        return false;
    }
    for (int i = 2; i <= Math.sqrt(num); i++) {
        if (num % i == 0) {
            return false;
        }
    }
    return true;
}
```

Ez a függvény egy természetes számot vár paraméterként, és eldönti, hogy prímszám-e, vagy sem. A függvény visszatérési értéke igaz, ha a szám prím, és hamis, ha nem. A függvény első lépéseként ellenőrzi, hogy a szám 1-nél kisebb-e, mivel 1-nél kisebb számok nem prímszámok. Ha a szám 1-nél nagyobb, akkor egy ciklust indít 2 és a szám négyzetgyöke között. Minden ciklusiterációban ellenőrzi, hogy a szám maradék nélküli osztója-e az aktuális ciklusváltozónak. Ha igen, akkor a szám nem prím, és a függvény hamis értékkel tér vissza. Ha a ciklus végére érve a függvény még mindig fut, akkor a szám prím, és igaz értékkel tér vissza.

Miért használja a Math.sqrt() függvényt?

A Math.sqrt() függvény a Java standard könyvtár része, és a paraméterként megadott szám négyzetgyökét adja vissza.

A függvény használata a fenti kódban azért jön jól, mert csak a szám négyzetgyökeig kell ellenőrizni, hogy az osztója-e az adott számnak.
Ha a szám négyzetgyöke után keresnénk az osztót, akkor már ismernénk az osztót előtte, a szám négyzetgyöke előtt.

Ez az algoritmus csökkenti az összes szám ellenőrzésére fordított időt.

Például:
Ha num=16, akkor a Math.sqrt(num) = 4
Ez azt jelenti, hogy 4-nél nagyobb osztók már nem lehetnek, mert az osztók párosak.
Mivel a szám 16, akkor az osztók 2,4,8,16.
Ezért az ellenőrzést 2-től 4-ig kell végezni, ez 4 db számot jelent, ami jóval kevesebb mint 16.

ZV-02

Írjon olyan függvényt vagy metódust, amely egy természetes számról eldönti, hogy tökéletes
szám-e, vagy sem! (pozitív osztóinak összege a szám kétszerese)

```
public static boolean isPerfectNumber(int num) {
    int sum = 0;
    for (int i = 1; i <= num/2; i++) {
        if (num % i == 0) {
            sum += i;
        }
    }
    return sum == num*2;
}
```

Ez a függvény egy természetes számot (int) vár paraméterül, és eldönti, hogy tökéletes-e a szám. A függvény belső logikája azt jelenti, hogy a szám osztóinak összegét számolja, és ha ez megegyezik a szám kétszeresével, akkor tökéletes számról van szó, és true-val tér vissza, különben false-val.

Az osztók keresése első feléig történik, mert a szám osztói párokban jelennek meg, és a nagyobbik osztó mindig a kisebbik osztóval megegyező számot ad.

ZV-03

Java: Írjon olyan függvényt vagy metódust, amely egy karakterláncban vagy sztringben
véletlenszerűen összekeveri a karaktereket (véletlenszám–generátor használható)!

```java
import java.util.Random;

public class StringShuffle {
    public static String shuffleString(String input) {
        char[] charArray = input.toCharArray();
        Random rand = new Random();
        for (int i = 0; i < charArray.length; i++) {
            int randomIndex = rand.nextInt(charArray.length);
            char temp = charArray[i];
            charArray[i] = charArray[randomIndex];
            charArray[randomIndex] = temp;
        }
        return new String(charArray);
    }
}
```

A fenti kód egy olyan `shuffleString` metódust definiál, amely egy bemeneti sztringet vesz és véletlenszerűen összekeveri a karaktereit. A metódus a sztringet karaktertömbbe helyezi át, majd egy `Random` objektumot használ a karakterek véletlenszerű összekeveréséhez. Végül a kevert karaktertömböt új sztringgé alakítja és visszaadja.

Optimalizáld!

```
import java.util.Random;

public class StringShuffle {
    public static String shuffleString(String input) {
        Random rand = new Random();
        char[] charArray = input.toCharArray();
        for (int i = charArray.length - 1; i > 0; i--) {
            int randomIndex = rand.nextInt(i + 1);
            char temp = charArray[i];
            charArray[i] = charArray[randomIndex];
            charArray[randomIndex] = temp;
        }
        return new String(charArray);
    }
}
```

Az előző megoldásban a kód a karakterek teljes tömbjén végigiterált és minden karakterre egy véletlenszerű indexet választott. Ennek az volt az eredménye, hogy a kód általában többször választott ki ugyanazt az indexet és ezáltal a karaktereket is. Az optimalizált megoldásban a kód az utolsó karaktertől kezdve végigiterál, és minden karakterre egy véletlenszerű indexet választ, amely kisebb mint a jelenlegi index. Így a kód a korábban kiválasztott indexeket nem veszi figyelembe és ezáltal a karakterek összekeverése is gyorsabb lesz.

ZV-04

A következő közelítő formulát használva írjon függvényt vagy metódust, amely egy valós szám négyzetgyökét adja vissza! Használja az $x_{k+1}=1/2*(x_k+a/x_k)$ sorozatot, amely a négyzetgyökéhez konvergál, ha $x_1=1$.

```
public static double squareRoot(double a) {
    double x = 1;
    for (int i = 0; i < 10; i++) {
        x = 0.5 * (x + a / x);
    }
    return x;
}
```

A fenti kódban a `for` ciklus fut 10 alkalommal, és minden futáskor frissíti az x értékét az $x_{k+1} = 1/2 * (x_k + a/x_k)$ képlet alapján. A ciklus futtatása 10 alkalommal elegendő a négyzetgyök kiszámításához, de ha nagyobb pontosságot szeretne, akkor növelnie kell a ciklus futtatásainak számát.

Miért elég 10 futás?

A fenti kódban a `for` ciklus fut 10 alkalommal a négyzetgyök kiszámításához, de ez csak egy példa. Az $x_{k+1}=1/2*(x_k+a/x_k)$ sorozat konvergenciája gyors, ami azt jelenti, hogy az x_k értékei hamar közelíteni fogják a valódi négyzetgyök értékét. Ha kevesebb futást használunk, akkor az x_k értékei valószínűleg nem lesznek elég pontosak, de ha több futást használunk, akkor az x_k értékei valószínűleg túl pontosak lesznek. 10 futás az egyensúlyt jelenti a pontosság és a teljesítmény között.

Azonban, el kell mondani, hogy ez a közelítő módszer nem minden esetben ad pontos eredményt, és a négyzetgyökök kiszámításánál jobb alternatívák is léteznek, pl. a `Math.sqrt()` függvény használata javascriptban vagy a `Math.sqrt()` metódus használata Java esetén.

ZV-05

Java nyelven!
Írjon függvényt vagy metódust, amely egy valós szám köbgyökét adja vissza! Használja az $x_{k+1}=1/3*(2*x_k+a/x_k^2)$ sorozatot, amely a köbgyökéhez konvergál, ha $x_1=1$.

```java
public static double cubeRoot(double a) {
    double x = 1;
    for (int i = 0; i < 100; i++) {
        x = (2 * x + a / (x * x)) / 3;
    }
    return x;
}
```

Ez a függvény egy valós szám köbgyökét adja vissza az $x_{k+1}=1/3*(2*x_k+a/x_k^2)$ sorozatot használva. A ciklus 100 iterációt végez, de a számítások számát a szükséges pontosságtól függően lehet módosítani.

Magyarázd el!

A fenti kód egy Java függvényt definiál, amelynek neve "cubeRoot" és egy valós számot vesz fel paraméterként. A függvény visszatérési értéke egy valós szám, ami a paraméterként kapott szám köbgyöke.

A függvény belsejében egy "for" ciklust használunk, amely 100 iterációt végez. Az iterációk során a ciklusváltozó (i) értékét nem használjuk, csupán a ciklus számát szabályozza. A ciklus belsejében a köbgyök számítását végző módszert használjuk, az $x_{k+1}=1/3*(2*x_k+a/x_k^2)$ sorozatot. Az x_{k+1} következő x értékét jelenti, x_k pedig az aktuális x értékét jelenti. Az a pedig a paraméterként kapott számot jelenti. A ciklus végén az x változót adja vissza a függvény, ami a köbgyök szám.

Fontos megemlíteni, hogy a sorozat konvergálni fog a köbgyökhöz, ha $x_1=1$.

ZV-06

Java nyelven!
Írjon függvényt vagy metódust, amely kiszámolja az n-edik Fibonacci számot!
A Fibonacci
sorozatot az $a_n=a_{n-2}+a_{n-1}$ rekurzióval definiálja (n>2), ahol $a_1=a_2=1$.

```java
public static int fibonacci(int n) {
    if (n <= 2) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

1	1	2	3	5	8	13	21

Ez a függvény rekurzívan meghívja magát, hogy kiszámolja az n-edik Fibonacci számot. Az első két eleme mindig 1, így ha n <= 2, akkor 1-et ad vissza. Ha n > 2, akkor a függvény a (n-1) és (n-2) Fibonacci számokat kérdezi le, és összeadja őket.

Egy másik megoldás, ami hatékonyabb, ha nagyszámú Fibonacci számokat számolunk, az iteratív megoldás:

```java
public static int fibonacci(int n) {
    int a = 0, b = 1, c;
    if (n == 0) return a;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

ZV-07

Írjon olyan függvényt vagy metódust, amely egy természetes számhoz visszaadja azt a legnagyobb egész kitevős hatványát, amely még éppen kisebb, mint 567!

```
public static int findMaxPower(int number) {
    int result = 0;
    while (Math.pow(number, result) < 567) {
        result++;
    }
    return result - 1;
}
```

Ez a függvény egy természetes számot vár paraméterül, és visszaadja azt a legnagyobb egész kitevős hatványát, amely még éppen kisebb, mint 567!. A függvény egy ciklust használ, amely addig megy, amíg a szám hatványa kisebb, mint 567. A ciklus minden iterációjának végén a kitevő értékét növeli 1-el. A ciklus után visszaadja a kitevőt 1-el kivonva, hogy az legyen a legnagyobb kitevő ami kisebb mint 567!.

Példákat is mutass!

```
public static void main(String[] args) {
    System.out.println(findMaxPower(2));   // 8
    System.out.println(findMaxPower(3));   // 4
    System.out.println(findMaxPower(5));   // 3
    System.out.println(findMaxPower(10));  // 2
    System.out.println(findMaxPower(20));  // 1
}
```

ZV-07

Ez a függvény egy természetes számot vár paraméterül, és visszaadja azt a legnagyobb egész kitevős hatványát, amely még éppen kisebb, mint 567!. A függvény egy ciklust használ, amely addig megy, amíg a szám hatványa kisebb, mint 567. A ciklus minden iterációjának végén a kitevő értékét növeli 1-el. A ciklus után visszaadja a kitevőt 1-el kivonva, hogy az legyen a legnagyobb kitevő ami kisebb mint 567!.

Példákat is mutass!

```
public static void main(String[] args) {
    System.out.println(findMaxPower(2));   // 8
    System.out.println(findMaxPower(3));   // 4
    System.out.println(findMaxPower(5));   // 3
    System.out.println(findMaxPower(10));  // 2
    System.out.println(findMaxPower(20));  // 1
}
```

A fenti példában a `main` függvény több példát ad a `findMaxPower` függvénnyel, amelyek a következők:

- 2 esetén a legnagyobb kitevő 8, mert 2^8 = 256 < 567
- 3 esetén a legnagyobb kitevő 4, mert 3^4 = 81 < 567
- 5 esetén a legnagyobb kitevő 3, mert 5^3 = 125 < 567
- 10 esetén a legnagyobb kitevő 2, mert 10^2 = 100 < 567
- 20 esetén a legnagyobb kitevő 1, mert 20^1 = 20 < 567

Az eredmények azt mutatják, hogy a függvény megfelelően működik és a várt kitevőt adja vissza.

ZV-08

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy természetes szám esetén kiírja, hogy a 9-es számjegyből hány darabot tartalmaz!

```java
public static int countNine(int number) {
    int count = 0;
    while (number > 0) {
        if (number % 10 == 9) {
            count++;
        }
        number /= 10;
    }
    return count;
}
```

Ez a függvény egy természetes számot vár paraméterként, és visszaadja, hogy hány darab 9-es számjegy van benne. A függvény while ciklussal végigmegy a számjegyeken, és ha talál egy 9-est, akkor növeli a számlálót.

ZV-09

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy természetes számról eldönti, hogy a kettes számrendszerbeli felírásában a jobbról második bitje 1 vagy 0!

```java
public static boolean isSecondBitOne(int num) {
    // Shift the number two bits to the right, then & with 1
    return (num >> 1 & 1) == 1;
}
```

A fenti kód egy metódust definiál, amely egy `int` típusú számot vár paraméterként, és visszaadja, hogy a kettes számrendszerbeli felírásában a jobbról második bitje 1 vagy 0. A `>>` operátorral a számot két bitet jobbra shifteljük, majd az `&` operátorral megvizsgáljuk, hogy az eredmény 1-e, ami azt jelenti, hogy a második bit 1. Ha az eredmény igaz, akkor a metódus `true`-t ad vissza, különben `false`.

Magyarázd el részletesebben!

A fenti kód egy `isSecondBitOne` nevű metódust definiál, amely egy `int` típusú számot vár paraméterként. A metódus visszaadja, hogy a kettes számrendszerbeli felírásában a jobbról második bitje 1 vagy 0.

Az első lépés, hogy a számot két bitet jobbra shifteljük a `>>` operátorral. Ennek hatására a jobbról második bitet az első bit helyére hozzuk, így az első bit a második helyen lesz.

A következő lépésben az `&` operátorral az első bitet kapjuk meg az első helyről, és megvizsgáljuk, hogy az 1-e. Ha az 1-e, akkor az eredmény 1 lesz, különben 0.

ZV-10

Írjon olyan függvényt vagy metódust, amelynek paramétere egy 1 < x < 10 természetes szám, és kiírja az 1,3,4,6,7,9,10,12,... sorozat első öt x-szel osztható elemét, azaz a sorozat i+1-edik tagja 2-vel nagyobb az i-ediknél, ha i páratlan, s eggyel nagyobb az i-ediknél, ha i páros!

```
public void printFirstFiveXDivisible(int x) {
    int i = 1;
    int count = 0;
    while (count < 5) {
        if (i % x == 0) {
            System.out.println(i);
            count++;
        }
        i += (i % 2 == 0) ? 1 : 2;
    }
}
```

ZV-10

Írjon olyan függvényt vagy metódust, amelynek paramétere egy 1 < x < 10 természetes szám, és kiírja az 1,3,4,6,7,9,10,12,... sorozat első öt x-szel osztható elemét, azaz a sorozat i+1-edik tagja 2-vel nagyobb az i-ediknél, ha i páratlan, s eggyel nagyobb az i-ediknél, ha i páros!

```
public static void printFirstFive(int x) {
    int counter = 0;

    while (counter < 5) {
        if (i % x == 0) {
            System.out.println(i);
            counter++;
        }
        if (i % 2 == 0) {
            i++;
        } else {
            i += 2;
        }
    }
}
```

A fenti kód egy `printFirstFive` nevű metódust definiál, amelynek egy `x` nevű paramétere van. A metódus a megadott feltételek szerint végigmegy az 1,3,4,6,7,9,10,12,... sorozaton, és kiírja az első öt x-szel osztható elemet. A számláló (counter) segítségével nyomon követi, hogy hány elemet írt ki a metódus, és amikor eléri a 5-öt, kilép a ciklusból.

ZV-11

Írjon olyan függvényt vagy metódust, amely a paraméterében megadott természetes szám
pozitív osztóinak számával tér vissza!

Itt van egy Java metódus, amely visszaadja a paraméterében megadott természetes szám pozitív osztóinak számát:

arduino

```
public static int getDivisorCount(int num) {
    int count = 0;
    for (int i = 1; i <= num; i++) {
        if (num % i == 0) {
            count++;
        }
    }
    return count;
}
```

ZV-12 String

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy karakterláncból vagy sztringből a számjegyek kivételével minden karaktert eltávolít! StringBuilder és isDigit nélkül!

Az alábbiakban egy egyszerű megoldást mutatok be, amely egy karakterláncból eltávolítja a számjegyeket és minden más karaktert megtartja. Ez a megoldás csak a karakterek ASCII-kódjait használja ki, ezért csak a 0-9 számjegyeket távolítja el helyesen.

```
public static String removeDigits(String str) {
    String result = "";
    for (int i = 0; i < str.length(); i++) {
        char c = str.charAt(i);
        if (c < '0' || c > '9') {
            result += c;
        }
    }
    return result;
}
```

A függvény a `str` karakterláncot kapja paraméterként, majd egy üres sztringgel (`result`) kezdi. A `for` ciklus végigiterál a karakterlánc minden karakterén, és az aktuális karaktert egy `char` típusú változóba (`c`) menti. Ha a karakter ASCII-kódja nem esik a '0' és a '9' közé, akkor hozzáadja a `result` sztringhez.

Végül a függvény visszatér az eredménysztringgel, amely csak a számjegyeket tartalmazó karaktereket hagyja el. Fontos megjegyezni, hogy a függvény nem javasolt hosszú karakterláncokra, mivel a `String` típusú változók konkatenációja hatékonytalan lehet nagy sztringeknél. Az ilyen esetekben a `StringBuilder` osztály használata javasolt.

ZV-12 StringBuilder

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy karakterláncból vagy sztringből a számjegyek kivételével minden karaktert eltávolít! isDigit nélkül!

Természetesen, itt van egy egyszerű megoldás, amely eltávolítja a számjegyeket a bemeneti karakterláncból, anélkül, hogy a `isDigit()` metódust használná:

```
public static String removeDigits(String input) {
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < input.length(); i++) {
        char c = input.charAt(i);
        if (c < '0' || c > '9') {
            result.append(c);
        }
    }
    return result.toString();
}
```

Ez a metódus egy `StringBuilder`-t használ annak érdekében, hogy az eredményt hatékonyan összeállítsa a karakterlánc végrehajtása során. A `for` ciklus végigiterál a bemeneti karakterlánc minden karakterén, és csak akkor adja hozzá az eredmény karakterláncához, ha az adott karakter nem számjegy. A feltétel az `if` állításban található, ahol az `input.charAt(i)` metódus adja vissza az aktuális karaktert, amelyet a `c` változóban tárolunk.

Ezután ellenőrizzük, hogy a karakter egy számjegy-e azáltal, hogy az ASCII értékeit hasonlítjuk össze a `0` és `9` karakterekével. Ha az aktuális karakter nem számjegy, akkor az eredmény karakterláncához hozzáadjuk a karaktert. Végül a metódus visszaadja az eredményként kapott karakterláncot, amely csak a számjegyeket tartalmazó karaktereket tartalmazza.

ZV-13

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy karakterláncról vagy sztringről eldönti, hogy palindróma-e! (Balról olvasva ugyanaz, mint jobbról olvasva.) For ciklussal!

Természetesen, itt van egy Java nyelvű függvény, amely egy karakterláncról eldönti, hogy palindróma-e, és for ciklust használ hozzá:

```
public static boolean isPalindrome(String str) {
    int len = str.length();
    for (int i = 0; i < len / 2; i++) {
        if (str.charAt(i) != str.charAt(len - i - 1)) {
            return false;
        }
    }
    return true;
}
```

Ez a függvény elkezdi a karakterlánc bejárását az első karaktertől a közepéig (azaz a hossz feléig), és ellenőrzi, hogy a megfelelő szemközti karakterek megegyeznek-e. Ha talál egy eltérést, azonnal visszatér hamissal, különben a függvény visszatér igazzal, ha a karakterlánc egy palindróma.

ZV-14

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy, az angol ábécé betűit tartalmazó karakterláncban vagy sztringben minden szó kezdőbetűjét nagybetűre alakítja!

```java
public static String capitalizeWords(String input) {
    String[] words = input.split("\\s");
    String result = "";
    for (String word : words) {
        result += Character.toUpperCase(word.charAt(0)) + word.substring(1) + " ";
    }
    return result.trim();
}
```

Magyarázd el!

Ez a függvény egy egyszerű megoldást kínál arra, hogy minden szó kezdőbetűjét nagybetűre alakítsa egy adott karakterláncban vagy sztringben.

A függvény először szétválasztja a bemeneti karakterláncot szavakra, ahol a szóhatárok szóközök. Ehhez a String.split() metódust használjuk, amely egy reguláris kifejezést vár paraméterként, amely szerint a karakterláncot fel kell bontani.

Ezután egy ciklusban végigmegyünk az összes szón, és a Character.toUpperCase() metódust használjuk az adott szó első karakterének nagybetűvé alakítására. A substring() metódussal az eredeti szó második karakterétől kezdve a végéig hozzáadjuk a szót az eredmény karakterláncunkhoz, majd egy szóközt is hozzáadunk az új szók közötti szóközök helyreállítása érdekében.

Végül a trim() metódust használjuk a karakterláncunk levágására az elejéről és végéről, hogy eltávolítsuk az utolsó szó utáni felesleges szóközt, majd visszaadjuk az eredmény karakterláncot. Az eredeti karakterláncot nem módosítjuk, mivel a String immutábilis (nem módosítható) adattípus, így mindig új String objektumot hozunk létre, ha módosítunk rajta.

ZV-15

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy karakterláncból vagy sztringből eltávolítja egy megadott karakter összes előfordulását!

Természetesen, itt van egy példa Java metódusra, amely eltávolítja egy adott karakter összes előfordulását egy sztringből:

```
public static String removeCharFromString(String str, char c) {
    // Létrehozunk egy StringBuilder objektumot, amely a sztring eredményét fogja tárolni.
    StringBuilder sb = new StringBuilder();

    // Végigmegyünk a sztring összes karakterén.
    for (int i = 0; i < str.length(); i++) {
        // Ellenőrizzük, hogy az aktuális karakter az, amelyet eltávolítani akarunk.
        if (str.charAt(i) != c) {
            // Ha az aktuális karakter nem az, amit eltávolítani akarunk, hozzáadjuk a StringBuilde
            sb.append(str.charAt(i));
        }
    }

    // Visszatérünk az eredmény sztringgel.
    return sb.toString();
}
```

ZV-16

Java nyelven!
Írjon olyan függvényt vagy metódust, amely megszámolja egy adott karakterlánc vagy sztring összes előfordulását egy másik karakterláncban vagy sztringben!

```java
public static int countSubstring(String str, String sub) {
    int count = 0;
    int index = 0;
    while ((index = str.indexOf(sub, index)) != -1) {
        count++;
        index += sub.length();
    }
    return count;
}
```

A fenti függvény `countSubstring` néven van elnevezve és két paramétert fogad el: `str` és `sub`. Az `str` paraméter egy sztring, amelyben keressük a `sub` paraméterben megadott karakterlánc előfordulásait.

A függvény visszatérési értéke egy egész szám, amely azt jelzi, hogy a `sub` karakterlánc hányszor fordul elő az `str` karakterláncban.

A függvény a `count` változót 0-val inicializálja, amely számolja a `sub` előfordulásainak számát az `str` karakterláncban. Az `index` változó szintén 0-val indul, amely megadja azt az indexet, ahol a `sub` először előfordul az `str` karakterláncban.

A függvény egy `while` ciklust használ, amely addig fut, amíg a `indexOf` függvénnyel keresett `sub` karakterlánc nem egyenlő a `-1`-el (ami jelenti, hogy a keresett karakterlánc nem található meg az `str` karakterláncban). Minden ciklusban megnöveljük a `count` változó értékét 1-el és növeljük az `index` változó értékét a `sub` karakterlánc hosszával, hogy a következő előfordulás keresésekor ne ismételje meg a már ellenőrzött részeket.

A ciklus végén a függvény visszaadja a `count` változó értékét, amely azt jelzi, hogy a `sub` karakterlánc hányszor fordul elő az `str` karakterláncban.

ZV-17

Java nyelven!
Írjon olyan függvényt vagy metódust, amely kiírja az angol kisbetűs ábécé azon betűit, melyek ASCII kódja négyzetszám!

```java
public static void kiirNegyzetASCII() {
    for (int i = 'a'; i <= 'z'; i++) {
        if (Math.sqrt(i) % 1 == 0) {
            System.out.println((char) i);
        }
    }
}
```

ZV-18

Java nyelven!
Írjon olyan függvényt vagy metódust, amely előállít egy 5 karakterből (angol kisbetűs ábécé karaktereit használva) álló véletlen karakterláncot vagy sztringet! Biztosítsa, hogy minden 5 hosszú különböző betűkből álló sztring egyenlő valószínűséggel kerüljön kiválasztásra, feltéve, hogy a választott programozási nyelv véletlenszám-generátora egyenletes eloszlást biztosít!

```java
import java.util.Random;

public class RandomString {
    public static String generateRandomString() {
        String alphabet = "abcdefghijklmnopqrstuvwxyz";
        Random random = new Random();
        char[] result = new char[5];
        for (int i = 0; i < 5; i++) {
            result[i] = alphabet.charAt(random.nextInt(alphabet.length()));
        }
        return new String(result);
    }
}
```

ZV-19

Java nyelven!
Írjon olyan függvényt vagy metódust, amely egy karakterláncba vagy sztringbe beszúr egy „a” karaktert véletlenül választott pozícióba (véletlenszám–generátor használható)!

```java
import java.util.Random;

public static String insertRandomA(String str) {
  Random random = new Random();
  int index = random.nextInt(str.length() + 1);
  return str.substring(0, index) + "a" + str.substring(index);
}
```

substring(int beginIndex): Visszaad egy olyan karakterláncot, amely ennek a karakterláncnak a részsorozata. A részlánc a megadott indexen lévő karakterrel kezdődik, és a karakterlánc végéig tart.

substring(int beginIndex, int endIndex): Visszaad egy olyan karakterláncot, amely ennek a karakterláncnak a részsorozata. A részlánc a megadott beginIndexnél kezdődik, és az endIndex - 1 indexnél lévő karakterig tart, így a részlánc hossza endIndex-beginIndex.

ZV-20

Java nyelven!
Adjon olyan függvényt vagy metódust, ami adott két pozitív egész paramétere esetén megadja (n alatt a k)=n!/k!(n-k)! értékét. Használjon rekurziót!

Ez a függvénnyel elkészíthető Java nyelven:

```java
public static int combination(int n, int k) {
    if (k == 0 || k == n) {
        return 1;
    }
    return combination(n-1, k-1) + combination(n-1, k);
}
```

Magyarázd el!

Ez a függvény a binomiális együtthatók értékét számítja ki. A binomiális együttható a kombinatorikai matematikában a `n` darabból `k` elemet kiválasztani képes lehetséges kombinációk számát jelenti.

A függvény rekurzívan működik. Az alap feltétel akkor teljesül, ha `k` vagy `n-k` 0-ra egyenlő, ilyenkor 1-et ad vissza. Egyébként a függvény magát hívja meg `n-1` és `k-1` vagy `n-1` és `k` paraméterekkel, amíg el nem éri az alap feltételt.

A rekurzív hívások eredményét összeadva kapjuk meg a kívánt binomiális együttható értékét.

Példán keresztül mutasd meg!

Például, ha számolni szeretnénk `n=5` és `k=2` esetén a binomiális együtthatót, akkor a függvény kombinatorikai matematikában a `n` darabból `k` elemet kiválasztani képes lehetséges kombinációk számát jelenti.

A függvény rekurzívan működik. Az alap feltétel akkor teljesül, ha `k` vagy `n-k` 0-ra egyenlő, ilyenkor 1-et ad vissza. Egyébként a függvény magát hívja meg `n-1` és `k-1` vagy `n-1` és `k` paraméterekkel, amíg el nem éri az alap feltételt.

A rekurzív hívások eredményét összeadva kapjuk meg a kívánt binomiális együttható értékét.

Példán keresztül mutasd meg!

Például, ha számolni szeretnénk `n=5` és `k=2` esetén a binomiális együtthatót, akkor a függvény így fog működni:

```
combination(5, 2)
= combination(4, 1) + combination(4, 2)
  combination(4, 1)
  = combination(3, 0) + combination(3, 1)
    combination(3, 0) = 1
  combination(4, 2)
  = combination(3, 1) + combination(3, 2)
    combination(3, 1)
    = combination(2, 0) + combination(2, 1)
      combination(2, 0) = 1
    combination(3, 2)
    = combination(2, 1) + combination(2, 2)
      combination(2, 1) = 1
      combination(2, 2) = 1
= 1 + (1 + 1) + (1 + 1) = 5
```

Tehát a `combination(5, 2)` értéke 5.

ZV-21

Adjon olyan metódust vagy függvényt, ami eldönti, hogy a paramétereként megadott (pozitív egészekből álló) nemüres tömbben van-e olyan szám, ami az összes többit osztja. (Maradékszámító függvény használható).

```
public static boolean vanEMindentOszto(int[] szamok) {
    int db;
    for (int i = 0; i < szamok.length; i++) {
        db = 0;
        for (int j = 0; j < szamok.length; j++) {
            if (szamok[j] % szamok[i] == 0) db++;
            if (db == szamok.length) return true;
        }
    }
    return false;
}
```