Féléves Feladat

A feladat megoldásaként a teljes solution mappát betömörítve a Moodle rendszerben kell leadni. A beadandó solution elnevezése a féléves feladat azonosítója és a saját neptunkódja legyen alulvonással elválasztva, nagybetűkkel: **AZONOSÍTÓ_NEPTUNKOD.zip**.

A feladat akkor kerül elfogadásra, ha a hallgató azt személyesen megvédte, aminek előfeltétele a kód szintaktikai és unit teszteken való megfelelése.

A feladattal kapcsolatos további információk az utolsó oldalon találhatók (ezen ismeretek hiányából adódó reklamációt nem fogadunk el!).

Egy sokemeletes házban szokatlan módon üzemeltetik a liftet. A lift az első szintről indult és mindig felmegy a legfelső szintre, majd visszetér az első szintre. Menet közben megáll minden olyan szinten, amelyik úti célja valamelyik liftben tartózkodó utasnak. Hasonlóan, olyan szinten is megáll, ahonnan utazni szándékozik valaki az aktuális irányban, feltéve, hogy még befér a liftbe (figyelembe véve az adott szinten kiszállókat).

Készítsen programot, amely kiszámítja, hogy legkevesebb hány menet szükséges ahhoz, hogy minden várakozó embert elszállítson a lift.

Bemenet

Az alkalmazáshoz bemeneti fájlként egy *input.txt* fájl tartozik, amit a *.exe mellől kell beolvasni:

- a fájl első sora két egész számot tartalmaz szóközzel elválasztva:
  - N - az épület szintjeinek száma
  - K - a lift kapacitása
- a következő N sor tartalmazza, az egyes szinteken várakozó embereket szóközzel elválasztva, ahol az értékek azt jelölik, hogy arról a szintről melyik szintre szeretnének utazni az emberek

Kimenet

Az alkalmazásnak az eredményt egy *output.txt* fájlba kell elmentenie a következők szerint:

- a fájlba egyetlen értéket kell írni, a legkevesebb menetek számát, amely az összes ember elszállításához szükséges

Megkötések

- $2 \leq N \leq 100$
- $1 \leq K \leq 10$
- $1 \leq$ *emberek száma a szinten* ≤ 200
- 1 menet = egyszer felmegy, majd lejön a lift
- minden szinten legalább egy ember szeretne utazni
- azonos szintre nem utazik senki
- adott szintről ugyan arra a szintre több ember is utazhat
- a bemeneti fájl minden esetben az `input.txt`; az alkalmazás ne akarjon beolvasni további fájlokat (*például:* `input2.txt`)
- az eredményt minden esetben az `output.txt` nevű fájlba mentse el
- az alkalmazás nem írhat ki és olvashat be semmit a *Console*-ról

Példa

input.txt

```
6 2
2 3 2
1 3
1 2
2 5
3 6 2
1 2 3
```

output.txt

```
3
```

Értelmezés

A bemeneti fájl első sora alapján az épület $N = 6$ emelet magas, ahol a liftbe egyszerre $K = 2$-en tartózkodhatnak.

A további $N = 6$ sor tartalmazza, hogy hány ember várakozik a szinten, és hogy ők hova szeretnének utazni, például az első szinten lévő 3 ember rendre a második, harmadik és a második emeletre.

- A lift az első menetben:
  - 1. szint: beszáll 2 ember a liftbe, akik a 2. és 3. emeletre utaznak
  - 2. szint: kiszáll 1 ember, beszáll aki a 3. utazik
  - 3. szint: kiszáll 2 ember, felfelé további emberek nem utaznak, így nem száll be senki
  - 4. szint: beszáll 1 ember, aki az 5. emeletre utazik
  - 5. szint: kiszáll 1 ember, beszáll az aki a 6. emeletre utazik
  - 6. szint: kiszáll 1 ember, beszáll 2 ember, akik az 1. és 2. emeletre utaznak
  - ... mivel tele a lift és nincs kiszállás, a 2. emelet lesz érdekes
  - 2. szint: kiszáll 1 ember, beszáll 1 ember, aki az 1. emeletre utazik
  - 1. szint: kiszáll 1 ember
- Végetért az első menet, a szinteken az alábbi emberek maradnak:
  - 1. szint: 2
  - 2. szint: -
  - 3. szint: 1 2
  - 4. szint: 2
  - 5. szint: 3 2
  - 6. szint: 3
- A lift a második menetben:
  - 1. szint: beszáll 1 ember a liftbe, akik a 2. emeletre utazik
  - 2. szint: kiszáll 1 ember
  - 3., 4., 5. szint: felfele nem utazik senki
  - 6. szint: beszáll 1 ember, aki a 3. emeletre utazik
  - 5. szint: beszáll 1 ember, aki a 3. emeletre utazik
  - ... mivel tele a lift és nincs kiszállás, a 2. emelet lesz érdekes
  - 3. szint: kiszáll 2 ember, beszáll 2 ember, akik az 1. és 2. emeletre utaznak
  - 2. szint: kiszáll 1 ember
  - 1. szint: kiszáll 1 ember
- Végetért a második menet, a szinteken az alábbi emberek maradnak:
  - 1. szint: -
  - 2. szint: -
  - 3. szint: -
  - 4. szint: 2
  - 5. szint: 2
  - 6. szint: -
- A lift a harmadik menetben:
  - ... felfele nem utazik senki, a lift felmegy a 6. emeletre, majd vissza indul az 1. emeletre
  - 5. szint: beszáll 1 ember a liftbe, akik a 2. emeletre utazik
  - 4. szint: beszáll 1 ember a liftbe, akik a 2. emeletre utazik
  - 2. szint: kiszáll 2 ember
- Végetért a harmadik menet, a szinteken az alábbi emberek maradnak:
  - 1. szint: -
  - 2. szint: -
  - 3. szint: -
  - 4. szint: -
  - 5. szint: -
  - 6. szint: -

Az eredmény alapján a kimeneti fájl egyetlen értéke a `3` lesz.

Tesztesetek

A feladat megoldásának helyes működését legalább az alábbi bemenetekkel tesztelje le!

input.txt	output.txt
2 1 2 1	1
3 2 2 2 3 1 3 2 1	1
5 2 5 2 2 3 5 3 4 4 5 1 5 2	2
6 2 2 3 2 1 3 1 2 2 5 3 6 2 1 2 3	3

A fenti tesztesetek nem feltétlenül tartalmazzák az összes lehetséges állapotát a be- és kimenet(ek)nek, így saját tesztekkel is próbálja ki az alkalmazás helyes működését!

A megoldás során tartsa be a tanult OOP alapelveket (egységbezárás, láthatóság, öröklés), törekedjen saját osztályok létrehozására. Ahol lehetséges alkalmazza a megtanult programozási tételeket, illetve használja a tanult technikákat.

Tájékoztató

A feladattal kapcsolatosan általános szabályok:

- A feladat megoldását egy Console Application részeként kell elkészíteni.
- A feladat megoldásaként beadni vagy a betömörített solution mappa egészét vagy a `Program.cs` forrásfájlt kell (hogy pontosan melyiket, azt minden feladat külön definiálja), melynek elnevezése a feladat azonosítója és a saját neptunkódja legyen alulvonással elválasztva, nagybetűkkel: **AZONOSÍTÓ_NEPTUNKOD**[.zip|.cs]
- A megvalósítás során lehetőség szerint alkalmazza az előadáson és a laboron ismertetett programozási tételeket és egyéb algoritmusokat figyelembe véve a *Megkötések* pontban definiáltakat, ezeket leszámítva viszont legyen kreatív a feladat megoldásával kapcsolatban.
- Az alkalmazás elkészítése során minden esetben törekedjen a megfelelő típusok használatára, illetve az igényes (*formázott, felesleges változóktól, utasításoktól mentes*) kód kialakítására, mely magába foglalja az elnevezésekkel kapcsolatos ajánlások betartását is (bővebben).
- **Ne másoljon vagy adja be más megoldását!** Minden ilyen esetben az összes (felépítésben) azonos megoldás duplikátumként lesz megjelölve és a megoldás el lesz utasítva.
- **Idő után leadott vagy helytelen elnevezésű megoldás vagy a kiírásnak nem megfelelő megoldás vagy fordítási hibát tartalmazó vagy (helyes bemenetet megadva) futásidejű hibával leálló kód nem értékelhető!**
- A feladat leírása az alábbiak szerint épül fel (* - opcionális):
  - *Feladat leírása* - a feladat megfogalmazása
  - *Bemenet* - a bemenettel kapcsolatos információk
  - *Kimenet* - az elvárt kimenettel kapcsolatos információk
  - *Megkötések* - a bemenettel, a kimenettel és az algoritmussal kapcsolatos megkötések, melyek figyelembevétele és betartása kötelező, továbbá az itt megfogalmazott bemeneti korlátoknak a tesztek minden eseteben eleget tesznek, így olyan esetekre nem kell felkészülni, amik itt nincsenek definiálva
  - **Megjegyzések* - további, a feladattal, vagy a megvalósítással kapcsolatos megjegyzések
  - *Példa* - egy példa a feladat megértéséhez
  - *Tesztesetek* - további tesztesetek az algoritmus helyes működésének teszteléséhez, mely nem feltétlenül tartalmazza az összes lehetséges állapotát a be- és kimenet(ek)nek
- **Minden eseteben pontosan azt írja ki és olvassa be az alkalmazás, amit a feladat megkövetel, mivel a megoldás kiértékelése automatikusan történik!** Így például, ha az alkalmazás azzal indul, hogy kiírja a konzolra a `"Kérem a számot:"` üzenetet, akkor a kiértékelés sikertelen lesz, a megoldás hibásnak lesz megjelölve, ugyanis egy számot kellett volna beolvasni a kiírás helyett.
- A kiértékelés során csak a *Megkötések* pont szerinti helyes bemenettel lesz tesztelve az alkalmazás, a "tartományokon" kívüli értéket nem kell lekezelnie az alkalmazásnak.
- Elősegítve a fejlesztést, a beadott megoldás utolsó utasításaként szerepelhet egyetlen `Console.ReadLine()` metódushívás.
- A kiértékelés automatikusan történik, így különösen fontos a megfelelő alkalmazás elkészítése, ugyanis amennyiben nem a leírtaknak megfelelően készül el a megoldás úgy kiértékelése sikertelen lesz, a megoldás pedig hibás.
- Az automatikus kiértékelés négy részből áll:
  - Unit Test-ek - az alkalmazás futásidejű működésének vizsgálatára
  - Szintaktikai ellenőrzés - az alkalmazás felépítésének vizsgálatára
  - Duplikációk keresése - az azonos megoldások kiszűrésére
  - Metrikák meghatározása - tájékoztató jelleggel
- A kiértékelések eredményéből egy HTML report generálódik, melyet minden hallgató megismerhet.
- A leadott megoldással kapcsolatos minimális elvárás:
  - Nem tartalmazhat fordítás idejű figyelmeztetést (`Solution contains 0 compile time warning(s)`).
  - Nem tartalmazhat fordítási hibát (`Solution contains 0 compile time error(s)`).
  - Minden szintaktikai tesztet teljesít (`0 test warning, 0 test failed`).
  - Minden unit test-et teljesít (`0 test failed, 0 optional test failed, 0 test was not run`).

- A feladat megoldásához minden esetben elegendő a ***.NET Framework 4.7.2***, illetve a ***C# 7.3***, azonban megoldását elkészítheti ***.NET 5***-öt, illetve a ***C# 9***-et használva is, viszont a nyelv újjításait nem használhatja. További általános, nyelvi elemekkel való megkötés, melyet a házi feladatok során nem használhat a megoldásában (*a felsorolás változásának jogát fenntartjuk, a mindig aktuális állapotot a report HTML fogja tartalmazni*):
  - Methods: `Array.Sort`, `Array.Reverse`, `Console.ReadKey`, `Environment.Exit`
  - LINQ: `System.Linq`
  - Attributes
  - Collections: `ArrayList`, `BitArray`, `DictionaryEntry`, `Hashtable`, `Queue`, `SortedList`, `Stack`
  - Generic collections: `Dictionary<K,V>`, `HashSet<T>`, `List<T>`, `SortedList<T>`, `Stack<T>`, `Queue<T>`
  - Keywords:
    - Modifiers: `protected`, `internal`, `abstract`, `async`, `event`, `external`, `in`, `out`, `sealed`, `unsafe`, `virtual`, `volatile`
    - Method parameters: `params`, `in`, `out`
    - Generic type constraint: `where`
    - Access: `base`
    - Contextual: `partial`, `when`, `add`, `remove`, `init`
    - Statement: `checked`, `unchecked`, `try-catch-finally`, `throw`, `fixed`, `foreach`, `continue`, `goto`, `yield`, `lock`, `break` - *in loop*
    - Operator and Expression:
      - Member access: `^` - *index from end*, `..` - *range*
      - Type-testing: `is`, `as`, `typeof`
      - Conversion: `implicit`, `explicit`
      - Pointer: `*` - *pointer*, `&` - *address-of*, `*` - *pointer indirection*, `->` - *member access*
      - Lambda: `=>` - *expression, statement*
      - Others: `?:` - *tenary*, `!` - *null forgiving*, `?.` - *null conditional member access*, `?[]` - *null conditional element access*, `??` - *null coalescing*, `??=` - *null coalescing assignment*, `::` - *namespace alias qualifier*, `await`, `default` - *operator, literal*, `delegate`, `is` - *pattern matching*, `nameof`, `sizeof`, `stackalloc`, `switch`, `with` - *expressiong*, `operator`
    - Types: `dynamic`, `interface`, `object`, `Object`, `var`, `struct`, `nullable`, `pointer`, `record`, `Tuple`, `Func<T>`, `Action<T>`,