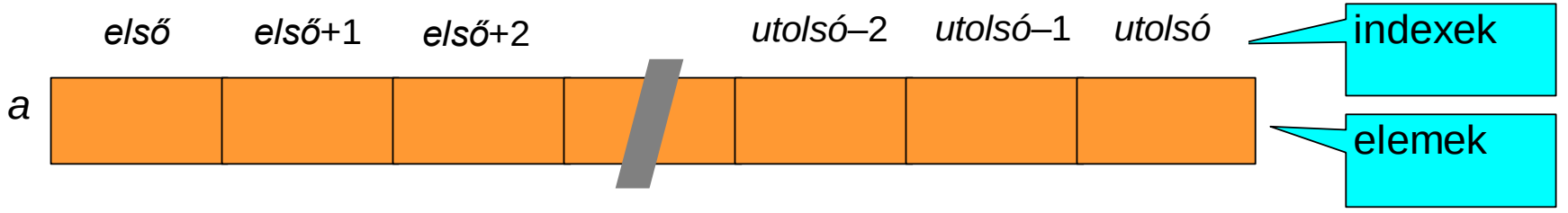
Tömbök

Dr. Ionescu Klára

clara@cs.ubbcluj.ro

A tömb lehet statikus és lehet dinamikus adatszerkezet

- A statikusan implementált tömbök számára lefoglalt tárrész (memória) ***ugyanaz és ugyanakkora méretű*** a lefoglalás pillanatától kezdődően a program végrehajtásának végéig.
- Ezek az adatszerkezetek ***megszületnek*** egy adott pillanatban és ***változatlanul léteznek*** a program bezárásáig.
- Ugyanakkor: a tömböket nem mindig implementáljuk statikusan!
- A tömb elemei azonos típusúak, tehát az adatszerkezet ***homogén***.



Az elemek

- A tömb bármely eleme elérhető (lekérdezhető, értéke megváltoztatható) ***O*(1)** időben, mivel a lekérdezés függetlenül attól, hogy milyen indexértékekkel és milyen elemtípusokkal dolgozunk, az elem címének kiszámítását jelenti.
- Egyszerű aritmetikai műveleteket kell elvégezni, amelyeknek száma csak a tömb dimenzióitól függ.

Az elem indexe lehet:

- egy ***állandó***
- egy ***kifejezés***, amelynek értékét a program fogja kiszámolni a végrehajtás alatt

A tömb

Megkülönböztetjük:

- az elemek típusát (***alaptípus***)
- az indexek típusától (***indextípus***).

A tömb ***<index, elem>*** párok halmaza, ahol minden létező indexnek megfelel egy elem.

Amikor egy programozási nyelvben létrehoztunk egy saját tömbtípust, ***két művelet áll a rendelkezésünkre***:

- ***egy elem lekérdezése***,
- ***egy elem értékének megváltoztatása***

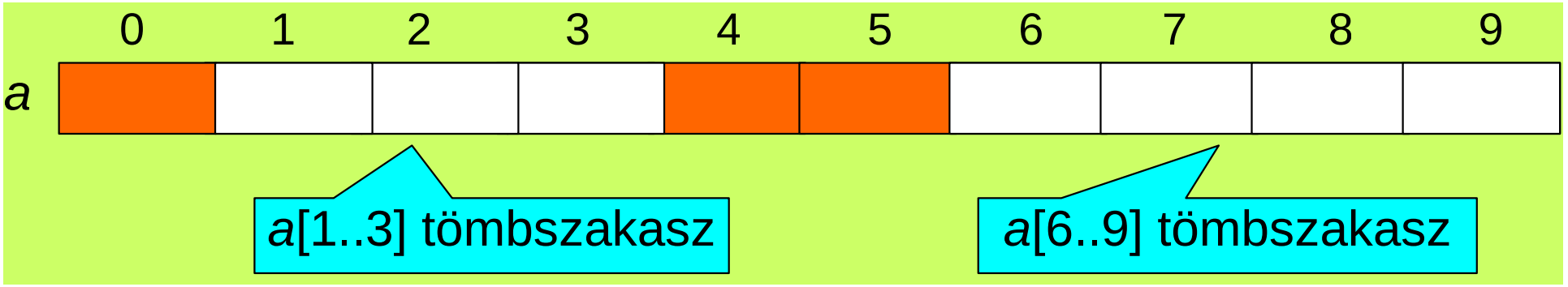
Tömbszakasz

A ***tömbszakasz*** (***szekvencia*** = olyan részsorozat , amelyben az eredeti indexek intervallumot fednek le) a tömb egy része, amely ***egymás után elhelyezkedő elemek***ből áll.

Jelölés: a[bal..jobb]

Elemei: a[bal], a[bal + 1],..., a[jobb]

A tömbszakasz hossza: jobb – bal + 1.



Példák C++-ban

```
const int maxDim = 7;
int negyzet[maxDim];
```

- A `negyzet` tömb számára lefoglalt memóriaterület bájtban kifejezett mérete a `sizeof(negyzet)` kifejezéssel pontosan lekérdezhető
- A `sizeof(negyzet[0]` kifejezés egyetlen elem méretét adja meg.
- Így a két kifejezés hányadosából (egész osztás) megtudható a tömb elemeinek száma:

```
int elemszam = sizeof(negyzet) / sizeof(negyzet[0]);
```

De `sizeof(negyzet[0]` helyett írhatjuk: `sizeof(int)`

Példák C++-ban

Megjegyzés:

- a C++ nyelv nem végez a tömb indexeire vonatkozó *ellenőrzést*
- az indexhatár átlépése a legkülönbözőbb futási hibákhoz vezethet

- Az indexhatár átlépésének ellenőrzéséhez használhatjuk az STL-ből a `vector` típust (template-et):

Példa

```
#include <vector>
using namespace std;
vector<int> negyzet(maxDim);
```

A vector típus használatának előnyei

- dinamikusan változtatható a tömb mérete: `negyzet.resize(ujMeret)`
- elem beszúrása a tömb végére (egyszerű): `negyzet.push_back(123)`
- visszatéríthető az elemek száma `negyzet.size()`

Stb.

Hátrány: a `vector` típusú paraméter bemásolódik a verembe

Inicializálás és értékadás

Általánosan:

elemtípus tömbnév[méret] = { vesszővel tagolt kezdőérték-lista };

Példák:

```
int primek[10] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
```

- A primek tömbben az elemszámnak megfelelő számú értékünk van.
- Ha több kezdőérték szerepel a listában, a fordító hibával jelzi azt.

```
char nev[8] = { 'I', 'v', 'á', 'n'};
```

- A tömb inicializációs listája a tömb elemeinek számánál kevesebb értéket tartalmaz. A tömb első 4 eleme felveszi a megadott értékeket, míg a többi elem értéke 0 lesz.
- Bármekkora tömböt egyszerűen nullázhatunk: `int nagy[2018] = {0};`

Inicializálás és értékadás

```
double szamok[] = { 1.23, 2.34, 3.45, 4.56, 5.67 };
```

- A **szamok** tömb elemeinek számát az inicializációs listában megadott konstansok számának (5) megfelelően állítja be a fordítóprogram.
- Az elemek számát a fentiekben bemutatott módszerrel tudhatjuk meg.
- Az inicializációs lista tetszőleges futásidejű kifejezést is tartalmazhat:

```
double eh[3]= { sqrt(2.3), exp(1.2), sin(3.14159265/4) };
```

Két azonos típusú és méretű tömb közötti értékadás

for ciklusban végezzük az elemek átmásolását

```
const int maxn = 8 ;
int forras[maxn]= { 2, 10, 29, 7, 30, 11, 7, 12 };
int cel[maxn];

   for (int i = 0; i < maxn; i++)
        cel[i] = forras[i]; // elemek másolása
...
```

Változó hosszúságú tömbök

- A C++11 szabvány (*Visual C++ 2012*) a változó méretű tömbök (*variable-length array*) bevezetésével bővíti a tömbök használatának lehetőségeit.
- A futásidőben létrejövő változó hosszúságú tömb csak lokális változó lehet, és a definíciójában nem kaphat kezdőértéket.
- Ezek a tömbök ***csak függvényben használhatók***, így előfordulhat, hogy a tömbök mérete minden híváskor más és más.
- A változó hosszúságú tömb mérete megadható, de így a létrehozást követően a méret nem módosítható.
- A változó méretű tömbökre a **sizeof** operátort alkalmazza a fordító.

Mutatók és a tömbök kapcsolata

- A C++ nyelvben a mutatók és a tömbök között szoros kapcsolat áll fenn.
- Minden művelet, ami tömbindexeléssel végezhető, megvalósítható mutatók segítségével is.
- Az egydimenziós tömbök (vektorok) és az egyszeres indirektségű („egycsillagos”) mutatók között teljes a tartalmi és a formai analógia.
- A többdimenziós tömbök és a többszörös indirektségű („többcsillagos”) mutatók esetén ez a kapcsolat csak formai.

Példa

```
int a[5];
```

- A vektor elemei a memóriában, adott címtől kezdve, folytonosan helyezkednek el.
- Mindegyik elemre `a[i]` formában hivatkozhatunk.
- Legyen `p` egy egészre mutató pointer;
- A „címe” operátor segítségével ráállítjuk az `a` tömb elejére (a `0`. elemre)!

```
int *p;
p = &a[0];
// vagy
p = a;
```

Hivatkozások

- A `p` mutató szerepe teljesen megegyezik az `a` tömbnév szerepével (mindkettő az elemek sorozatának kezdetét jelöli ki a memóriában).
- Lényeges különbség a két entitás között: míg a `p` mutató változó (tehát értéke tetszőlegesen módosítható), addig az `a` egy konstans értékű mutató, amelyet a fordító rögzít a memóriában.

Az alábbi hivatkozások azonosak:

- A tömb `i`-dik elemének címe: `&a[i]`, `&p[i]` , `a + i`, `p + i`
- A tömb `0`-dik eleme: `a[0]`, `p[0]`, `*a`, `*p`, `*(a + 0)`, `*(p + 0)`
- A tömb `i`-dik eleme: `a[i]`, `p[i]`, `*(a + i)`, `*(p + i)`

Dinamikus helyfoglalású tömbök

- A hagyományos tömbök használatának nagy hátránya: méretük már a fordítás során eldől, és csak futás közben derül ki, ha túl nagy memóriát igényeltünk.
  - Különösen igaz ez a függvényekben létrehozott lokális tömbökre.
  - Általános szabály, hogy függvényen belül, csak kisméretű tömböket definiáljunk.
- Amennyiben nagyobb adatmennyiséget kell tárolnunk, alkalmazzuk a dinamikus memóriafoglalás eszközeit, hisz a ***halomterületen*** (heap-memória) sokkal több szabad tárterület áll a rendelkezésünkre, mint a veremben.

Emlékeztető

- **new**: egy, vagy több, egymás után elhelyezkedő elemnek foglalhatunk helyet a memóriában.
- **delete[]**: a lefoglalt tárterület felszabadítása.

Szabványos könyvtárfüggvények a **cstdlib** fejállományban:

- **malloc ()***:* adott bájtméretű területet foglal le, és visszaadja a kezdőcímét
- **calloc ()**: a tárterület méretét elemszám, elemméret formában kell megadni; a **calloc ()** lenullázza a lefoglalt memóriablokkot
- **realloc ()***:* újraméretezi a már lefoglalt memóriaterületet, megőrizve annak eredeti tartalmát (ha nagyobbat foglalunk)
- **free ()***:* felszabadítja a lefoglalt területet.

Egydimenziós dinamikus tömbök

Általánosan

```
típus *mutató;
mutató = new típus [elemszám];
```

- Tömbök esetén különösen fontos, hogy ne feledkezzünk meg a lefoglalt memória szabaddá tételéről:

```
delete [] mutató;
```

Példa:

Töltsük fel az ***n*** elemű ***a*** tömböt véletlenszámokkal.

```
int main() {
  long *adatok, meret;
  cout << "A tomb merete = "; cin >> meret;   cin.get();
  try {
    adatok = new long [meret];                          // memóriafoglalás
  }
  catch (bad_alloc) {                                   // sikertelen foglalás
     cerr  <<  endl << "Nincs eleg memoria ! " << endl;
     return -1;                                         // kilépünk a programból
  }
   srand(unsigned(time(0)));
   for(int i = 0; i < meret; i++) {
      adatok[i] = rand() % 2020;                  // a tömb feltöltése véletlen számokkal
  for(int i = 0; i < meret; i++)
      cout  <<  adatok[i]  <<  " ";
  cout  << endl;
  delete[] adatok;                                  // a lefoglalt memória felszabadítása
}
```

Háromszögű mátrix

A tömbök bizonyos tulajdonságai lehetővé teszik, hogy jelentős méretű tárrészt ***mentesítsünk*** (takarítsunk meg).

A háromszögű mátrix egy négyzetes kétdimenziós tömb, amelyben a ***főátló fölötti elemek 0-val egyenlők*** ($a_{ij} = 0$, *ha* $j > i$).

Példa: első = 1 és ***utolsó*** = n. A mátrix alakja:

$$
\begin{pmatrix}
a_{11} & 0 & 0 & \dots & 0 \\
a_{21} & a_{22} & 0 & \dots & 0 \\
a_{31} & a_{32} & a_{33} & \dots & 0 \\
\dots & & & & \\
a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn}
\end{pmatrix}
$$

A nullától különböző elemek száma: $n \times (n + 1)/2$.

Háromszögű mátrix

Ha szükség van a tárigény csökkentésére, a háromszögű mátrix elemeit egydimenziós tömbben tároljuk:

Cím	*Elemek*	*Sor*
b	$a[1, 1]$	1
$b + h$	$a[2, 1]$	2
$b + 2 \times h$	$a[2, 2]$	2
$b + 3 \times h$	$a[3, 1]$	3
$b + 4 \times h$	$a[3, 2]$	3
$b + 5 \times h$	$a[3, 3]$	3
$b + 6 \times h$	$a[4, 1]$	4
…	…	…
$b + (n - 1) \times h$	$a[n, n]$	n

Háromszögű mátrix

- Ez a tárolás a klasszikussal összehasonlítva $h \times n \times (n-1)/2$ bájtot mentesít azáltal, hogy a **0**-val egyenlő elemeket nem tárolja.

Az eredeti tömb a_{ij} elemének címe:

$$cím(a_{ij}) = b + (1 + 2 + 3 + \ldots + (i-1)) \times h + (j-1) \times h =$$

$$= b + h \times i \times (i-1)/2 + (j-1) \times h$$

- át kell ugornunk $i-1$ sort ahhoz, hogy az i-edik sorba jussunk, ahol i darab nullától különböző elem található, amelyek közül az első $j-1$ elemet átugorjuk.

Ritka tömbök

Példa:

0	0	5	0	0	0
8	0	0	0	-2	0
0	7	0	0	0	0

- A ***ritka tömb legtöbb eleme egy bizonyos értékkel*** (például 0-val) ***egyenlő***.
  - Ha nem a kétdimenziós tömböt, hanem ***egy tömörített egydimenziósat*** tárolunk, amelyben minden elemérték mellett tároljuk az eredeti tömbben elfoglalt hely ***indexeit is***, jelentős számú bájtot takaríthatunk meg.
  - Legyen a nullától különböző elemek száma ***nn***.
  - A klasszikus ábrázolás elpazarol $(utolsó_1 \times utolsó_2 - nn) \times h$ bájtot.

A ritka mátrix ábrázolása

- A sűrített tömb elemei: ***<sor, oszlop, érték>***
- A sűrítést sorfolytonosan végezzük, az egy sorhoz tartozó elemek az oszlopindexük növekvő sorrendjében követik egymást.
- A 0-tól különböző elemek száma: ***elemsz***
- A sorok és oszlopok száma: ***sorsz, oszlopsz***
- ***MaxElemSzám***: a legnagyobb szám, amely a 0-tól különböző elemek száma

Ritka tömbök

Index	sor	oszlop	érték
1	1	3	5
2	2	1	8
3	2	5	-2
4	3	2	7

- Ez az ábrázolás bonyolítja a mátrix elemeit feldolgozó algoritmusokat, hiszen a feldolgozandó elemet meg kell keresnünk.
- Az elérés már nem közvetlen, mivel ***szükség van egy keresésre***.

Ábrázolás

- A ***tömörített egydimenziós*** tömbben, minden elemérték mellett tároljuk az eredeti tömbben elfoglalt hely ***indexeit is***

```
const int nnmax = 50;                    // a nem sajátos értékek (pl. 0) maximális száma
struct elem {                            // a ritka mátrixot ábrázoló tömb elemeinek típusa
        int sor, oszlop;
        float ertek;
};
struct ritka {                                  // a ritka mátrixot ábrázoló struktúra
  elem tomb[nnmax];                                          // a sűrített tömb
  int hanysor, hanyoszlop;                      // az eredeti tömb sor- és oszlopszáma
  int nn;                                                    // a sűrített tömb hossza
};
```

1. Maximális összegű leghosszabb tömbszakasz

Adott egy ***n*** egész számból álló számsorozat, amely *biztosan tartalmaz legalább egy pozitív számot*. Írjunk programot, amely meghatározza azt a *leghosszabb tömbszakaszt*, amelynek *összege a lehető legnagyobb*.

Példa: ***n* = 10**, a sorozat: **1, 2, –6, 3, 4, 5, –2, 10, –5, –6**.

- Maximális összeg: **20**.
- A tömbszakasz hossza: **5**
- A tömbszakasz első elemének sorszáma: **4**
- A tömbszakasz utolsó elemének sorszáma: **8**
- A keresett tömbszakasz: **3, 4, 5, –2, 10**.

Maximális összegű leghosszabb tömbszakasz

1. Megoldás

- Generálunk minden *bal* és *jobb* egész számpárt, amelyekre:
- $1 \leq bal \leq jobb \leq n$, és kiszámítjuk a megfelelő $t_{bal}, t_{bal+1}, \ldots, t_{jobb}$ tömbszakaszok összegét. Közben, kiválasztjuk azt a tömbszakaszt, amelynek összege maximális.
- Ennek az algoritmusnak a bonyolultsága $O(n^3)$ mivel három egymásba ágyazott ciklusból áll.
- Ez a bonyolultság elfogadhatatlan, ezért előbb keresünk egy négyzetes algoritmust, majd egy lineárist!

```
Algoritmus MaxÖsszegűTömbszakasz_1(n, t, MaxS):
                                        // bemeneti adatok: n, t; kimeneti adat: MaxS
  MaxS ← t1                                          // a keresett maximális összeg
    Minden bal = 1, n végezd el:          // bal: a tömbszakasz első elemének indexe
        Minden jobb = bal, n végezd el:          // jobb: az utolsó elemének indexe
            össz ← 0                          // össz: az aktuális tömbszakasz összege
            Minden i = bal, jobb végezd el:
                össz ← össz + ti
            vége(minden)
            Ha MaxS < össz akkor
                MaxS ← össz { stb… }
            vége(ha)
        vége(minden)
    vége(minden)
Vége(algoritmus)
```

Maximális összegű leghosszabb tömbszakasz

2. Megoldás

- Észrevesszük, hogy a $t_{bal}, t_{bal+1}, \ldots, t_{jobb}$ tömbszakasz összegét ki lehet számítani az előző lépésben kiszámolt $t_{bal}, t_{bal+1}, \ldots, t_{jobb-1}$ tömbszakasz összegének segítségével.
- Minden $t_{bal}, t_{bal+1}, \ldots, t_{jobb}$-nak megfelelő összeget összehasonlítunk az eddig megtalált maximális összeggel azonnal a kiszámolás után.
- Az algoritmusunk így négyzetes lesz, mivel csak két egymásba ágyazott ciklust fog tartalmazni:

```
Algoritmus MaxÖsszegűTömbszakasz_2(n, t, MaxS):
        MaxS ← t1                          // bemeneti adatok: n, t; kimeneti adat: MaxS
        Minden bal=1,n végezd el:
                össz ← 0
                Minden jobb = bal, n végezd el:
                        össz ← össz + tjobb
                        Ha MaxS < össz akkor
                                MaxS ← össz… stb.
                        vége(ha)
                vége(minden)
        vége(minden)
Vége(algoritmus)
```

Maximális összegű leghosszabb tömbszakasz

3. Megoldás

- ***A feladat garantálja, hogy a sorozatban van legalább egy pozitív szám!***
- Következik, hogy, ha a sorozatnak $n-1$ negatív eleme és egyetlen pozitív eleme lenne, akkor a maximális összegű tömbszakasz ebből az egyetlen pozitív számból állna.
- Következik, hogy amíg egy aktuális összeg pozitív, addig az ***AktMax***-hoz hozzáadjuk a t_i-t.
- Ha, egy adott pillanatban ***AktMax*** negatívvá válik, akkor ***AktMax*** új értéke t_i lesz.

```
Algoritmus MaxÖsszegűTömbszakasz_3(n, t, eleje, vége, MaxS):
                                    // bemeneti adatok: n, t; kimeneti adatok: MaxS, eleje, vége
    MaxS ← t1; összeg ← MaxS; kezd ← 1; eleje ← 1; vége ← 1
    Minden i = 2, n végezd el:
        Ha összeg < 0 akkor
            összeg ← ti; kezd ← i
        különben összeg ← összeg + ti
        vége(ha)
        Ha MaxS < összeg akkor
            MaxS ← összeg; eleje ← kezd; vége ← i
        különben
            Ha MaxS = összeg és vége – eleje < i – kezd akkor
                eleje ← kezd; vége ← i
            vége(ha)
        vége(ha)
    vége(minden)
Vége(algoritmus)
```

2. Körkörös palindrom

- Egy természetes számokat tartalmazó sorozatot ***palindrom***nak nevezünk, ha balról jobbra olvasva, ugyanazt a sorozatot kapjuk, mintha jobbról balra haladva olvasnánk.
- Például az **(1, 2, 3, 2, 1)** sorozat palindrom, míg az **(1, 2, 3, 2, 4)** nem palindrom.
- Egy természetes számokat tartalmazó sorozat ***körkörös palindrom***, ha az elemeinek néhány körkörös permutációjával palindrommá alakítható. Az elemek körkörös permutációja alatt a sorozat elemeinek egy pozícióval balra tolását értjük (kivételt képez az első elem, amely a sorozat utolsó pozíciójára kerül).

Körkörös palindrom

- Írjunk programot, amely eldönti, hogy az ***n*** elemű, természetes számokat tartalmazó a sorozat körkörös palindrom-e vagy sem, és kiír egy megfelelő üzenetet (Igen/Nem).
- Ha a döntés eredménye Igen, a program meghatározza azoknak a körkörös permutációknak a számát, amelyekkel a sorozat palindrommá alakítható.

- ***1. Példa:*** az ***a* = (1, 1, 2, 2)** sorozat az **(1, 2, 2, 1)** egyetlen körkörös permutációval palindrommá alakítható.
- ***2. Példa:*** az ***a* = (3, 4, 3, 2, 1, 1, 2)** sorozat öt körkörös permutációval palindrommá alakítható: **(4, 3, 2, 1, 1, 2, 3); (3, 2, 1, 1, 2, 3, 4); (2, 1, 1, 2, 3, 4, 3); (1, 1, 2, 3, 4, 3, 2); (1, 2, 3, 4, 3, 2, 1)**.
- ***3. Példa:*** az ***a* = (1, 2, 3)** sorozat nem alakítható palindrommá körkörös permutációkkal.

Körkörös palindrom

1. Megoldás

- Nem generáljuk a sorozat körkörös permutációit.
- Ehelyett az ***a*** sorozatot meghosszabbítjuk úgy, hogy a végére másoljuk a teljes ***a*** sorozatot.
- Ebben a sorozatban megtalálhatók az ***a*** sorozat körkörös permutációi, amelyek ***n*** hosszúságú tömbszakaszai ennek a megkétszerezett sorozatnak.
- Minden lépésben eldöntjük egy-egy ilyen tömbszakaszról, hogy palindrom tulajdonságú, vagy sem. A tömbszakaszok első elemének indexét az ***eleje*** változóban, az utolsóét a ***vége*** változóban tároljuk.

Körkörös palindrom

- A **palindrom(n, a, eleje, vége)** algoritmus eldönti, hogy az ***a*** sorozat aktuális tömbszakasza palindrom tulajdonságú vagy sem.
- Az algoritmusban rendre összehasonlítjuk a sorozat (tömbszakasz) két végén, illetve a végektől egyenlő távolságra levő elemeit.
- A sorozat bejárása addig tart, amíg az elemek egyenlőek, illetve a két összehasonlítandó elem indexére érvényes, hogy az első nagyobb vagy egyenlő, mint a második.
- Ha sikerült bejárni a sorozatot az ***eleje*** és a ***vége*** között, az alprogram *igaz*-at térít, különben *hamis*-at.

Körkörös palindrom

- A palindrom(n, a, eleje, vége) alprogramot meghívja a ciklikusPalindrom(n, a) alprogram, amely implementálja az előbb leírt trükköt.
- Így elérjük, hogy nem kell generálnunk a körkörös permutációkat és rendre vizsgáljuk (legtöbb ***n***-szer), hogy az ***eleje*** és ***vége*** közti tömbszakasz palindrom-e vagy sem.
- Amikor a palindrom(n, a, eleje, vége) algoritmus *igaz*-at térít, leállunk, hiszen megtaláltuk az ***a*** sorozat azon körkörös permutációját, amely palindrom tulajdonságú.

Körkörös palindrom

- Térítjük az (***i*** **– 1**) értéket, mivel az **Amíg** ciklus akkor talált palindrom tulajdonságú részt, amikor az ***i.*** lépésben hívta meg a palindrom(n, a, eleje, vege) alprogramot (de utána ***i*** még nőtt eggyel).
- Ez az a szám, ahányszor a körkörös permutálást el kellett volna végeznünk.
- Ha az **Amíg-ból** való kilépés azután történik, miután az ***n.*** körkörös permutációnak megfelelő lépést is végrehajtottuk, (következne az **(*n* + 1).**), levonjuk a következtetést, hogy az ***a*** sorozat nem alakítható palindrommá körkörös permutálásokkal, és **-1**-et térítünk.

3. Beszúrás

- Adott az ***n*** elemű (**0 < *n* ≤ 10 000**) ***a*** sorozat, amelynek elemei **30 000**-nél kisebb szigorúan pozitív természetes számok. Írjunk programot, amely a sorozat minden eleme után beszúr egy új számot, amely 2-nek az a legnagyobb hatványa, amely kisebb vagy egyenlő az adott elem értékével.
- ***Példa:*** ha ***n* = 4** és ***a* = (3, 1, 24, 9)**, akkor az új ***a*** sorozat:
- **(8)** és az új ***n* = 8**.

Beszúrás

Elemzés

Egy hatékony algoritmus egyszer fogja bejárni a sorozatot és nem használ segéd adatszerkezeteket.

- Abból a célból, hogy a sorozatot ne kelljen minden lépésben eltolni egy-egy elemmel jobbra (ezáltal helyet szorítva a kettőhatványnak), a sorozatot a végétől az eleje felé haladva dolgozzuk fel.
- Egy bizonyos lépésben elhelyezzük az aktuális elemet a végleges helyére, majd a kettőhatványt utána.
- A sorozatot kettesével járjuk be. Minden elemet egyszer dolgozunk fel, segéd adatszerkezet nélkül.

4. Megfeleltetés

- Legyen az ***n*** elemű (**$1 \le n \le 10\,000$**) ***a*** és az ***m*** elemű (**$1 \le m \le 10\,000$**) ***b*** sorozat, amelyeknek elemei **30 000**-nél kisebb természetes számok.
- Az ***a*** sorozat „*megfeleltethető*” a ***b*** sorozatnak, ha az ***a*** sorozatot fel tudjuk osztani diszjunkt tömbszakaszokra úgy, hogy teljesüljenek a következő tulajdonságok:
  - ha az összes tömbszakaszt, a felosztás sorrendjében, egymás után ragasztjuk, megkapjuk az ***a*** sorozatot;
  - ha az összes tömbszakaszt, a felosztás sorrendjében, behelyettesítjük a megfelelő tömbszakasz elemeinek összegével, megkapjuk, rendre a ***b*** sorozat elemeit.

4. Megfeleltetés

Írjunk programot, amely eldönti, hogy az ***a*** sorozat *megfeleltethető vagy sem* a ***b*** sorozatnak. Ha igen, írjuk ki annak a ***b*** sorozatban található elemnek a ***k*** indexét, amely egyenlő az ***a*** sorozat leghosszabb tömbszakaszához (***maxHossz*** hosszúságú) tartozó elemek összegével. Ha több leghosszabb tömbszakasz létezik, az elsőt vesszük figyelembe. Ha ***válasz*** értéke ***hamis***, ***k*** és ***maxHossz*** értéke **-1**.

1. Példa: ha ***n* = 12**, ***a* = (6, 3, 4, 1, 6, 41, 8, , 6, 1, 7, 7)**, ***m* = 4** és ***b* = (13, 7, 18, 16)**, akkor ***válasz = igaz***, mivel: **6 + 3 + 4 = 13**, **1 + 6 = 7, 4 + 6 + 1 + 7 = 18, 1 + 8 + 7 = 16**. Ezek szerint ***k* = 3** és ***maxHossz* = 4**.

2. Példa: ha ***n* = 17**, ***a* = (10, 12, 11, 2, 2, 3, 2, 3, 13, 3, 41, 5, 4, 5, 6, 5, 2)**, ***m* = 6** és ***b* = (33, 4, 15, 41, 25, 2)**, akkor ***válasz = hamis***, mivel: **10 + 12 + 11 = 33**, **2 + 2 = 4**, de **3 + 2 + 3 < 15**, és **3 + 2 + 3 + 13 > 15**. Tehát a $b_3 = 15$ értéket nem tudjuk megfeleltetni az ***a*** sorozat egyik tömbszakaszának sem.

Megfeleltetés

Megoldás

- Ha az ***a*** sorozat mérete kisebb, mint a ***b*** sorozaté, nincs megoldás, hiszen – bármilyen értékei is lennének a két sorozat elemeinek – a ***b*** sorozatban maradnának elemek, amelyeknek nincs mit megfeleltetni.
- Célszerű a két sorozatot párhuzamosan bejárni. Az ***a*** sorozat elemeit addig adjuk össze az ***összeg*** változóban amíg ez az összeg kisebb vagy egyenlő a ***b*** sorozat aktuális elemének értékével.
- Ha a parciális összeg egy adott elem összeadása után egyenlő a ***b*** sorozat aktuális elemével, haladunk tovább mindkét sorozatban.

Megfeleltetés

- Ha a parciális összeg nagyobb, mint a ***b*** sorozat aktuális eleme, az algoritmus ***hamis-at*** fog téríteni, mivel nem lehet elvégezni a megfeleltetést.
- Abban a különleges esetben, amikor az ***a*** sorozat bejárása véget ér, de a ***b*** sorozatban még vannak elemek, szintén ***hamis***-at térítünk.

- Az ***a*** sorozat maximális hosszúságú tömbszakaszának ***maxHossz*** hosszát meghatározzuk a két sorozat bejárása során, anélkül, hogy újra bejárnánk a sorozatokat vagy a tömbszakaszokat.
- A maximális hosszúságú tömbszakasz méretét akkor aktualizáljuk, amikor sikerült „előállítani” a ***b*** sorozat aktuális elemét.

5. „Erős” számok

- Egy nullától különböző ***sz*** természetes számnak az ***erőssége k***, ha bináris alakjában pontosan ***k*** darab 1-es számjegy található.
- Például, a **23** erőssége **4** (kettes számrendszerben felírva, **4** darab 1-es számjegye van).
- Adott számsorozat ***k erősségű csoport***jának nevezzük azt a részsorozatot, amely a sorozat ***k*** erősségű elemeit tartalmazza, az elemek eredeti sorrendjében.
- Például, az ***s* = (7, 12, 3, 13, 24, 19)**, sorozat ***k* = 2** *erősségű csoportja* a **(12, 3, 24)** részsorozat.

„Erős” számok

- Írjunk programot, amely meghatároz ***minden erősségi csoportot***, amelyek az ***n*** (**1 < *n* < 100**) elemű ***x*** sorozat elemeiből létrehozhatók.
- Az elemek különböző természetes számok és kisebbek, mint **30000**.
- A kimenet a csoportok ***csSzáma*** darabszáma és a ***csoportok*** (a létrehozott csoportok, erősségük szerint növekvően rendezve; a csoporton belül az elemek sorrendje tetszőleges).

Példa: ha ***n* = 6** és ***x* = (12, 3, 24, 16, 15, 32)**, akkor ***csSzáma* = 3**, és a ***csoportok***: **(16, 32), (12, 3, 24), (15)**.

„Erős” számok

Elemzés

- A megoldás alapművelete az *erősség meghatározása* vagyis a szám bináris alakjában található 1-es számjegyek számának meghatározása.
- A következő gondunk: hogyan ábrázoljuk a csoportokat?
- A megoldásban szétosztjuk a számokat erősségük szerint egy kétdimenziós tömbbe.
- Az ***i*** erősségű számok az ***i***. sorba kerülnek.
- A csoport számosságát a **0** indexű elem tárolja.

```
Algoritmus erősség(szám):
    erő ← 0
    Ismételd
        szám ← szám & (szám – 1)          // bitenkénti és művelet
        erő ← erő + 1
    ameddig szám = 0                      // kilépünk, ha a szám 0-vá vált
    térítsd erő
Vége(algoritmus)
```

```
Algoritmus csoportok(n, x, f):                          // csoportok meghatározása
// az aktuális elemet az erősségnek megfelelő sorba helyezzük
// a csoport számosságát a 0 indexű elemben tároljuk
    cs ← 0
    Minden i = 1, 16 végezd el:
        f[i][0] ← 0                                     // a csoportok számosságának kezdőértékei
    vége(minden)
    Minden i = 1, n végezd el:                          // feldolgozzuk a sorozatot
        erő ← erősség(x[i])                             // az x[i] elem erőssége
        Ha f[erő][0] = 0 akkor                          // az x[i]-vel új csoport kezdődik
            cs ← cs + 1                                 // nő a csoportok száma
        vége(ha)
        f[erő][0] ← f[erő][0] + 1                       // nő az aktuális erőnek megfelelő sor hossza
        f[erő][f[erő][0]] ← x[i]                        // elhelyezzük az elemet az erősségének megfelelő sorba
    vége(minden)
    térítsd cs
Vége(algoritmus)
```

6. Előszelet

Sors-számjegynek hívjuk azt a természetes számot, amelyet adott természetes számra a következőképpen számítunk ki: összeadjuk a szám számjegyeit, majd a kapott összeg számjegyeit, és így tovább, amíg a kapott összeg nem válik egyszámjegyű számmá.

Például, a **182** *sors-számjegye* **2** (**1 + 8 + 2 = 11, 1 + 1 = 2**).

Egy pontosan ***k*** számjegyű ***p*** számot egy legkevesebb ***k*** számjegyű ***q*** szám ***előszeleté***nek nevezünk, ha a ***q*** szám első ***k*** számjegyéből alkotott szám (balról jobbra tekintve) egyenlő ***p***-vel.

Például, **17** előszelete **174**-nek, és **1713** előszelete **1 713 242**-nek.

Előszelet

Legyen az ***sz*** természetes szám (**$0 < sz \leq 10\,000$**) és az ***m*** sorral és ***n*** oszloppal (**$0 < m \leq 100, 0 < n \leq 100$**) rendelkező ***A*** mátrix (kétdimenziós tömb), amelynek elemei **30 000**-nél kisebb természetes számok.

Írjunk programot, amely meghatározza és kiírja az ***sz*** szám ***leghosszabb előszeletét, amelyet az adott mátrix elemeinek megfelelő sors-számjegyeiből fel lehet építeni***. Egy ilyen sors-számjegyet akárhányszor fel lehet használni.

Ha nem építhető fel előszelet, a program írja ki a „*nem létezik előszelet*” üzenetet.

```
Algoritmus sorsSzámjegy(x):
    Amíg x > 9 végezd el:                    // x-nek több, mint egy számjegye van
        y ← x                                // másolat x-röl
        s ← 0                                // x számjegyeinek összege
        Amíg y > 0 végezd el:
            s ← s + y mod 10
            y ← y / 10
        vége(amíg)
        x ← s                                // x-et felülírjuk számjegyeinek összegével
    vége(amíg)
    térítsd x                                // x-nek most egy számjegye van
Vége(algoritmus)
```

```
Algoritmus prefixMaxSzámjeggyel(szám, m, n, A, számjegyek, számjegyekSzáma):
    Minden i = 0, maxSzjSz -1 végezd el:                    // előfordulások tömbje
        előfordulások[i] ← 0
    vége(minden)
    Minden i = 0, m - 1 végezd el:
        Minden j = 0, n - 1 végezd el:
            előfordulások[sorsSzámjegy(A[i][j])] ← 1
        vége(minden)
    vége(minden)
    számjegyekSzáma ← 0  // a szam számjegyeinek darabszáma
    Amíg szám > 0 végezd el:
        számjegyekSzáma ← számjegyekSzáma + 1
        számjegyek[számjegyekSzáma] ← szám mod 10
        szám ← szám / 10
    vége(amíg)
```

```
    i ← számjegyekSzáma – 1                    // bejárjuk a számjegyek sorozatát
    Amíg i ≥ 0 és előfordulások[számjegyek[i]] végezd el:
                                    // létezik sorsszámjegy, amely egyenlő az aktuális számjeggyel
        i ← i - 1
    vége(amíg)
    térítsd számjegyekSzáma - i - 1
Vége(algoritmus)
```