Adrián Patrik, 12. évf.
Baross Gábor Középiskola, Szakiskola és Kollégium, Debrecen

A feladat kritikája

A megoldás részletes ismertetése előtt versenyzőként szükségesnek éreztem, hogy a feladatkiírás számomra kritikus pontjait részletesen értelmezzem és rámutassak néhány, a megoldás során felmerült értelmezési és tartalmi problémára.

Ha csak azt nem feltételezzük, hogy a program futtatásához használt számítógép 7 bites bájtot használ, akkor máris elérkeztünk az első problémához. Az ASCII (American Standard Code for Information Interchange) egy 7 bites kódtábla, azaz a $0 \ldots 127$ egész számokhoz rendel karaktereket. Egy 8 bitből álló bájtokat használó számítógépen azonban tetszőleges bináris állományban a bájtok 256 értéket vehetnek fel, aminek következésképp a felét az ASCII nem tartalmazza. A feladat egyértelműen azt mondja, hogy csak az ASCII által nem nyomtatható és a szóköz karakterek helyett jelenítsük meg a numerikus értéküket, egyébként magát a karaktert (bájtot) írjuk ki.

Ez önmagában elfogadható is lenne, hiszen értelmezhető úgy, hogy az ASCII által nem definiált karakterek is kiírandók. (Más értelmezésben pedig nem is mond semmit ezekről a karakterekről.) És akkor eltekintettünk attól, hogy a `0x7F` kódú `DEL` karakter ugyancsak nem nyomtatható (vezérlőkarakter), pedig azt még az ASCII is tartalmazza – a példában azonban bináris alakban szerepel.

Az is kérdéses, hogy a programnak bináris vagy szöveges állományokkal kell dolgoznia, mint bemenet. A tesztállományokból az derül ki, hogy mindkettővel; ekkor azonban célszerű volna kerülni a karakter megnevezést, mert az egyértelműen szöveges fájlokhoz köthető, ami újabb kérdéseket vonhat maga után, mint például a bemenet kódolása. Itt ugyan egyértelmű, hogy az egyes bájtok értékeinek valamely 8 bites karaktertábla kódpontjainak való 1:1 megfeleltetéséről beszélünk, úgy, ahogyan azt az ilyen karaktertábláknál használni szokás. Konkrét utalást azonban erre sem találunk, csupán sejthetjük, hogyan is képezhetők le karakterekre a bemenet bájtjai.

Ha mégis karakterekben gondolkodunk, érdemes volna szót ejteni arról is, hogy a 128–255 tartományba eső értékekhez melyik karaktertábla alapján rendeljük az értékeket. Ha maradunk a Unicode által is használt Latin-1 (ISO-8859-1) karaktertáblánál, akkor azért kerülnek még elő vezérlőkarakterek, egészen pontosan a `0x80`–`0x9F` tartományban. (Persze akármelyik ASCII-kompatibilis kódtáblát is választhatnánk, csak a gyakorisága miatt maradtunk a Latin-1-nél.) Ennél azonban tovább is mehetünk, és feltehetjük azt a kérdést is, hogy mi legyen a `0xA0` és `0xAD` karakterekkel? Előbbihez a Latin-1 és a Unicode a nem törhető szóközt (non-breaking space, NBSP), utóbbihoz a feltételes kötőjelet (soft/discretionary hyphen, SHY) rendeli. Abból kiindulva, hogy a feladat a szóközt is számmal megjelenítve kérte, feltételezhető, hogy kitűzésekor az volt az elsődleges szempont, hogy a betűképpel (glyph) rendelkező karaktereket a program írja ki a kimenetre, a többit pedig a számértékével jelenítse meg.

Ugyancsak kérdéses még az ábécésorrend értelmezése is. (Melyik ábécé szerint? A legtöbb nyelvnek saját ábécéje van.) Azt feltételezzük, hogy a kiírók ezalatt a karakterek kódpontjuk szerinti növekvő sorrendjét értették, ami – az ASCII felépítése miatt – majdnem megfelel az ábécérendnek az angol ábécé különböző betűire, viszont olyan esetekben is értelmezhető, amikor a két karakter a) közül legfeljebb az egyik betű, b) ugyanazon betűnek kis- és nagybetűs alakja c) esetleg az ASCII-n kívül eső tartományból ugyanazon betű valamely, csak az ékezetben eltérő alakja. Helyesen például az *é*, *i*, *U* betűk ebben a sorrendben követik egymást a magyar ábécében, nem pedig ahogyan a Latin-1 kódpontjaiból következik (*U*, *i*, *é*). A példák itt sem követik a leírást, mert kódpont és nem ábécé szerint rendezettek; ellenkező esetben a *T* betűt nem követhetné az *a* betű.

Mindehhez már csak adalék, hogy a feladatleírás a karaktereknek a kódhossz szerint növekvő sorrendű kiírását kérte, azon belül is ábécésorrendben; a minták ezt sem teljesítik.

Mitévő legyen ilyenkor a versenyző? – merülhet fel a kérdés. A *T* betű megelőzze, vagy kövesse az *i* betűt? Az egyik egyezik a nyomtatott mintával, a másik az ábécérenddel. A végső döntés a kódpontok sorrendjének követése és az ábécérend elvetése, több okból is. a) Egyszerűbb. b) A karakterek értelmezéséhez szükség lenne a bemenet minden bájtjának karaktereket megfeleltető karaktertábla meghatározására, amiről nem szólt a feladat. c) Még a karaktertábla ismeretében is valamilyen rendezési sorrend felállítására lenne szükség, ami – még ha a

betűk esetén többé-kevésbé egyértelmű is (az eltérő ábécék azonos latin betűből származtatott alakjainak sorrendje persze kérdéses) – a további karaktereken újból a definíció igényét vetné fel. (Vagy lemásolhatnánk pl. a MySQL `latin1_general_ci` rendezési sorrendjét.) A témával ennél jóval részletesebben foglalkozik a Unicode vonatkozó algoritmusa[1].

Annak eldöntésére pedig, hogy egy karaktert karakterként vagy számként írjunk a kimenetbe végül – *aurea mediocritas* – az `isgraph` függvényt használjuk az alapértelmezett `"C"` lokális beállítással. Az ASCII tartományra ez garantáltan a kiírásnak megfelelően működik, az ezen kívül pedig annyit pontosan tudni róla, amennyi a kiírásban szerepel.

És végül azt is célszerű megjegyezni, hogy a feladat nem rendelkezik azokról az esetekről, amikor a bemenő fájl üres vagy nem tartalmaz különböző értékeket. (Egy karakter ismétlődik.) Előbbi esetben nyilván elintézhetjük az egészet annyival, hogy nem csinálunk semmit; utóbbi azonban azért nehéz kérdés, mert ekkor – a szabály szerint – a fa egyetlen levélcsúcsot tartalmazna; a biteket azonban az élekhez rendeljük, következésképp a gyökércsúcs kódja üres. Ezzel persze lehet kódolni, de dekódolni már nem. Megoldásképp ilyenkor az egyetlen bájt kódjának egyetlen bitet, a 0-t feleltetjük meg, amihez szigorúan véve nem tartozik az algoritmus által előállítható fa, de legalább dekódolható valahogyan. (Ilyenkor persze egyszerűbb lenne a fájl méretét kiírni bájtokban, mert azáltal is rekonstruálható lenne a tartalom, de így legalább csak a kódok előállításának és nem az alkalmazásának a szabályát sértettük meg.)

Az algoritmus

A megoldáshoz használandó algoritmust, a Huffmann-kódolás eljárását már röviden tartalmazza a kiírás, érdemes azonban a feladatot formálisan is megfogalmazni. Adott karakterek $C = \{c_1, c_2, \ldots, c_n\}$ halmaza és ezek $W = \{w_1, w_2, \ldots, w_n\}$ súlya. C pontosan azokat a karaktereket tartalmazza, amelyek a bemenetben szerepeltek, W pedig az egyes karakterek előfordulási gyakoriságát, azaz hogy hányszor fordultak elő. A (c_i, w_i) rendezett párok az algoritmus használatához elegendő információval szolgálnak.

Az algoritmus elején n darab triviális bináris fát hozunk létre, melyek mindegyike egyetlen gyökércsúcsból áll. Minden csúcs tartalmazza a *karakter*, *súly*, *kód*, *bal* és *jobb* mezőket, igaz, ezek nem mindegyikét értelmezzük minden csúcs esetén. Azon csúcsok, melyek *karakter* mezője értelmezve van levelek, a többi mind belső csúcsok. (Az algoritmus úgy építi a fát, hogy ez teljesül.)

Nevezzük szabadnak az olyan csúcsokat, melyeket még nem *választottunk ki*; kezdetben mindegyik ilyen. Ezek után ismételjük az alábbi lépést addig, amíg van legalább két ki nem választott csúcs: Válasszuk ki a még ki nem választott csúcsok közül a két minimális súlyút, legyenek ezek A és B, ezen kívül hozzunk létre egy új csúcsot is, ez legyen P. P súlya legyen A és B súlyának összege, A legyen a bal, B a jobb gyermeke, amit a *bal* és *jobb* mezőkben jelölünk. P biztosan belső csúcs, és mivel újonnan hoztuk létre, még nincs kiválasztva.

Az algoritmus biztosan véget ér, mert minden lépésben 2 csúcsot választunk ki és 1 újat hozunk létre, ezáltal a ki nem választott csúcsok száma mindig 1-gyel csökken, a kezdeti n csúcsból kiindulva $n - 1$ lépésben, ami azt is jelenti, hogy ennyi belső csúcsot hoz az algoritmus létre.

Az egyetlen megmaradó csúcs a kódolási fa gyökere. Hogyan használhatjuk ezt a fát? Legyen a gyökérhez tartozó bináris kód az üres sztring, majd az összes többi csúcshoz rendeljük hozzá a szülőjének kódját ahhoz jobbról egy 0-s vagy 1-es bitet írva attól függően, hogy a csúcs bal vagy jobb gyermeke a szülőjének. Ezzel a fa leveleihez rendelt kódok egy prefixum-mentes kódrendszert alkotnak, vagyis teljesül, hogy egyik kód sem prefixuma a másiknak.

Térjünk még egy kicsit vissza a fa építésére. Azt mondtuk, hogy minden lépésben válasszuk ki a két legkisebb

[1] `http://www.unicode.org/reports/tr10/tr10-22.html`

súlyú csúcsot: de hogyan? Több alternatíva is kínálkozik. A naiv, hogy az összes szabad csúcson végigiterálva válogassuk ki a megfelelőket; ez azonban $O(n)$ komplexitású. Mivel mindig csak a legkisebb elemek meghatározására van szükségünk, ezért használhatunk prioritási sort is, ami megfelelő adatszerkezettel valóban gyorsabban, $O(\lg n)$ időben teszi lehetővé a legkisebb súlyú csúcs megkeresését és eltávolítását.

Egy harmadik megoldás azonban ennél jobbat, $O(1)$ műveletenkénti időt garantál. Ehhez szervezzük az adatokat két *sor* adatszerkezetbe, legyenek ezek L és I. L az algoritmus elején a leveleket tartalmazza olyan sorrendben, hogy azok súlyai az eltávolításuk sorrendjében nemcsökkenő sorozatot alkossanak. A másik sor kezdetben üres, ebbe kerülnek majd a belső csúcsok, súlyuk szerint növekvő sorrendben.

Az algoritmus lényegi lépését megvalósító eljárás a következő: Távolítsuk el abból a sorból az első elemet, amelynél ez kisebb, mint a másiknál, vagy a másik üres; legyen ez az elem A. Tegyük meg ugyanezt még egyszer, a most törölt csúcs legyen B. Hozzuk létre P-t, aminek bal és jobb gyerekei A és B, súlya A és B súlyainak összege, majd helyezzük be P-t az I sor végére.

Azt, hogy ez az algoritmus ugyanazt az eredményt adja, mint a fentebb közölt, azzal bizonyíthatjuk, hogy belátjuk: mindkét sor nemcsökkenő sorrendben tárolja az elemeit. A levelek sorára ez nyilvánvalóan teljesül, mert azt egyszer így megépítjük, ezután nem adunk hozzá csúcsot, tehát ez a tulajdonság nem is sérülhet. Amint I-hez először adunk csúcsot, úgy annak súlya – egyedülálló elem lévén – biztosan nemcsökkenő sorozatot alkot, eddig jó. A további lépések pedig megtartják ezt a tulajdonságot, mert a sorokból törölt két elem súlya mindig a lehető legkisebb, az új csúcs súlya pedig ezek összege. Az előző lépésben, amikor az I sor utolsó eleme bekerült, akkor az a legkisebb képezhető összeg volt. Mivel a két újonnan választott csúcs súlyának összege ennél nem kisebb (különben már hamarabb kiválasztásra kerültek volna), ezért az új csúcs is nem kisebb súlyú lesz, mint a sorban lévő utolsó, ezáltal a nemcsökkenő tulajdonság megmarad. Formálisan ha a két egymást követő lépésben a kiválasztott csúcsok súlyai (a_1, b_1) és (a_2, b_2), akkor mivel $a_1 \le b_1 \le a_2 \le b_2$, $a_1 + b_1 \le a_2 + b_2$ is teljesül, mert az összeadandókra (a tranzitivitás miatt) páronként is teljesül az egyenlőtlenség.

Elemzés

Mielőtt rátérnénk az algoritmus implementációs részleteire, ejtsünk néhány szót az algoritmusunk komplexitásáról is. A bemenet beolvasása – igaz, ez nem a Huffmann-kódolás része – $\Theta(\sum w_i)$ ideig tart. Hogy a karaktereket egyenként konstans időben leszámlálhassuk, egy konstans keresési idejű adatszerkezetet, jelen esetben tömböt használtunk. Hogy ebből előállíthassuk a rendezett L sort egy összehasonlító rendezési algoritmust, a gyorsrendezést használjuk, ami átlagosan $O(n \lg n)$, legrosszabb esetben $O(n^2)$ időt vesz igénybe. Az algoritmus egy lépésének végrehajtása konstans idejű, de mivel $n-1$ lépésből áll, ezért összesen $(n-1) \cdot O(1) = O(n)$ komplexitást ad.

A fa bejárása ismét $O(n)$ időt igényel, a kódok előállítása egyenként $O(b)$ időt, ahol b a maximális kódhossz, ami megegyezik a fa d magasságával. Ez összesen $O(nd)$ idő. A kódok ismeretében a leveleket újból rendezni kell, ami egy újabb gyorsrendezést jelent átlagosan $O(n \lg n)$ idő alatt. Végül a kódok kírása egy újbóli $O(nd)$ komplexitású feladat, amivel véget is ér a program.

A memóriaigény $O(n) + O(d)$.

Fontos még megemlítenünk, hogy a csúcsok maximális száma egy valódi implementációban nem függ a felhasználó által megadott bemenettől, csak a futtató architektúrán egy bájtként kezelt bitek számától. Ha egy bájt 8 bitet tartalmaz, akkor a fának legfeljebb $2^8 = 256$ levele lehet. (Általánosan ha egy bájt k bites, akkor a levelek maximális száma 2^k.)

A program algoritmusa ugyan gyors és várhatóan kevés csúccsal dolgozik, elég nagy bemenetre azonban a futásidő jelentősen megnőhet a fájlműveletek lassúsága miatt; ez ellen sajnos nem igazán tehetünk semmit, mert a

késés a háttértárak sebességéből adódik.

Az implementáció

A programban a fa csúcsainak tárolására létrehoztunk egy `struct node` típust, ami tárolja az adott csúcshoz rendelt súlyt (`weight`), a gyermekeit (`left`, `right`), a hozzárendelt bitsorozatot karaktersorozatként (`encoded_form`) és ennek hosszát (`encoded_length`). A csúcsok nem tartalmazzák a hozzájuk rendelt karaktert, mert az a csúcs memóriacíméből visszakövetkeztethető. Az `unsigned char` típust `byte`-ra `typedef`eltük, ami így jobban tükrözi a funkcióját. (Az `unsigned char` a szabvány szerint mindig 1 bájtos.) A másik `typedef`-ünk a `long long` típusnak ad új nevet (`lsize_t`); ez egy biztosan legalább 64 bites változó, ami megfelelően tudja számolni azon eseteket is, ahol 32 bit már nem elég. Az egy bájton tárolható különböző értékek számát a `CHAR_BIT` értékéből számolva a program `MAXNODES` néven tárolja; ennyi levele lehet a fának.

A szükséges memória nagyrészt statikus foglalású. A `leaves` és `inner` tömbök a fa levél és belső csúcsait tárolják, míg a `leaf_queue` és `inner_queue` az ezekre való mutatókat tároló sorok, amiknek részletes funkciójára az algoritmus bemutatásánál kitértünk. A `_size` és `_front` végződésű változók a sorok elemeinek számát és az első elemek indexeit tárolják. A `codestack` verembe a fa bejárásakor mindig az éppen aktuális csúcshoz tartozó kód kerül. A grafikusan (karakterként) és numerikusan megjelenítendő karakterek számára is globálisan definiáljuk a formátumsztringeket.

A `filebuf` puffert a beolvasáskor használjuk azért, hogy ne egyesével kelljen a bájtokat kérni, ami nagyobb fájlokra igencsak lassú lenne a sok függvényhívás miatt; a puffer mérete 64 KiB. A program 32 bites alkalmazásként fordítva lehet, hogy csak korlátozott méretű fájlokat tud kezelni (a pontos érték implementációfüggő); 64 bites rendszeren ilyen probléma a gyakorlatban aligha merülhet fel. Fontos, hogy a bemenetet mindig bináris módban nyissuk meg, mert így a standard könyvtár nem végez fordítást.

A `main` függvénynek még az elején rögzítjük egy `atexit` hívással, hogy a program kilépéskor a fájlokat zárja le, így ezzel a programkódban nem kell törődni. A bemenet megnyitása után beolvassuk a fájlt és minden bájtnál megnöveljük az őt reprezentáló csúcs súlyát. Ezután kiszűrjük a bemenetben nem szereplő értékeket: csak azok kerülnek a `leaf_queue`-ba, melyeknek legalább 1 a súlya. Ha nincs ilyen érték, akkor a bemenet üres volt, kilépünk.

A `leaf_queue` elemeit súlyuk szerint növekvően rendezzük, majd végrehajtjuk a már leírt fakészítő algoritmust. Ekkor az `inner_queue` egyetlen megmaradó eleme, amire `inner_queue[inner_queue_front]` mutat, a fa gyökere. (Megj.: Az `inner_queue` tömb ugyanolyan sorrendben tárolja az elemeket, mint az `inner`. Erre a redundanciára azért van szükségünk, mert a `leaf_queue` a rendezés miatt eltér a `leaves` sorrendjétől, és így egyeznek a típusok.)

Miután a fát előállítottuk, inorder módon egy rekurzióval bejárjuk, ami során a leveleknél rögzítjük (lemásoljuk) az aktuális kódot, amit a `codestack` tárol. Ehhez dinamikusan foglaljuk le a memóriát. Az `encoded_length` változót kiválthatnánk az `strlen(encoded_form)` hívással, de így megspórolunk valamennyi időt.

A kódok ismeretében a levélcsúcsokat újból rendezni kell, ezúttal a kiírásnak megfelelő feltétel szerint: elsősorban `encoded_length` szerint, egyezés esetén kódpont-sorrendben. Ehhez a `leaf_queue` tömböt használjuk fel, csupán helyspórolás miatt. Végül a tömbön végigiterálva kiírjuk a fa leveleit a megfelelő formában, majd a standard kimeneten is megjelenítjük a szükséges információt.

A program az ISO/IEC 9899:1999 C szabványnak megfelelően, C nyelven készült. Az UTF-8 kódolású programkód fordítható gcc-vel a `gcc s68.c -o s68.exe` vagy a `cl s68.c` paranccsal Visual C++ használata esetén. A forráskód UTF-8 kódolású.