Definíció: Az alsó egészrész függvény

Az *alsó egészrész* függvény minden valós számhoz egy egész számot rendel hozzá, éppen azt, amely a tőle nem nagyobb egészek közül a legnagyobb. Az alsó egészrész függvény jele: $\lfloor x \rfloor$, ahol x valós szám. Tömören:

$$\lfloor x \rfloor = \max_{\substack{k \le x \\ k \in Z}} k \tag{3}$$

Más szavakkal formálisan: $\lfloor x \rfloor = k$, ahol k olyan egész szám, hogy $k \le x < k+1$.

példa:

x	$-5{,}2$	-5	5	$5{,}2$
$\lfloor x \rfloor$	-6	-5	5	5

Definíció: A felső egészrész függvény

A *felső egészrész* függvény minden valós számhoz egy egész számot rendel hozzá, éppen azt, amely a tőle nem kisebb egészek közül a legkisebb. A felső egészrész függvény jele: $\lceil x \rceil$, ahol x valós szám. Tömören:

$$\lceil x \rceil = \min_{\substack{k \ge x \\ k \in Z}} k \tag{4}$$

Más szavakkal formálisan: $\lceil x \rceil = k$, ahol k olyan egész szám, hogy $k-1 < x \le k$.

példa:

x	$-5{,}2$	-5	5	$5{,}2$
$\lceil x \rceil$	-5	-5	5	6

Az alsó és felső egészrész függvények fontos tulajdonságait az alábbi táblázatban foglaljuk össze: (Lássuk be, hogy ezek valóban teljesülnek!)

1.	Ha a egész szám, akkor	$\lfloor a \rfloor = a$,	$\lceil a \rceil = a$,
2.	Ha x valós, a egész szám, akkor	$\lfloor x \pm a \rfloor = \lfloor x \rfloor \pm a$,	$\lceil x \pm a \rceil = \lceil x \rceil \pm a$,
3.	Ha x és y valós számok, akkor	$\lfloor x \pm y \rfloor \ge \lfloor x \rfloor \pm \lfloor y \rfloor$,	$\lceil x \pm y \rceil \le \lceil x \rceil \pm \lceil y \rceil$,
4.	Ha x valós szám, akkor	$-\lfloor x \rfloor = \lceil -x \rceil$,	$-\lceil x \rceil = \lfloor -x \rfloor$,
5.	Ha $x \le y$ valós számok, akkor	$\lfloor x \rfloor \le \lfloor y \rfloor$,	$\lceil x \rceil \le \lceil y \rceil$

Definíció: A kerekítő függvény

A *kerekítő függvény* minden valós számhoz a hozzá legközelebb eső egész számot rendeli hozzá. Ha a legközelebbi egész szám nem egyértelmű, akkor a nagyobbat választja. A kerekítő függvény jele: *Round(x)*, ahol x valós szám.

$$Round(x) = \left\lfloor x + \frac{1}{2} \right\rfloor \tag{5}$$

A legközelebbi egészre kerekít. Pozitív számok esetén, ha a tizedesrész 5/10, vagy annál nagyobb, akkor felfelé, kisebb esetben lefelé kerekít. Negatív számok esetén ha a tízes számrendszer szerinti felírásban a tizedesrész kisebb, mint 5/10, vagy egyenlő vele, akkor felfelé, egyébként lefelé kerekít.

példa:

x	-6	$-5{,}8$	$-5{,}5$	$-5{,}2$	-5	5	5,2	5,5	5,8	6
$Round(x)$	-6	-6	-5	-5	-5	5	5	6	6	6

Definíció: A törtrész függvény

A *törtrész függvény* minden valós számhoz azt a számot rendeli hozzá, amely azt mutatja meg, hogy a szám mennyivel nagyobb az alsó egészrészénél. A törtrész függvény jele: $\{x\}$, ahol x valós szám. Tömören:

$$\{x\} = x - \lfloor x \rfloor. \tag{6}$$

Mindig fennáll a $0 \le \{x\} < 1$ egyenlőtlenség.

példa:

x	$-5{,}8$	$-5{,}2$	-5	5	5,2	5,8
$\{x\}$	0,2	0,8	0	0	0,2	0,8

Felhívjuk a figyelmet két műveletre:

Definíció: Az egész hányados képzése, a div művelet

Legyen a és b egész szám, $b \neq 0$. Definíció szerint az egész osztás műveletén az a/b osztás eredményének alsó egész részét értjük. Tömören:

$$a \ div \ b = \lfloor a / b \rfloor \tag{7}$$

példa: $-9 \ div \ 4 = -3$, $9 \ div \ 4 = 2$

Definíció: Az egész maradék képzése, a mod művelet

Legyen a és b egész szám. Definíció szerint

$$a \bmod b \overset{def}{=} \begin{cases} a, & ha \ b = 0 \\ a - \lfloor a/b \rfloor \cdot b = a - (a \, div \, b) \cdot b, & ha \ b \neq 0 \end{cases} \tag{8}$$

példa: $-9 \bmod 4 = 3$, $9 \bmod 4 = 1$, $-9 \bmod (-4) = -1$, $9 \bmod (-4) = -3$

Speciális jelentése van az $a \bmod 1$ jelölésnek. Ezt minden valós a-ra értelmezzük és jelentése az a valós szám törtrésze, azaz

$$a \bmod 1 \overset{def}{=} \{a\} \tag{9}$$

Egy szám lejegyzésekor a használt számrendszer alapszámát mindig tizes számrendszerben adjuk meg és a szám jobb alsó sarkához írjuk indexként. Ha a számrendszer alapja a $b \geq 2$ egész szám, akkor az x pozitív egész szám számjegyei:

$$c_n, c_{n-1}, \ldots, c_1, c_0 \tag{10}$$

ahol $0 \leq c_k < b$, $k = 0,1,\ldots,n$ és az x szám értéke ezekkel a számjegyekkel és az alappal kifejezve:

$$x = c_n \cdot b^n + c_{n-1} \cdot b^{n-1} + \ldots + c_1 \cdot b^1 + c_0 \cdot b^0 \tag{11}$$

Az n értékét úgy határozzuk meg, hogy $c_n \neq 0$ legyen, és minden $c_k = 0$, ha $k > n$. Ha a számrendszer alapszáma tíznél nagyobb, akkor a 0,1,2,...,9 számjegyek mellett új számjegyeket kell bevezetni a tíz, tizenegy, ..., $b-1$ számértékekre. Kényelmi és nyomdatechnikai okok miatt a latin ábécé nagybetűit használjuk ezen célra. Ilyen módon tehát az A=10, B=11, C=12, ..., Z=35 jelek használatosak. (Lehet találkozni vegyes jelöléssel is, ahol a számjegyeket tizes számrendszerben jegyzik le. Mi nem fogjuk ezt alkalmazni.)

példa: 2006 számjegyei tizes számrendszerben $c_3 = 2$, $c_2 = 0$, $c_1 = 0$, $c_0 = 6$. Itt $n=3$ és $2006 = 2 \cdot 10^3 + 0 \cdot 10^2 + 0 \cdot 10^1 + 6 \cdot 10^0$.

Nyílvánvalóan: $c_0 = x \bmod b$ és $c_n c_{n-1} \ldots c_2 c_1 = x \; div \; b$. A következő séma alkalmas a számjegyek egymást követő fordított irányú előhozására:

x	b
$x_1 = x \; div \; b$	$c_0 = x \bmod b$
$x_2 = x_1 \; div \; b$	$c_1 = x_1 \bmod b$
...	...
$x_k = x_{k-1} \; div \; b$	$c_{k-1} = x_{k-1} \bmod b$
...	...
$x_n = x_{n-1} \; div \; b$	$c_{n-1} = x_{n-1} \bmod b$
0	$c_n = x_n \bmod b$

(12)

példa: Írjuk fel a 2006-ot kettes (= *bináris*) számrendszerben és 16-os (= *hexadecimális*) számrendszerben.

2006	2
1003	0
501	1
250	0
125	0
62	1
31	0
15	1
7	1
3	1
1	1
0	1

2006	16
125	6
7	$13=D_{16}$
0	7

Tehát $2006_{10}=11111010010_2=7D6_{16}$

Átírás tizes alapra a (11) formula átrendezésével történik az úgynevezett *Horner séma* szerint.

$$x = (\ldots((c_n)\cdot b + c_{n-1})\cdot b + \ldots + c_1)\cdot b + c_0 \tag{13}$$

Ezzel azt érjük el, hogy kevés műveletet kell használni, a műveletek azonos jellegűek, másrészt a műveleteket végezhetjük tizes számrendszerben.

példa: $7D6_{16}$=((7)·16+13) ·16+6=2006
11111010010_2=((((((((((1) ·2+1) ·2+1) ·2+1) ·2+1) ·2+0) ·2+1) ·2+0) ·2+0) ·2+1) ·2+0
2006_{10}=(((2) ·10+0) ·10+0) ·10+6

Kellemes az átváltás a két számrendszerbeli ábrázolás között, ha történetesen a bináris és a hexadecimális számrendszerről van szó. Ekkor hexadecimális alakról binárisra történő átírás esetén minden hexadecimális jegyet a jegy bináris megfelelőjével helyettesítünk. Binárisról hexadecimálisra történő átírásnál pedig a törtvesszőtől jobbra is és balra is négyes csoportokra osztva a bináris szám számjegyeit, minden csoportot helyettesítünk a hexadecimális megfelelőjével. A megfeleltetés a hexadecimális számjegyek és a bináris négyjegyű csoportok között az alábbi táblázatban látható:

0000 ↔ 0	0100 ↔ 4	1000 ↔ 8	1100 ↔ C
0001 ↔ 1	0101 ↔ 5	1001 ↔ 9	1101 ↔ D
0010 ↔ 2	0110 ↔ 6	1010 ↔ A	1110 ↔ E
0011 ↔ 3	0111 ↔ 7	1011 ↔ B	1111 ↔ F

(Lássuk be, hogy a javasolt módszer helyes eredményre vezet!) Módszerünkkel kikerüljük egyrészt a tizes számrendszerre történő átmeneti átalakítást, másrészt nem kell nem tizes alapú számrendszerben műveleteket végezni

Pozitív egész szám *b* alapú logaritmusa és a szám *b* alapú számrendszerbeli számjegyei számának a kapcsolatát világítja meg az alábbi tétel.

3. Tétel: A számjegyek számáról
Pozitív *x* egész szám számjegyeinek a száma *b* alapú számrendszerben eggyel több, mint a szám *b* alapú logaritmusának az alsó egészrésze, azaz ha a szám számjegyei $c_n, c_{n-1}, \ldots, c_1, c_0$, akkor a jegyek száma

$$n+1 = \lfloor \log_b x \rfloor + 1 \tag{14}$$

Bizonyítás

$$\begin{aligned} x &= c_n \cdot b^n + c_{n-1}\cdot b^{n-1} + \ldots + c_1 \cdot b^1 + c_0 \cdot b^0 = \\ &= b^n \cdot \underbrace{\left(c_n + c_{n-1}/b + \ldots + c_1/b^{n-1} + c_0/b^n\right)}_{=y} = \\ &= b^n \cdot y \end{aligned} \tag{15}$$

Világos, hogy $1 \le c_n \le y < b$. Innen az *y* logaritmusára kapjuk, hogy

$$0 \le \log_b y < 1 \tag{16}$$

(15)-ből logaritmálással

$$\log_b x = n \cdot \log_b b + \log_b y = n + \log_b y \tag{17}$$

adódik. Azaz

$$n + \log_b y = \log_b x \tag{18}$$

(18) mindkét oldalán az alsó egészrészt véve $n = \lfloor \log_b x \rfloor$ adódik, mivel *n* egész szám és (16) fennáll. Egyet hozzáadva mindkét oldalhoz kapjuk az állításunkat. ■

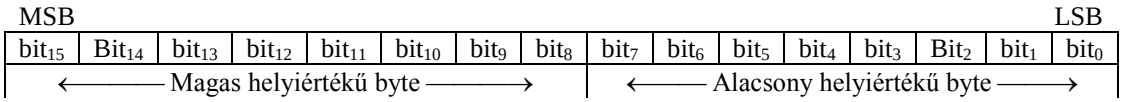
Általában nem fog minket érdekelni, hogy a tanulmányunk tárgyai, objektumai hogyan valósíthatók meg fizikailag (hogyan realizálhatók). Ennek ellenére nem árt egy kis kitekintés a számítástechnikára. Az egyik fontos dolog, hogy az egész számok hogyan kerülnek tárolásra a számítógép memóriájában. A memóriát úgy lehet elképzelni, mint egymás mellett lineárisan felsorakoztatott tárolórekeszek sorozata. A rekeszeket egymástól a sorban elfoglalt helyük különbözteti meg, amit egy indexszel (címmel) írunk le. A rekesz fogalom szemléletes, de fizikailag nem pontos. A fizikai rekesz ma a byte (= 8 bit). A byte a memória legkisebb fizikailag címezhető egysége. A memóriából kiolvasni, vagy oda beírni egy byte-nál kevesebb adatmennyiséget nem lehet. Ha csak egy bitet akarunk megváltoztatni, akkor is ki kell olvasni az őt tartalmazó byte-ot, a kívánt bitet átírni, majd a byte-ot visszaírni a memóriába. A byte tartalmát a jobb áttekinthetőség miatt hexadecimális számrendszerben szoktuk megadni. Például az 11001001_2 tartalmú byte hexadecimális alakban $C9_{16}$. A byte bitjeit jobbról balra indexeljük. A jobbszélső bit a nullás indexű bit (a legkevésbé szignifikáns bit, Least Significant Bit, LSB), tőle balra áll az egyes indexű bit, és így tovább. A byte balszélén áll a hetes bit (a legszignifikánsabb bit, a legnagyobb helyiértékű bit, Most Significant Bit, MSB).

	MSB							LSB
	bit_7	bit_6	bit_5	bit_4	bit_3	bit_2	bit_1	bit_0
bitindex	7	6	5	4	3	2	1	0

Byte és bitjei

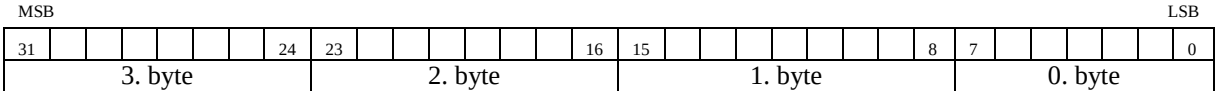
Már egyetlen byte is alkalmas szám tárolására, csak a számtartomány kicsi. A nyolc bit mindegyike lehet zérus, vagy egy. Ennek megfelelően $2^8 = 256$ egymástól különböző byte létezhet. Minden ilyen bitvariációhoz hozzárendelünk egy számot. Természetes módon kínálkozott és a gyakorlat is ezt részesítette előnyben, hogy előjel nélküli (nemnegatív) illetve előjeles számokat különböztessünk meg. A byte tartalmát aszerint interpretáljuk, hogy milyen típusú adatot akarunk benne látni. Előjel nélküli esetben a hozzárendelés a byte tartalomhoz a bitsorozat kettes számrendszerben leírt számként való értelmezése. Ekkor a legkisebb szám a zérus lesz (tiszta zérus bitek), a legnagyobb a 255 (tiszta egyes bitek). Ha előjeles számokat szeretnénk tárolni, akkor az úgynevezett *kettes komplemens* ábrázolást szokás előnyei miatt alkalmazni. Ekkor az előjeles egész szám hozzárendelése úgy történik, hogy ha a MSB=0, akkor a byte tartalmat az előjelmentes esetnek megfelelően értelmezzük. Ha MSB=1, akkor annyival kell többet tennünk, hogy a kapott előjelmentes számból kivonjuk a $2^8 = 256$ számot, ami biztosan negatív lesz. Ezen a módon a legkisebb szám –128 lesz (MSB=1, a többi bit zérus), a

legnagyobb szám 127 lesz (MSB=0, a többi bit egyes). Egy adott bitmintázathoz tartozó kettes komplemensnek a meghatározására egy egyszerű szabály a következő: a mintázat jobbvégétől balfelé haladva leírjuk az összes bitet változatlanul az elsőként előforduló egyes bittel bezárólag, majd ezután minden további bitet ellentétesre változtatunk. (Lássuk be, hogy a módszer valóban helyes! Van-e olyan bitmintázat, amelynek a kettes komplemense saját maga, és ha van, akkor hány van és melyek ezek?) Egy byte tartalmát értelmezhetjük azonban karakterként is. A ***karakter*** gyűjtőfogalom, a betű, számjegy, írásjel, vezérlőjel összefoglaló neve. Ha egy byte tartalmát karakterként kívánjuk interpretálni, akkor használhatjuk a szabványos és elterjedt ASCII kódtáblázatot, amely a C. Függelékben megtalálható. Nem célunk most itt belemenni ennek az egyszerű dolognak a gyakorlat által történő elbarokkosításába, amikoris a karakter milyensége függhet az eszköztől, amire azt kiküldjük (képernyő, nyomtató, stb.), a nyelvi megállapodástól stb. Bármilyen meglepő ezekután, egy byte-ot nyolc bitnek is lehet interpretálni, amikor az egyes biteknek más és más a jelentése. Ilyenkor jelzőbitekről (flag-ek) beszélünk. Az egyes bitek szemafor szerepet játszanak, pillanatnyi értéküktől függően döntünk. Visszatérve az egész számok ábrázolására azonnal látszik, hogy az egyetlen byte nagyon szűk mozgásteret enged, kicsi az átfogott számtartomány. A fizikai eszközeink (a hardware) lehetőséget biztosítanak több byte összekapcsolására. Ez azonban a software útján is megvalósítható, csak az kevésbé hatékony. Két byte már 16 bitet ad, ami 2^{16}=65536 különböző szám megadhatóságát jelenti. A byte-párt *szó*nak (word) nevezik. A szóban a MSB a szó balszélén a 15-ös indexű bit.



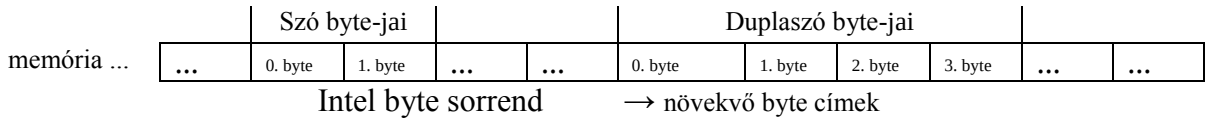
Szó és bitjei

Előjel nélküli számok esetén a legkisebb érték 0 (16 zérus bit) a legnagyobb szám 2^{16}-1 = 65535 (16 egyes bit). Előjeles számok esetén ha MSB=1, akkor kivonni a 2^{16}=65536-ot kell. A legkisebb szám -2^{15} = -32768 (MSB=1, a többi bit zérus), a legnagyobb 2^{15}-1 = 32767 (MSB=0, a többi bit egyes). Mód van négy byte összekapcsolására. Ez a ***dupla szó*** (double word). A dupla szóban a MSB bit indexe 31.

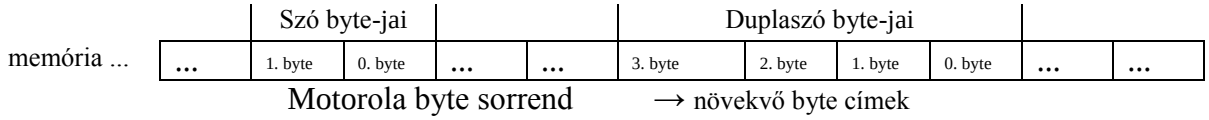


Duplaszó byte-jai és bitjei

A lehetőségek száma 2^{32} = 4294967296. Előjel nélküli számok esetén a legkisebb szám a zérus, a legnagyobb 2^{32}-1 = 4294967295. Előjeles esetben ha MSB=1, akkor kivonni a 2^{32} = 4294967296 számot kell. A legkisebb szám a -2^{31} = -2147483648, a legnagyobb 2^{31}-1 = 2147483647. Ha a byte-okat az egyik számítógépről valamely másikra visszük át, akkor nem lényegtelen, hogy az összekapcsolt byte-ok esetén mi a helyes byte-sorrend. Az Intel processzorok az összekapcsolt byte-okat fordított sorrendben rakják le egymás után a memóriában, ami azt jelenti, hogy elöl a legalacsonyabb helyiértékeket tartalmazó byte áll, majd azt követik az egyre magasabb helyiértékeket adó byte-ok. (A Motorola processzorok az egyenes sorrend hívei, először a legmagasabb helyiértékű byte, majd a csökkenő helyiérték szerintiek következnek.)



Intel byte sorrend → növekvő byte címek



Az egész számokkal végzett műveletek pontos eredményt adnak, ha a végeredmény és az összes részeredmény az ábrázolási tartományba esik. Összefoglalva a számtartományokat:

	előjel nélküli szám		előjeles szám	
	legkisebb	legnagyobb	legkisebb	legnagyobb
byte	0	2^8-1	-2^7	2^7-1
szó	0	2^{16}-1	-2^{15}	2^{15}-1
dupla szó	0	2^{32}-1	-2^{31}	2^{31}-1

Más a helyzet a valós számokkal. Ezen a téren ma már szabvány létezik, az IEEE 754-es szabvány, amely 1985 óta érvényes és William Kahan a Berkeley egyetem professzora nevéhez fűződik. A szabvány pontos előírásokat ad a valós számok ábrázolására. Pontosabban nem is valós számoknak nevezik az ílymódon kezelhetö számokat, hanem *lebegőpontos számok*nak. Ezen ábrázolási formát nem tárgyaljuk teljes részletezettséggel, de a két együtt ismertethető esetről - az egyszeres pontosság és a dupla pontosság esete - szólunk néhány szót. Előtte azonban meg kell ismerkedni egy a tizedes törteket más alapú számrendszerbe történő lehetséges átírási módszerrel. Tekintsük egy szám törtrészének a tizes számrendszerbeli felírását. A szám legyen $0,c_1c_2c_3\ldots$ alakú, ahol $c_1,c_2,c_3\ldots$ a törtrész egymást követő tizedesjegyei. Ha 10-zel szorzunk, akkor az első tizedesjegy kicsúszik az egészek helyére, amit levághatunk. További szorzásokkal a többi jegy is előjön egymás után. Ha minket egy b alapú számrendszerbeli felírás jegyei érdekelnek, akkor világos, hugy a 10 helyett b-vel kell szorozgatni. A tevékenység egy sémába foglalható, a számjegyek egyenes sorrendben keletkeznek:

b	x
$c_1 = \lfloor x \cdot b \rfloor$	$x_1 = (x \cdot b) \bmod 1$
$c_2 = \lfloor x_1 \cdot b \rfloor$	$x_2 = (x_1 \cdot b) \bmod 1$
...	...
$c_k = \lfloor x_{k-1} \cdot b \rfloor$	$x_k = (x_{k-1} \cdot b) \bmod 1$
...	...

A visszaalakítás pedig történhet szintén egy Horneres séma szerint. Az $x = 0,c_1c_2c_3\ldots c_n$ szám értelmezése ugyanis

$$x = c_1/b + c_2/b^2 + c_3/b^3 + \ldots + c_n/b^n \tag{19}$$

Ez úgy is számolható, hogy

$$x = (\ldots(((c_n)/b + c_{n-1})/b + c_{n-2})/b + \ldots c_1)/b \tag{20}$$

A séma kényelmes, felváltva kell számjegyenként osztást és összeadást végezni.

12. példa: Írjuk fel a 0,52734375-öt kettes (= *bináris*) számrendszerben és 16-os (= *hexadecimális*) számrendszerben.

2	0,52734375
1	0,0546875
0	0,109375
0	0,21875
0	0,4375
0	0,875
1	0,75
1	0,5
1	0,0

16	0,52734375
8	0,4375
7	0,0

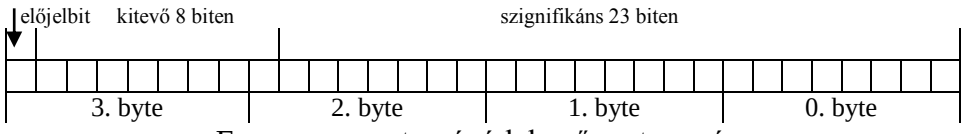
Tehát $0{,}52734375_{10}=0{,}10000111_2=0{,}87_{16}$

13. példa: Visszaírás tizes számrendszerre: $0{,}87_{16}=0+((7)/16+8)/16=0{,}52734375$
$0{,}10000111_2=0+(((((((1)/2+1)/2+0)/2+0)/2+0)/2+0)/2+1)/2$
$0{,}52734375_{10}=0+(((((((5)/10+7)/10+3)/10+4)/10+3)/10+7)/10+2)/10+5)/10$

A tizedes törtek átalakításának van egy a számítógép oldaláról tekintve kellemetlen oldala. Az ugyanis, hogy ami véges tizedes tört az egyik alap mellett, az nem biztos, hogy szintén véges lesz a másik alap mellett. Például $0{,}1_{10}$ binárisan végtelen sok törtjegyet tartalmaz. (Ellenőrizzük!) Tárolni viszont mindig csak véges sok bitet tudunk. Valahol a végtelen sorozatot el kell vágni. Ez azt jelenti, hogy az átalakított szám már nem fog megegyezni értékét tekintve az eredetivel. Tehát például, ha egy program bekér egy lebegőpontos számot és 0,1-et billentyűzünk be, akkor az kettes számrendszerbelivé átalakítva értékét tekintve nem fog megegyezni az eredeti tízes számrendszerbeli 0,1-del. Ha egy törtjegyeket tartalmazó számnak az egészek helyén álló része nem zérus, akkor először átalakítjuk az egészek részét, majd a törtrészt és a kettőt egy törtvesszővel elválasztva egymás mellé helyezzük.

14. példa: $2006{,}52734375_{10}=11111010010{,}10000111_2=7D6{,}87_{16}$

A szabvány szerinti lebegőpontos számábrázolás négy byte-on történik egyszeres pontosság esetén és nyolc byte-on dupla pontosság esetén. A kettő között eltérés igazán csak a pontosságban és az átfogott számtartományban van, az ábrázolás elve azonos. Tekintsük először a *normalizált szám* esetét. Legyen a szám nemzérus. Ekkor a binárisan felírt számot átalakítjuk olyan formára, hogy a törtvesszőt a legelső egyes jegyet közvetlenül követően helyezzük el és megjegyezzük, hogy ezen művelethez a törtvesszőt hány bitpozícióval kellett balra mozgatni. Ez a szám balra mozgásnál pozitív, jobbra mozgásnál negatív lesz és azt mutatja, hogy az átalakítás utáni számot a 2 milyen kitevőjű hatványával kell megszorozni, hogy a kiinduló számot megkapjuk. A törtvesszőt követő bitek sorozatának neve: *szignifikáns*. Ezen információkat kell elhelyeznünk a rendelkezésre álló négy illetve nyolc byte-on. A bitek kiosztása az egyszeres pontosság esetén:



Egyszeres pontosságú lebegőpontos szám

Az előjelbit pozitív szám esetén zérus, negatív szám esetén 1. A kitevő részére fenntartott 8 bites mezőbe a kitevő 127-tel megnövelt (eltolt) értékét helyezzük el előjel nélküli egész

számként. A szignifikáns (a vezető egyes nélkül, implicit egyes bit) kerül a hátramaradt 23 bites mezőbe.

15. példa:Példa: 2006,52734375 hogyan néz ki egyszeres pontosságú lebegőpontos számként? A szám binárisan, ahogy már kiszámoltuk: $11111010010{,}10000111_2$. Normalizált alakban: 1,111101001010000111 ahol a kitevő decimálisan 10, ami 127-tel eltolva $10+127=137=10001010_2$. (Negatív kitevőt 8 biten kettes komplemens módon tárolunk, majd így adjuk hozzá a 127-et.) A szignifikáns 23 bitre zérusokkal kiegészítve: 11110100101000011100000. Végül az egyszeres pontosságú lebegőpontos ábrázolás 32 bitje 0100 0101 0111 1010 0101 0000 1110 0000, vagy hexadecimálisan 45 7A 50 E0.

A szám ebben a formában történő ábrázolása csak akkor megengedett, ha az eltolt kitevő nem zérus és nem 255. Ez a két szélső eset más célra van fenntartva. A tiszta zérus biteket tartalmazó kitevő mező és a zérus szignifikáns együtt *zérus*ként van definiálva. Van pozitív zérus és negatív zérus az előjeltől függően, de valójában a lebegőpontos processzor (koprocesszor) ezeket azonosként kezeli. Ha a kitevő mező zérus, de a szignifikáns mező nem zérus, akkor *nem normalizált* (denormalizált) lebegőpontos számról beszélünk. Ekkor az implicit egyes bit is tárolásra kerül, mint a szignifikáns része, mivel a törtvessző mögé kerül. Denormalizált tárolásnál komoly jegyveszteségre lehet számítani! Például a 2^{-126} még normalizált módon tárolódik, de a tőle kisebb kitevőjűeknél már az eddig elhagyott egyes bitet is tároljuk. Az alábbi táblázat illusztrál néhány esetet.

Szám	Byte-ok binárisan	Byte-ok hexában
2^{-126}	0000 0000 1000 0000 0000 0000 0000 0000	00 80 00 00
2^{-127}	0000 0000 0100 0000 0000 0000 0000 0000	00 40 00 00
2^{-128}	0000 0000 0010 0000 0000 0000 0000 0000	00 20 00 00
…	…	…
2^{-149}	0000 0000 0000 0000 0000 0000 0000 0001	00 00 00 01

A kitevő mező legmagasabb értékéhez szintén két eset tartozik. Ha a szignifikáns mező zérus, akkor a tárolt információ előjeles végtelenként van definiálva.

Szimbólum	Byte-ok binárisan	Byte-ok hexában
$+\infty$	0111 1111 1000 0000 0000 0000 0000 0000	7F 80 00 00
$-\infty$	1111 1111 1000 0000 0000 0000 0000 0000	FF 80 00 00

A végtelen kezelése során a processzor a végtelennel végezhető műveletek tulajdonságait megtartja. Például végtelen plusz véges eredménye végtelen, vagy véges osztva végtelennel zérust ad. Ha a szignifikáns rész nem zérus, akkor ezt a szituációt *nem szám*ként definiálták (NaN=Not a Number).

Szimbólum	Byte-ok binárisan
NaN	0111 1111 1xxx xxxx xxxx xxxx xxxx xxxx
NaN	1111 1111 1xxx xxxx xxxx xxxx xxxx xxxx

A sémában az x-szel jelölt bitek nem lehetnek egyszerre mind zérusok. Ilyen eset (NaN) lehet például a végtelen osztva végtelennel művelet eredménye. Azok a számok, értékek, állapotok, amelyek a fenti sémába nem férnek bele, nem ábrázolhatók, velük közvetlen módon számolni nem tudunk.

A dupla pontosságú esetben a nyolc byte-ban a kitevő mező 11 bites, a szignifikáns mező 52 bites. A kitevő eltolás mértéke 1023. A kitevő mező két kitüntetett értéke a zérus és az 2047. Egy táblázatban mellékeljük a lebegőpontos aritmetika lehetőségeit, korlátait:

Jellemzők	Egyszeres pontosság	Dupla pontosság
Előjelbitek száma	1	1
Kitevő bitek száma	8	11
Törtrész bitek száma	23	52
Összes bitek száma	32	64
Kitevő ábrázolása	127-es eltolás	1023-as eltolás
Kitevő tartománya	-126 - +127	-1022 - +1023
Legkisebb normalizált szám	2^{-126}	2^{-1022}
Legnagyobb normalizált szám	kb. 2^{128}	kb. 2^{1024}
Decimális számtartomány	kb. 10^{-38} - 10^{38}	kb. 10^{-308} - 10^{308}
Értékes decimális jegyek száma	kb. 6-7	kb. 15-16
Legkisebb nem normalizált szám	kb. 10^{-45}	kb. 10^{-324}

A számítógép programozásáról

A számítógépes programozás területéről több fogalomra lesz szükségünk annak ellenére, hogy igazán egyetlen programozási nyelv mellett sem kötelezzük el magunkat. A számításaink, adatokon végzett tevékenységeink elvégzéséhez gépi utasítások, parancsok rögzített sorozatára lesz szükségünk. Ezeket összefogva *program*nak fogjuk nevezni. A programot valamilyen magas szintű programozási nyelven (az ember gondolkodásmódjához közel álló nyelven) írjuk meg, majd azt a gép nyelvére egy fordítóprogram (*compiler*) segítségével fordítjuk le (remélhetően jól). Ha van *interpreter* program, akkor azzal is megoldható a feladatvégzésnek a gépre történő átvitele. A programok általában *procedúrák* (eljárások) sokaságát tartalmazzák. Ezek a zárt programegységek egy-egy kisebb feladat elvégzésére specializáltak. A program többi részével csak a paramétereik révén tartják a kapcsolatot. Fekete doboznak kell őket tekintenünk. A dobozra rá van írva, hogy miből mit csinál. Vannak (lehetnek) bemenő (*input*) és vannak (lehetnek) kimenő (*output*) paraméterei. A bemenetet alakítják át a kimenetté. Ha ismerjük a procedúra belső szerkezetét — mert mondjuk mi készítettük —, akkor fehér doboz a neve, ha nem ismerjük — mert nem vagyunk kíváncsiak rá, vagy másoktól kaptuk —, akkor fekete doboz szerkezet a neve. Például készíthetünk olyan procedúrát, amely bekéri (input) az *a, b, c* három valós számot, melyeket egy ax^2+bx+c kifejezés (itt *x* valós szám, változó) konstans együtthatóinak tekint, majd eredményül (output) meghatározza a kifejezés valós gyökeinek a számát és ha van(nak) gyök(ök) , akkor az(oka)t is megadja. Példa egy lehetséges másik procedúrára: egy file nevének ismeretében a procedúra a file rekordjait valamilyen szempont szerint megfelelő sorrendbe rakja (rendezi). A procedúrák által használt memóriarekeszek — a paramétereket kivéve — a zártságnak köszönhetően *lokálisak* a procedúrára nézve. Csak addig foglaltak, míg a procedúra dolgozik, aktív. A procedúrát munkára fogni az aktivizáló utasítással lehet. Ezt *eljáráshívás*nak is nevezik. Az aktivizált procedúra lehet saját maga az aktivizáló is, ekkor *rekurzív hívás*ról beszélünk, a procedúrát pedig *rekurzív procedúrá*nak nevezzük. A procedúra munkája végén a vezérlés visszaadódik az aktivizáló utasítást követő utasításra. Ezt a mechanizmust a *verem* (stack) révén valósítjuk meg. A verem a memória egy erre a célra kiválasztott része. A procedúra aktivizálásakor ide kerülnek beírásra a procedúra paraméterei és a *visszatérési cím* (az aktivizáló utasítást követő utasítás címe). A procedúrából való visszatéréskor ezen cím és információk alapján tudjuk

folytatni a munkát, a programot. A visszatéréskor a veremből az aktivizálási információk törlődnek. Ha a procedúra aktivizál egy másik procedúrát, akkor a verembe a korábbiakat követően az új aktivizálási információk is bekerülnek, azt mondjuk, hogy a verem mélyül, a veremmélység szintszáma eggyel nő. Kezdetben a verem üres, a szintszám zérus, procedúrahíváskor a szintszám nő eggyel, visszatéréskor csökken eggyel. A dolog pikantériájához tartozik, hogy a procedúra a lokális változóit is a verembe szokta helyezni, csak ezt közvetlenül nem érzékeljük, mivel a visszatéréskor ezek onnan törlődnek, a helyük felszabadul. Időnként azonban a hatás sajnálatosan látványos, amikor *verem túlcsordulás* (stack overflow) miatt hibajelzést kapunk és a program futása, a feladat megoldásának menete megszakad. Adódhat azonban úgy is, hogy mindenféle hibajelzés nélkül „lefagy a gép". A veremnek a valóságban van egy felső mérethatára, amelyet nagyon nem tanácsos túllépni.

Nézzünk egy példát a veremhasználatra. Tegyük fel, hogy van még olyan elvetemült informatikus, aki nem tudja, hogy $1+2+3+\ldots+n=\frac{n\cdot(n+1)}{2}$, és ezért egy kis procedúrát ír ennek kiszámítására. Amennyiben az illető a fent említett hibája mellett teljesen normális, akkor igen nagy eséllyel az alábbi módon oldja meg a problémát. A procedúra neve legyen Summa és legyen egy paramétere az *n*, hogy 1-től kezdve meddig történjen az összeadás. Feltételezzük a procedúra jóhiszemű használatát és így az *n* pozitív egész szám kell legyen. (Nem írjuk meg a procedúrát első lépésben még „bolondbiztosra".) Kirészletezzük egy kissé a procedúra teendőit. Szükség lesz egy gyűjtőrekeszre, ahol az összeget fogjuk képezni és tárolni. Legyen ennek a neve *s*. A procedúra munkájának végén ez lesz a végeredmény, ezt kapjuk vissza, ez lesz a procedúra output paramétere. Szükség lesz továbbá egy számlálóra, legyen a neve *k*, amellyel egytől egyesével elszámolunk *n*-ig és minden egyes értékét az *s*-hez a számlálás közben hozzáadjuk. Az *s*-et a munka kezdetén természetesen ki kell nullázni, hiszen nem tudjuk, hogy mi van benne az induláskor. Ezek után a kósza meggondolások után egy kissé rendezettebb alakban is írjuk le a teendőket.

A Summa procedúra leírása

Összefoglaló adatok a procedúráról:

A procedúra neve:	Summa.
Bemenő paraméter:	*n*, megadja, hogy 1-től meddig kell az összeadást elvégezni.
Kijövő paraméter:	*s*, tartalmazza az összeget a végén.
Lokális változó:	*k*, számláló, amely egytől elszámol *n*-ig egyesével.

A procedúra tevékenysége:

1. lépés: *s* kinullázása
2. lépés: *k* beállítása 1-re, innen indul a számlálás
3. lépés: *s* megnövelése *k*-val (*s*-hez hozzáadjuk a *k*-t és az eredmény *s*-ben marad.
4. lépés: Eggyel megnöveljük a *k* számláló értékét
5. lépés: Ellenőrizzük, hogy a *k* számláló nem lépett-e túl az *n*-nen, a végértéken.
   Ha még nem, akkor folytatjuk a munkát a 3. lépésnél.
   Ha igen, akkor pedig a 6. lépéshez megyünk.
6. lépés: Készen vagyunk, az eredmény az *s*-ben található.

Ezután ha szükségünk van, mondjuk, 1-től 5-ig a számok összegére, akkor csak leírjuk, hogy Summa(Input:5, Output s), vagy rövidebben Summa(5,s). Esetleg függvényes alakot használva

az s=Summa(5) is írható. Az aktivizálás hatására a verembe bekerül az 5-ös szám, valamint az *s* rekesz memóriabeli címe és a visszatérési cím, hogy a procedúra munkája után hol kell folytatni a tevékenységet. Miután most nincs több teendő, ezért ez a cím olyan lesz, amelyből ez a tény kiderül. Jelezhetjük ezt formálisan mondjuk egy STOP-pal. Valahogy így néz ki a verem formálisan:

5	*s* címe	STOP

Kezdetben üres volt a verem, most egy szint került bele bejegyzésre. Amikor a procedúra munkája véget ér, akkor ez a bejegyzés a veremből törlődik, így az újra üres lesz. (Tulajdonképpen a számláló számára lefoglalt helyet is fel kellett volna tüntetni a bejegyzésben, de ez a számunkra most nem fontos.)

Minden nagyon szép, minden nagyon jó, mindennel meg vagyunk elégedve, és akkor jön egy rekurzióval megfertőzött agyú ember, aki így gondolkodik. Egytől *n*-ig összeadni a számokat az ugyanaz, mint az egytől *n*-1-ig összeadott számok összegéhez az *n*-et hozzáadni. A feladatot visszavezettük saját magára, csak kisebb méretben. Egytől *n*-1-ig persze megint úgy adunk össze, hogy az *n*-2-ig képezett összeghez adjuk az *n*-1-et. Ez a rekurzió. Arra kell vigyázni, hogy valahol ennek a visszavezetésnek véget kell vetni. Amikor már csak egytől egyig kell az összeget képezni, akkor azt nem vezetjük vissza tovább, hiszen ott tudjuk az eredményt, ami triviálisan éppen egy. Tehát a rekurzív agyú ember egy függvényt alkot, mondjuk *RekurzívSumma* néven, és az alábbi módon definiálja azt:

$$RekurzívSumma(n)=\begin{cases}1, & ha\ n=1\\ RekurzívSumma(n-1)+n, & ha\ n>1\end{cases} \qquad (24)$$

Ha most leirjuk, hogy *s*=*RekurzívSumma*(5), akkor ezt úgy kell kiszámolni, hogy:

s=*RekurzívSumma*(5) = *RekurzívSumma*(4)+5
=(*RekurzívSumma*(3)+4)+5
=((*RekurzívSumma*(2)+3)+4)+5
=(((*RekurzívSumma*(1)+2)+3)+4)+5
=((1+2)+3)+4)+5
=((3+3)+4)+5
=(6+4)+5
=10+5
=15

Lássuk ezekután hogyan alakul a verem története. A *RekurzívSumma*(5) hatására az üres verembe egy bejegyzés kerül:

5	eredmény	Az eredmény *s*-be írási címe

A továbbiakban pedig a verem az egyes rekurzív hívások hatására a következőképpen alakul:

RekurzívSumma(5):	5	eredmény	Az eredmény s-be írási címe
RekurzívSumma(4):	4	eredmény	Összeadás helye
RekurzívSumma(3):	3	eredmény	Összeadás helye
RekurzívSumma(2):	2	eredmény	Összeadás helye

RekurzívSumma(1):	1	eredmény	Összeadás helye

Itt a rekurzió megakad, további rekurzív hívás már nem lesz, a végleges veremmélység 5, a rekurzív hívások száma 4 (a legelső aktivizálás még nem rekurzív hívás). A legutolsó hívás már tud számolni, és az eredmény 1 lesz, ami a veremben meg is jelenik:

RekurzívSumma(5):	5	eredmény	Az eredmény *s*-be írási címe
RekurzívSumma(4):	4	eredmény	Összeadás helye
RekurzívSumma(3):	3	eredmény	Összeadás helye
RekurzívSumma(2):	2	eredmény	Összeadás helye
RekurzívSumma(1):	1	1	Összeadás helye

Ezután az utolsó előtti hívásbeli összeadás (1+2) elvégezhető, a hívás befejeződik és a veremből a legutolsó bejegyzés törlődik. A továbbiakban rendre az alábbi veremállapotok állnak elő:

RekurzívSumma(5):	5	eredmény	Az eredmény *s*-be írási címe
RekurzívSumma(4):	4	eredmény	Összeadás helye
RekurzívSumma(3):	3	eredmény	Összeadás helye
RekurzívSumma(2):	2	3	Összeadás helye

RekurzívSumma(5):	5	eredmény	Az eredmény *s*-be írási címe
RekurzívSumma(4):	4	eredmény	Összeadás helye
RekurzívSumma(3):	3	6	Összeadás helye

RekurzívSumma(5):	5	eredmény	Az eredmény *s*-be írási címe
RekurzívSumma(4):	4	10	Összeadás helye

RekurzívSumma(5):	5	15	Az eredmény *s*-be írási címe

Innen a visszatérés az értékadáshoz, az *s*-be történő eredmény elhelyezéshez történik, miáltal a verem kiürül. Az elmondottak alapján látszik, hogy a feladat elvégzéséhez szükséges maximális veremmélység 5 és összesen 4 rekurzív hívás történt.

Itt akár fel is lélegezhetnénk, de ekkor egy újabb, még súlyosabb állapotban lévő fazon jelenik meg, aki azt mondja, hogy lehet ezt még szebben is csinálni. Ő a rekurziót arra építi, hogy az összeg képezhető úgy is, hogy az összeadandó számok halmaza első felének összegéhez hozzáadja a halmaz második felének összegét. A felezést további felezéssel számolja, mígcsak az aprózódás révén el nem jut egytagú ősszegekig. Röviden és tömören ő egy másik függvényt definiál, amely kétváltozós, neve *RekSum*(*m*,*n*), és *m*-től *n*-ig adja össze a számokat. Ezzel az általánosabb függvénnyel egytől *n*-ig összeadni *RekSum*(1,*n*)-nel lehet. Speciálisan a mi fenti problémánk esetében : *RekSum*(1;5) számolandó. Az ő definíciója így néz ki:

$$RekSum(m,n)=\begin{cases} m, & ha\ m=n \\ RekSum\left(m,\left\lfloor\frac{m+n}{2}\right\rfloor\right)+RekSum\left(\left\lfloor\frac{m+n}{2}\right\rfloor+1,n\right), & ha\ m<n \end{cases} \qquad (25)$$

Nézzük csak hogyan is számol ez a ravasz módszer a mi speciális *s*=*RekSum*(1;5) esetünkben?

s=*RekSum*(1;5) =*RekSum*(1;3) + *RekSum*(4;5)

=(*RekSum*(1;2) + *RekSum*(3;3)) + (*RekSum*(4;4) + *RekSum*(5;5))
=((*RekSum*(1;1) + *RekSum*(2;2)) + 3) + (4 + 5))
=((1 + 2) + 3) + (4 + 5))
=(3 + 3) + (4 + 5)
=(6 + 9)
=15

Hogyan alakul a verem sorsa ebben az esetben? Az első aktivizáló hívás után a verem:

1	5	eredmény	Az eredmény *s*-be írási címe

Ezután következik a *RekSum*(1;3) hívás. A hatása:

RekSum(1,5)	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	3	eredmény	Összeadásjel

Most jön a *RekSum*(1;2) hívás a *RekSum*(1;3)-on belül. A hatás:

RekSum(1,5)	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	3	eredmény	Összeadásjel
RekSum(1,2)	1	2	eredmény	Összeadásjel

Ez megint nem számolható közvetlenül, tehát jön a *RekSum*(1;1), mire a verem új képe:

RekSum(1,5)	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	3	eredmény	Összeadásjel
RekSum(1,2)	1	2	eredmény	Összeadásjel
RekSum(1,1)	1	1	eredmény	Összeadásjel

Itt már van eredmény, átmenetileg nincs több rekurzív hívás. Az eredmény 1.

RekSum(1,5)	1	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	1	3	eredmény	Összeadásjel
RekSum(1,2)	1	1	2	eredmény	Összeadásjel
RekSum(1,1)	1	1	1	1	Összeadásjel

A hívás befejezte után a veremből kiürül a legutolsó bejegyzés, visszatérünk az összeadásjelhez, amely után azonban egy újabb rekurzív hívás keletkezik, a *RekSum*(2;2). hatására a verem képe:

RekSum(1,5)	1	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	1	3	eredmény	Összeadásjel
RekSum(1,2)	1	1	2	eredmény	Összeadásjel
RekSum(2,2)	2	2	2	eredmény	Összeadás befejezése

Az innen történő visszatérés után a verem képe:

RekSum(1,5)	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	3	eredmény	Összeadásjel
RekSum(1,2)	1	2	3	Összeadásjel

Az összeadás elvégzéséhez itt azonban egy újabb rekurzív hívás szükséges, a *RekSum*(3;3).

RekSum(1,5)	1	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	1	3	eredmény	Összeadásjel
RekSum(3,3)	3	3	3	eredmény	Összeadás befejezése

Innen

RekSum(1,5)	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(1,3)	1	3	6	Összeadásjel

következik, majd pedig egy újabb hívás, a *RekSum*(4;5). A veremállapot:

RekSum(1,5)	1	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(4,5)	4	4	5	eredmény	Összeadásjel

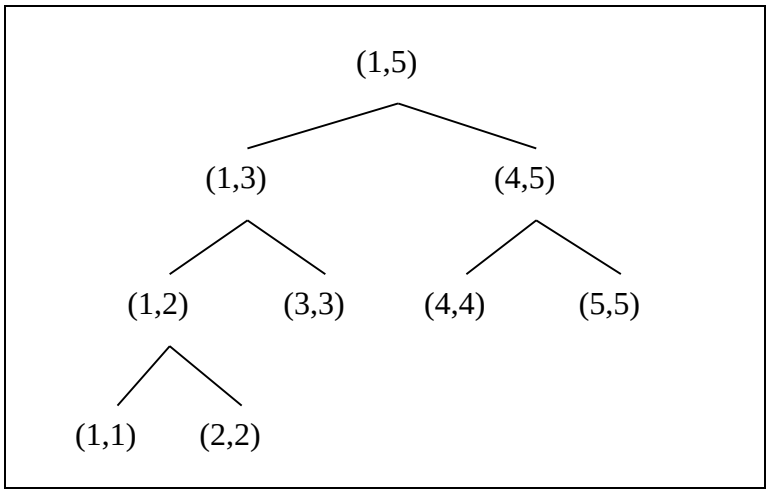
Újabb hívás szükséges a *RekSum*(4;4). A veremállapot:

RekSum(1,5)	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(4,5)	4	5	eredmény	Összeadás vége
RekSum(4,4)	4	4	eredmény	Összeadásjel

Ennek befejezte után és a veremből történő törlést követően még kell egy hívásnak lennie, ez pedig a *RekSum*(4;4). A veremállapot:

RekSum(1,5)	1	1	5	eredmény	Az eredmény *s*-be írási címe
RekSum(4,5)	4	4	5	eredmény	Összeadás vége
RekSum(5,5)	5	5	5	eredmény	Összeadás vége

Innentől kezdve a verem már csak ürül, további rekurzív hívásokra nincs szükség. A feladat elvégzéséhez kevesebb szintből álló verem is elég volt, mint az előző esetben, most a maximális veremmélység csak 4 volt. A rekurzív hívások száma azonban megnőtt, összesen nyolc rekurzív hívás volt. Ebben a rekurzióban minden hívás, kivéve a legalsóbb szinten levőket két újabbat eredményezett, de ezek a veremnek ugyanazon szintjét használták. A hívások szerkezetét egy úgynevezett *hívási fa séma*val tudjuk ábrázolni, melyben csak a paraméter értékeket tüntetjük fel. Íme:



Az ábrán jól látszik a verem négy szintje. A legfelső szint kivételével a többi szinten lévő hívások rekurzívak. Az azonos szinten lévő hívások a verem azonos szintjét használják csak eltérő időben.

FELADATOK

1. Töltsük ki az alábbi táblázatot! (Soron belül ugyanaz a szám más számrendszerben)

Számrendszer alapszáma és a szám						
2	3	8	10	12	16	36
10101110						
	120120					
		1234567				
			1973			
				1234		
					9AB	
						XYZ

2. Töltsük ki az alábbi táblázatot! (Soron belül ugyanaz a szám más számrendszerben)

Számrendszer alapszáma és a szám						
2	3	8	10	12	16	36
101110,111						
	120,111					
		4567,111				
			973,111			
				234,111		
					AB,111	
						YZ,111

8. a. Az *n*! (*n faktoriális)* fogalmát pozitív egész számokra az alábbi rekurzív formulával definiáljuk:

$$n! = \begin{cases} 1, & ha\ n = 1 \\ n \cdot (n-1)!, & ha\ n > 1 \end{cases}$$

Procedúrahívással kiszámítottuk a 7! értékét. Hogyan alakult a verem felépítése, mélysége? Mekkora a minimális méretű verem, amely a feladat elvégzéséhez szükséges? Hány rekurzív hívás volt a számolás során?

b. Definiáljunk egy $P(m,n)$ függvényt, amely pozitív egész *m* és *n* argumentumokra van definiálva és $m \le n$ fenn kell, hogy álljon.

$$P(m,n) = \begin{cases} m, & ha\ m = n \\ P\left(m, \left\lfloor \frac{m+n}{2} \right\rfloor\right) \cdot P\left(\left\lfloor \frac{m+n}{2} \right\rfloor + 1, n\right), & ha\ m \neq n \end{cases}$$

Világos (de azért lássuk is be), hogy *n*! ezzel a függvénnyel kiszámítható a $P(1,n)$ paraméterezéssel. Procedúrahívással számíttassuk ki a 7! számot! Hogyan alakul a verem felépítése, mélysége? Mekkora a minimális méretű verem, amely a feladat elvégzéséhez szükséges? Hány rekurzív hívás lesz a számolás során?

c. A binomiális együttható szimbólum definíciója: $\binom{n}{k} \overset{def}{=} \frac{n!}{k!\cdot(n-k)!}$. A szimbólum kiolvasása „*n* alatt a *k*”. A szimbólumot nemnegativ egész *n* és *k* értékekre definiáljuk, ahol $0 \le k \le n$ fönn kell álljon, továbbá megegyezés szerint $0! \overset{def}{=} 1$. Bizonyítsuk be, hogy $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. Ezt felhasználva adjunk rekurzív módszert $\binom{n}{k}$ kiszámítására. Konkrétan határozzuk meg $\binom{7}{3}$ értékét rekurzívan! Hogyan alakul a verem felépítése, mélysége? Mekkora a minimális méretű verem, amely a feladat elvégzéséhez szükséges? Hány rekurzív hívás lesz a számolás során? Tudnánk-e ugyanezen kérdésekre általános választ is adni?

d. A nemnegatív egész számokon értelmezzük a következő függvényt rekurzív módon:

$$F(n) = \begin{cases} 0, & ha \quad n = 0 \\ 1, & ha \quad n = 1 \\ F(n-1) + F(n-2), & ha \quad n > 1 \end{cases}$$

Procedúrahívással számíttassuk ki az F(7) számot! Hogyan alakul a verem felépítése, mélysége? Mekkora a minimális méretű verem, amely a feladat elvégzéséhez szükséges? Hány rekurzív hívás lesz a számolás során?

e. Legyen az alábbi kétváltozós függvényünk, amelyet nemnegatív egész argumentumokra értelmezünk rekurzívan:

$$P(a,b) = \begin{cases} 0, & ha \quad b = 0 \\ P(2a, b \; div \; 2), & ha \quad b > 0 \; és \; b \; páros \\ a + P(a, b-1), & egyébként \end{cases}$$

Procedúrahívással számíttassuk ki a P(2,5) számot! Hogyan alakul a verem felépítése, mélysége? Mekkora a minimális méretű verem, amely a feladat elvégzéséhez szükséges? Hány rekurzív hívás lesz a számolás során?

9. a. Töltsük ki az alábbi táblázatot!

Jegyek száma / Számrendszer	2	10	16
Szám			
2^2			
10^{10}			
16^{16}			

b. Ha egy pozitív egész szám 34 jegyű 2-es számrendszerben, hány jegyű 16-osban? Adható-e általános formula bináris *n*-jegyű számok esetére? Ha igen, adjon ilyet, ha nem, indokolja meg, miért nem!

c. Ha egy pozitív egész szám 9 jegyű 16-os számrendszerben, hány jegyű 2-es számrendszerben? Adható-e általános formula hexadecimális *n*-jegyű számok esetére? Ha igen, adjon ilyet, ha nem, indokolja meg, miért nem!