Feladatmegoldás Pythonban és Javában

Különböző programozási nyelveket illetően felmerül a kérdés, hogy adott esetekben milyen eszközökkel és mennyire hatékonyan oldanak meg egyes problémákat. A *Python* és a *Java* nyelvekre vonatkozóan két magas szintű objektumorientált programozási nyelvről beszélünk, azonban vannak olyan nyelvi elemek, amelyek hatékonyabbá teszik az egyiket a másikhoz képest.

A *Java* egy általános célú, objektumorientált, osztály alapú magas szintű programozási nyelv. Szintaxisát főleg a C és C++ nyelvektől örökölte.[1]

Ehhez hasonlóan, a *Python* is egy általános célú magas szintű programozási nyelv amely többek között a funkcionális, objektumorientált, az imperatív és a procedurális programozási paradigmákat támogatja. Úgynevezett interpreteres nyelv, mivel a forrás- és a tárgykód nincs különválasztva, ezáltal a megfelelő futtató rendszer birtokában a program máris futtatható. A nyelv alapvetően az olvashatóságot és a programozói munka megkönnyítését helyezi előtérbe a futási sebességgel szemben. Egyszerűbb eszközökkel rendelkezik, azonban adott helyzetekre kevesebb megoldási alternatívát kínál.[2]

[1] https://hu.wikipedia.org/wiki/Java_(programoz%C3%A1si_nyelv)
[2] https://hu.wikipedia.org/wiki/Python_(programoz%C3%A1si_nyelv)

Imperatív programozási elemek

A *Python* és a *Java* az imperatív programozási nyelvekben elérhető vezérlési szerkezetekkel rendelkezik. Az imperatív programozás alap vezérlési szerkezetei a szekvencia, a szelekció és az iteráció.

A szekvencia a különféle utasítások, blokkok sorrendbe történő végrehajtását jelenti. Ebben az esetben a program egymás után, sorról sorra hajtja végre őket. Javában az utasítások végét ; karakter zárja, több utasítást pedig { és } karakterek közé írva tudunk blokkokba rendezni. A Python esetében a sor eleji behúzásoknak van hasonló jelzésértékük. Az azonos blokkokba tartozó elemeknek azonos mennyiségű space/tab karakterekkel kell kezdődnie.

```
int a = 2;
int b = 3;
int c = a + b;
```

Java

```
a = 2
b = 3
c = a + b
```

Python

A szelekció valamilyen feltételhez kötött vezérlésátadást valósít meg. Lehetővé teszi, hogy egy utasítás csak akkor hajtódjon végre, ha a hozzá tartozó feltétel teljesül. Mind két nyelv esetében két fajtáját különböztethetjük meg: kétirányú, illetve többirányú elágaztató utasítás.

A kétirányú elágaztató utasítás a *Java* esetében az *if* kulcsszóval kezdődik, mögötte kerek zárójelben egy logikai kifejezés szerepel. Ha a logikai kifejezés értéke igaz, az *if* után lévő utasítás hajtódik végre. Ha a kifejezés értéke hamis, és a szelekció rendelkezik *else* ággal (hosszú if szerkezet), akkor az *else* kulcsszó utáni utasítás hajtódik végre. Ha az utasításnak nincs else ága (rövid if szerkezet), akkor a program futása az if szerkezetet követően szekvenciálisan folytatódik. Ha több utasítást szeretnénk a szelekción belül végrehajtani, azokat blokkba kell rendezni. A Python nyelv esetében csupán szintaktikai eltéréseket figyelhetünk meg.

```
int a = 20;
int b = 18;
if ( a > b) {
    System.out.println( ’’ a
    nagyobb, mint b.”);
}
else {
    System.out.println( ’’ a nem
    nagyobb, mint b.”);
}
```

Java

```
a = 20
b = 18
if a > b:
    print(’’a nagyobb, mint
    b”)
else:
    print(’’a nem nagyobb,
    mint b”)
```

Python

A *Java* nyelvnek van egy speciális operátora, amely szemantikáját tekintve hasonlóan működik, mint a kétirányú elágaztató utasítás. Ez a ?: operátor. Ennek szintaktikája: <kifejezés> ? <kifejezés, ha igaz> : <kifejezés, ha hamis>.

Működését tekintve, ha a kérdőjel karakter előtt szereplő kifejezés értéke a kiértékelést követően igaz, akkor a ? karakter után elhelyezkedő kifejezés, ellenkező esetben pedig a : karakter utáni kifejezés értékelődik ki.

```
public class Test {
   public static int abs(int number) {
      int res = (number < 0) ? number * -1 : number;
      return res;
   }
}

System.out.println(abs(-4));   -> 4
System.out.println(abs(3));  -> 3
```

Bizonyos helyzetekben ezzel az operátorral felírt kifejezés helyettesítheti a kétirányú elágaztató utasítást. Ezáltal rövidíthető vele a kód, ez azonban nem biztos, hogy segíti az olvashatóságot.

A többirányú elágaztató utasítás olyan utasítás, ami a program egy adott pontján több tevékenység végrehajtása közül kínál fel egyet. Lehetővé teszi, hogy egy tevékenységet hajtsunk végre sok lehetőség közül. A többirányú elágaztató utasítás szintaktikája és szemantikája nyelvenként eltérő lehet.

Többirányú elágaztatást Java nyelven vagy az if utasítások egymásba ágyazásával, vagy a switch utasítással, Python nyelven if...elif...else szerkezettel lehet megvalósítani.

Ha az *if* kulcsszó utáni kifejezés értéke hamis, a szelekció következő ágának feltétele értékelődik ki, majd annak függvényében hajtódnak végre az utasítások. Egy szelekciós utasításon belül mindig pontosan egy ág hajtódik végre, mivel egy ág végrehajtása után a program már nem is vizsgálja a később előforduló feltételek lehetséges teljesülését.

```
int a = 20;
if ( a > 0) {
    System.out.println( ’’Pozitív.”
}
else if (a < 0) {
    System.out.println(”Negatív.”
}
else {
    System.out.println(” 0 ”);
}
```

Java

```
a = 20
if a > 0:
    print(”Pozitív.”)
elif a < 0:
    print(”Negatív.”)
else:
    print(” 0 ”)
```

Python

A *switch* utasítás szintaktikailag a switch kulcsszóból, egy kerek zárójelpárba írt kifejezésből és egy kapcsos zárójelek közötti utasításlistából áll. Az utasításlistának az elemei, az egyes utasítások a case kulcsszót követő konstans kifejezésekkel vagy a default címkével címkézhetők. Egy utasításnak több címkéje is lehet. Arra kell figyelni, hogy a címkék konstans kifejezései nem vehetik fel ugyanazt az értéket, egyedieknek kell lenniük. A *default* címke helye nincs kötve, bárhol, bármelyik utasítás előtt megjelenhet a *switch* szerkezetben, legfeljebb egyszer.

A *switch* végrehajtásakor a kulcsszó után elhelyezkedő kifejezés kiértékelését követően arra a *case* címkére kerül a vezérlés, ahol a kapott érték megegyezik a konstans kifejezés értékével. Ekkor a program végrehajtása ettől a ponttól kezdve az utasításlista utasításainak szekvenciális végrehajtásával folytatódik.

Ha nincs egyezés és van a szerkezetben default címke, akkor a *default* címkére kerül a vezérlés és a program végrehajtása innen folytatódik szekvenciálisan az utasításlista utasításainak végrehajtásával.

Ha a switch utáni kifejezés nem egyezett meg egyik case címke konstans kifejezésével sem, és default címke sincs a szerkezetben, akkor a program végrehajtása a switch szerkezetet követő utasítással szekvenciálisan folytatódik tovább.

```
switch(<kifejezés>) {
        case <konstans kifejezés>:
                <utasítás>;
        case <konstans kifejezés>:
        case <konstans kifejezés>:
                <utasítás>;
                break;
        …
        default
                <utasítás>;
}
```

Bizonyos speciális esetekben a Python *elif* szerkezete egyenértékű lehet a Java *switch* utasításával. Nevezetesen akkor, ha ugyanarra a kifejezésre vonatkozóan mindig egyenlőséget vizsgál.

Az iteráció adott utasítások ismételt végrehajtását eredményező szerkezet. A Java esetében három típusát különböztethetjük meg egymástól, a kezdőfeltételes, végfeltételes valamint a felsorolásos ciklust. A Python tekintetében beszélhetünk továbbá az előírt lépésszámú ciklusról is.

A kezdőfeltételes ciklus esetében a ciklusmagban lévő utasítások végrehajtásának száma a feltétel teljesülésétől függ. Lehetséges, hogy a ciklusmag egyszer sem fog

lefutni. A feltétel kiértékelése az utasítások végrehajtása előtt történik meg. Minden esetben, amikor a ciklusmag végrehajtódik, a kód visszatér a feltétel újra kiértékeléséhez. A kezdőfeltételes ciklust illetően a ciklus belsejében kell arról gondoskodnunk, hogy a feltétel az ismételt kiértékelések során megváltozhasson, ellenkező esetben végtelen ciklust kaphatunk.

```
int a = 1;
while ( a < 10) {
    System.out.println(a);
    a++;
}
```

Java

```
a = 1
while a < 10:
    print(a)
    a += 1
```

Python

A Java esetében a kezdőfeltételes ciklusok közé tartozik a *for* ciklus azon alakja, amely két ; karaktert tartalmaz a fejében, amely három kifejezést választ el egymástól (az első kifejezés helyén egy deklarációs utasítás is állhat). Ez a három kifejezés szabályozza a ciklus működését.

Az első kifejezés a ciklus elérésekor fog kiértékelődni, illetve amennyiben a helyén deklarációs utasítás áll, végrehajtódni.

A második kifejezés a ciklusmag lefutása előtt értékelődik ki. Valahányszor igaz az értéke, végrehajtódik a ciklusmag. Amennyiben a kifejezés értéke hamis, a ciklus futása befejeződik és a program végrehajtása a ciklust követő utasítással folytatódik.

A harmadik kifejezés a ciklusmag lefutása után értékelődik ki. Ezután a vezérlés a második kifejezés újbóli kiértékelésével folytatja a programot.

Amint az látható, nem feltétlenül szükséges a ciklus fejében változót deklarálni, nem biztos, hogy csak egy változót használunk benne, és az sem biztos, hogy valamiféle számlálás történik az esetében. Számos olyan használati lehetőséggel rendelkezik,

ami miatt nem az előírt lépésszámú ciklusok, hanem a kezdőfeltételes ciklusok közé sorolhatjuk. Vegyük példának a legnagyobb közös osztó meghatározásának problémáját. Ebben az esetben például nem tudhatjuk előre, hogy pontosan hányszor fog lefutni a ciklus.

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int a = sc.nextInt();
    int b = sc.nextInt();
    for (int m; (m = a%b) != 0; a = b, b = m)
        ;
    System.out.println(b);
}
```

Ezzel ellentétben a Python *for* ciklusa tökéletes példa a felsorolásos ciklusra. Ebben az esetben a ciklus rendelkezik ciklusváltozóval, aminek a hatóköre a ciklus blokkja. Értékét a ciklus belsejében nem tudjuk megváltoztatni. A ciklusmag annyiszor fog végrehajtódni, ahány értéket a ciklusváltozó felvesz. A klasszikus előírt lépésszámú ciklus a *range()* függvény segítségével használható. Ezen belül megadhatjuk a kezdő és a végső értéket (ha a kezdő érték nincs megadva, alapértelmezetten 0-tól indul a ciklus), valamint a lépésközt is (ha nincs megadva, alapértelmezetten 1). Nem szabad azonban elfelejtenünk, hogy a *range()* függvény elöl zárt, hátul nyílt intervallumot ad vissza.

A néhány bekezdéssel korábban leírt for ciklussal a Java nyelvű programokban is szimulálhatunk előírt lépésszámú ciklust. Ekkor a *for* kulcsszót követő kerek zárójelpár között kell megadnunk a ciklusváltozóra vonatkozó kifejezéseket. Az első kifejezésben a ciklusváltozó kezdőértékét adjuk meg, a második kifejezés a ciklus leállási feltétele, a harmadik pedig a ciklusváltozó léptetésére szolgál.

```
for ( int i = 0; i < 6; i++ ) {
    System.out.println(i);
}
# 0
# 1
# 2
# 3
# 4
# 5
```

Java

```
for i in range(0, 6, 2):
    print(i)

# 0
# 2
# 4
```

Python

A végfeltételes ciklus esetében a ciklusmag legalább egyszer végrehajtódik. A feltétel kiértékelése a ciklusmag végrehajtása után történik meg. Ennek eredményétől függően a vezérlés visszatér a ciklus elejére.

A Java nyelvben a végfeltételes ciklus a do...while szerkezettel valósítható meg.

A Python esetében külön kulcsszóval jelzett hátultesztelős ciklus nincsen, azonban más szerkezettel itt is megvalósítható.[3] Például indíthatunk egy végtelen ciklust, amiből a ciklusmagban feltételhez kötött *break* utasítással léphetünk ki.

```
int a = 1;
do {
    System.out.println(a);
    a++;
} while (a < 10);
```

Java

```
a = 1
while True:
    print(a)
    a += 1
    if a == 10:
        break
```

Python

[3] https://sulipython.fandom.com/hu/wiki/Ciklus

Felsorolásos ciklust érdemes alkalmazni, ha egy gyűjtemény minden elemére pontosan egyszer végre akarjuk hajtani a ciklusmagban lévő utasításokat. Legfőbb jellemzője a ciklusszámláló hiánya.

```
int t = {1,2,3};
for ( int item : t) {
    System.out.println(item);
}
```

Java

```
t = [1,2,3]
for item in t:
    print(item)
```

Python

Mindkét nyelv esetében találkozhatunk vezérlésátadó utasításokkal. Ezek a *for*, a *while* és a *do*...*while* ciklusokban egyaránt használhatók. Ide tartozik a *break* utasítás, aminek hatására a program kiugrik az őt tartalmazó ciklusból, valamint a *continue* utasítás, amelynek következtében a vezérlés a ciklusmag végére kerül át. Ezt követően a ciklus úgy folytatódik, mintha normálisan ért volna véget, csak kihagyta a continue és a ciklusmag vége között lévő utasításokat.

Típusrendszerek kérdése

A két nyelv tekintetében a Java komolyabb típusrendszerrel rendelkezik, ezáltal erősebb típusos nyelvnek tekinthető. Esetében mindent deklarálni kell, mindennek pontosan megadott típussal kell rendelkeznie.

Típusrendszerét két csoportra oszthatjuk, beszélhetünk primitív és referencia típusokról. A primitív típusok közé tartoznak a számjellegű valamint a logikai típusok. A numerikus típusok közé sorolható a *byte*, a *short*, az *int*, a *long*, a *char*, a *float*, valamint a *double* típus. Emellett a logikai típusok közé tartozik a *boolean* típus, amely egy logikai értéket reprezentál a *true* és a *false* literálokkal jelezve.

A Java nyelv referencia típusai közé sorolhatók az *osztályok*, az *interfészek*, a *típusparaméterek*, valamint a *tömbök*. Osztály, interfész és tömb típusú (referencia) változókkal ilyen típusú objektumokra tudunk hivatkozni. [4]

A Python nyelv a Javával ellentétben nem követeli meg a változók típusainak előre deklarálást. Ennek értelmében a változók típusa folyamatosan (dinamikusan) változhat, ezt nevezzük dinamikus tipizálásnak. Ugyanakkor, a megfelelő operátorok használatának érdekében sokszor fontos lehet tudni, milyen típussal rendelkeznek az adott változók. Ennek következtében fontos lehet a változók megfelelő elnevezése is.

A nyelv számszerű típusai közé tartoznak az *int*, a *float* valamint a *complex* típusok. Ezen felül a logikai értékek is az egész számok altípusai közé sorolhatók.

A Python tekintetében beszélhetünk továbbá szekvencia típusokról, ide tartoznak a *list*, a *tuple*, illetve a *range* típusok. Ezek mellett a szöveg szekvencia típusok közé a *string* típus, a bináris szekvencia típusok közé pedig a *bytes*, a *bytearray* és a *memoryview* típusok.

A nyelv típusrendszerében találkozhatunk továbbá halmaz típusokkal (set, frozenset), valamint mapping típussal (dict).[5]

[4] https://docs.oracle.com/javase/specs/jls/se12/jls12.pdf
[5] https://docs.python.org/3/tutorial/index.html

Eljárásorientált elemek

A programozási nyelvek eljárásorientált elemei közé tartoznak az alprogramok (függvények, eljárások), amelyek a funkcionális absztrakció eszközei, az újrafelhasználhatóságot valósítják meg.

A Java esetében az alprogramok (metódusok) szintaktikájukat tekintve három részből épülnek fel: fej, törzs és vég. A fej tartalmazza a metódus specifikációját: a metódus módosítóit, a metódus nevét, formális paraméterlistáját, a metódus törzsében nem kezelt, ám ott előforduló ellenőrzött kivételek típusát, valamint a függvénymetódusok esetében a metódus típusát is. A törzs felépítése az alprogram algoritmusának utasításait foglalja magába. A függvénymetódusok befejezése minden esetben return utasítással történik.

A Python nyelvben az alprogramok létrehozása a def kulcsszóval kezdődik. A kulcsszót követően az alprogram neve, a formális paraméterlistája, valamint egy : karakter foglal helyet. Az alprogram törzsében a végrehajtani kívánt utasítások helyezkednek el. A függvények esetében szükség van egy return utasításra, amely a függvény visszatérési értékét adja meg. Ha az alprogram nem rendelkezik return utasítással, akkor esetében eljárásról beszélünk. A Javával ellentétben, a Python nyelvben nincs szükség a függvény típusának megadására.[6]

Az alprogramok aktiválásához azok nevével és aktuális paraméterlistájával ellátott hívására van szükség. A függvényhívás kifejezésekben operandusok helyén, az eljáráshívás utasítások helyén szerepelhet.

[6] Pearson – The Practice of Computing Using Python (Third Edition)

```
public int plus (int a, int b) {
        return a + b;
}
```

Java

```
def plus (a, b):
        return a + b
```

Python

Objektumorientált elemek

Az objektumorientált elemek közé tartoznak az osztályok, az interfészek valamint az enumerációk.

Az osztály megvalósítja az egységbezárás elvét, modell szintjén összevonja az adat- és funkcionális modellt. Segítségével olyan típust hozunk létre, amely megvalósítja az absztrakt adattípust, azaz a bezárást és az információrejtést. Az osztályokon belül megkülönböztethetünk adattagokat, amelyek az adatmodellt leíró részek, valamint metódusokat, amelyek a modell viselkedését definiáló alprogramok.

Két vagy több osztály között fennálló asszimetrikus kapcsolatot az osztályok közötti öröklődésnek nevezzük. Az öröklődés során az alosztály (Java és a Python esetében gyermekosztály (child class)) örökli a szuperosztály (Java és a Python esetén szülőosztály (parent class)) minden, a bezárás által megengedett attribútumát és metódusát. Az alosztály ezen felül rendelkezhet saját attribútumokkal és metódusokkal, illetve megváltoztathatja az öröklött metódusok implementációit, ezáltal polimorf metódusokat létrehozva. Ezeken felül meg tudja változtatni az örökölt eszközök tulajdonságait, bezárási szintjüket, típusukat.

Az öröklődés tekintetében beszélhetünk egyszeres öröklődésről, amely esetében egy alosztálynak legfeljebb egy szuperosztálya lehet, valamint többszörös öröklődésről, amikor egy alosztálynak több szuperosztálya is lehet. A Java esetében az osztályok között egyszeres, az interfészek esetén pedig többszörös öröklődés lehetséges. Ezzel szemben a Python nyelvben az osztályok tekintetében is lehetőség van a többszörös öröklődésre.

Az öröklődési hierarchiának különböző szintjeit különböztethetjük meg egymástól. Egy osztály leszármazottainak nevezzük az ő gyermekosztályait, valamint azok gyermekeit stb. egészen a hierarchia aljáig. Ezzel szemben egy osztály ősosztályainak (elődosztályának) nevezzük az ő szülőosztályát, valamint annak a szülőosztályát stb. egészen a hierarchia tetejéig. Egy osztály kliens osztályának nevezzük azt az osztályt, amely se nem leszármazott osztálya, se nem őse az adott

osztálynak. A hierarchia levél osztályai azok az osztályok, amelyekből valamilyen okból kifolyólag már nem származtatunk újabb osztályokat.

Az osztályok adattagjai és metódusai esetében különböző láthatósági szinteket különböztethetünk meg egymástól. A legtágabb szinttől a legszűkebb felé haladva beszélhetünk publikus, védett, valamint privát bezárási szintről.

A publikus bezárási szintű elemek láthatóak a definiáló osztály mellett az abból leszármazott osztályokban és a kliensosztályokban egyaránt. A védett elemek a definiáló és a belőle származtatott osztályokban, még a privátok csak a definiáló osztályokban láthatók.

A láthatósági szintek mellett beszélhetünk osztályszintű és példányszintű attribútumokról és metódusokról is. Az osztályszintű elemek példányfüggetlenek, az adott osztályhoz köthetők, egymagukban jellemzőek az osztályból származtatott összes objektumra. Közös tulajdonságokat írnak le, ezáltal érdemes őket egy helyen tárolni. Ezzel szemben a példányszintű elemek aktiválásához szükség van az adott osztály egy példányára, példányonként eltérőek lehetnek.

Az osztályok tekintetében fontos megemlíteni az absztrakt osztályt, amely nem példányosítható, de örökölhető osztály, ezáltal a hierarchiának általában nem levéleleme. Absztrakt osztályról akkor beszélünk, ha a modellező az osztályt absztraktnak definiálta, vagy pedig van absztrakt metódusa. Az absztrakt metódusok különlegessége, hogy nem rendelkeznek implementációval, csak a specifikációjukat ismerjük.

Python nyelv esetén a fentebb említett láthatósági szinteket különböztethetjük meg egymástól. Alapértelmezetten az osztályok minden attribútuma és metódusa publikus láthatósági szintű. A védett láthatósági szintet egy darab aláhúzás karakterrel, a privát láthatósági szintet pedig két aláhúzás karakterrel jelölhetjük az attribútumok és metódusok neve előtt. (Például: self.name - publikus, self._name - védett, self.__name - privát)[7]

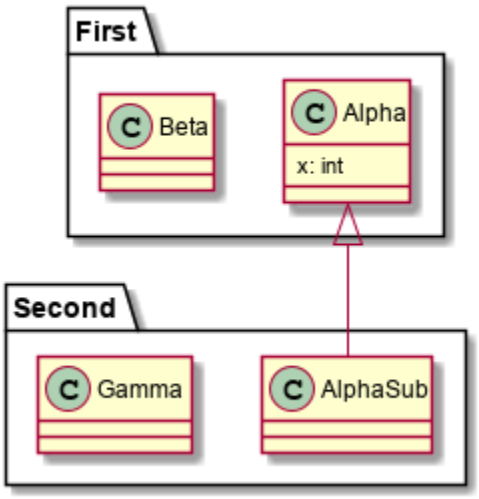
Pythonban az osztályszintű adattagokat a konstruktoron kívül, a példányszintű adattagokat pedig azon belül deklaráljuk. Az osztályszintű metódusok jelölésére a

[7] Pearson – The Practice of Computing Using Python (Third Edition)

@staticmethod decorator-t használjuk. Ha egy metódus előtt ez nem szerepel, akkor a metódus példányszintű.

Java nyelvben az osztályszintű adattagokat és metódusokat a static kulcsszó jelöli.

A Java esetében a korábban említett bezárási szintek mellett létezik egy csomagszintű láthatósági szint is, amely a védettnél szűkebb, a privátnál bővebb láthatóságot tesz lehetővé. Ennek következtében a nyelvben a láthatósági szintek tulajdonságai eltérnek az általánostól. Vegyük példának az alábbi UML ábrát.



Az x attribútum különböző láthatóságok szerint a következő osztályokból érhető el:

	Alpha	Beta	AlphaSub	Gamma
publikus	x	x	x	x
védett	x	x	x	
csomagszintű	x	x		
privát	x			

Általánosan tehát, egy osztálytag a láthatósági szinttől függően elérhető a(z):

	definiáló típusban	azonos csomagban lévő típusban	leszármazott típusban	bármely típusban
publikus	x	x	x	x
védett	x	x	x	
csomagszintű	x	x		
privát	x			

A Java nyelv interfész típusa szintaktikáját tekintve hasonlít az absztrakt osztályokhoz. Olyan viselkedéseket, metódusokat definiál, amelyeket a hierarchia valamely osztályával valósíthatunk meg. Tartalmazhat public static final módosítójú adattagokat, valamint publikus absztrakt, statikus és default metódusokat (utóbbi kettőt a Java 8-as verziója óta). Az interfészek adattagjai és metódusai egyaránt publikusak, nincs szükség külön a láthatósági szintjük megjelölésére. Egy interfész akár több interfészt is kiterjeszthet, azaz esetünkben megengedett a többszörös öröklődés.

Az enumeráció speciális típusa a Java nyelvnek. Arra szolgál, hogy előre definiált konstansokat hozzunk létre a segítségével. Szintaktikája hasonló az osztályéhoz, azonban az enumeráció nem örökölthető. Létrehozása az enum kulcsszóval történik.

Feladatmegoldások

Az elkövetkezőkben komolyabb példákon keresztül mutatnám be azokat az eszközöket, amelyek az eddigiek során elméleti szinten tárgyaltam.

Tekintsük első példaként egy, a futballal kapcsolatos probléma megoldását.

Football (aka Soccer)[8]

A futball a világ legnépszerűbb sportja. Ahogy azt mindenki tudja, Brazília büszkélkedhet a legtöbb világbajnoki címmel (szám szerint öttel: 1958, 1962, 1970, 1994 és 2002). Mivel a nemzeti bajnokságukban sok csapat vesz részt (sőt még a regionális bajnokságaikban is sok csapat van), nagyon nehéz feladat az állás nyomon követése ilyen sok csapat és lejátszott mérkőzés mellett!

A feladatod tehát nagyon egyszerű: írj egy programot, amely megkapja a bajnokság nevét, a csapatneveket és a lejátszott mérkőzéseket, és kiírja a bajnokság jelenlegi állását.

Egy csapat akkor nyer meg egy mérkőzést, ha több gólt szerez, mint az ellenfele. Nyilvánvalóan veszít egy csapat, ha kevesebb gólt szerez. Ha mindkét csapat ugyanannyi gólt szerez, akkor ezt döntetlennek nevezzük. Egy csapat minden győzelemért 3 pontot, minden döntetlenért 1 pontot és minden vereségért 0 pontot kap.

A csapatokat a következő szabályok alapján rangsoroljuk (ebben a sorrendben):

1. A legtöbb megszerzett pont.
2. A legtöbb győzelem.
3. A legjobb gólkülönbség (azaz a rúgott gólok számából kivonva a kapott gólok számát).
4. A legtöbb rúgott gól.

[8] https://progcont.hu/progcont/100013/?pid=10194

5. A legkevesebb lejátszott mérkőzés.
6. Lexikografikus sorrend.

A bemenet specifikációja

A bemenet első sora egy N egész számot tartalmaz önmagában (0 < N < 1000). Ezt N darab bajnokságleírás követi, amelyek mindegyike a bajnokság nevével kezdődik egyetlen sorban. A bajnokság neve tartalmazhat bármilyen betűt, számjegyet, szóközöket stb. A bajnokságnév legfeljebb 100 karakter hosszúságú lehet. A következő sorban egy T egész szám szerepel (1 < T ≤ 30), amely a bajnokságban részt vevő csapatok számát adja meg. Ezután T sor következik, amelyek egy-egy csapatnevet tartalmaznak. A csapatnevekben minden olyan karakter előfordulhat, amelynek az ASCII kódja nagyobb vagy egyenlő 32-nél (szóköz), kivéve a „#” és a „@” karaktereket, amelyek soha nem szerepelnek csapatnevekben. Egyik csapatnév sem áll 30-nál több karakterből.
A csapatneveket követően egy G nemnegatív egész szám áll egy sorban, amely a bajnokságban eddig lejátszott mérkőzések számát adja meg. G nem nagyobb, mint 1000. Ezt G sor követi a lejátszott mérkőzések eredményeivel. Ezek a sorok a következő formájúak:

csapatnév_1#gólok1@gólok2#csapatnév_2

Vegyük például a következő sort:
A csapat#3@1#B csapat

Ez azt jelenti, hogy az A csapat és a B csapat közötti mérkőzésen az A csapat 3 gólt szerzett, a B csapat pedig 1-et. Minden gólszám egy 20-nál kisebb nemnegatív egész. Felteheted, hogy nem szerepelnek nem létező csapatnevek (azaz minden csapatnév, amely előfordul a mérkőzéseredmények között, korábban előfordult a csapatnévlistában is), valamint hogy egyik csapat sem játszik saját maga ellen.

A kimenet specifikációja

Minden egyes bajnokság esetén ki kell írnod a bajnokság nevét egy sorban. A következő T sorban az állást kell kiírnod a fenti szabályoknak megfelelően. Ha a helyezést a lexikografikus sorrend dönti el, akkor a kis- és nagybetűket nem kell megkülönböztetni. A kiírandó sorok a következő formájúak:
[a]) Csapatnév [b]p, [c]g ([d]-[e]-[f]), [g]gd ([h]-[i])
A szögletes zárójelekbe írt betűk jelentése a következő:

- [a] = a csapat helyezése
- [b] = szerzett pontok
- [c] = lejátszott mérkőzések
- [d] = győzelmek
- [e] = döntetlenek
- [f] = vereségek
- [g] = gólkülönbség
- [h] = rúgott gólok
- [i] = kapott gólok

A mezők között egyetlen szóköznek kell állnia, az egyes bajnokságokhoz tartozó kimeneteket pedig egyetlen üres sornak kell elválasztania egymástól (lásd a példa kimenetet).

A feladat megoldásának egyik lehetséges alternatívája Java nyelven a következő:

```
import java.util.*;

public class Futball implements Comparable<Futball> {

  private String csapatnev;
  private int jatszottMeccs;
  private int gyozelem;
  private int vereseg;
  private int rugottGol;
  private int kapottGol;
```

```
public Futball(String csapatnev) {
    this.csapatnev = csapatnev;
}

@Override
public String toString() {
    return csapatnev + " "
                + (gyozelem * 3 + getDontetlen())
                + "p, " + jatszottMeccs + "g (" + gyozelem
                + "-" + getDontetlen() + "-" + vereseg + ")," + getGolkulonbseg() +
                "gd (" + rugottGol + "-" + kapottGol + ")";
}

public String getCsapatnev() {
    return csapatnev;
}

public int getjatszottMeccs() {
    return jatszottMeccs;
}

public int getGyozelem() {
    return gyozelem;
}

public int getRugottGol() {
    return rugottGol;
}

public int getPont() {
    return gyozelem * 3 + getDontetlen();
}

public int getGolkulonbseg() {
    return rugottGol-kapottGol;
}
public int getDontetlen() {
   return jatszottMeccs - (gyozelem + vereseg);
}


public void addJatszottMeccs() {
    this.jatszottMeccs ++;
```

```
}

public void addGyozelem() {
    this.gyozelem ++;
}

public void addVereseg() {
    this.vereseg ++;
}

public void plusRugottGol(int rugottGol) {
    this.rugottGol += rugottGol;
}

public void plusKapottGol(int kapottGol) {
    this.kapottGol += kapottGol;
}



@Override
public int compareTo(Futball o) {
    int res = Integer.compare(o.getPont(),this.getPont());
    if (res != 0) {
        return res;
    }
    res = Integer.compare(o.getGyozelem(),this.gyozelem);
    if (res != 0) {
        return res;
    }
    res = Integer.compare(o.getGolkulonbseg(),this.getGolkulonbseg());
    if (res != 0) {
        return res;
    }
    res = Integer.compare(o.getRugottGol(),this.rugottGol);
    if (res != 0) {
        return res;
    }
    res = Integer.compare(this.jatszottMeccs,o.getjatszottMeccs());
    if (res != 0) {
        return res;
    }
```

```
    return this.csapatnev.toLowerCase()
                .compareTo(o.getCsapatnev().toLowerCase());
}


public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    int n = Integer.parseInt(sc.nextLine());

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            System.out.println();
        }

        String bajnoksag = sc.nextLine();

        int t = Integer.parseInt(sc.nextLine());
        Map<String, Futball> map = new TreeMap<>();
        for (int j = 0; j < t; j++) {
            String name = sc.nextLine();
            map.put(name,new Futball(name));
        }

        int g = Integer.parseInt(sc.nextLine());
        String[] merkozesek = new String[g];
        for (int k = 0; k < g; k++) {
            merkozesek[k] = sc.nextLine();
        }

        for (String merkozes : merkozesek) {
            String[] adatok = merkozes.split("#|@");
            int ad1 = Integer.parseInt(adatok[1]);
            int ad2 = Integer.parseInt(adatok[2]);
            Futball elso_csapat = map.get(adatok[0]);
            Futball masodik_csapat = map.get(adatok[3]);
            elso_csapat.addJatszottMeccs();
            elso_csapat.plusRugottGol(ad1);
            elso_csapat.plusKapottGol(ad2);
            masodik_csapat.addJatszottMeccs();
            masodik_csapat.plusRugottGol(ad2);
            masodik_csapat.plusKapottGol(ad1);
```

```
                if (ad1 > ad2) {
                    elso_csapat.addGyozelem();
                    masodik_csapat.addVereseg();
                }
                else if (ad2 > ad1) {
                    masodik_csapat.addGyozelem();
                    elso_csapat.addVereseg();
                }
            }

            System.out.println(bajnoksag);

            Futball[] csapatok = map.values().toArray(new Futball[0]);

            Arrays.sort(csapatok);
            for (int r = 1; r <= t; r++) {
                System.out.println(r + ") " + csapatok[r - 1]);
            }
        }
    }
}
```

A feladat megoldásához létrehoztunk egy Futball osztályt, amelynek objektumai később az egyes csapatok lesznek. Az osztályban deklaráltuk a szükséges adattagokat, létrehoztunk egy egy paraméteres konstruktort, valamint újraimplementáltuk a toString() metódust. Az utóbbi a kimenet egy részének megfelelő formátumáért lesz felelős.

Ezt követően létrehoztuk az osztály azon metódusait amelyek az adattagok értékének lekérdezését (getter metódusok) és változását (add és plus metódusok) teszik lehetővé.

Az egyes csapatok feladat szerinti megfelelő sorba rendezéséhez szükséges volt az osztály számára egy természetes rendezést kialakítanunk. Ezt a Comparable interfész compareTo() metódusának implementálásával tudtuk megtenni. A compareTo() metóduson belül a feladat szövege szerint, különböző utasítások segítségével határoztuk meg, milyen adatokat figyelembe véve tudja a programunk a focicsapatok közötti megfelelő sorrendet kialakítani.

A Futball osztályon belül létrehoztunk egy main() metódust, amely az input adatok kezeléséért és a feladat végrehajtásáért lesz felelős. Az adatokat úgy kezeltük, hogy egy mapben tároltuk az egyes csapatokhoz tartozó csapatneveket kulcsként, a hozzájuk tartozó értékként pedig a saját Futball típusú objektumuk került. A mérkőzéseket külön tömbben tároltuk. Miután minden adat bekerült a memóriába, egy ciklus segítségével megvizsgáltuk az egyes mérkőzések adatait, a résztvevő csapatokat és a rúgott gólok számát. Ezek függvényében változtattuk a csapatnevekhez tartozó objektumok egyes adatait. Miután minden mérkőzés eredményét elkönyveltük, sorba rendeztük a csapatokat és a megfelelő formátumban kiírattuk őket.

A Futball osztály objektumainak rendezhetőségét a Comparable interfész compareTo metódusának implementálásán túl, más eszközökkel is megvalósíthatjuk. Ilyen esetekben nem szükséges a Comparable interfész implementálása.

Lehetőségünk van például arra, hogy a sort() metóduson belül adjuk meg a rendezést segítő, a Comparator interfészt megvalósító példányt.

```
Arrays.sort(csapatok, new Comparator<Futball>() {
        @Override
        public int compare(Futball o1, Futball o2) {
            int res = Integer.compare(o2.getPont(),o1.getPont());
            if (res != 0) {
                return res;
            }
            res = Integer.compare(o2.getGyozelem(),o1.getGyozelem());
            if (res != 0) {
                return res;
            }
            res = Integer.compare(o2.getGolkulonbseg(),o1.getGolkulonbseg());
            if (res != 0) {
                return res;
            }
            res = Integer.compare(o2.getRugottGol(),o1.getRugottGol());
            if (res != 0) {
                return res;
```

```
            }
            res = Integer.compare(o1.getjatszottMeccs(),o2.getjatszottMeccs());
            if (res != 0) {
                return res;
            }
            return
        o1.getCsapatnev().toLowerCase().compareTo(o2.getCsapatnev().toLowerCase());
          }
});
for (int r = 1; r <= t; r++) {
       System.out.println(r + ") " + csapatok[r - 1]);
}
```

Ezen felül lehetőségünk van a sort() metódus második paramétereként lambda kifejezés megadására is.

```
Arrays.sort(csapatok, (o1,o2) -> {
        int res = Integer.compare(o2.getPont(),o1.getPont());
        if (res != 0) {
            return res;
        }
        res = Integer.compare(o2.getGyozelem(),o1.getGyozelem());
        if (res != 0) {
            return res;
        }
        res = Integer.compare(o2.getGolkulonbseg(),o1.getGolkulonbseg());
        if (res != 0) {
            return res;
        }
        res = Integer.compare(o2.getRugottGol(),o1.getRugottGol());
        if (res != 0) {
            return res;
        }
        res = Integer.compare(o1.getjatszottMeccs(),o2.getjatszottMeccs());
        if (res != 0) {
            return res;
        }
        return o1.getCsapatnev().compareTo(o2.getCsapatnev());
});

for (int r = 1; r <= t; r++) {
          System.out.println(r + ") " + csapatok[r - 1]);
}
```

Mindezek mellett a rendezést megvalósíthatjuk a Comparator interfész comparing() és thenComparing() metódusainak egymást követő hívásaival is. Ekkor ezek paraméterlistáján a Futball osztály publikus metódusreferenciáit is használhatjuk.

```
Arrays.sort(csapatok, Comparator
      .comparing(Futball::getPont, Comparator.reverseOrder())
      .thenComparing(Futball::getGyozelem, Comparator.reverseOrder())
      .thenComparing(Futball::getGolkulonbseg, Comparator.reverseOrder())
      .thenComparing(Futball::getRugottGol, Comparator.reverseOrder())
      .thenComparing(Futball::getjatszottMeccs, Comparator.reverseOrder())
      .thenComparing(Futball::getCsapatnev));

for (int r = 1; r <= t; r++) {
   System.out.println(r + ") " + csapatok[r - 1]);
}
```

A feladat Python nyelven írt lehetséges megoldása a következő:

```
import sys
import re

class Futball():

  def __init__(self, csapatnev):
    self.__csapatnev = csapatnev
    self.__jatszott_meccs = 0
    self.__gyozelem = 0
    self.__vereseg = 0
    self.__rugott_gol = 0
    self.__kapott_gol = 0

  def __str__(self):
    return "{} {}p, {}g ({}-{}-{}), {}gd ({}-{})".format(self.__csapatnev,self.__gyozelem * 3 +
                                                         (self.__jatszott_meccs - (self.__gyozelem +
                                                         self.__vereseg)),self.__jatszott_meccs,
                                                         self.__gyozelem,(self.__jatszott_meccs -
                                                         (self.__gyozelem +
                                                         self.__vereseg)),self.__vereseg,
                                                         (self.__rugott_gol -
                                                         self.__kapott_gol),self.__rugott_gol,self.__kapott_
                                                         gol)

  def getCsapatnev(self):
    return self.__csapatnev

  def getJatszott_meccs(self):
    return self.__jatszott_meccs

  def getGyozelem(self):
    return self.__gyozelem

  def getRugott_gol(self):
    return self.__rugott_gol

  def getPont(self):
    return self.__gyozelem * 3 + (self.__jatszott_meccs - (self.__gyozelem +
      self.__vereseg))

  def getGolkulonbseg(self):
    return self.__rugott_gol - self.__kapott_gol

  def addJatszottMeccs(self):
    self.__jatszott_meccs += 1
```

```
    def addGyozelem(self):
        self.__gyozelem += 1

    def addVereseg(self):
        self.__vereseg += 1

    def plusRugottGol(self, rugottGol):
        self.__rugott_gol += rugottGol

    def plusKapottGol(self, kapottGol):
        self.__kapott_gol += kapottGol

    def __lt__(self, other):
        if self.getPont() != other.getPont():
            return self.getPont() > other.getPont()

        if self.__gyozelem != other.getGyozelem():
            return self.__gyozelem > other.getGyozelem()

        if self.getGolkulonbseg() != other.getGolkulonbseg():
            return self.getGolkulonbseg() > other.getGolkulonbseg()

        if self.__rugott_gol != other.getRugott_gol():
            return self.__rugott_gol > other.getRugott_gol()

        if self.__jatszott_meccs != other.getJatszott_meccs():
            return self.__jatszott_meccs < other.getJatszott_meccs()

        return self.__csapatnev.lower() < other.getCsapatnev().lower()



def main():

    n = int(sys.stdin.readline())

    for i in range(0, n):
        if i > 0:
            print()

        bajnoksag = sys.stdin.readline()

        t = int(sys.stdin.readline())
        dict = {}
        for j in range(0, t):
            name = sys.stdin.readline()[:-1]
```

```
            dict.update({name:Futball(name)})

        g = int(sys.stdin.readline())
        merkozesek = []
        for k in range(0, g):
            merkozesek.append(sys.stdin.readline()[:-1])

        for merkozes in merkozesek:
            adatok = re.split('#|@',merkozes)
            ad1 = int(adatok[1])
            ad2 = int(adatok[2])
            elso_csapat = dict.get(adatok[0])
            masodik_csapat = dict.get(adatok[3])
            elso_csapat.addJatszottMeccs()
            elso_csapat.plusRugottGol(ad1)
            elso_csapat.plusKapottGol(ad2)
            masodik_csapat.addJatszottMeccs()
            masodik_csapat.plusRugottGol(ad2)
            masodik_csapat.plusKapottGol(ad1)

            if ad1 > ad2:
                elso_csapat.addGyozelem()
                masodik_csapat.addVereseg()
            elif ad2 > ad1:
                elso_csapat.addVereseg()
                masodik_csapat.addGyozelem()

        print(bajnoksag, end='')

        csapatok = sorted(dict.values())
        for r in range(1,t+1):
            print("{}) {}".format(r, csapatok[r-1]))


if __name__ == '__main__':
    main()
```

A feladat Python nyelvű megoldása logikáját tekintve ugyanaz, mint a Java nyelvű megoldás. A különbség a szintaktikában, az egyes metódusok elnevezésében, és a sorbarendezés megoldásában van.

A Python esetében a lower then metódus (__lt__(self,other)) újraimplementálásával oldható meg a feladat. Ebben az esetben a hasonlító operátor (<) újraimplementálása történik. Ennek segítségével a sorted függvény képes a Futball tipusú objektumok között is értelmezni a hasonlító operátort, ezáltal képes a megfelelő sorrend kialakítására.

A következő feladattal először Python nyelven találkoztam, ahol a feladat megoldásának eredményeképpen megszületett programkód lassúsága volt a legnagyobb probléma. Több módszerrel próbáltam hatékonyabb kódot létrehozni, azonban az algoritmus optimalizálása nem oldotta meg a problémát. Ennek oka, hogy maga az algoritmus listakezelése az, ami sokkal lassabb, mint a Java esetében, habár a két nyelvű megoldás a szintaktikát leszámítva ugyanaz.

Nézzük először a feladat szövegét:

Hármasok[9]

A feladatod, hogy pozitív egész számok egy sorozata esetén meghatározd a sorozatban található hármasok számát. Ebben a feladatban akkor és csak akkor tekintjük az (x, y, z)-t hármasnak, ha x + y = z. Az (1, 2, 3) tehát hármas, míg a (3, 4, 5) nem az.

A bemenet specifikációja
A bemenet számos tesztesetet tartalmaz. Minden teszteset N-nel, egy pozitív egész számmal kezdődik (3 ≤ N ≤ 5000). A következő néhány sorban N pozitív egész szám szerepel. A bemenetet a fájl vége jel (EOF) zárja.

A kimenet specifikációja
Minden tesztesetre a sorozatban található hármasok számát kell a kimenetre írni egy sorban.

A feladat Python nyelvű megoldása a következő:

```
import sys

def triples(v):
    v = sorted(v)
    key = []
    freq = []
    ind = -1
```

[9] https://progcont.hu/progcont/100023/?locale=hu&pid=11386

```
        for num in v:
            if num not in key:
                key.append(num)
                freq.append(1)
                ind += 1
            else:
                freq[ind] += 1
        db = 0
        for i in range(0,len(key)):
            z= key[i]
            x = 0
            y = i
        while x<=y:
            if key[x] + key[y] > z:
                y -= 1
            elif key[x] + key[y] < z:
                x += 1
            elif x == y:
                db += ((freq[x]*(freq[x]-1))/2) * freq[i]
                x += 1
                y -= 1
            else:
                db += freq[x] * freq[y] * freq[i]
                x += 1
                y -=1

        return int(db)


def main():

    g = sys.stdin.readline()[:-1]
    while g:
        v = sys.stdin.readline().split(' ')
        vi = []
        for i in v:
            vi.append(int(i))
        print(triples(vi))
        g = sys.stdin.readline()[:-1]

if __name__ == '__main__':
        main()
```

Ugyanez Java esetén:

```
import java.util.Arrays;
import java.util.Scanner;


public class Triples {

  public static void main(String[] args) {

    Scanner sc = new Scanner(System.in);
    while (sc.hasNext()) {
      int n = Integer.parseInt(sc.nextLine());
      String[] v = sc.nextLine().split(" ");
      int[] vi = new int[n];
      for (int i = 0; i<n; i++) {
        vi[i] = Integer.parseInt(v[i]);
      }
      System.out.println(Triples.triples(vi));
    }
  }

  public static long triples(int[] v) {
    Arrays.sort(v);
    long[] key = new long[v.length+1];
    int[] freq = new int[v.length+1];
    int ind = 1;
    for (int num:v) {
      if (key[ind-1] == num) {
        freq[ind-1]++;
      }
      else {
        key[ind] = num;
        freq[ind] = 1;
        ind++;
      }
    }


    long db = 0;
    for (int i = 1; i < ind; i++) {
      long z = key[i];
      int x = 0;
      int y = ind - 1;
```

```
            while (x <= y) {
                if (key[x] + key[y] > z) {
                    y --;
                }
                else if (key[x] + key[y] < z) {
                    x ++;
                }
                else if (x == y) {
                    db += 1L * ((freq[x] * (freq[y]-1))/2) * freq[i];
                    x ++;
                    y --;
                }
                else {
                    db += 1L * freq[x] * freq[y] * freq[i];
                    x ++;
                    y --;
                }
            }
        }
        return db;
    }
}
```

A megoldás algoritmusa mindkét nyelv esetében ugyanaz. Az adatok beolvasása és eltárolása után, azokat megvizsgálva létrehoztunk két tömböt. Az első tömbünk (key) a beolvasott értékeket tárolja el úgy, hogy minden érték csak egyszer szerepel benne, akkor is, ha a beolvasás során több azonos számot tároltunk el. A második tömbünk (freq) a különböző számok gyakoriságát tárolja el. A két tömb esetében az azonos indexeken az összetartozó adatok szerepelnek.

Ezt követően a key tömb elemein sorban végig haladtunk és minden (z) elemhez megvizsgáltuk, van-e olyan lehetőség amellyel hármast alkothatunk belőle és két másik számból. Ehhez segítségül használtunk két segéd változót (x,y), amelyekkel indexeket jelölve a key tömbön jobbról valamint balról végighaladva, folyamatosan megvizsgáltuk, hogy az adott két számmal a kiinduló számunk (z) hármast alkothat-e. Ha a segédváltozókkal kijelölt számok összege nagyobb, mint a z, akkor az y segédváltozóval léptünk egyet balra, ha kisebb, akkor az x segédváltozóval léptünk egyet jobbra. Reménykedve abban, hogy egyszer az x és y által kijelölt számok

összege pontosan z lesz. A megfelelő számítások elvégzését követően változtattuk az indexek értékét, valamint a talált hármasok számát.

A következő feladat egy ismert logikai játék megoldása.

A híd[10]

n ember szeretne átkelni egy hídon éjszaka. Legfeljebb két ember kelhet át egyszerre, akiknél lennie kell egy elemlámpának. Az n embernél csak egy elemlámpa van, így egyfajta ingázást kell végezniük, hogy visszajuttassák az elemlámpát, és ezáltal további emberek is átkelhessenek.
Az egyes személyek nem feltétlenül azonos sebességgel tudnak átkelni a hídon; egy csoport sebességét a lassabb tagjának a sebessége határozza meg. A feladatod, hogy meghatározz egy olyan stratégiát, amellyel az n ember minimális idő alatt juthat át a túloldalra.

A bemenet specifikációja

A bemenet egy olyan sorral kezdődik, amely egyetlen pozitív egész számból, a tesztesetek számából áll. Ezt a sort egy üres sor követi, mint ahogy egy-egy üres sor van két egymást követő teszteset között is.
Minden teszteset első sorában n értéke szerepel. Ezt n sor követi, amelyek az egyes emberek átkelési idejeit tartalmazzák. Legfeljebb 1000 ember lesz, és egyiküknek sem tart tovább 100 másodpercnél a hídon való átkelés.

A kimenet specifikációja

Az egyes tesztesetekhez tartozó kimeneteknek az alábbi leírást kell követniük. Két egymást követő teszteset kimenetét egy-egy üres sorral kell elválasztani.
Minden tesztesetre a kimenet első sorában az n ember átkeléséhez szükséges másodpercek számának kell szerepelnie. A további sorokba egy olyan stratégiát kell kiírni, amellyel ez az idő érhető el. Mindegyik további sornak egy vagy két egész számot kell tartalmaznia, amelyek a következőnek átkelő személyt vagy személyeket adják meg. (Az egyes személyeket a bemeneten megadott átkelési idejükkel azonosítjuk. Bár több embernek is lehet ugyanaz az átkelési ideje, ennek a kétértelműségnek nincs következménye.) Ne feledjük, hogy az átkelések váltakozó

[10] https://progcont.hu/progcont/100067/?locale=hu&pid=10037

irányúak, mivel mindig vissza kell juttatni az elemlámpát, hogy mások is átkelhessenek. Ha több stratégiával is elérhető a minimális idő, bármelyik megadható.

A feladat Python nyelvű megoldása a következő:

```
import sys

def bridge(v):
  v = sorted(v)
  if len(v) == 1:
    print(v[0])
    print(v[0])
  elif len(v) == 2:
    print(v[1])
    print('{} {}'.format(v[0],v[1]))
  elif len(v) == 3:
    print(v[0]+v[1]+v[2])
    print('{} {}'.format(v[0],v[1]))
    print(v[0])
    print('{} {}'.format(v[0],v[2]))
  else:
    ind = len(v)-1
    sum=0
    while ind > 2:
      if v[1]*2 < v[0]+v[ind-1]:
        sum+=v[0]+2*v[1]+v[ind]
      else:
        sum+=2*v[0]+v[ind-1]+v[ind]
      ind -= 2
    if ind == 2:
      sum +=v [0]+v[1]+v[2]
    elif ind == 1:
      sum += v[1]
    print(sum)

    while len(v) > 3:
      if v[1]*2 < v[0]+v[len(v)-2]:
        print('{} {}'.format(v[0],v[1]))
        print(v[0])
        print('{} {}'.format(v[len(v)-2],v[len(v)-1]))
        print(v[1])
```

```
            else:
                print('{} {}'.format(v[0],v[len(v)-2]))
                print(v[0])
                print('{} {}'.format(v[0],v[len(v)-1]))
                print(v[0])
            v.remove(v[len(v)-1])
            v.remove(v[len(v)-1])
        if len(v) == 3:
            print('{} {}'.format(v[0],v[1]))
            print(v[0])
            print('{} {}'.format(v[0],v[2]))
        elif len(v) == 2:
            print('{} {}'.format(v[0],v[1]))



def main():
    n=int(sys.stdin.readline())

    for i in range(0,n):
        speed=[]
        s=sys.stdin.readline()
        number=int(sys.stdin.readline())
        for j in range(0,number):
            next = int(sys.stdin.readline())
            speed.append(next)
        bridge(speed)
        print()

if __name__ == '__main__':
    main()
```

Ez a megoldás az esetszétválasztás elvét követi. A program először megnézi, hogy az adott bemenet esetében 1, 2, vagy pedig 3 ember szeretne-e átkelni a hídon, és ha igen, akkor az elágaztató utasítás if és elif ágai egyből kiírják a megoldást. Ha több ember várakozik a híd előtt, akkor a program megvizsgálja a lehetséges átmeneteleket és kiszámolja, hogy melyik esetben gyorsabb az átkelés. Miután kiszámolta és kiírta a program az átkelések minimális költségét, újra lejátssza a folyamatot, ekkor a párok kombinációját kiíratva.

A program és a kód sokkal rövidebbé tehető, ha nem szedjük külön az eseteket, az átkelések sorrendjét pedig a költség kiszámítása során egy listában tároljuk. Ebben az esetben a programnak az átkelések sorrendjét is csak egyszer kell megvizsgálnia, második körben már elegendő a listában összegyűjtött lépéseket kiíratnunk.

```
import sys

def bridge(v):
  v = sorted(v)
  ind = len(v)-1
  sum=0
  res = []
  while len(v) > 3:
    if v[1]*2 < v[0]+v[ind-1]:
      sum+=v[0]+2*v[1]+v[ind]
      res.append('{} {}'.format(v[0],v[1]))
      res.append(v[0])
      res.append('{} {}'.format(v[len(v)-2],v[len(v)-1]))
      res.append(v[1])
    else:
      sum+=2*v[0]+v[ind-1]+v[ind]
      res.append('{} {}'.format(v[0],v[len(v)-2]))
      res.append(v[0])
      res.append('{} {}'.format(v[0],v[len(v)-1]))
      res.append(v[0])
    v.remove(v[len(v) - 1])
    v.remove(v[len(v) - 1])
      # ind -= 2
  if len(v) == 3:
    sum +=v [0]+v[1]+v[2]
    res.append('{} {}'.format(v[0],v[1]))
    res.append(v[0])
    res.append('{} {}'.format(v[0],v[2]))
  elif len(v) == 2:
    sum += v[1]
    res.append('{} {}'.format(v[0],v[1]))
  else:
    sum += v[0]
    res.append(v[0])
  print(sum)
  for i in res:
    print(i)
```

```
def main():
  n=int(sys.stdin.readline())

  for i in range(0,n):
    speed=[]
    s=sys.stdin.readline()
    number=int(sys.stdin.readline())
    for j in range(0,number):
      next = int(sys.stdin.readline())
      speed.append(next)
    bridge(speed)
    print()

if __name__ == '__main__':
  main()
```

Java nyelven az esetszétválasztással történő megoldás a következő:

```
import java.util.Arrays;
import java.util.Scanner;

public class bridge {

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = Integer.parseInt(sc.nextLine());
    for (int i = 0; i<n; i++) {
      String s = sc.nextLine();
      int number = Integer.parseInt(sc.nextLine());
      int[] speed = new int[number];
      for (int j=0; j<number; j++){
        int next = Integer.parseInt(sc.nextLine());
        speed[j] = next;
      }
      bridge(speed);
      System.out.println(System.lineSeparator());
    }
  }
```

```
public static void bridge(int[] v){
    Arrays.sort(v);
    if (v.length == 1) {
        System.out.println(v[0]);
        System.out.println(v[0]);
    }
    else if (v.length == 2) {
        System.out.println(v[1]);
        System.out.println(v[0]+" "+v[1]);
    }
    else if (v.length == 3) {
        System.out.println(v[0]+v[1]+v[2]);
        System.out.println(v[0]+" "+v[1]);
        System.out.println(v[0]+" "+v[2]);
    }
    else {
        int ind = v.length - 1;
        int sum = 0;
        while (ind > 2) {
            if (v[1]*2 < v[0]+v[ind-1]){
                sum += v[0]+2*v[1]+v[ind];
            }
            else {
                sum += 2*v[0]+v[ind-1]+v[ind];
            }
            ind -= 2;
        }
        if (ind == 2) {
            sum += v[0]+v[1]+v[2];
        }
        else if (ind == 1) {
            sum += v[1];
        }
        System.out.println(sum);

        ind = v.length - 1;
        while (ind > 2){
            if (v[1]*2 < v[0]+v[ind-1]){
                System.out.println(v[0]+" "+v[1]);
                System.out.println(v[0]);
                System.out.println(v[ind-1]+" "+v[ind]);
                System.out.println(v[1]);
            }
            else {
```

```
                System.out.println(v[0]+" "+v[ind-1]);
                System.out.println(v[0]);
                System.out.println(v[0]+" "+v[ind]);
                System.out.println(v[0]);
            }
            ind -= 2;
            if (ind == 2) {
                System.out.println(v[0]+" "+v[1]);
                System.out.println(v[0]);
                System.out.println(v[0]+" "+v[2]);
            }
            else if (ind == 1) {
                System.out.println(v[0]+" "+v[1]);
            }
        }
    }
  }
}
```

Az esetek összevonásával a Java nyelvű megoldás a következő:

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Scanner;

public class bridge {

  public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     int n = Integer.parseInt(sc.nextLine());
     for (int i = 0; i<n; i++) {
        String s = sc.nextLine();
        int number = Integer.parseInt(sc.nextLine());
        intr[] speed = new int[number];
        for (int j=0; j<number; j++){
           int next = Integer.parseInt(sc.nextLine());
           speed[j] = next;
        }
        bridge(speed);
        System.out.println(System.lineSeparator());
     }
  }
```

```
    public static void bridge(int[] v){
        Arrays.sort(v);
        StringBuilder sb = new StringBuilder();
        String line = System.lineSeparator();
        int ind = v.length - 1;
        int sum = 0;
        while (ind > 2) {
            if (v[1]*2 < v[0]+v[ind-1]){
                sum += v[0]+2*v[1]+v[ind];
                sb.append(v[0]+" "+v[1]+line);
                sb.append(v[0]+line);
                sb.append(v[ind-1]+" "+v[ind]+line);
                sb.append(v[1]+line);
            }
            else {
                sum += 2*v[0]+v[ind-1]+v[ind];
                sb.append(v[0]+" "+v[ind-1]+line);
                sb.append(v[0]+line);
                sb.append(v[0]+" "+v[ind]+line);
                sb.append(v[0]+line);
            }
            ind -= 2;
        }
        if (ind == 2) {
            sum += v[0]+v[1]+v[2];
            sb.append(v[0]+" "+v[1]+line);
            sb.append(v[0]+line);
            sb.append(v[0]+" "+v[2]+line);
        }
        else if (ind == 1) {
            sum += v[1];
            sb.append(v[0]+" "+v[1]+line);
        }
        else {
            sum += v[0];
            sb.append(v[0]+line);
        }
        System.out.println(sum);

        System.out.println(sb.toString());
    }
}
```

Leghosszabb közös részsorozat[11]

Két adott karaktersorozat esetén írd ki a két sorozat leghosszabb közös részsorozatának hosszát!
Az „abcdgh” és az „aedfhr” sorozatok leghosszabb közös részsorozata például az „adh” sorozat, amelynek hossza 3.

A bemenet specifikációja

A bemenet sorpárokból áll. A pár első sora tartalmazza az első sztringet, a második sora pedig a másodikat. Minden sztring külön sorban van, és legfeljebb 1000 karakterből áll.

A kimenet specifikációja

Minden bemeneti sorpárra egy sort kell a kimenetre írni, amely egyetlen egész számot tartalmaz a fent leírtaknak megfelelően.

A feladat Python nyelvű megoldása a következő:

```
import sys
import numpy as np

def lkr(n,k,m):
  nh=len(n)
  kh=len(k)

  if nh==0 or kh==0:
    if m[nh][kh]==-1:
      sum=0
      m[nh][kh]=sum
    else:
      sum= m[nh][kh]
  elif n[nh-1]== k[kh-1]:
    if m[nh][kh] == -1:
      sum= 1+lkr(n[:nh-1],k[:kh-1],m)
```

[11] https://progcont.hu/progcont/100128/?locale=hu&pid=10405

```
            m[nh][kh]=sum
        else:
            sum=m[nh][kh]
    else:
        if m[nh][kh] == -1:
            sum=max(lkr(n[:nh-1],k,m),lkr(n,k[:kh-1],m))
            m[nh][kh]=sum
        else:
            sum=m[nh][kh]
    return sum

def main():
    n=sys.stdin.readline()[:-1]

    while n:
        k=sys.stdin.readline()[:-1]
        m=np.random.randint(-1,0,(len(n)+1,len(k)+1))
        print(lkr(n,k,m))
        n = sys.stdin.readline()[:-1]

if __name__ == '__main__':
    main()
```

A megoldás Java nyelven:

```
import java.util.Scanner;

public class Lkr {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()){
            String n = sc.nextLine();
            String k = sc.nextLine();
            int[][] m = new int[n.length()+1][k.length()+1];
            for (int row = 0; row < n.length()+1; row++){
                for (int col = 0; col < k.length()+1; col++){
                    m[row][col] = -1;
                }
            }
            System.out.println(lkr(n,k,m));
        }
```

```
    }

    public static int lkr(String n, String k, int[][] m){
        int nh = n.length();
        int kh = k.length();
        int sum = 0;
        if (nh == 0 || kh == 0){
            if (m[nh][kh] == -1){
                sum = 0;
                m[nh][kh] = sum;
            }
            else {
                sum = m[nh][kh];
            }
        }
        else if (n.charAt(nh-1) == k.charAt(kh-1)){
            if (m[nh][kh] == -1){
                sum = 1 + lkr(n.substring(0,nh-1),k.substring(0,kh-1),m);
                m[nh][kh] = sum;
            }
            else {
                sum = m[nh][kh];
            }
        }
        else {
            if (m[nh][kh] == -1){
                sum = Math.max(lkr(n.substring(0,nh-1),k,m),lkr(n,k.substring(0,kh-
1),m));
                m[nh][kh] = sum;
            }
            else {
                sum = m[nh][kh];
            }
        }
        return sum;
    }
}
```

A feladatot rekurzív módon oldottuk meg. Egy kétdimenziós tömbben tároltuk a rekúrzívan meghívott függvény/metódus eredményeit, majd ha a bemenetben kapott stringek vizsgálatának végére értünk, a kétdimenziós tömb megfelelő indexén lévő értékkel tértünk vissza.