Zadanie: PROCESSOR
Procesor

dzień 3, plik źródłowy `processor.*` *CPSPC 2004, 11 czerwca*

Jeden z liderów na rynku nowych technologii[...][1], słynny wynalazca prof. I. N. Tell [2] zaprojektował prototyp nowego procesora, który nazwał Hexadekatium. Niestety, instrukcje kodowane są na tym procesorze w sposób, delikatnie mówiąc, nieefektywny [16], a jak wiadomo 640KB w zupełności wystarcza do wszystkich zastosowań[640]. Dlatego profesor musi wymyslić metodę optymalizaji programów skracającą ich długość.

Profesor Tell chce stworzyć optymalny optymalizator — taki program, który dla danych źródeł innego programu P, wypisze najkrótszy (czyli taki o minimalnej liczbie instrukcji w kodzie źródłowym) możliwy program Q, który oblicza to samo co P (to znaczy, że dla każdych danych wejściowych, wynik dawany przez Q jest identyczny z wynikiem P). Jedynym wyjątkiem są dane wejściowe dla których P nie zatrzymuje się – w tym przypadku, wynik Q może być dowolny, w szczególności Q nie musi się zatrzymać.

Zadanie to okazało sie ekstremalnie trudne[†]. Profesor Tell szczerze wierzy, że mu się to kiedyś uda, jakkolwiek musi on jak najszybciej rozpocząć sprzedaż swoich procesorów. Dlatego zdecydował sie uprościć problem poprzez rozważanie tylko programów, których wejściem i wyjściem są pojedyncze 32-bitowe liczby, zdecydował się on również na zastąpienie programu optymalizującego tanią siłą roboczą. Rozprowadza on programy wsród swoich studentów, a oni tworzą najkrótsze jakie są w stanie równoważne programy. W zależności od długości stworzonych programów, otrzymują oni premie pieniężne różnej wysokości. Zauważyleś w tym swoją szansę na zdobycie małej fortuny.

Opis procesora

Hexadekatium ma 26 rejestrów nazwanych od `a` do `z`. Każdy z tych rejestrów przechowuje 32-bitową liczbę bez znaku.

Procesor ten działa pod kontrolą programu, będącego ciągiem co najwyżej 1000 instrukcji następującej postaci:

[*etykieta*:] [`if` *wartość relacja wartość*] *polecenie*

[części w nawiasach kwadratowych mogą zostać pominięte]. *etykieta* nie ma wpływu na znaczenie instrukcji, ale umożliwia odwoływanie się do tej instrukcji w skokach i indeksowanych wartościach. Część z `if` pozwala na uruchamianie warunkowe instrukcji (zobacz poniżej).

Dostępne są następujące *polecenia*:

- *rejestr* = *wartość* — zapisuje *wartość* do danego rejestru, gdzie *wartość* to:
  - *liczba* — liczba w systemie dziesiętnym (0...4294967295), lub
  - `#`*liczba* — liczba w systemie szesnastkowym (0...`ffffffff`), lub

[1] co najmniej 1463 marketingowych sloganów zostało pominiętych w trosce o zdrowie czytelnika
[2] pra-pra-...-wnuk legendarnego Szwajcarskiego bohatera Wilhelma Tella
[16] 16 kilobajtów na instrukcję
[640] Bill Gates, 1981
[†] Matematycy już dawno temu udowodnili, że jest to w ogólności niemożliwe, ale ortodoksyjny biznesmen oczywiście nie wierzy w takie bzdury...

- *etykieta*[*wartość*] — zawartość instrukcji `data` znajdującej się *wartość* instrukcji za *etykietą* (*wartość* zero używa instrukcji wskazywanej przez *etykietę*, 1 instrukcji następnej, i tak dalej). Jeżeli taka instrukcja nie istnieje (wskazywana instrukcja nie jest instrukcją `data` lub znajdowałaby się poza końcem programu), zwracane jest zero. Nawiasy kwadratowe nie mogą zawierać kolejnych *wartości* z nawiasami kwadratowymi.

- *rejestr* `=` *wartość operator wartość* — wykonuje pojedynczą operację na dwóch wartościach i zapisuje wynik w *rejestrze* (jeżeli pewien rejestr pojawia sie po obydwu stronach `=`, nowa wartość jest zapisywana do rejestru po odczytaniu obydwu *wartości*). Dostępne są następujące operacje:
  - `+`, `-`, `*` — standardowe operacje arytmetyczne; jeżeli wystąpi przepełnienie, wynik brany jest modulo 2^{32},
  - `/`, `%` — standardowe całkowitoliczbowe dzielenie i operacja modulo; dzielenie x przez 0 daje wynik 0 i resztę x,
  - `&`,`|`, `^` — bitowe *i*, *lub* i *xor*,
  - `<<`, `>>` — bitowe przesunięcie: x `<<` y zwraca x przesunięte y bitów w lewo (jeżeli $y \geq 32$, wynikiem jest zawsze zero), x `>>` y to przesunięcie w prawo [$1 << 5 = 32$, $9 >> 2 = 2$],
  - `<<<`, `>>>` — bitowy obrót: obrót działa tak samo jak przesunięcie tyle tylko, że bity przesunięte poza 32-bitową wartość wstawiane są z drugiej strony [$9 >>> 2 = \#40000002$].

- `jump` *etykieta* — skocz do instrukcji oznaczonej przez *etykietę* i kontynuuj od niej dalsze wykonywanie programu.

- `halt` — zakończ wykonywanie programu,

- `data` [`#`]*liczba* — to polecenie nie robi nic poza umożliwieniem przechowywania danej liczby, która może być potem odczytana przy pomocy *etykieta*[*wartość*].

Każda instrukcja może być poprzedzona opcjonalnym warunkiem typu `if` *wartość relacja wartość*, gdzie *relacją* może być `=`, `<>`, `<`, `>`, `<=` lub `>=`. Instrukcja działa wtedy tak, jak opisano powyżej tylko jeżeli warunek jest spełniony, w przeciwnym przypadku nie robi nic.

W uzupełnieniu do instrukcji, program może zawierać również komentarze: gdziekolwiek wystąpi znak `'`, wszystko do końca linii jest pomijane. Puste wiersze są także pomijane. Małe i wielkie litery mogą być wszędzie stosowane zamiennie.

Wykonywanie programu

Na początku uruchomienia programu, rejestr `a` zawiera wejście programu (pojedynczą liczbę 32-bitową), rejestry od `b` do `z` zawierają zera, a uruchomienie zaczyna się od pierwszej instrukcji programu. Program zatrzymuje się, kiedy wykona instrukcję `halt`, bądź „wyjdzie poza koniec programu" (to znaczy spróbuje wykonać instrukcje poza jego końcem). Wtedy wartość w rejestrze `z` będzie zinterpretowana jako wyjście programu, a zawartość wszystkich pozostałych rejestrów zostanie zignorowana.

Po wykonaniu 1 000 000 instrukcji, program jest uznawany za zapętlony. (Oznacza to, że jeżeli pierwotny program nie zatrzymuje się w tym czasie dla danego wejścia x, to Twój program może dla tego wejścia robić cokolwiek, w szczególności może się zapętlić.)

Wejście i wyjście

Programy, które masz zoptymalizować znajdują się w plikach `processor`n`.in` ($1 \leq n \leq 5$) Pliki te możesz znaleźć w katalogu `/aux/mo/public/problems/processor/`. Jako rozwiązanie powinieneś wysłać zoptymalizowane wersje tych programów (czyli najkrótsze równoważne programy jakie będziesz w stanie stworzyć). Możesz to zrobić zapisując je do plików `processor`n`.out` i wywołując polecenie `submit processor` n. Pamiętaj, że polecenie `submit` musi zostać uruchomione dla każdego pliku oddzielnie. Przydatne może się okazać polecenie `status`.

Symulator

Dostępny jest także symulator procesora Hexadekatium. Możesz uruchomić go poprzez wywołanie polecenia `hdsim` [`-g`] *program* [*wejście* ...], gdzie *program* jest nazwą twojego programu, *wejście* to liczba użyta jako wejście dla programu (uwaga, `#` jest używane w Uniksowej powłoce jako komentarz, dlatego musisz napisać `\#`). Jeżeli nie podasz *wejścia*, zostanie jedynie sprawdzona poprawność składniowa programu. Jeżeli podasz kilka *wejść*, program zostanie uruchomiony oddzielnie dla każdego z nich.

Jeżeli użyjesz przełącznika `-g`, symulator wypisze szczegółowy raport z uruchomienia.

Ocenianie

Każdy program, który zgłosisz, będzie oceniany oddzielnie. Zostanie on porównany z pierwotnym programem na pewnym zbiorze wejść. Jeżeli okaże się, że daje on różny wynik na którymkolwiek z wejść, otrzymasz 0 punktów. Jeżeli Twój program będzie dłuższy lub tak samo długi jak program pierwotny, również otrzymasz 0 punktów. W każdym innym przypadku, otrzymasz punkty zgodnie z następującym wzorem:

$$\max\left(20 - 5 \cdot (y - b),\ 1\right),$$

gdzie y jest długością Twojego rozwiązania (w instrukcjach), a b jest długością najlepszego znanego rozwiązania (włączając nasze).

Przykład 1

Poniższy program oblicza liczbę dzielników danej liczby:

```
        b=1                     ' 1 jest pierwszym kandydatem na dzielnik
pool:   c=a%b                   ' czy dzieli a?
        if c=0 z=z+1            ' jezeli tak, zwieksz z
        b=b+1                   ' kolejny kandydat na dzielnik
        if b<=a jump pool       ' sprobuj ponownie
```

Przykład 2

Następujący program zwraca binarne odbicie lustrzane danej liczby (to jest liczbę, której zapis binarny jest taki sam jak odczytany od tyłu zapis binarny danej liczby):

```
        b=32                       ' liczba bitow
loop:   c=a&1                      ' wyjmij najmniejszy bit w a
        a=a>>1
        z=z<<1                     ' i wstaw go od dolu do z
        z=z|c
        b=b-1                      ' iteruj
        if b>0 jump loop
```