Fibonacciho kód

Termín odevzdání:	**27.03.2023 11:59:59**	823341.913 sec
Pozdní odevzdání s penalizací:	**21.05.2023 23:59:59** (Penále za pozdní odevzdání: 100.0000 %)	
Hodnocení:	**5.5000**	
Max. hodnocení:	**5.0000** (bez bonusů)	
Odevzdaná řešení:	19 / 30 Volné pokusy + 20 Penalizované pokusy (-2 % penalizace za každé odevzdání)	
Nápovědy:	4 / 3 Volné nápovědy + 2 Penalizované nápovědy (-10 % penalizace za každou nápovědu)	

Úkolem je vytvořit C/C++ funkce, které dokáží překódovat soubor v kódování UTF8 do Fibonacciho kódu a zpět.

Znaky ukládáme jako čísla - indexy do tabulky kódů, dnes nejčastěji UNICODE. Znaků je v UNICODE mnoho - 1048576+65536. Pokud by byly ukládané přímo jako binární čísla, byly by potřeba na reprezentaci jednoho znaku vždy 4 bajty. Většina znaků ale má kódy nižší, např. hodnoty menší než 128 (65536). Toho využívá kódování UTF-8, které podle zapisované hodnoty znak ukládá kompaktněji do 1 až 4 bajtů.

Kódování UTF-8 pro běžné textové soubory snižuje celkový objem ukládaných dat, není ale optimální. Při návrhu kódování UTF-8 byl zvolen způsob, kdy kódy jednotlivých znaků vždy končí na hranici celého bajtu. To je praktické pro rychlé načítání, ale není to optimální z hlediska délky.

Pro paměťově efektivní ukládání celých čísel byly vyvinuté jiné kódy, např. kód Fibonacciho, který je založen na Fibonacciho posloupnosti. Připomeňme, že Fibonacciho posloupnost je posloupnost celých čísel, kde každé další číslo posloupnosti vznikne jako součet dvou předešlých čísel. Posloupnost má podobu 1, 1, 2, 3, 5, 8, ... Čísla ve Fibonacciho kódu se ukládají jako bitové posloupnosti, které nastaveným bitem udávají přítomnost příslušného prvku Fibonacciho posloupnosti. Například číslo 9 lze zapsat jako:

```
                          1  1  2  3  5  8  ....
  9 = 1 + 8               1  0  0  0  0  1
  9 = 1 + 8               0  1  0  0  0  1
  9 = 1 + 3 + 5           1  0  0  1  1
  9 = 1 + 1 + 2 + 5       1  1  1  0  1
  ...
```

Možných zápisů je mnoho. Ze všech variant se vyberou pouze takové zápisy, kde se žádné číslo z Fibonacciho posloupnosti neopakuje vícekrát (tedy zápis 1 + 1 + 2 + 5 nebude použit). Vzhledem k tomuto omezení nemusíme ve Fibonacciho posloupnosti uvažovat dvě počáteční jedničky, šla by stejně použít pouze jedna z nich. Dále nepřipustíme zápisy, kde se vyskytují sousední členy Fibonacciho posloupnosti (tedy zápis 1 + 3 + 5 bude také nepřípustný, 3 a 5 jsou sousedi). Dvojici sousedních prvků Fibonacciho posloupnosti lze vždy nahradit jejich součtem, tedy následujícím prvkem v posloupnosti.

S těmito omezeními je již Fibonacciho kód jednoznačný. Pro prvních několik celých čísel má podobu:

```
                          1  2  3  5  8
  1 = 1                   1
  2 = 2                   0  1
  3 = 3                   0  0  1
  4 = 1 + 3               1  0  1
  5 = 5                   0  0  0  1
  6 = 1 + 5               1  0  0  1
  7 = 2 + 5               0  1  0  1
  8 = 8                   0  0  0  0  1
  9 = 1 + 8               1  0  0  0  1
 10 = 2 + 8               0  1  0  0  1
 11 = 3 + 8               0  0  1  0  1
 12 = 1 + 3 + 8           1  0  1  0  1
 ...
```

Celá čísla tedy lze ukládat pomocí Fibonacciho kódu. Všimněte si, že malá čísla jsou kódována kratšími bitovými sekvencemi, velká čísla pak sekvencemi delšími. Problémem je ale rozpoznání, kde kód jednoho čísla končí a kde začíná kód další. Pro binární zápis s fixním počtem bitů je to triviální - bitové posloupnosti rozdělíme do skupin po 8/16/32/64 bitech podle fixní velikosti datového typu. Zde bychom ale chtěli pro úsporu místa použít proměnlivý počet bitů. Představme si situaci, kdy na vstupu je sekvence bitů `00101`. Tu lze chápat jako jedno číslo (11) nebo jako dvě čísla 3 a 2, pokud bychom sekvenci bitů rozdělili do skupin `001 01`.

Pro označení konce čísla musíme do sekvence přidat domluvenou značku. Pro Fibonacciho sekvenci je to snadné, mezi čísla přidáme jeden bit s hodnotou 1. Protože kód vždy končí jedničkou, přidáním další jedničky dostaneme sekvenci dvou jednotkových bitů. Vzhledem k vlastnostem Fibonacciho kódu se taková sekvence nemůže vyskytovat uprostřed kódovaného čísla, tedy bude jednoznačně identifikovat konec kódu. Samotné číslo 11 z příkladu by tedy bylo kódované jako posloupnost `001011`, posloupnost dvou čísel 3 a 2 by se zakódovala jako `0011011`.

Fibonacciho kód v daném provedení nedokáže uložit hodnotu 0. Toto ošetříme jednoduše tím, že celý kód posuneme o 1 pozici:

```
                          1  2  3  5  8
  0 ~ 1                   1  1
  1 ~ 2                   0  1  1
  2 ~ 3                   0  0  1  1
  3 ~ 1 + 3               1  0  1  1
  4 ~ 5                   0  0  0  1  1
  5 ~ 1 + 5               1  0  0  1  1
  6 ~ 2 + 5               0  1  0  1  1
  7 ~ 8                   0  0  0  0  1  1
  8 ~ 1 + 8               1  0  0  0  1  1
  9 ~ 2 + 8               0  1  0  0  1  1
 10 ~ 3 + 8               0  0  1  0  1  1
 11 ~ 1 + 3 + 8           1  0  1  0  1  1
 ...
```

Poslední problém spočívá v zarovnávání. Pokud máme zakódovat např. posloupnost čísel 0 5 10, pak potřebujeme uložit celkem 2 + 5 + 6 = 13 bitů. Do souboru jsme schopni ukládat na celé bajty, nevyužité bity (max 7 na konci souboru) tedy nastavíme na hodnotu nula. Díky tomu, že toto doplnění (padding) neobsahuje sekvenci dvou po sobě jdoucích jedniček, nemůže toto doplnění nechtěně přidat další kódovaný znak.

`Ukázka zakódování sekvence znaků s hodnotami 0 5 10:`

```
Kódy znaků:              0   5      10
Bitové sekvence:         11  10011  001011
Seskupení:               1110011001011
Doplnění na celé bajty:  1110011001011000
Bajty (LSB vlevo):       11100110   01011000
Bajty (LSB vpravo):      01100111   00011010
Bajty hexadecimálně:     67         1A
```

Poznámka: LSB znamená Least Significant Bit. Při vysvětlování kódů jsem uvažoval zápis s LSB vlevo, kódy jsou názornější. Při zápisu bajtů a v hexdumpu však pracujeme s LSB vpravo. Pro implementaci si stačí rovnou zvolit konvenční postup (LSB vpravo) a tomu podřídit skládání bitů do bajtu. Zápis s LSB vlevo chápejte pouze jako pomůcku pro snazší pochopení.

Úkolem je realizovat dvě funkce s rozhraním níže. Obě funkce mají parametrem dvě jména souborů: zdrojový a cílový. Funkce čtou zdrojový soubor a zapisují překódovaný výsledek do cílového souboru. Návratovou hodnotou obou funkcí je příznak úspěchu (`true`) nebo chyby (`false`). Pokud se během požadované operace cokoliv nepodaří (otevřít soubor / vytvořit soubor / číst zdroj / zapisovat cíl / nesprávný formát dat / ...), funkce bude vracet hodnotu `false`.

Poznámky:

- Pečlivě ošetřujte souborové operace. Testovací prostředí úmyslně testuje Vaši implementaci pro soubory neexistující, nečitelné nebo soubory s nesprávným datovým obsahem.
- Princip kódování pomocí UTF-8 byl naznačen v PA1, detailní informace najdete na Wikipedii nebo v e-learningových materiálech PA1.
- Testovací prostředí předkládá soubory, které jsou neplatné:
  - jsou UTF-8 kódy znaků poškozené (např. chybí nějaké mezilehlé bajty kódu znaku, kódovaný znak je mimo rozmezí 0 až 0x10ffff),
  - Fibonacciho kódy jsou poškozené (kódováno je číslo překračující rozsah UNICODE 0 až 0x10ffff, neukončený Fibonacciho kód). Každý znak kódovaný Fibonacciho kódem musí být ukončen dvojicí po sobě jdoucích jedničkových bitů (viz vysvětlení nahoře). Jedinou výjimku tvoří případná sekvence nulových bitů použitá jako padding na konci souboru.

Očekávanou reakcí na neplatné vstupy je samozřejmě navrácení hodnoty neúspěch (`false`).

- Při implementaci lze použít C i C++ rozhraní pro práci se soubory, volba je na Vás.
- Testovací prostředí zadává náhodná jména souborů. Neočekávejte, že soubory budou mít nějaká konkrétní jména nebo přípony.
- V přiloženém archivu ale najdete sadu testovacích souborů v UTF8 (přípona .utf8) a jim odpovídajících ekvivalentů ve Fibonacciho kódu (přípona .fib). Testovací sada obsahuje i neplatné soubory, pro takové chybí jejich ekvivalent ve druhém kódu.
- Vstupní a výstupní soubory mohou být velké, větší než je velikost dostupné paměti. Obecně se proto při práci se soubory snažíme data zpracovávat průběžně. Není rozumné celý vstupní soubor načíst do paměti a pak jej v paměti zpracovávat. Poslední test kontroluje paměťové nároky Vašeho řešení. Selže, pokud se pokusíte udržovat v paměti najednou celé soubory nebo jejich velké části.

Vzorová data: Download

Odevzdat: Choose File No file chosen Odevzdat

Referenční řešení

- **Hodnotitel: automat**
  - Program zkompilován
  - Test 'Zakladni test se soubory z ukazky': Úspěch
    - Dosaženo: 100.00 %, požadováno: 100.00 %
    - Celková doba běhu: 0.002 s (limit: 10.000 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Nespravne vstupy (utf8ToFib)': Úspěch
    - Dosaženo: 100.00 %, požadováno: 50.00 %
    - Celková doba běhu: 0.004 s (limit: 9.998 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Nespravne vstupy (fibToUtf8)': Úspěch
    - Dosaženo: 100.00 %, požadováno: 50.00 %
    - Celková doba běhu: 0.000 s (limit: 9.994 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Nahodny test (utf8ToFib)': Úspěch
    - Dosaženo: 100.00 %, požadováno: 50.00 %
    - Celková doba běhu: 0.272 s (limit: 9.994 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Nahodny test (fibToUtf8)': Úspěch
    - Dosaženo: 100.00 %, požadováno: 50.00 %
    - Celková doba běhu: 0.338 s (limit: 9.722 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Test osetreni I/O chyb': Úspěch
    - Dosaženo: 100.00 %, požadováno: 50.00 %
    - Celková doba běhu: 0.596 s (limit: 9.384 s)
    - Úspěch v závazném testu, hodnocení: 100.00 %
  - Test 'Test pametove narocnosti': Úspěch
    - Dosaženo: 100.00 %, požadováno: 50.00 %
    - Celková doba běhu: 1.468 s (limit: 15.000 s)
    - Úspěch v nepovinném testu, hodnocení: 100.00 %

- Celkové hodnocení: 100.00 % (= 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00 * 1.00)

- Celkové procentní hodnocení: 100.00 %
- Bonus za včasné odevzdání: 0.50
- Celkem bodů: 1.00 * (5.00 + 0.50) = 5.50

SW metriky:

	Celkem	Průměr	Maximum	Jméno funkce
Funkce:	**9**	--	--	--
Řádek kódu:	**179**	**19.89 ± 12.17**	38	`COutWrapper::writeFibonacci`
Cyklomatická složitost:	**45**	**5.00 ± 3.37**	12	`CInWrapper::readUtf8`