Припрема за писмени

(решења одабраних домаћих задатака: стрингови, низови, битови, рекурзија, упити распона)

1.
Низ слова ABACABADABACABAEABACABADABACABAFABACA... се може формирати на следећи начин:

1. Низ је на почетку празан.
2. На низ се допише прво следеће веће (лексикографски гледано) велико слово енглеског алфабета које се не појављује у формираном делу низа, а иза тог слова се понове сва слова која су се појавила пре њега.
3. Корак 2 се понови потребан број пута

Тако после прве примене корака 2 добијамо низ A, после друге низ ABA, после треће низ ABACABA итд.

Одредити слово које се појављује на n-том месту у низу, бројећи места од 1. Лексикографски редослед слова у енглеском алфабету је ABCDEFGHIJKLMNOPQRSTUVWXYZ.

Улаз: Један природан број мањи од 67 108 864.

Излаз: Једно велико слово енглеског алфабета.

Пример 1
Улаз
8

Излаз
D

Пример 2
Улаз
65

Излаз
A

Пример 3
Улаз
100

Излаз
C

Решење:

Решење грубом силом је да се у меморији креира цео низ карактера и да се затим прочита карактер са одговарајуће позиције. Ово решење троши превише меморије, а и времена док се дугачак низ карактера не изгради. Тестирајте решење и видећете да и на аутоматској евалуацији ћете добити МЛЕ грешку на већим тест примерима.

```
#include <iostream>
#include <string>

using namespace std;

int main() {
  unsigned n;
  cin >> n;
  string s = "";
  char slovo = 'A';
  while (s.size() <= n - 1) {
    s = s + slovo + s;
    slovo++;
  }
  cout << s[n-1] << endl;
}
```

Задатак се може решити без креирања дугачке ниске карактера, ако пажљиво проучимо правилност по којој се слова појављују.
Прво слово `A` се налази на месту 1,
прво слово `B` на месту `2`,
прво слово `C` на месту `4`,
прво слово `D` на месту `8` итд.,
па се може наслутити
да се прва појављивања слова налазе на местима која су степени броја 2.

Дакле, ако дужину низа карактера пре уметања новог слова абецеде обележимо са d[k], важи да је d[0] = 0 (пре уметања слова `A` не налази се ниједан карактер)
и да је d[k+1] = 2d[k] + 1 (пре уметања новог слова налази се ниска која је добијена тако што је претходно слово спојено са два појављивања ниске која се појављивала пре тог претходног слова).

Зато је d[k] = 2^k - 1 (важи да је 2^0 - 1 = 0 и да је 2^{k+1} - 1 = 2 (2^k - 1) + 1, док је прва позиција слова k (ако се слова броје од нуле) једнака 2^k.

Дакле, ако је унети број n неки степен двојке n = 2^k, онда је у питању позиција на којој се слово први пут појављује и важи да је k =$\log_2$ n
(Подсетимо се да $\log_2$ n означава број k такав да је n = 2^k, нпр. $\log_2$ 256 = 8, јер је 2^8 = 256).
Дакле k-то слово можемо одредити сабирајући вредност k са ASCII/UNICODE кодом слова `A`.

У супротном, проблем можемо свести на проблем мање димензије. Нека је 2^k < n < 2^{k+1}, тј. нека је k највећи степен двојке мањи од n.
Карактер који тражимо налази се, дакле, иза позиције 2^k у делу низа који је добијен копирањем дела низа испред позиције 2^k.
Зато је n-ти карактер у целом низу једнак карактеру који је на позицији n - 2^k у копираном делу, а пошто део који се копира почиње на почетку низа, једнак је n-2^k-том карактеру у целом низу.

Овим смо описали рекурзивни поступак којим ефикасно можемо доћи до решења.

$$f(n)=\log_2 n, за\ n=2^k\ односно$$
$$f(n)=f(n-2^k), за\ 2^k<n<2^{k+1}$$

На пример, f(23) = f(23-16) = f(7) = f(7 - 4) = f(3) = f(3 - 2) =f(1) = 0,
па се на месту 23 налази слово `A`.
Слично, на пример, важи да је
f(40) = f(40 - 32) = f(8) = 3, па се на месту 40 налази слово `D`.

На основу претходне дефиниције би се могла имплементирати рекурзивна функција (за то би било потребно испитати да ли је дати број степен броја 2, наћи логаритам таквог броја, и наћи највећи степен броја 2 од ког је дати број већи или једнак). Ипак, за тим нема потребе.

Анализирајући рад такве рекурзивне функције можемо унапред закључити шта ће њен резултат бити, без потребе за њеним извршавањем.
Претпоставимо да знамо бинарни запис броја n тј. да знамо да се број n представља као збир неких степенова двојке
$$2^{s_m}+2^{s_{m-1}}+...+2^{s_0}$$

Током рекурзије од броја ће се одузимати један по један степен двојке, све док не остане само 2^{s_0} и тада ћемо знати да је k = s_0. Дакле важи да је резултат једнак најмањем степену двојке који учествује разлагању броја на збир

степенова двојке тј. да је тражени број k једнак позицији најдешње јединице у бинарном запису броја (претпостављамо да се позиције броје од 0, здесно). До траженог резултата је могуће доћи дељењем броја са 2_0^s , тј. узастопним дељењем броја са 2 све док последњи сабирак не постане 1, тј. све док је број паран (то ће се догодити тачно s_0 пута).

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  int k = 0;
  while (n % 2 == 0) {
    n /= 2;
    k++;
  }
  cout << (char)('A' + k) << endl;
  return 0;
}
```

Сличан резултат можемо добити и баратајући директно са интерним записом броја у рачунару, коришћењем битовских оператора. Позицију крајње десне јединице једноставно можемо израчунати шифтовањем (померањем) бинарног записа броја удесно за једно место све док последња цифра не постане једнака 1 (последњу цифру можемо испитати битовском конјункцијом са 1).

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  int k = 0;
  while ((n & 1) == 0) {
    n >>= 1;
    k++;
  }
  cout << "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[k] << endl;
  return 0;
}
```

Савремени хардвер често поседује и инструкцију *count trailing zeroes* којом се одређује број нула иза последње јединице у бинарном запису (што је баш позиција последње јединице). Иако програсмки језици обично не стандардизују приступ овој операцији, неки компилатори нуде подршку за то (на пример, ако се користи GCC, може се употребити функција `__builtin_ctz`, а ако се користи Microsoft Visual C++, може се употребити функција `_BitScanReverse`).

```
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  cout << (char)('A' + __builtin_ctz(n)) << endl;
  return 0;
}
```

2. Напиши програм који израчунава вредност аритметичког израза у којем се јављају природни бројеви и између њих оператори `+`, `-` и `*`. Нпр. `3+4*5-7*2`.

Улаз: Једина линија стандардног улаза садржи описани израз.
Излаз: Стандардни излаз треба да садржи само тражену вредност учитаног израза.
Пример
Улаз
3+4*5-7*2
Излаз
9

Решење:

1. Рачунање вредности текућег сабирка

Једна могућност је да памтимо вредност израза без последњег сабирка и посебно вредност последњег (текућег) сабирка. Кад кажемо сабирак, мислимо на операнд испред ког стоје аритметички оператори +`, `-`.

Вредност последњег сабирка иницијализујемо на 1, читамо број и оператор иза њега, вредност последњег сабирка множимо са бројем и ако је прочитани оператор `+` или `-` или ако смо стигли до краја ниске, завршили смо управо са израчунавањем тог сабирка, и његову додајемо или одузимамо од резултата у зависности од тога који је био претходни адитивни оператор. Ако смо прочитали оператор `+` или `-`, памтимо га.

Вредност првог сабирка можемо израчунати засебно и резултат иницијализовати на њега, а можемо на почетку претпоставити да је претходни адитивни оператор био `+` (иако он не пише испред израза) и први сабирак обрадити као и све остале.

Прикажимо рад овог алгоритма на изразу `8+4*5*2-7*2`.
На почетку, вредност резултујућег израза постављамо на нулу, вредност претходног сабирка на један, а претходни оператор на `+` (еквивалентно, можемо увести променљиву у којој памтимо знак сабирка и можемо је иницијализовати на 1).
Након читања броја `8`, множимо претходни сабирак који има иницијалну вредност `1` са њим, добијамо `8` и пошто смо наишли на`+`, завршили смо са обрадом једог сабирка и вредност текућег резултата која је `0` увећавамо за производ вредности `8` (то је вредност управо обрађеног сабирка) и `1` (то је вредност знака).

Вредност знака опет постављамо на `1` (јер смо наишли на `+` и у наредном кораку ће опет бити потребно увећање вредности), а вредност текућег сабирка поново враћамо на иницијалну вредност `1`. Читамо затим `4` и множимо текући сабирак са `4` (добијамо вредност `4`), затим читамо `5` и множимо га са `5` (добијамо вредност `20`) и затим читамо `2` и множимо га са `2` (чиме добијамо `40`). Пошто је следећи оператор `-`, завршили смо са обрадо још једног сабирка и резултат увећавамо за вредност текућег сабирка помножену вредношћу знака (добијамо вредност `48`). Знак постављамо на `-1` (јер ће се наредни сабирак одузимати) и вредност текућег сабирка поново враћамо на иницијалну вредност `1`. Након тога читамо `7` и њиме множимо вредност текућег сабирка (добијамо вредност `7`), затим читамо `2` и њоме множимо вредност текућег сабирка (добијамо вредност `14`) и пошто смо стигли до краја, резултат увећавамо за производ знака и вредности текућег сабирка (тј. на збир додајемо `-14`), чиме добијамо коначну вредност `34`.

```
#include <iostream>
#include <string>

using namespace std;

int vrednost(const string& s) {
  int rezultat = 0;
  int znakTekucegSabirka = 1;
  int tekuciSabirak = 1;
  int tekuciBroj = 0;
  for (int i = 0; i <= s.length(); i++)
    if (i < s.length() && isdigit(s[i]))
      // procitali smo cifru
      // dodajemo je kao poslednju cifru tekuceg broja
      tekuciBroj = 10 * tekuciBroj + s[i] - '0';
    else {
      // dosli smo do nekog operatora ili kraja broja
```

```
      // tekuci broj je faktor tekuceg sabirka
      tekuciSabirak *= tekuciBroj;
      // zavrsili smo sa obradom tekuceg broja i priremamo se za
      // citanje narednog
      tekuciBroj = 0;

      // ako smo stigli do kraja ili procitali aditivni operator
      // zavrsili smo obradu tekuceg sabirka
      if (i == s.length() || s[i] == '+' || s[i] == '-') {
          // rezultat uvecavamo ili umanjujemo za tekuci sabirak u
          // zavisnosti od ranije odredjenog znaka
          rezultat += znakTekucegSabirka * tekuciSabirak;
          if (i < s.length()) {
            // priremamo se za obradu narednog sabirka
            tekuciSabirak = 1;
            // njegov znak postavljamo u zavisnosti od operatora na koji
            // smo naisli
            znakTekucegSabirka = s[i] == '+' ? 1 : -1;
          }
      }
    }
  return rezultat;
}

int main() {
  string s;
  getline(cin, s);
  cout << vrednost(s) << endl;
  return 0;
}
```

2. Спекулативно израчунавање

Интересантна техника коју можемо применити у овом задатку је израчунавање редом једног по једног подизраза. Размотримо израз `3+4-2+5*7`.

Израчунавамо вредност израза `3`, затим `3+4`, затим `3+4-2`, затим `3+4-2+5` и на крају `3+4-2+5*7`.
Вредност претходно израчунатог израза помаже да се израчуна вредност наредног израза. На пример, ако знамо вредност израза `3`, тада вредност израза `3+4` добијамо тако што на ту вредност додајемо вредност броја `4`. То важи и у општем случају. Ако знамо вредност израза `e`, тада вредност израза `e+x` добијамо тако што на ту вредност додамо вредност броја `x`. Слично, ако знамо

вредност израза `e`, тада вредност израза `e-x` добијамо тако што од те вредности одузмемо вредност броја `x`.

Случај множења је мало компликованији. Ако знамо вредност израза `3+4-2+5`, вредност израза `3+4-2+5*7` не можемо израчунати једноставним множењем са `7`.

Наиме, у изразу `3+4-2+5` број `5` представља одређени вишак и он је додат на текућу суму спекулативно (под претпоставком да се иза њега неће наћи оператор множења). Ако се тај оператор ипак појави, онда ту вредност `5` треба одузети од текућег збира (тако да се добије вредност израза `3+4-2`), затим је помножити са `7` и на крају тај производ додати на вредност израза.

У општем случају, ако имамо израз облика `e+e'*x`, и знамо вредност израза `e+e'`, вредност новог израза добијамо тако што од вредности израза `e+e'` одузмемо вредност `e'` а затим додамо вредност `e'*x`. Да би ово било могуће уз вредност сваког текућег израза који мало по мало проширујемо, увек памтимо и вредност последњег сабирка који у њему учествује (тај сабирак може бити и негативан, ако је испред њега знак минус).

Прикажимо рад овог алгоритма на изразу `8+4*5*2-7*2`. У старту вредност израза постављамо на нулу, као и вредност последњег сабирка. Након тога наилазимо на вредност `8`, увећавамо вредност израза на `8`, што је уједно и вредност последњег сабирка. Након тога наилазимо на вредност `4`, увећавамо вредност израза на `12`, а вредност последњег сабирка постављамо на `4`. Пошто након тога наилазимо на множење бројем `5`, од вредности `12` одузимамо вредност `4` и додајемо `4*5`, чиме добијамо `28`, док вредност последњег сабирка постављамо на `20`. Пошто наилазимо на још једно множење опет од вредности `28` одузимамо вредност `20`, а затим додајемо `20*2` чиме добијамо вредност `48`, док вредност последњег сабирка постављамо на `40`. Након тога долазимо до одузимања вредности `7` тако да вредност израза постаје `41`, а последњи сабирак постављамо на `-7`.
На крају, од збира одузимамо тих `-7` и увећавамо га за `-7*2` чиме добијамо `34`, а последњи сабирак постављамо на `-14`. Коначна вредност израза је `34`.

```
#include <iostream>
#include <string>

using namespace std;

int vrednost(const string& s) {
  int rezultat = 0;
  char op = '+';
  int tekuciBroj = 0;
  int poslednjiSabirak = 0;

  for (int i = 0; i <= s.length(); i++)
```

```
    if (i < s.length() && isdigit(s[i]))
      // procitali smo cifru
      // dodajemo je kao poslednju cifru tekuceg broja
      tekuciBroj = 10 * tekuciBroj + s[i] - '0';
    else {
      switch (op) {
      case '+':
          // rezultat uvecavamo za tekuci broj (nadajuci se da iza njega
          // ne ide *)
          rezultat += tekuciBroj;
          poslednjiSabirak = tekuciBroj;
          break;
      case '-':
          // rezultat umanjujemo za tekuci broj (nadajuci se da iza
          // njega ne ide *)
          rezultat -= tekuciBroj;
          poslednjiSabirak = -tekuciBroj;
          break;
      case '*':
          // rezultat je u prethodnom koraku greskom uvecan za poslednji sabirak
          rezultat -= poslednjiSabirak;
          // poslednji sabirak treba da ukljuci i tekuci broj kao faktor
          poslednjiSabirak = poslednjiSabirak * tekuciBroj;
          // uvecavamo rezultat za azuirarani poslednji sabirak,
          // nadajuci se da se iza njega nece vise javljati znak *
          rezultat += poslednjiSabirak;
          break;
      }
      // zavrsili smo sa obradom tekuceg broja i priremamo se za
      // citanje narednog
      tekuciBroj = 0;
      if (i < s.length())
          // pamtimo operator pre citanja narednog broja, jer nam on
          // govori kako naredni broj koji budemo procitali treba
          // ukljuciti u rezultat
          op = s[i];
    }
  return rezultat;
}

int main() {
  string s;
  getline(cin, s);
  cout << vrednost(s) << endl;
  return 0;
```

}

3. (Битови + Структура података)
Напиши програм који на основу неозначеног целог броја n формира и исписује његов 32-битни бинарни запис.
Са стандардног улаза се уноси број n ($0 \leqslant n \leqslant 2^{32}-1$
На стандардни излаз исписати 32-битни бинарни запис броја n.

Пример 1
Улаз
123456

Излаз
00000000000000001111000100100000

Пример 2
Улаз
16777215

Излаз
00000000111111111111111111111111

Решење:
Нека је низ од 32 логичке вредности попуњен вредности `false`.
Бинарни запис одређујемо тако што одређујемо једну по једну бинарну цифру броја, здесна налево. У сваком кораку петље одређујемо остатак при дељењу броја n са 2, и на наредно место у низу (у кораку i на место i) уписујемо `true` ако је тај остатак једнак 1. На крају петље, исписујемо садржај низа уназад.

```
#include <iostream>

using namespace std;

int main() {
  // broj koji se prevodi
  unsigned long n;
  cin >> n;

  // niz binarnih cifara, redom, od cifre najmanje do cifre najvece
  // tezine
  bool binarneCifre[32] = {false};
  // prevodjenje
  for (int i = 0; n > 0; i++, n /= 2)
    binarneCifre[i] = n % 2;
```

```
  // ispisujemo rezultat (od cifre najmanje tezine
  for (int i = 31; i >= 0; i--)
    cout << (binarneCifre[i] ? '1' : '0');
  cout << endl;

  return 0;
}
```