NEM KORLÁTOZOTT ÁBRÁZOLÁSI TARTOMÁNYÚ EGÉSZ ARITMETIKA

PROGRAMOZÓI DOKUMENTÁCIÓ

A program célja az elméletileg bárhány számjegyből álló számokkal való számolás, az alapműveletek elvégzése és ezek segítségével két ilyen szám legnagobb közös osztójának a kiszámolása. A C nyelvben a legbővebb ábrázolási tartományú szám típus, amiről tanultunk, az a long long int, ami legfeljebb 19 számjegyű számokkal tud számolni, ez nem lenne elég a feladat megvalósításához. A matematikai műveleteket megfigyelve észrevehetjük, hogy elemi szinten bitenként hajtódnak végre és így viszonylag egyszerűen lehet őket algoritmizálni.

Ennek alapján lett felépítve ez a program is, a lényege, hogy a számokkal számjegyenként végzi el a műveleteket.

A program tartalma és felépítése röviden:

egesz_aritmetika.c	**file_muvelet.c**	**szamolasok.c**	**seged.c**	**vezerlo_seged.c**
vezérlés	beolvasás, kiírás, struktúrába rendezés	a szamitasi muveletek	seged figgvenyek	a vezérléshez függvények
debugmalloc.h	**file_muvelet.h**	**szamolasok.h**	**seged.h**	**vezerlo_seged.h**
dinamikus memóriakezelés ellenőrzés	string.h seged.h struct kifejezes	seged.h struct marszam	stdlib.h stdbool.h stdio.h debugmalloc.h struct szam	file_muvelet.h szamolasok.h
mod.c				
a lineáris kongruencia megoldó függvény és segéd függvényei				

ADATSZERKEZETEK

Az adatok tárolásához a program dinamikus tömböket használ:

```
typedef struct szam
{
    int *szjegyek;
    int hossz;
}szam;
typedef struct kifejezes
{
    szam *szamok;
    char *muveletek;
    int szamdb;
}kifejezes;
```

A számok a számjegyeik dinamikus tömbjéből és a hosszúságukból állnak. Ez azért praktikus, mert a számjegyenként dolgozó függvények majd gyorsan el fogják tudni érni az aktuális szám adott helyiértéken lévő számjegyét és azért jó, hogy dinamikus tömbről van szó, mert így nem kell előre tudnia a programnak a szám hosszát beolvasás során, vagy a műveletvégzések során az eredmény kiszámolásánál. Illetve nagyon sokszor kéne bejárni a láncolt listát a műveletek során, hogyha azt használna a program, és az lényeges futásidőbeli különbséget jelentene.

Egy kifejezés fő résztvevői a számok és a műveleti operátorok, ezek dinamikus *szam*- és karaktertömbökben tárolódnak, mivel a beolvasás során nem tudhatja a program, hogy hány szám és hány művelet lesz, illetve a

számolások során praktikusnak tűnik a tömbökkel való munka. A számok darabszámát jelző integer változó praktikusságból van benne a struktúrában, mivel így egyszerűbbek lesznek a számolások. Ami fontos még ahhoz, hogy egyértelmű legyen a kifejezés tartalma, az az, hogy a műveleti operátorok között pár szám is van, melyek azt mondják meg, hogy hányadik szám után kell azt a műveletet végrehajtani. A műveletek sorrendjével nem kell különösebben törődnie a programnak, mert fordított lengyel jelöléssel kell a felhasználónak megadnia a kifejezéseket, aminek fő ismérve, hogy amilyen sorrendben vannak beírva a műveletek, olyan sorban kell őket végrehajtani a számokon, hátulról előrefelé.

A felhasználó által megadott kifejezések, melyeknek az eredményeit szeretné megkapni egy kifejezések által alkotott végjeles tömbben vannak elhelyezve. Ez már egy háromdimenziós dinamikus tömb.

```
typedef struct marszam
{
    szam eredmeny;
    szam maradek;
}marszam;
```

A program használ még egy *marszam* struktúrát is, amiben kettő *szam* típusú szám szerepel, azért, hogy a maradékos osztást végző függvény vissza tudjon térni a hányadossal és a maradékkal is.

EGESZ_ARITMETIKA.C

Ez a modul hívja meg a program működését vezérlő függvényeket, amik után felszabadít minden korábban fel nem szabadított dinamikusan lefoglalt memória területet:

- **`kifejezes* kiftomb(void)`**

  Ez a függvény hívja meg a beolvasó függvényt, majd a kapott szöveget struktúrába rendezi. A kifejezéseket soronként dolgozza fel, karakterenként, és ezeket rakja be egy végjeles dinamikus tömbbe, melynek az utolsó elemének egy üres kifejezést állít be. A létrehozott kifejezések tömbjének a pointerével tér vissza.

```
void plusz(kifejezes *tomb, int j, int k, bool *szamvan, szam *eredmeny, char error_l[]);
void minusz(kifejezes *tomb, int j, int k, bool *szamvan, szam *eredmeny, char error_l[]);
void csillag(kifejezes *tomb, int j, int k, bool *szamvan, szam *eredmeny, char error_l[]);
void per(kifejezes *tomb, int j, int k, bool *szamvan, szam *eredmeny, char error_l[]);
```

Ezek a függvények (melyek a vezerlo_seged.c modulban találhatóak) számoltatják és íratják ki a részszámításokat, a nevüknek megfelelő karakter előfordulása esetén lesznek meghívva. Egységesen megkapják a kifejezések tömbjének a pointerét, 3 egész számot és egy logikai típusú tömböt, cím szerint. A **`kifejezes *tomb`** azért kell, mert ezek a függvények módosítják az adatait, a már nem szükséges számok számjegyeinek a tömbjét felszabadítják és az eredményeket pedig beleírják. A **`j, k, l`** egészek a számok helyeit határozzák meg, amikkel a műveleteteket végre kell hajtatniuk a függvényeknek az adott művelethez tartozó számoló függvénnyel. A **`bool *szamok`**-nak pedig az a funkciója, hogy egy **igaz** értékkel jelzi az adott indexén, hogy a számok tömbjében az adott indexen van-e olyan szám, melynek még nincsenek felszabadítva a számjegyei. Ezek a függvények a **`void er_kiir(szam szam1, szam szam2, szam eredmeny, char c)`**-t használják a részeredménynek a képernyőre való kiírása céljából, ez a függvény a két számot, az eredményüket és a műveleti jelet kapja meg.

- **`eredmeny_fkiir(j + 1, ". kifejezes eredmenye: ", eredmeny)`**

Kiírja az *eredmenyek.txt* file-ba az adott kifejezés eredményét. Ezt a file-t a program moduljainak a könyvtárába menti.

A modul függvényeihez szükséges könyvtárakat a vezerlo_seged.h tartalmazza.

VEZERLO_SEGED.C

Tartalmazza a vezérlő függvényeket és a

```
void er_kiir(szam szam1, szam szam2, szam eredmeny, char c);
```

függvényt, ami kiírja a képernyőre az adott résszámítást.

FILE_MUVELET.C

Ez a modul végzi el a file műveleteket, a beolvasást, a beolvasott adatok struktúrába töltését és a kiírást. A program használati útmutatását is a beolvasó függvény írja ki:

1) Írja be a kifejezéseket (szammal kell kezdődniük), vagy adja meg a beolvasandó file (például: "C:/Users/admin/Desktop/vmi.txt" vagy "../test.txt") eléresi útját (nem kezdődhet szammal!)!

2) A kiszámítandó kifejezést fordított lengyel jelöléssel írja be! Ha Euklideszi algoritmust szeretne végrehajtani, akkor a 2 szam után egy szóközzel elválasztva ezt írja be: "e.a."!

Ha lineáris kongruencia egyenletet szeretne megoldani, akkor ilyen alakban írja be a számokat (a, b, m a beírandó számok, csak ezeket es utánuk egy "m" betűt írjon be):

a * x kongruens b (mod m)

3) Ha befejezte a kifejezések beírását a programablakba, akkor nyomjon egy "x" gombot es egy ENTER-t!

Ezután a megfelelő helyről a függvény beolvassa egy sztringbe a szöveget, majd a

```
kifejezes *kiftomb(void)
```

függvény eltárolja egy tömbben a beírt kifejezéseket. Addig olvassa a karaktertömböt, amíg lezáró nullához nem ér. Ezen belül egy ciklussal feldolgozza soronként egy-egy kifejezést létrehozva és feltöltve. A soron belül karakterenként dolgozik, amíg számjegyek vannak, addig azokat berakosgatja egy dinamikus integertömbbe. Utána mikor szóköz jön és az elozo1 enum-nak digit az értéke, létrehoz egy *szam* típusú számot a számjegyek tömbjéből és darabszámából, majd ezt hozzáadja a kifejezés *szamok* nevű tömbjébe (ez is dinamikus tömb). Ha a

```
while ((s = szoveg[i]) != '\0') {
  kifejezes kifejezes1;
  kifejezes1.szamok = (szam *) malloc1(sizeof(szam) * 1);
  char *muveletek;
  muveletek = (char *) malloc1(sizeof(char) * 2);
  elozo elozo1;
  while ((s = szoveg[i]) != '\n') {
    int *szamtomb;
    szamtomb = (int *) malloc1(sizeof(int) * 1);

    while (47 < (s = szoveg[i]) && (s = szoveg[i]) < 58) {...}
    if (s == ' ') {...}
    else if (s == '+' || s == '-' || s == '/' || s == '*' || s == 'm' || s == 'e' || s == 'a' ||s == '.'){...}
    else {...}
  }
  muveletek[muvdb] = '\0';
  kifejezes1.muveletek = muveletek;
  kifejezes1.szamdb = szamdb;
  kiftomb = realloc1(kiftomb, sizeof(kifejezes) * (k + 1));
  kiftomb[k] = kifejezes1;
  k++;
  i++;
  muvdb = 0;
  szamdb = 0;
```

meghatározott műveleti jelek egyike a következő karakter, akkor azt berakja a *muvtomb*-be, illetve, ha az elozo1 értéke digit, akkor egy számot is rak az operátor karakter elé, ami az operátor előtti szám sorszámával egyezik meg. Ezeket a lépéseket a sor vége jelig csinálja a függvény, majd mindet berakja a *kifejezes1* megfelelő adat helyére, végül hozzáadja a kifejezést a kifejezések tömbjéhez. Ha elérte a sztring végét, akkor hozzáad a kifejezések tömbjéhez egy üres kifejezést, melyben az egészek hely 0-ák vannak és a pointerek helyén NULL pointerek.

SZAMOLASOK.C

Ebben a modulban vannak a legfontosabb függvények, amikkel a számolásokat lehet majd megoldani, összeadás, kivonás, szorzás, osztás és az Euklideszi algoritmus.

```
szam osszeadas(szam szam1, szam szam2)
```

Az összeadást számjegyenként összeadva végzi el a függvény. Az új számjegy az a kettő szám adott helyiértékű számjegyeinek és a többletnek az összegének a 10-zel osztva vett maradéka, az új többlet pedig az összeg 10-zel osztva vett hányadosa. Ha az összeg túllépné a számrendszer alapját (10-et), akkor a következő számjegyösszeghez 1-et továbbadva.

```
szam kivonas(szam szam1, szam szam2)
```

A kivonást szintén számjegyenként elvégezve kell csinálnia a programnak, ha kisebb lenne a különbség 0-nál, akkor a következő helyiértéknél eggyel többet levonva kell majd számolnia.

```
szam szorzas (szam szam1, szam szam2)
```

A szorzást is számjegyenként végezi el, hasonlóan az összeadáshoz, ha több lenne a helyiértéki szorzat a számrendszer alapjánál, akkor a megfelelő mennyiséget hozzá kell majd adni a következő szorzathoz. A szorzó minden egyes számjegyével végig szorozza a szorzandó számot a függvény, helyiérték szerint növekvő sorrendben, a `szam konst_szorz(szam szam1, int konst)` segítségével, ami megszorozza az összes számjegyét egy szam típusú számnak egy egész típusú, 10-nél kisebb számmal és a szorzott számmal tér vissza. Az egyes rész-szorzatokat pedig a szorzó aktuális helyiértéke szerint kell majd eltolva összeadni (a nagyobb helyiértékkel szorzott számhoz a helyiérték indexével megegyező 0-át kell hozzáadni), ezt a `szam tizhatvany (szam szam1, int kitevo)` segítségével teszi meg, ami a seged.c modulban található és megszorozza a kapott számot a 10 *kitevo*-edik hatványával és ezzel tér vissza. Például, mint feljebb látható.

31329 * 653565437 =

1. lépés: 31329 * 7 = 219303 (7 a szorzó tényezőben 10^0-on helyiértéken van, a 0 megegyezik a helyiérték indexével)

2. lépés: 31329 * 3 = 93987 (3 a szorzó tényezőben 10^1-on helyiértéken van, itt az index az 1)

3. …

```
marszam osztas (szam osztando, szam oszto)
```

Paraméterként kap egy *osztando* es egy *oszto* számot, az *oszto* nem lehet 0, ezt a vezérlő függvény vizsgálja, először megvizsgál pár triviális esetet, ha az *oszto* 1, vagy ha az *osztando* kisebb, mint az *oszto*. Addig vegzi a kivonasokat, amig nagyob az *osztando* az *oszto*-nál. Az *osztando*-ból mindig az id [1] -nek a legnagyobbszorosát vonja ki a függvény, így csökkentve a lépésszámot. A végeredményhez hozzáadja mindig a függvény az adott ciklus lépésben az osztó kivont darabszámát (az id a lehető legnagyobb számmal vett szorzatát) a maradék az a megváltozott osztandó lesz, ami már kisebb az osztónál.

```
szam ea (szam szam1, szam szam2)
```

Ez a függvény számolja ki két szám legnagyobb közös osztóját az Euklideszi algoritus segítségével. Ameddig nem lesz az osztás maradéka nulla, addig mindig elosztja a *szam1*-et a *szam2*-vel. Ezek folyamatosan csökkennek, mert minden osztás

[1] ideiglenes szám, az osztónak a szorzata az osztó és az osztandó szám nagyságrend béli különbségével

után a *szam1* megkapja a *szam2* értékét, a *szam2* pedig az osztás maradékát. Az algoritmus minden lépést kiír, így a képernyőn jól követhetőek a számításai, illetve, hogy hogyan is működik.

MOD.C

Itt számítja ki a program a lineáris kongruencia egyenlet megoldását.

A modulon belül a vezérfüggvény a kiszámítandó kifejezés számait kapja meg és az eredménnyel tér vissza:

```
szam m(szam ax, szam b, szam mod)
```

Ellenőrzi az egyenlet megoldhatóságát, illetve, hogy van-e triviális megoldása, ezt ennek a függvénynek a segítségével:

```
bool trivialis(szam ax, szam b, szam mod, szam *megoldas)
```

Ez a megoldást pointerként kapja, hogy tudja esetlegesen módosítani a függvény. Ha van egyértelmű megoldása az egyenletnek, akkor igazzal tér vissza.

A vezérlő függvény a fő, ismétlendő számításokat ezzel a függvénnyel hajtatja végre:

```
void m_ciklusmag(szam *a0, szam *ax, szam *b0, szam *b, const szam mod)
```

Cím szerint kapja a változókat, hogy tudja az összeset változtatni (kivéve a modulust persze). Mindig kivonja a 2-vel előtti egyenletből az eggyel előttinek a legnagyobb egész-szorosát, hogy az ax pozitív maradjon, amíg ax nem lesz 1. Ez esetben ugyanis elérkezett az egyenlet megoldásához.

```
void m_kiir(szam ax, szam b, szam mod
```

Kiírja a képernyőre a kongruencia egyenletet a megadott számok alapján.

```
void egyszerusites(szam *ax, szam *b, const szam mod)
```

A kongruencia definíciója alapján a lehető legkisebb egészekre csökkenti ax és b számokat.

SEGED.C

Ebben a modulban a kisebb, segéd függvények találhatók, melyeket akár több modul is használ.

A kettő leggyakrabban használt:

```
void *malloc1(int size_t)
void *realloc1(void *p, int size_t)
```

Ezek olyan dinamikus memória kezelő függvények, amelyek tartalmaznak hiba kezelést is, tehát nem kell minden alkalommal megvizsgálni a használatuk után, hogy sikerült-e lefoglalniuk a kívánt nagyságú területet. Ha nem sikerülne, akkor kiírja, hogy nem sikerült és aztán tér csak vissza NULL pointerrel, egyébként a lefoglalt tömb pointerével térnek vissza.

```
bool egyenlo (szam szam1, szam szam2)
```

Megvizsgálja, hogy két *szam* típusú szám egyenlő-e és igazzal, vagy hamissal tér vissza.

```
bool nagyobb (szam szam1, szam szam2)
```

Megvizsgálja, hogy az első *szam* típusú szám nagyobb-e a másodiknál és igazzal, vagy hamissal tér vissza.

```
szam tizhatvany (szam szam1, int kitevo)
```

Megszoroz egy *szam* típusú számot a 10 *kitevo*-edik hatványával, amit úgy tesz meg, hogy lefoglal egy dinamikus tömböt az új szám számjegyeinek, az eredeti hosszának és a kitevőnek az összegével egyenlő méretűre.

```
szam szamcpy(szam szam1)
```

Mély másolatot készít egy *szam* típusú számról és visszatér vele.

```
void kiiras(szam szam1)
```

Kiírja a megkapott szam típusú számot a képernyőre számjegyenként.

FELHASZNÁLÓI DOKUMENTÁCIÓ

Az program olyan nagy számokkal való számítások elvégzésére való, amiket már átlagos számológépek nem tudnak kiszámolni (kicsi számokra is ugyanolyan jól működik).

A program parancssorból, vagy akár a mappájából is futtatható, a .exe file megnyitásával.

A felhasználónak vagy a *.txt file elérési útját kell megadnia, amiben azok a kifejezések szerepelnek, amiket meg szeretne oldani, vagy pedig be kell írnia a kifejezéseket sortöréssel elválasztva. A kifejezéseket minden esetben fordított lengyel jelöléssel kell megadnia, bármennyit beírhat egymás után. Ha ki szeretne lépni a programból, akkor amikor megkérdezi ezt a program, akkor „q” gombot és egy ENTER-t kell nyomnia a felhasználónak, egyébként bármi mást és írhatja is be a további kiszámítandó feladatot.

A kifejezéseket minden esetben számjeggyel kell kezdeni, illetve az elérési utat nem szabad azzal kezdeni. A számok és a műveletek között szóközt kell hagyni. Ahhoz, hogy elvégezhetők legyenek a műveletek, eggyel több számnak kell lennie a kifejezésben, mint amennyi művelet jel.

Ha legnagyobb közös osztót szeretne számolni a felhasználó, akkor a kettő szám után szóközzel elválasztva ezt kell beírnia: „e.a.”.

Ha lineáris kongruencia egyenletet szeretne megoldani, akkor a 3 szám (az x együtthatója, amivel kongruens a bal oldad és a modulus) után szóközzel elválasztva egy „m” betűt kell írnia.

A részszámítások eredményeit a képernyőn fognak megjelenni ilyen formátumban, a végeredmények pedig az eredmenyek.txt[2] file-ba is beírva, hasonló módon.

Részszámítások:

---

1. kifejezes:

…

---

2. kifejezes:

876652837698231698437087 + 7896871346846355478 = 876660734569578544792565

6547646138721561262 * 876660734569578544792565 = 57400642736733084770805703549526167296170 30

---

3. kifejezes:

Euklideszi algoritmus alapjan:

…

[2] A program végrehajtó .exe file-jának a mappájának a szülőmappájában lesz

Minta a megoldasok.txt kinézetére:

---

1. kifejezes eredmenye: 0

---

2. kifejezes eredmenye: 57400642736733084770805703549526167296170 30

---

3. kifejezes:

A ket megadott szam legnagyob kozos osztoja: 7