A JELADÓ-feladat elemzése

1. feladat

Vizsgálat

- **Beolvasás:** A **File.ReadAllLines** használata kényelmes, és mivel a feladat maximum 1000 sort rögzít, ez nem okoz memóriaproblémát.
- **Adatszerkezet:** A párhuzamos tömbök ***(ora[], perc[], stb.)*** használata megfelel a komplex elemek kizárásának és a feladat követelményeinek is.
- **Feltételezések:** Mivel a feladat szerint feltételezhetjük az adatok helyességét, az **int.Parse** és a fix indexelés **(sor[0]-sor[4])** biztonságos.
- **Apró észrevétel:** A mintamegoldásban használt **d://jel.txt** elérési út is működik (a .NET kezeli a dupla perjelet), bár a C#-ban hagyományosan a **@"d:\jel.txt"** vagy **"d:\\jel.txt"** használatos, de ez csak technikai megjegyzés.

Hatékonyság

A jelenlegi megoldás hatékonysága a megadott keretek között (1000 sor) **kiváló**.

- **Időbeli összetettség:** **O(n),** ahol **n** a sorok száma. Egyszer végigolvassa a fájlt, egyszer pedig végigmegy a sorokon.
- **Memóriakezelés:** A **ReadAllLines** a teljes fájlt egyszerre a memóriába tölti ***stringként***, majd a tömbökben is eltárolja az egészeket. 1000 sornál ez elhanyagolható (pár tíz kilobájt), de több millió sornál már nem lenne szerencsés.

Alternatív megoldások (Tömbökkel)

Ha még "nyersebb" és memóriatakarékosabb megoldást szeretnénk (elkerülve a teljes fájl ***stringként*** való tárolását), használhatunk **StreamReader**-t és fix méretű tömböket, mivel tudjuk a maximum korlátot (1000 sor)

```
int[] ora = new int[1000];
int[] perc = new int[1000];
int[] mp = new int[1000];
int[] x = new int[1000];
int[] y = new int[1000];
int n = 0;

using (StreamReader sr = new StreamReader("d://jel.txt"))
{
    while (!sr.EndOfStream && n < 1000)
    {
        string[] sor = sr.ReadLine().Split(' ');
        ora[n] = int.Parse(sor[0]);
        perc[n] = int.Parse(sor[1]);
        mp[n] = int.Parse(sor[2]);
        x[n] = int.Parse(sor[3]);
        y[n] = int.Parse(sor[4]);
        n++;
    }
}
```

```
StreamReader sr = new StreamReader("d://jel.txt");
int n = 0;

// Mivel tudjuk a maximumot, fix méretű tömböket használunk
int[] ora = new int[1000];
int[] perc = new int[1000];
int[] mp = new int[1000];
int[] x = new int[1000];
int[] y = new int[1000];

while (!sr.EndOfStream)
{
    string sor = sr.ReadLine();
    string[] adatok = sor.Split(' ');

    ora[n] = int.Parse(adatok[0]);
    perc[n] = int.Parse(adatok[1]);
    mp[n] = int.Parse(adatok[2]);
    x[n] = int.Parse(adatok[3]);
    y[n] = int.Parse(adatok[4]);
    n++;
}

sr.Close(); // Ez kritikus!
```

Előnye: Nem tárolja el a teljes fájlt egyetlen óriási **string[]** tömbben, csak soronként dolgozik.

Hátránya: Több gépelést igényel, és a **using** blokk vagy a manuális **Close()** miatt kicsit komplexebb a szerkezete.

Mit kell tudni erről a megoldásról?

- **A Close() fontossága:** Ha elmarad a **sr.Close(),** a program "rajta tartja a kezét" a fájlon. Ez azt jelentheti, hogy amíg a program fut, más alkalmazások nem tudják módosítani vagy törölni a fájlt, sőt, néha még olvasni sem.
- **Kockázat:** A **using** blokk azért jobb, mert ha a beolvasás közben hiba történik (például egy hibás adat miatt elszállna az **int.Parse**), a **using** akkor is automatikusan lezárja a fájlt. Manuális megoldásnál ilyenkor a **Close()** sorra sosem jut el a vezérlés, és a fájl nyitva marad a memóriában. Ez akár adatvesztéssel is járhat.
- **Feltételezés:** Mivel a feladatleírás kimondja, hogy feltételezhetjük az adatok helyességét és szabályosságát, a manuális **Close()** használata egy érettségi szintjén teljesen elfogadható és pontot érő megoldás.
- **Hatékonyság:** Ez a módszer **memóriatakarékosabb**, mint a **File.ReadAllLines**, mert nem olvassa be az egész fájlt egyszerre egy óriási **string**-tömbbe, hanem csak soronként dolgozik.

Ha nagyon precízek akarunk lenni **using** nélkül, akkor egy **try-finally** blokkba tehetnénk a lezárást, de ez már túllépne az általános keretrendszeren.

```
StreamReader sr = null;
try
{
    // Megpróbáljuk megnyitni a fájlt
    sr = new StreamReader("d://jel.txt");

    // Fájl feldolgozása (például a maximum 1000 sor beolvasása)
}
catch (Exception e)
{
    // Ide ugrik a vezérlés, ha hiba történik (pl. hiányzó fájl)
    Console.WriteLine("Hiba történt a fájl olvasásakor: " + e.Message);
}
finally
{
    // Ez a rész MINDENKÉPPEN lefut
    if (sr != null)
    {
        sr.Close();
        // Garantáljuk a fájl lezárását
    }
}
```

A blokkok feladata:

- **try**: Ebben a blokkban helyezzük el a "veszélyes" műveleteket. Ilyen a fájl megnyitása és a sorok számmá alakítása (**int.Parse**).
- **catch**: Ha a **try** blokkban bárhol hiba történik, a futás azonnal megszakad és ide ugrik. Itt kezelhetjük a hibát (pl. hibaüzenet kiírása), ahelyett, hogy a program hibaablakkal leállna.
- **finally**: Ez a legfontosabb rész az erőforrások (például fájlok) kezelésénél. Mivel ez a blokk akkor is lefut, ha minden rendben ment, és akkor is, ha hiba történt, ideális hely a **Close()** metódus meghívására.

Bár a feladatleírás szerint feltételezhetjük, hogy az adatok helyesek és a fájl rendelkezésre áll, éles környezetben ez a szerkezet teszi stabillá a szoftvert.

2. feladat

Vizsgálat:

- **Sorszámozás kezelése:** A feladat kiköti, hogy a sorszámozás 1-től indul. A kódban az **`int k = int.Parse(Console.ReadLine()) - 1;`** sor helyesen korrigálja ezt az eltolást, így a 0-tól indexelt tömbökből a megfelelő adatot nyeri ki.
- **Kiírás formátuma:** A kimenet megfelel az elvárásoknak: tartalmazza a feladat sorszámát, utal a tartalomra, és a koordinátákat a kért **`x=... y=...`** formában jeleníti meg.
- **Adatbevitel:** Megjelenik a kért üzenet arról, hogy milyen értéket vár a program a felhasználótól.
- **Egyszerűség:** Mivel a leírás szerint nem kell ellenőrizni a megadott adatok helyességét, az **`int.Parse`** és a közvetlen indexelés használata ebben a környezetben teljesen szabályos.

A **2. feladat** megoldása alapvetően megfelel az elvárásoknak, de van néhány pont, ahol egy valós programban – vagy egy szigorúbb környezetben – probléma adódhat.

Lehetséges problémák

- **Túlindexelés (`IndexOutOfRangeException`):** Ha a felhasználó olyan sorszámot ad meg, amely nagyobb, mint a beolvasott adatok száma (például 1500-at ír, miközben csak 1000 sorunk van), a program hibaüzenettel leáll.
- **Érvénytelen bemenet:** Ha a felhasználó nem számot, hanem szöveget ír be, az **`int.Parse`** metódus összeomlaszthatja a programot.
- **0 vagy negatív sorszám:** Mivel a sorszámozás 1-től indul, a 0 vagy negatív érték bevitele logikai hibát vagy tömbindex-hibát okoz.

A feladatleírás kifejezetten rögzíti, hogy a felhasználó által megadott adatok helyességét és érvényességét nem kell ellenőriznie. Így a fenti problémák kezelése technikailag nem elvárás, de a hiányuk korlátozza a kód robusztusságát.

Alternatív és hatékonyabb megoldások

Bár a feladat egyszerű, a kód olvashatósága és biztonsága növelhető az alábbi módszerekkel:

1. Biztonságos beolvasás (Range Check)

Még ha nem is kötelező a validáció, egy egyszerű feltétel megvédi a programot az összeomlástól:

```
Console.Write("Adja meg a jel sorszamat! ");
int sorszam = int.Parse(Console.ReadLine());
int k = sorszam - 1;

if (k >= 0 && k < n) // Ellenőrizzük, hogy a tartományon belül van-e
{
    Console.WriteLine($"x={x[k]} y={y[k]}");
}
else
{
    Console.WriteLine("Nincs ilyen sorszamu jel!");
}
```

2. String Interpoláció használata

Az **"x=" + x[k] + " y=" + y[k]** helyett modernebb és olvashatóbb a C# string interpolációja: **Console.WriteLine($"x={x[k]} y={y[k]}");** Ez elkerüli a sok + jelet és átláthatóbbá teszi a kimeneti formátumot.

3. Memóriahatékonyság

A jelenlegi megoldás párhuzamos tömböket használ **(x[], y[]).** Egy alternatív "tömbös" megoldás lehetne egyetlen kétdimenziós tömb használata (pl**. int[,] koordinatak = new int[1000, 2]**), ahol a **[k, 0]** az **x**, a **[k, 1]** pedig az **y** koordináta. Ez egy fokkal rendezettebbé teszi az összetartozó adatokat komplexebb típusok (***struct/class***) használata nélkül is.

3. feladat

A **3. feladat** lényege egy olyan függvény elkészítése, amely kiszámítja a két időpont között eltelt időt másodpercekben.

Vizsgálat

A kódban jelenleg szereplő függvény:

```
static int Eltelt(int o1, int p1, int m1, int o2, int p2, int m2)
{
    return (o2 - o1) * 3600 + (p2 - p1) * 60 + (m2 - m1);
}
```

Helyesség: A matematikai logika tökéletes. A különbséget tagonként számolja ki, ami ekvivalens azzal, mintha mindkét időpontot külön-külön másodpercre váltanád és kivonnád őket egymásból.

Paraméterezés: A feladat megengedte a tetszőleges paraméterátadást (például hat darab egész számot), így ez a megoldás teljesen szabályos.

Használat: A kódban a függvényt később a 4. és a 7. feladatnál is használjuk, így teljesítjük azt az előírást is, hogy legalább egy későbbi feladatnál fel kell használni.

Lehetséges probléma

Bár a feladat szerint az adatok egy napon belüliek és időrendben vannak, a függvényed csak akkor ad helyes (pozitív) eredményt, ha a második időpont valóban későbbi, mint az első. Ha éjfélen

átnyúló adatokat kellene kezelni (ahol az óra értéke "átfordul"), ez a formula módosításra szorulna, de a feladatleírás alapján itt ettől nem kell tartani.

Alternatív megoldások

1. **Teljesen másodperc alapú megközelítés:**

Sokan átláthatóbbnak tűnhet, ha először mindkét időpontot átváltjuk a nap elejétől eltelt összes másodpercre, majd ezeket vonjuk ki egymásból:

$$Eredmény = (o_2 \cdot 3600 + p_2 \cdot 60 + m_2) - (o_1 \cdot 3600 + p_1 \cdot 60 + m_1)$$

```
static int Eltelt(int o1, int p1, int m1, int o2, int p2, int m2)
{
    // Az első időpont összes másodperce
    int mp1 = o1 * 3600 + p1 * 60 + m1;

    // A második időpont összes másodperce
    int mp2 = o2 * 3600 + p2 * 60 + m2;

    // A két érték különbsége adja meg az eltelt időt
    return mp2 - mp1;
}
```

Miért jobb ez néha?

- **Átláthatóbb**: Világosan elkülönül a két időpont „abszolút” értéke, és csak a legvégén történik a kivonás.
- **Kevésbé hibaveszélyes**: Az eredeti megoldásnál negatív részeredmények keletkezhetnek (például ha a másodperc 10-ről 05-re vált), ami bár matematikailag a végén kijön, nehezebb fejben követni vagy debuggolni.
- **Bővíthetőség**: Ha később éjfélen átnyúló időpontokat kellene kezelni, ennél a logikánál elég lenne egyetlen if (eredmeny < 0) eredmeny += 86400; feltételt betenni a végére.

2. **Tömbök használata:** Mivel párhuzamos tömbökben tároljuk az adatokat, a függvény kaphatna két indexet és a tömböket, vagy csak két kisebb tömböt (pl. **int[] t1, int[] t2**), ahogy azt a feladat is javasolta lehetőségként. Ez akkor lehet hasznos, ha nem akarjuk a hívás helyén egyesével kiirogatni a koordinátákat vagy időadatokat.
   A függvény nem 6 darab int értéket kap, hanem a két vizsgálandó jel sorszámát (indexét) és magukat az adattömböket.

```
static int Eltelt(int i, int j, int[] ora, int[] perc, int[] mp)
{
    // Kiszámoljuk az i-edik indexű időpontot másodpercben
    int ido1 = ora[i] * 3600 + perc[i] * 60 + mp[i];

    // Kiszámoljuk a j-edik indexű időpontot másodpercben
    int ido2 = ora[j] * 3600 + perc[j] * 60 + mp[j];

    return ido2 - ido1;
}
```

A hívás sokkal rövidebb lesz, mert csak az indexeket (első jel: 0, utolsó jel: n - 1) és a tömbök nevét kell átadni:

```
// 4. feladat: az első és az utolsó észlelés között eltelt idő
int ido = Eltelt(0, n - 1, ora, perc, mp);
```

- **Kevesebb gépelés a hívásnál:** Nem kell hatszor beírni a tömbök neveit és indexeit a zárójelbe.
- **Tisztább kód:** A függvényen belül dől el, hogy melyik adathoz nyúlunk hozzá, a Main metódusban csak azt mondjuk meg, hogy „számold ki az időt az első és az utolsó között".
- **Rugalmasság:** A feladat szövege kifejezetten említi, hogy a paraméterátadás módja tetszőleges, így ez a megoldás is teljes mértékben elfogadható.

3. **Beépített típus (TimeSpan):** C# környezetben létezik a **TimeSpan** típus is, de egy ilyen vizsgafeladatnál talán a manuális számítás jobban elvárt, mert ez mutatja meg az algoritmus szintű gondolkodást. De azért, hogyan is működik/működne?

A .NET keretrendszer **TimeSpan** típusa pont időtartamok és időkülönbségek kezelésére lett kitalálva. Ezzel elkerülhető a manuális szorzás (3600, 60).

```
static int Eltelt(int o1, int p1, int m1, int o2, int p2, int m2)
{
    // Létrehozunk két TimeSpan objektumot az óra, perc, másodperc adatokból
    TimeSpan t1 = new TimeSpan(o1, p1, m1);
    TimeSpan t2 = new TimeSpan(o2, p2, m2);

    // A két időpont kivonásával egy újabb TimeSpan-t kapunk,
    // aminek a TotalSeconds tulajdonsága megadja a különbséget másodpercben.
    return (int)(t2 - t1).TotalSeconds;
}
```

Miért érdekes ez a megoldás?

- **Beépített logika:** Nem nekünk kell kiszámolnunk, hogy egy óra hány másodperc, a keretrendszer tudja.
- **Precizitás:** A **TotalSeconds** ***double*** értéket ad vissza (tizedmásodpercek miatt), de mivel itt egész számokkal dolgozunk, nyugodtan kényszeríthetjük (**cast**) ***int*** típusra.
- **Paraméterezés:** Megfelel a feladatnak, hiszen tetszőleges módon adhatjuk át a paramétereket.
- Fontos megjegyzés: Érettségin a javítókulcs általában a matematikai megoldást (szorzásokat) várja el, de a **TimeSpan** használata is abszolút szabályos és elegáns programozói megoldás.

4. feladat

A **4. feladat** célja az első és az utolsó észlelés közötti időtartam meghatározása és formázott kiírása.

Logika: Mindez használja az előzőleg megírt **Eltelt** függvényt az első (**0. index**) és az utolsó (**n-1. index**) adatsor között.

Váltás: A másodpercek visszaváltása órára, percre és másodpercre a maradékos osztás segítségével a mintamegoldásban matematikai szempontból pontos.

Formázás: A **{0:D2}** formátumkód használata kiváló választás, mert biztosítja a vezető nullákat (pl. 09:05:01), ami megfelel az óra:perc:mp kijelzési hagyományoknak és a mintának.

Alternatív megoldások és hatékonyság

1. **A TimeSpan használata a kiíráshoz**: Ha már megvan az összes másodperc értéke (**ido**), a visszaalakítást rábízhatjuk a .NET-re is, ami rövidebb kódot eredményezhet:

```
TimeSpan t = TimeSpan.FromSeconds(ido);
Console.WriteLine($"Idotartam: {t.Hours:D2}:{t.Minutes:D2}:{t.Seconds:D2}");
```

   Ez olvashatóbb, mert nem kell kézzel bűvészkedni a 3600-as és 60-as számokkal.

2. **Egyszerűsített kiírás (ha nem kritikus a vezető nulla)**: Bár a mintában szerepel a vezető nulla, ha a feladat nem írja elő szigorúan a két számjegyet, a legegyszerűbb összefűzés is működne: **Console.WriteLine($"Idotartam: {h}:{m}:{s}");**

Lehetséges probléma

A feladatleírás nem említi, de ha az időtartam meghaladná a 24 órát, a **h = ido / 3600** továbbra is jól működne (kiírná pl. a 26 órát), míg bizonyos beépített formátumok (mint a **t.ToString(@"hh\:mm\:ss"))** ilyenkor újraindulnának nulláról. A manuális osztás tehát ebből a szempontból **biztonságosabb**.

Egyéb észrevétel

Kötelező függvényhasználat: Az előírás szerint az **Eltelt** függvényt legalább egy feladatban (például itt) kötelező felhasználni.

Adatok köre: Mivel a forrásfájl pontosan egy nap adatait tartalmazza időrendben, az „első” és „utolsó” észlelés mindig a fájl legelső (**0. index**) és legutolsó (**n-1. index**) sorát jelenti.

Időrend: A feladat rögzíti, hogy az adatok időrendben vannak, így nem kell rendezéssel vagy azzal foglalkozni, hogy melyik időpont volt korábban.

5. feladat

Az **5. feladat** célja a jeladó mozgási területét lefedő legkisebb, tengelyekkel párhuzamos oldalú téglalap (befoglaló téglalap) meghatározása. Ehhez a rögzített koordináták közül a legkisebb és legnagyobb **x** és **y** értékeket kell megkeresni.

Algoritmus: A kód szélsőérték-keresés logikája hibátlan. Az első elemre való inicializálás, majd a többi elemen való végighaladás a legbiztosabb módszer.

Hatékonyság: A megoldás ***O(n)*** időbeli összetettségű, ami a lehető leggyorsabb, hiszen minden koordinátát pontosan egyszer kell megvizsgálni.

Kimenet: A kiírt szöveg **(„Bal also... jobb felso...")** formátuma és a koordináták sorrendje pontosan követi a példát.

Lehetséges problémák és finomítások

- **Üres fájl esete**: Ha a fájl üres lenne (**n=0**), az **x[0]** hivatkozás hibát dobna. Bár a feladat szerint az adatok helyesek, egy **if (n > 0)** ellenőrzés növelné a biztonságot.
- **Több koordináta**: Mivel a koordináták -10,000 és 10,000 közé esnek, az **int** típus bőven elegendő a tárolásukra.
- **Hibás inicializálás:** Ha a **minX** vagy **minY** értékét nem az első elemre, hanem fixen 0-ra állítanád, az hibát okozna, ha minden koordináta pozitív (vagy negatív) lenne. Ez a megoldás (**x[0]**) ezt helyesen elkerüli.
- **Koordináta-tartomány:** A feladat szerint a koordináták -10 000 és 10 000 között változhatnak. Emiatt fontos, hogy a tároló változók (pl. **int**) képesek legyenek kezelni ezeket az értékeket.
- **A "Bal alsó" és "Jobb felső" definíciója:** A téglalap bal alsó sarkát a (**min X**, **min Y**), a jobb felsőt pedig a (**max X**, **max Y**) pontok alkotják. Ha véletlenül összekeveredne egy minimum és egy maximum, a téglalap nem a legkisebb befoglaló idom lenne.

Alternatív megoldások

Tömbök esetén, az alábbi lehetőségek vannak a kód tömörítésére vagy átláthatóbbá tételére:

Megoldás	**Leírás**	**Előny/Hátrány**
Math metódusok	**minX = Math.Min(minX, x[i]);**	Olvashatóbb, de funkcionálisan azonos az if-es megoldással.
Beépített rendezés	**Array.Sort(x);** majd a **0.** és az **n-1.** elem.	Nagyon kevés kód, de lassabb **(O(n log n)** a sima **O(n)** helyett).
Szélsőérték konstansok	**minX = int.MaxValue;**	Biztonságos kezdőérték, ha nem akarjuk az első elemet használni az indításhoz.

Példa a Math.Min/Max használatára:

Ez a módszer elkerüli a sok elágazást, és tisztább vizuálisan:

```
for (int i = 1; i < n; i++)
{
    minX = Math.Min(minX, x[i]);
    maxX = Math.Max(maxX, x[i]);
    minY = Math.Min(minY, y[i]);
    maxY = Math.Max(maxY, y[i]);
}
```

6. feladat

A **6. feladat** lényege a jeladó által megtett összes út hosszának kiszámítása, feltételezve, hogy a pontok között egyenes vonalban haladt.

A kódban szereplő megoldás matematikai és programozási szempontból is pontos:

- **Logika**: A ciklus **n - 1**-ig fut, ami helyes, mert az utolsó elemnek már nincs „következő” párja, amivel távolságot számolhatnánk.
- **Képlet**: A távolságot a $\sqrt{(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2}$ képlettel számoljuk ki, ami megfelel az előírásnak.
- **Típusválasztás**: A **double** használata az **ossz** változóhoz szükséges a tizedesjegyek miatt.
- **Kiírás**: A **{0:0.000}** formátumkód biztosítja a kért három tizedesnyi pontosságot.

Lehetséges problémák és alternatívák

- **Hatékonyság**: A **dx * dx** használata sokkal gyorsabb, mint a **Math.Pow(dx, 2)**, így a megoldás ezen a téren is optimális.
- **Túlcsordulás**: Bár a koordináták abszolút értéke akár 10 000 is lehet, a különbségük négyzete (20 000^2 = 400 000 000) még bőven belefér az **int** típusba (ami ~2 milliárdig kezel számokat), így nem kell tartani túlcsordulástól a gyökvonás előtt.

Alternatív megközelítés (Tiszta tömbökkel)

Ha még olvashatóbbá szeretnénk tenni, kiszervezhetjük a távolságszámítást egy külön függvénybe, hasonlóan az **Eltelt** függvényhez:

```
static double Tavolsag(int x1, int y1, int x2, int y2)
{
    return Math.Sqrt(Math.Pow(x1 - x2, 2) + Math.Pow(y1 - y2, 2));
}
```

Ez tisztábbá tenné a főciklust, de a jelenlegi megoldás is teljesen korrekt és hatékony.

7. fejezet

A 7. feladat az állat mozgása során keletkezett **adatvételi hibákat** hivatott naplózni a ***kimaradt.txt*** fájlba.

1. A hiba detektálásának szabályai

A jeladó két esetben küld jelet, amit a vevőnek rögzítenie kell:

- **Időalapú:** Ha az utolsó jel óta eltelt **5 perc** (300 másodperc).
- **Koordináta-alapú:** Ha bármelyik koordináta (x vagy y) változása elérte a **10 egységet**.

Ha két rögzített pont között ennél nagyobb az eltérés, akkor kimaradás történt.

2. A kimaradt jelek számának kiszámítása

Ez a legkritikusabb rész, ahol a példák segítenek a pontos képlet meghatározásában:

- **Koordináta-eltérés:** A mintaértékek szerint ha a koordináta eltérése **29 egység**, az legalább **2 vétel** kimaradását jelenti.
    - *Matematikailag:* Ha 10 egységenként jön jel, akkor a 29-ben a 10 kétszer van meg maradékkal.
    - *Képlet:* **max_elmozdulás / 10**. (Ha az elmozdulás 29, akkor 29/10=2).
    - *Pontosítás:* Ha az elmozdulás pontosan 20, az is 2 jel lenne, de a 20. egységnél pont esedékes egy jel, tehát a *kimaradás* csak akkor több, ha túlléptük a határt. A legbiztosabb képlet: **(max_eltérés - 1) / 10**.
- **Időeltérés:** Ha 5 percnél több idő telt el, de a 10 percet nem haladta meg, az legalább **1 vétel** kimaradása.
    - *Matematikailag:* Ha 300 másodpercenként jön jel, akkor egy 305 másodperces résnél 1 jel maradt ki.
    - *Képlet:* a fentiek alapján **(eltelt_idő - 1) / 300**.

3. Összeférhetetlenség kezelése

Gyakori, hogy mind az idő, mind a koordináta alapján kimaradást észlelünk.

- A szabály: Mindig a **nagyobb értéket** kell a fájlba írni.
- Ha az értékek egyenlőek, bármelyik ok (idő vagy koordináta) kiírható.

4. A kimeneti formátum (kimaradt.txt)

A fájlnak pontosan követnie kell a mintát:

- Az adatsor elején az **észlelés időpontja** (óra perc mp) szerepeljen (az a pont, amely előtt a hiányt észleltük).
- Ezt követi a hiba oka: időeltérés vagy koordináta-eltérés.
- A sor végén a **kimaradt jelek száma**.

Lehetséges problémák (Ahol elcsúszhat a logika)

- **A "határesetek" hibás kezelése:** A forrás szerint a jeladó jelet küld, ha az elmozdulás **elérte** a 10 egységet. Ez azt jelenti, hogy a pontosan 10 egységnyi elmozdulás még **nem** kimaradás, hanem egy szabályos jel.
    - Ugyanez igaz az időre: az 5 perc (300 mp) leteltekor küldött jel szabályos.
    - **Probléma:** Ha sima egész osztást használunk **(diff / 10),** akkor a 10-re 1-et kapunk, ami azt sugallná, hogy 1 jel kimaradt, pedig az maga a rögzített jel. Ezért kell a korrekció: **(diff - 1) / 10**.
- **A prioritási szabály elvétése:**
    - A feladat előírja: ha az idő- és koordináta-eltérésből is adódik jelkimaradás, a **nagyobbat** kell kiírni.
    - **Probléma:** Ha két külön **if**-et használunk és mindkettő kiír valamit a fájlba, az hibás, mert egy észleléshez csak egy bejegyzés tartozhat a ***kimaradt.txt***-ben.
- **Fájlfolyam nyitva maradása:**
    - Ha a ciklusban hiba történik (pl. váratlan adat), és nem **using** blokkot **vagy try-finally**-t használunk, a ***kimaradt.txt*** zárolva maradhat, vagy a puffer tartalma nem íródik ki a lemezre.

Alternatív megoldások a megvalósításra

Mivel tömbökkel dolgozunk, a struktúra kötött, de a logika tálalása változhat:

1. "Szelektív" változók használata (Tisztább kód)

Ahelyett, hogy bonyolult ***if-else*** ágakat írnunk a **StreamWriter**-be, érdemes előre kiszámolni mindent:

```
int kIdo = (dt - 1) / 300;
int kCoord = (Math.Max(dx, dy) - 1) / 10;

int maxKimaradt = Math.Max(kIdo, kCoord); //
string hibaOk = kIdo >= kCoord ? "idoelteres" : "koordinata-elteres"; //

if (maxKimaradt > 0) {
    sw.WriteLine($"{ora[i]} {perc[i]} {mp[i]} {hibaOk} {maxKimaradt}");
}
```

Előnye: Könnyen olvasható, és a prioritási szabály (idő vs koordináta) egyetlen sorban eldől.

2. Logikai elágazás (Hatékonyság)

Ha a teljesítmény a cél, elkerülhetjük a felesleges számításokat:

```
if (dt > 300 || dx > 10 || dy > 10)
{
    int kIdo = (dt - 1) / 300;
    int kCoord = (Math.Max(dx, dy) - 1) / 10;

    // Itt dől el a fájlba írás...
}
```

Mit csinál ez az 5 sor?

1. **A feltétel (if):** Ez a "szűrő". Csak akkor foglalkozunk a kimaradt jelekkel, ha az idő több mint **300 mp** (5 perc), **VAGY** ha valamelyik koordináta elmozdulása elérte a **10 egységet**. Ha egyik sem teljesül, nincs hiba, megyünk a következő adatra.
2. **Az időalapú számítás (kIdo):**
    - Itt számoljuk ki, hányszor telt le az 5 perc a két vétel között.
    - **Miért van ott a -1?** Mert ha pontosan 300 másodperc telt el, az osztás eredménye 1 lenne, de a feladat szerint 300 mp-nél még **nincs** kimaradás (akkor jön a szabályos jel). A -1 miatt a számítás csak 301-nél fog 1-et adni: (301-1) / 300 = 1.
3. **A koordináta-alapú számítás (kCoord):**
    - A **Math.Max(dx, dy)** kiválasztja, hogy az X vagy az Y irányú elmozdulás volt-e a nagyobb (mivel a feladat szerint bármelyik elérheti a 10-et, a nagyobbat kell nézni).
    - A **-1** itt is ugyanazt a trükköt csinálja: pontosan 10 egység elmozdulásnál még nem jelez hibát, de 11-nél már igen: (11-1) / 10 = 1.

Eset	Eltérés	Számítás ([...])	Kimaradt jelek
Példa 1	29 egység	(29 - 1) / 10	**2**
Példa 2	305 mp	(305 - 1) / 300	**1**
Határeset	10 egység	(10 - 1) / 10	**0** (nincs kimaradás)

A feladat kiköti, hogy ha az idő- és koordináta-eltérésből is adódik jelkimaradás, a **nagyobb értéket** kell kiírni. Az általunk választott szerkezetben ezt egy egyszerű **Math.Max** vagy egy **if-else** ág segítségével tudjuk véglegesíteni a fájlba írás előtt.

```
// Példa a prioritás kezelésére az if-en belül
int maxKimaradt = Math.Max(kIdo, kCoord);
if (kIdo >= kCoord) {
    // időeltérés kiírása sw.WriteLine-nal...
} else {
    // koordináta-eltérés kiírása...
}
```

A programban felhasznált legfontosabb algoritmusok:

1. Beolvasás és adattárolás

- **A feladat:** Be kell olvasni a **jel.txt** állományt.
- **Kötelező elem:** Az adatok sorrendjének megőrzése (időrend).
- **Buktató:** Az adatok száma nem ismert előre (csak a maximum: 1000). Ezért fontos egy adatszam változóban tárolni, hány sort olvastunk be ténylegesen.
- **Trükk:** Ha nem használunk saját típust (struktúrát/osztályt), akkor 5 külön tömböt kell kezelnünk. Itt nagyon figyeljünk, hogy azonos index (i) tartozzon minden adathoz!
- **Alternatíva:** Használhatnánk **struct**-ot vagy **class**-t is, de a párhuzamos tömbök a legegyszerűbbek.

2. Adott jel sorszámának lekérdezése

- **A feladat:** Bekérni egy sorszámot és kiírni az X, Y koordinátát.
- **Buktató (Indexelés):** Ez a leggyakoribb hiba! A felhasználó „emberi” sorszámot ad meg (1, 2, 3...), de a programozásban a tömbök **0-tól** kezdődnek. Tehát a sorszam - 1 indexet kell használni.
- **Trükk:** Mindig ellenőrizzük a kiírás formátumát (pl. x=... y=...), pontosan kövessük a példát!

3. Az Eltelt függvény (A feladat szíve)

- **A feladat:** Két időpont közötti különbség másodpercben.
- **Didaktikai tanács:** Soha ne próbáljunk meg órákat és perceket kivonni egymásból! Mindent váltsunk át a legkisebb egységre (másodperc).
- **Képlet:** (óra * 3600) + (perc * 60) + másodperc.
- **Trükk:** A függvény megkönnyíti a 4. és a 7. feladatot is. Ez a "modularitás" elve: egyszer írjuk meg, de többször használjuk.

4. Időtartam kiírása (Visszaváltás)

- **A feladat:** Az első és utolsó jel között eltelt idő ó:p:mp formátumban.
- **Algoritmus:** Maradékos osztás (%) és egész osztás (/).
- **Példa:** 3665 mp = 1 óra (3665 / 3600), 1 perc (65 / 60), 5 mp (65 % 60).

5. Szélsőérték keresés (Befoglaló téglalap)

- **A feladat:** **MinX**, **MaxX**, **MinY**, **MaxY** megkeresése.
- **Kötelező elem:** A téglalap két szemközti sarkát kell megadni.
- **Buktató:** Sokan a 0-t állítják be kezdőértéknek a keresésnél. Ez hiba, ha minden koordináta negatív! Mindig az **első elem** (**tomb[0]**) legyen a kezdőérték.

6. Elmozdulások összege (Távolság)

- **A feladat:** Pitagorasz-tétel alkalmazása minden szomszédos pontpárra.
- **Trükk:** A ciklus csak adatszam - 1-ig mehet, mert az utolsó elemnek már nincs "következő" párja (**IndexOutOfRangeException** veszélye!).
- **Matematika:** **Math.Sqrt(dx*dx + dy*dy).** A **Math.Pow** lassabb, de olvashatóbb.
- **Formázás:** A F3 vagy N3 formátumkód biztosítja a 3 tizedesjegyet.

7. Jelkimaradás (A legnehezebb rész)

- **A feladat:** Naplózni a hiányzó jeleket egy fájlba.
- **Logikai bukta:** A feladat azt kéri, hogy nézzük meg, az időeltérés **VAGY** a távolság alapján hány jel maradt ki.
    - Időnél: ha több mint 5 perc (300 mp) telt el. Ha 1000 mp telt el, az 1000 / 300 = 3,33, tehát legalább 3 jel maradt ki. A pontos képlet: (különbség - 1) / 300.
    - Koordinátánál: ha bármelyik irányban > 10 az elmozdulás. Képlet: (**max_eltérés - 1) / 10**.
- **Kötelező elem:** Ha mindkettő alapján van kimaradás, a **nagyobbat** kell kiírni. Ezt egy **Math.Max()** vagy egy egyszerű **if** megoldja.
- **Fájlba írás:** Figyelj, hogy a fájl neve pontosan kimaradt.txt legyen.

Feladat	Algoritmus	Mire figyelj?
1.	Beolvasás	**Split(' '), int.Parse()**
2.	Kiválasztás	**index = sorszam - 1**
3.	Transzformáció	Összes másodperc képlet
4.	Maradékos osztás	**/ 3600, % 3600 / 60, % 60**
5.	Szélsőérték keresés	Kezdőérték a **tomb[0]** legyen!

Feladat	Algoritmus	Mire figyelj?
6.	Összegzés	**i** és **i+1** elemek távolsága
7.	Összetett feltétel	**StreamWriter**, **Math.Max**, egész osztás

1. Sorozatszámítás (Összegzés) tétele

Ezt a tételt akkor használjuk, amikor egy sorozat elemeit valamilyen szempont szerint összesítenünk kell.

- **A programban:** A **6. feladatnál**, ahol az elmozdulások (távolságok) összegét számoljuk ki. Egy **tavolsag** nevű változóban gyűjtjük össze az egymást követő pontok távolságát.
- **Működése:** szumma = szumma + uj_ertek

2. Minimum- és maximumkiválasztás tétele

Ez az algoritmus egy adatsor legkisebb és legnagyobb elemét keresi meg.

- **A programban:** Az **5. feladatnál**, ahol a jeladó által bejárt terület határaira (a befoglaló téglalapra) vagyunk kíváncsiak. Külön-külön megkeressük az x és y koordináták minimumát és maximumát.
- **Működése:** Feltételezzük, hogy az első elem a legkisebb/legnagyobb, majd végigmenve a tömbön frissítjük az értéket, ha találunk nála kisebbet/nagyobbat.

3. Megszámlálás tétele (Módosított változat)

Ezt a tételt akkor használjuk, ha meg akarjuk tudni, hány elem felel meg egy feltételnek.

- **A programban:** A **7. feladatban** közvetve jelenik meg, amikor kiszámoljuk, hogy egy-egy észlelés között hány jel maradt ki (kimaradtIdo, kimaradtKoord). Bár nem a teljes sorozatra számolunk egy végeredményt, az elv ugyanaz: egy feltétel (idő- vagy távolságküszöb) alapján határozunk meg egy mennyiséget.

4. Eldöntés és Kiválasztás (Keresés)

Bár a program minden adaton végigmegy, a **2. feladat** egyfajta "közvetlen elérésű" kiválasztás: egy adott sorszámú (indexű) elemet kell megjelenítenünk a tömbből.

5. Elemi algoritmusok: Transzformáció és Konverzió

Bár nem "tételként" emlegetjük, a program fontos részét képezik:

- **Időkonverzió:** Az óra:perc:másodperc formátum átváltása összes másodpercre. Ez elengedhetetlen az időbeli különbségek egyszerű kiszámításához.
- **Koordinátageometria:** A két pont közötti távolság kiszámítása a koordináták alapján (Pitagorasz-tétel alkalmazása).

6. Szekvenciális feldolgozás (Fájlkezelés)

A program alapja a fájlban tárolt adatok soronkénti feldolgozása. Ez magában foglalja a **szétvágást (Split)** és a szöveges adatok **számmá alakítását (Parse)**, ami a bemeneti adatok előkészítésének (***input sanitization***) alapvető lépése.

Összegzés a kód szerkezete alapján: A megoldás egy klasszikus **szekvenciális algoritmus**, amely a beolvasás után több különálló lépésben (feladatonként) dolgozza fel ugyanazt az adathalmazt, amit memóriában (tömbökben) tárolunk.

Egyéb elemek

1. A moduláris felépítés (Függvényhasználat)

A feladat **kötelezően előírja** az **Eltelt** függvény elkészítését. Ez nem csak egy ajánlás.

- **Miért fontos?** Megmutatja, hogy tudunk-e paramétereket átadni és visszatérési értéket kezelni.
- **Trükk:** Sokan elfelejtik, hogy a függvénynek int típust kell visszaadnia. Ha csak kiíratnánk az eredményt a függvényen belül, az pontlevonással járna.
- **Didaktikai tanács:** A függvény legyen "tiszta", azaz ne használjon globális változókat, csak azt, amit paraméterként megkap.

2. Formázott kiírás (String Interpolation)

A modern C# (6.0+) egyik legszebb eleme a $ jellel kezdődő string.

- **Példa: `Console.WriteLine($"x={x[i]} y={y[i]}");`**
- **Miért jobb?** Sokkal olvashatóbb, mint az összeadogatás **("x=" + x[i] + ...)** vagy a **{0}, {1}** formátum. Az érettségin a pontosság a lényeg, és ez segít elkerülni a hiányzó szóközöket.

3. A 7. feladat „Határeset” logikája

Ez a rész választja el a kiváló programozót az átlagostól. A feladat azt mondja: *„legalább hány jel maradt ki”*.

- **A matematikai buktató:** Ha két jel között 601 másodperc telik el, az hány kimaradt jel?
    - 601 / 300 = 2,003.
    - Mivel 5 percenként (300 mp) *kell* jelet küldenie, itt 2 jel maradt ki (a 300. és a 600. másodpercnél).
    - A képlet: **(eltelt_idő - 1) / 300**. Azért kell a -1, mert ha pontosan 300 mp telt el, az még pont szabályos, nem maradt ki semmi.
- **Ugyanez koordinátánál: (eltelt_távolság - 1) / 10**.

4. Erőforrás-kezelés (Fájl lezárása)

Bár a modern operációs rendszerek sokszor lezárják a fájlokat a program végén, az érettségin ez kötelező pont.

- **A using kulcsszó:** Ez a legbiztonságosabb.
- **Hiba:** Ha elmarad a lezárás (Close()), a fájl tartalma üres maradhat, mert a puffer nem ürül ki.

5. Beolvasási stratégia: Split és Parse

A ***jel.txt*** soraiban szóközök választják el az adatokat.

- **A folyamat:** 1. Beolvassuk a sort egy stringbe. 2. **Split(' ')** metódussal daraboljuk (ez egy **string** tömböt ad). 3. **int.Parse()**-al minden darabot számmá alakítunk.
- **Veszély:** Ha véletlenül két szóköz van az adatok között, a sima **Split(' ')** üres stringeket is eredményezhet.
  Biztonságosabb a **Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries)** használata, bár érettségi fájloknál ritka az ilyen típusú probléma.

6. Algoritmusok vizualizációja

A program logikai váza alapvető vezérlési szerkezetekre épül. Érdemes fejben (vagy vázlatban) látni a folyamatot:

7. Tiszta kód alapelvek egyszerű eszközökkel

- **Névadás:** Ne a, b, c tömböket használjunk. A **koordinataX, percek** nevek segítenek, hogy ne keverjük össze, melyik tömbben mi van a 6. feladat bonyolultabb képleténél.
- **Kommentelés:** Érdemes a feladatszámokat kommentként odaírni a kódba. Ez segít a javító tanárnak, és nekünk is, ha vissza kell néznünk valamit.

Összefoglalva: A legfontosabb "maradék" elem a **pontos típuskonverzió** (egész osztás vs. lebegőpontos számítás a távolságnál) és a **határértékek (0. index és utolsó-1 index)** kezelése.