2. ВТОРОЙ ОТБОРОЧНЫЙ ЭТАП

Второй отборочный этап проводится в командном формате в сети интернет, работы оцениваются автоматически средствами системы онлайн-тестирования. Продолжительность второго этапа составляет 37 дней. Задачи условно разделены на задачи по математике и информатике, но носят междисциплинарный характер и помогают отработать те навыки, которые потребуются для решения командной задачи заключительного этапа. Для каждой из параллелей (9 класс или 10-11 класс) предлагается свой набор задач по математике, задачи по информатике общие — для всех участников. Участники не были ограничены в выборе языка программирования для решения задач.

Объем и сложность задач этого этапа подобраны таким образом, чтобы решение всех задач одним человеком было маловероятно. Это призвано обеспечить включение командной работы и распределения обязанностей. Решение каждой задачи дает определенное количество баллов. Баллы зачисляются в полном объеме за правильное решение задачи. В данном этапе можно получить суммарно от 0 до 99 баллов.

Все условия задач по математике доступны участникам с первого дня второго отборочного этапа. Задачи по программированию выкладывались двумя партиями: в начале второго этапа и через три недели после начала. Команды могут выполнять задачи в любом порядке. Задачи допускают неограниченное число попыток сдать решение.

Задачи второго этапа

6.1. Задачи по математике (9 класс)

Задача 6.1.1. Простой криптоанализ (3 балла)

Используя частотный криптоанализ, расшифруйте следующее сообщение

$$MUMUHUBKSAORUSQKIUEVJUDJXUVHUGK$$

$$UDSOCUJXETDUUTIBEDWUHSYFXUHJUNJ$$

в преположении, что шифрование происходило при помощи «шифра Цезаря» (каждая буква латинского алфавита сдвигается на некоторое число d по модулю 26). Например, если $d = 16$, то сообщение

$$PAYMENOW \longrightarrow 15\ 0\ 24\ 12\ 4\ 13\ 14\ 22 \longrightarrow$$

$$\longrightarrow 5\ 16\ 14\ 2\ 20\ 3\ 4\ 12 \longrightarrow FQOCUDEM$$

Рекомендуем использовать вычислительную технику.

Решение

Находим самый часто встречающийся символ в сообщении: латинская буква U встречается 15 раз. Вероятнее всего ей должна изначально соответствовать латинская буква E. Следовательно, сдвиг d равен 10. После сдвига всех символов на d по модулю 26 получаем осмысленное сообщение.

Ответ: WEWERELUCKYBECAUSEOFTENTHEFREQUENCYMETHOD NEEDSLONGERCIPHERTEXT

Задача 6.1.2. Теоремы Ферма и Эйлера (5 баллов)

Про каждое утверждение ниже выясните: верно оно или нет. В ответ запишите строку, состоящую из восьми цифр 0 или 1, где 1 соответствует верному утверждению, а 0 — неверному. (*Например 01111001*).

а) Если найдется хотя бы одно натуральное число a, для которого $a^{n-1} \not\equiv 1 \pmod{n}$, то n — составное число.

б) Натуральное число n свободно от квадратов (если $n \vdots p$, то $n \not\vdots p^2$). Тогда уравнение $n = 7 \cdot \varphi(n)$ имеет ровно одно натуральное решение, где φ — функция Эйлера.

в) Существует лишь конечное число натуральных n, что $5^n - 1$ делится на 2^{100}.

г) $\varphi(n) \vdots \varphi(m)$ тогда и только тогда, когда $n \vdots m$.

д) Если для любого натурального числа a выполняется сравнение $a^n \equiv a \pmod{n}$, то n — простое число.

е) Известно, что натуральное число $n = p \cdot q$, где p и q различные простые числа. Тогда $p, q = \frac{\gamma \pm \sqrt{\gamma^2 - 4n}}{2}$, где $\gamma = n - \varphi(n) + 1$.

ж) Пусть p — нечетное простое число и $p - 1 = m \cdot 2^h$, где m — нечетное число, h — натуральное. Тогда при a взаимно простом с p выполняется

$$a^m \equiv 1 \pmod{p} \text{ или } \exists t\ (0 \leqslant t < h) :\ a^{m2^t} \equiv -1 \pmod{p}.$$

з) Для любых натуральных m и n верно $\varphi(m \cdot n) = \varphi(m) \cdot \varphi(n)$.

Ответ: 10000110

Задача 6.1.3. Тест Ферма по основанию 2 (6 баллов)

Пусть n – нечетное число. Если число 2^{n-1} при делении на n дает отстаток отличный от 1, то тест говорит, что n – составное число. К сожалению, в обратную сторону этот тест не работает (если остаток 1, то n может быть составным). Выясните сколько ложных результатов дает тест в промежутке $1 < n \leqslant 10^8$. Рекомендуем использовать вычислительную технику.

Решение

Ответ получается из непосредственного перебора всех чисел в промежутке $1 < n \leqslant 10^8$ (на компьютере). На каждом шаге бинарного возведения в степень храним только остаток по модулю n.

Ответ: 2057

Задача 6.1.4. Теория вероятности (5 баллов)

Про каждое утверждение ниже выясните: верно оно или нет. В ответ запишите строку, состоящую из восьми цифр 0 или 1, где 1 соответствует верному утверждению, а 0 — неверному. (*Например 01111001*).

а) Пусть $A = \{$при броске двух игральных кубиков сумма значений равно 6$\}$. Тогда $P(A) = 1/6$.

б) Из колоды игральных карт (52 карты) выбирается одна случайная карта. Пусть

$$A = \{\text{карта бубен}\} \text{ и } B = \{\text{карта: валет, дама, король или туз}\}.$$

Тогда $P(A \cap B) = 3/52$.

в) Среди 100 учеников 30 занимаются олимпиадной математикой, 20 — олимпиадным программированием, а 10 — и тем и другим. Тогда вероятность того, что случайно выбранный ученик занимается олимпиадной математикой или олимпиадным программированием равна $2/5$.

г) Дан конечный набор попарно непересекающихся событий $\{A_i\}_{i=1}^n$ и некоторое событие B. Если $P(A_i) > 0$ для любого i, то верна формула

$$P(B) = \sum_{i=1}^{n} P(A_i)P(B|A_i).$$

д) Существует набор событий $A_1, A_2, \ldots, A_n$ такой, что верно

$$P(A_1 \cup A_2 \cup \ldots \cup A_n) = P(A_1) \cdot P(A_2|A_1) \cdot \ldots \cdot P(A_n|A_1 \cap A_2 \cap \ldots \cap A_{n-1})$$

е) В семье три ребенка. Рассмотрим события

$$A = \{\text{есть мальчик и девочка}\} \text{ и } B = \{\text{есть хотя бы один мальчик}\}.$$

Тогда события A и B зависимы.

ж) События A и B независимы тогда и только тогда, когда независимы события $\overline{A}$ и $\overline{B}$.

з) Для того, чтобы события A, B и C были попарно независимыми необходимо и достаточно

$$P(A \cap B \cap C) = P(A) \cdot P(B) \cdot P(C)$$

Ответ: 01101010

Задача 6.1.5. Теория вероятности (5 баллов)

Известно, что 80% всех мобильных телефонов производится в Китае, 15% — в США и 5% — в Японии. Оказалось, что с вероятностью 0.04 телефон, произведенный в Китае, с дефектом; с вероятностью 0.06 — в США и с вероятностью 0.09 — в Японии. Посчитайте вероятность того, что случайно выбранный телефон, который оказался с дефектом, произведен в США. Ответ округлите до тысячных.

Решение

Вероятность дефекта при покупке телефона равна $0.8 \cdot 0.04 + 0.15 \cdot 0.06 + 0.05 \cdot 0.09 = 0.0455$. Вероятность того, что случайно выбранный телефон, оказавшийся с дефектом, произведен в США равна доле брака мобильников из США среди всего брака

$$\frac{0.15 \cdot 0.06}{0.8 \cdot 0.04 + 0.15 \cdot 0.06 + 0.05 \cdot 0.09} \approx 0.198.$$

Ответ: 0.198

Задача 6.1.6. Modular Exponentiation (6 баллов)

Сообщение

$$29273,\ 320901,\ 52885,\ 107960,\ 12432,\ 169798$$

было зашифровано при помощи криптографического алгоритма *modular exponentiation* с публичным ключем $(350377, 8461)$ (первое число модуль, а второе — основание). Расшифруйте сообщение (расставьте правильно пробелы). Рекомендуем использовать вычислительную технику.

Решение

Число 350377 изначально является простым, поэтому $\varphi(350377) = 350376$. Следовательно,

$$8461^{-1} \equiv 117565 \pmod{350376}.$$

Возведем каждое число из зашифрованного сообщения в степень 117565 по модулю 350377 и получим следующую последовательность чисел

$$190010,\ 041204,\ 191424,\ 142017,\ 110400,\ 030417.$$

Разбиваем каждое сообщение на блоки по 2 цифры и получаем порядковые номера латинских букв. В итоге после декодирования получаем

$$TAK,\ EME,\ TOY,\ OUR,\ LEA,\ DER.$$

Наконец, правильно расставляем пробелы.

Ответ: TAKE ME TO YOUR LEADER

6.2. Задачи по математике (10-11 класс)

Задача 6.2.1. Безопасный пароль (3 балла)

В научно-исследовательском институте чародейства и волшебства (НИИЧАВО) используется устройство для проверки безопасности пароля. Паролем может быть любая непустая последовательность символов из множества $\{a, b, c\}$. Будем обозначать такие наборы большими латинскими буквами. Устройство преобразует введенный в него пароль X в другую последовательность символов $Y = f(X)$. Само устройство f держится в секрете, но зато известно, что оно обладает тремя свойствами

(1) $f(aX) = X$

(2) $f(bX) = f(X)af(X)$

(3) последовательность символов $f(cX)$ получается из последовательности $f(X)$ выписыванием в обратном порядке.

Пароль признается безопасным, если $f(X) = X$. Например, пароль bab – безопасный ($f(bab) = f(ab)af(ab) = bab$). Найдите хотя бы один безопасный пароль, состоящий из 11 символов.

Решение

Все ответы получаются из простого рекурсивного алгоритма (на компьютере). Также подобрать хотя бы один ответ можно руками.

Ответ: Любой из: bccccabcccc, cbcccacbccc, ccbccaccbcc, cccbcaccccbc, ccccbacccccb

Задача 6.2.2. Теоремы Ферма и Эйлера (5 баллов)

Натуральное число n называется псевдопростым по основанию a, если a и n взаимно просты и $a^{n-1} \equiv 1 \pmod{n}$.

Про каждое утверждение ниже выясните: верно оно или нет. В ответ запишите строку, состоящую из восьми цифр 0 или 1, где 1 соответствует верному утверждению, а 0 — неверному. (*Например 01111001*).

а) Если найдется хотя бы одно натуральное число a, для которого $a^{n-1} \not\equiv 1 \pmod{n}$, то n — составное число.

б) Уравнение $n = 7 \cdot \varphi(n)$ имеет ровно одно натуральное решение, где φ — функция Эйлера.

в) Существует лишь конечное число натуральных n, что $13^n - 1$ делится на $2^{100} \cdot 3^{200} \cdot 5^{300}$.

г) Пусть $a > 1$ и n — взаимно простые числа. Натуральное число m удовлетворяет сравнению $a^m \equiv 1 \pmod{n}$ тогда и только тогда, когда $m \vdots \varphi(n)$.

д) Если для любого натурального числа a выполняется сравнение $a^n \equiv a \pmod{n}$, то n — простое число.

е) Известно, что натуральное число $n = p \cdot q$, где p и q различные простые числа.

Тогда $p, q = \frac{\gamma \pm \sqrt{\gamma^2 - 4n}}{2}$, где $\gamma = n - \varphi(n) + 1$.

ж) Пусть p — простое число и $p - 1 = m \cdot 2^h$, где m — нечетное число, h — натуральное. Тогда при a взаимно простом с p выполняется

$$a^m \equiv 1 \pmod{p} \text{ или } \exists t\ (0 \leqslant t < h) :\ a^{m2^t} \equiv -1 \pmod{p}.$$

з) Если n – псевдопростое число одновременно по основаниям a и ab, то оно псевдопростое и по основанию b.

Ответ: 10000111

Задача 6.2.3. Тест Ферма по основаниям 2, 3 и 5 (5 баллов)

Натуральное число n такое, что НОД$(n,\ 30) = 1$. Если хотя бы одно число из 2^{n-1}, 3^{n-1} и 5^{n-1} при делении на n дает отстаток отличный от 1, то тест говорит, что n – составное число. К сожалению, в обратную сторону этот тест не работает (если все три числа дают остаток 1, то n может быть составным). Выясните сколько ложных результатов дает тест в промежутке $1 < n \leqslant 10^8$. Рекомендуем использовать вычислительную технику.

Решение

Ответ получается из непосредственного перебора всех чисел в промежутке $1 < n \leqslant 10^8$ (на компьютере). На каждом шаге бинарного возведения в степень храним только остаток по модулю n.

Ответ: 257

Задача 6.2.4. Теория информации: энтропия Шеннона (5 баллов)

Про каждое утверждение ниже выясните: верно оно или нет. В ответ запишите строку, состоящую из восьми цифр 0 или 1, где 1 соответствует верному утверждению, а 0 — неверному. (*Например 01111001*).

а) Энтропия Шеннона принимает минимальное значение на равномерном распределении (для любого i верно $p_i = \frac{1}{n}$) среди всех возможных распределений на таком же числе исходов.

б) $H(\alpha \mid \beta) = H((\alpha, \beta)) - H(\beta)$.

в) Для любого вероятностного распределения $H(\alpha) \geqslant 0$.

г) $H((\alpha, \beta)) \leqslant H(\alpha)$ для любых двух вероятностных распределений α и β.

д) $H((\alpha, \beta)) = H(\alpha) \cdot H(\beta)$ тогда и только тогда, когда α и β — независимые вероятностные распределения.

е) $H((\alpha, \beta, \gamma)) \leqslant H(\alpha) + H(\beta) + H(\gamma)$ для любых трех вероятностных распределений α, β и γ.

ж) $I(\alpha : \beta) = 0$ тогда и только тогда, когда α и β — независимые вероятностные распределения.

з) Существует вероятностное распределение α такое, что $p_1 = p_2 = p_3 = \frac{1}{100}$ и $H(\alpha) = 0$.

Ответ: 01100110

Задача 6.2.5. Теория вероятности (6 баллов)

На множестве $\{0,1\}^3$ задано совместное распределение (α, β, γ). Причем оказалось, что распределение равномерно на множестве значений $\alpha + \beta + \gamma \equiv 0 \pmod 2$ (во всех остальных случаях вероятность равна 0). Вычислите величину

$$I(\alpha : \beta : \gamma) = H(\alpha) + H(\beta) + H(\gamma) - H((\alpha, \beta)) - H((\beta, \gamma)) - H((\gamma, \alpha)) + H((\alpha, \beta, \gamma)).$$

Решение

Возможны варианты $(\alpha, \beta, \gamma) = \{(0,0,0),\ (1,1,0),\ (1,0,1),\ (0,1,1)\}$. Следовательно, вероятность каждого исхода равна $1/4$.

Заметим, что $P(\alpha = 1) = P(\alpha = 0) = 1/2$, следовательно $H(\alpha) = \log_2 2$. Также $P(\alpha = i, \beta = j) = 1/4$, следовательно $H(\alpha, \beta) = \log_2 4$. И наконец, мы знаем, что вероятность каждого исхода $P(\alpha = i, \beta = j, \gamma = k) = 1/4$, где $i + j + k \equiv 0 \pmod 2$. Поэтому $H(\alpha, \beta, \gamma) = 4 \cdot 1/4 \cdot \log_2 4 = \log_2 4$.

В итоге получаем: $I(\alpha : \beta : \gamma) = 3 \cdot \log_2 2 - 3 \cdot \log_2 4 + \log_2 4 = -1$.

Ответ: -1

Задача 6.2.6. Алгоритм RSA (6 баллов)

Сообщение

$$2163315218,\ 5629365529,\ 9458153711,\ 1721672107,\ 10120672595$$

было зашифровано при помощи криптографического алгоритма RSA с публичным ключом $(11170948057, 11987)$ (первое число модуль, а второе — основание). Расшифруйте сообщение (расставьте правильно пробелы и удалите лишние символы). Рекомендуем использовать вычислительную технику.

Решение

Для того, чтобы расшифровать сообщение, мы должны разложить на множители $11170948057 = 139273 \cdot 80209$. Следовательно,

$$\varphi(11170948057) = 139272 \cdot 80208 = 11170728576$$

и

$$11987^{-1} \equiv 7586627291 \pmod{11170728576}.$$

Возведем каждое число из зашифрованного сообщения в степень 7586627291 по модулю 7586627291 и получим следующую последовательность чисел

$$0408060719,\ 1402111402,\ 1000130300,\ 1111081822,\ 0411112525.$$

Разбиваем каждое сообщение на блоки по 2 цифры и получаем порядковые номера латинских букв. В итоге после декодирования получаем

$$EIGHT,\ OCLOC,\ KANDA,\ LLISW,\ ELLZZ.$$

Наконец, правильно расставляем пробелы и откидываем лишние символы ZZ.

Ответ: EIGHT OCLOCK AND ALL IS WELL

6.3. Задачи по информатике

Задача 6.3.1. Little-Endian, Big-Endian (1 балл)

В современном мире любые данные в памяти компьютера и при их передаче по каналам связи представлены в виде последовательности байтов. В разных системах и протоколах байты принято упорядочивать по-разному: от старшего к младшему (big-endian) или от младшего к старшему (little-endian).

Например:

$$\begin{aligned} 2017_{10} &= 11111100001_2 \\ Big-endian: \quad 2017_{10} &= 00000111 \quad 11100001_2 \\ Little-endian: \quad 2017_{10} &= 11100001 \quad 00000111_2 \end{aligned} \tag{6.1}$$

В мире блокчейн разницу между кодированием чисел можно сразу увидеть: Bitcoin использует последовательность little-endian, Ethereum использует последовательность big-endian.

Напишите программу, которая преобразует число к формату с little-endian с учетом того для гипотетической вычислительной системы, где используются байти состоящие только из 7 бит.

Формат входных данных

На вход подается целое число N ($1 \leqslant N \leqslant 2^{28} - 1$).

Формат выходных данных

Необходимо вывести последовательность из 28 символов 0 или 1 (биты) – представление входного числа в формате little-endian.

Примеры

Пример №1

Стандартный ввод
2

Стандартный вывод
0000010000000000000000000000

Пример №2

Стандартный ввод
268435305
Стандартный вывод
11010011111110111111111111111

Решение

По условию задачи мы знаем, что длина двоичного представления числа, подающегося на вход не превышает 28 символов. Переведем данное число в двоичную систему счисления и добавим недостающие нули в начало получившегося числа.

Осталось разделить полученную строку на 4 подстроки по 7 элементов и вывести их в обратном порядке.

Пример программы

Ниже представлено решение на языке Python3

```
n = int(input())
n = bin(n)[2:]
addition = "0" * (28-len(n))
n = addition + n
for i in range(21, -1, -7):
    print(n[i:i+7], end = "")
```

Задача 6.3.2. Кодирование Base58 (3 балла)

Base58 кодирование используется в Bitcoin для получение человекочитаемого адреса, который используется для получения крпитовалюты. Для этого число, определяющее адрес, переводится в строку, для кодирования которой используется алфавит состоящий из 58 символов, включая строчные и заглавные буквы латинского алфавита и цифры, исключая неоднозначные в некоторых шрифтах символы, такие как 0 (ноль), O (заглавная o), I (заглавная i) и l (строчная L).

Напишите программу, которая бы определяла символ алфавита, который используется в конкретном адресе Bitcoin чаще других символов.

Формат входных данных

Входной файл содержит 10 строк, где каждая строка – последовательность из нулей и единиц - двоичное представление адреса Bitcoin. Длина каждой строки - 176 символов.

Формат выходных данных

Укажите символ, который используется чаще всего в Base58 представлениях указанных адресов. Если самый частый символ - единица, то укажите следующий по частоте символ. Гарантируется, что других символов с такой же частотой встречаемости нет.

Примеры

Пример №1

Стандартный ввод
0000000000000000100010110110010100001101011101000011110101101011000011011
0000001111101100000100110100100101101010100011011011000001110100111001000
0111111111000001000001100010111
0000000000000000100000101000000010011010100011001011011011001100100001011
0001111110100101110011011011011111011001111110111110011010111000101100110
1110111110101110000111101110111
0000000000000000100011010110001111010001100111011010111111110000010001111
1000110101011100110110001010000111110100000011010000000011000011101011101
1010000101011110100010111101111
0000000000000000101010110001011011101110010000101001010001100010110110110
1001001110111101100011100011101010011101011010111111010101111110011100110
1110111011001001101101000110011
0000000000000000100111100011011010011011110101000010001010010101110001100
0001111000011101000111000000011010011101000000110010011001010100001110000
1001100111001010111001111010100
0000000000000000110010101111000100110100000000101010000010001101100111101
1110100111101010101111001111101101111001101110001110110100001011000100001
1110110010100010111111111001111
0000000000000000001001010110100100010000000100001010011101010010000000000
1001000111101010001010011100000101101100000000101111000010011001010010111
1011110010010001000110100100110
0000000000000000010100111100110000101010000110000111111001100111100100001
1011110110101011011001011000000011010100000000011001111110010011101000001101
0010010100000010011011000111000
0000000000000000111101100000010111101100000001101000000011001110101111110
1000011100110110111101001110011111100110011100101100101101111001100101111
1010100110110111011000100100011
0000000000000000111111110011001000000110000101101010000001010011110111101
0110100100001111010100100001001110000000111101101001000111100000011011011
0101101100100111100110010001100

Стандартный вывод
d

Комментарии

Используйте библиотеку Python `bitcoin` для генерации base58 адреса Bitcoin из его численного представления.

Решение

Для решения данной задачи воспользуемся функцией *hex_to_b58check* из библиотеки Python `bitcoin`.

Согласно документации Bitcoin, адрес генерируется с помощью base58 кодирования, применяемого к хэшу публичного ключа. Необходимый хэш получается путем последовательного применения функций sha256 и ripemd160 к публичному ключу аккаунта и состоит ровно из 20 байт (`https://github.com/bitcoinbook/bitcoinbook/blob/second_edition/ch04.asciidoc`).

Таким образом, для кодирования необходимо перевести полученное на вход число в шестнадцатеричную систему счисления и убедиться, что оно содержит ровно 40 символов. В случае необходимости мы просто добавляем незначащие нули в начало хэша.

Проходясь по каждому адресу, находим наиболее часто встречающийся символ и в конце выводим ответ.

Пример программы

Ниже представлено решение на языке Python3

```
import bitcoin
n = 10
dict = {}
for i in range(n):
    f = input()
    f = hex(int(f, 2))[2:]
    addition = 40 - len(f)
    f = '0'*addition + f
    f = bitcoin.hex_to_b58check(f)
    for j in f:
        if j in dict:
            dict[j] += 1
        else:
            dict[j] = 1

a = sorted(list(dict.items()), key = lambda x: x[1], reverse=True)
if a[0][0] != '1':
    print(a[0][0])
else:
    print(a[1][0])
```

Задача 6.3.3. Сериализация данных в Ethereum (3 балла)

Человек привык работать с символьным отображением информации, цифровые устройства хранят, обрабатывают и пересылают любую информацию в виде чисел, а если быть более точным, в виде двоичных битов. Разные структуры данных по разному выглядят для человека: числа - последовательности цифр, строки - последовательности символов, списки - последовательности символов со специальным форматированием. Цифровому устройству нужно использовать специальные алгоритмы, для того, чтобы сохранить данные структуры в памяти или передать по коммуникационному каналу. Процесс перевода структур данных в формат, используемый для хранения и пересылки, называется *сериализацией*, обратный процесс называется *десериализацией*.

Программное обеспечение Ethereum для сериализации и десериализации данных использует метод RLP (Recursive Length Prefix, `https://github.com/ethereum/wiki/wiki/RLP`.

Например,

Изначальные данные	Сериализированные данные в шестнадцатиричном формате
42	2a
31415	827ab7
'ethereum'	88657468657265756d
['a task', 4, 'you']	cc8661207461736b0483796f75

Напишите программу, которая меняет порядок элементов на обратный в сериализованной с использованием RLP кодирования структуре данных по следующему правилу:

- Если данные - число, то в нем меняется порядок с big-endian на little-endian.
- Если данные - строка, то ее элементы представляются в обратном порядке.
- Если данные - список, то его элементы представляются в обратном порядке.

Формат входных данных

На вход подается одна строка - сериализованные данные в шестнадцатиричном виде. Длина строки - не более 65535 символов.

Формат выходных данных

Необходимо вывести сериализованные с использованием RLP кодирования данные в шестнадцатиричном виде.

Примеры

Пример №1

Стандартный ввод
`81df`
Стандартный вывод
`81df`

Пример №2

Стандартный ввод
`83017d45`
Стандартный вывод
`83457d01`

Пример №3

Стандартный ввод
`87426974436f696e`
Стандартный вывод
`876e696f43746942`

Пример №4

Стандартный ввод
`cb8277658673616c75746575`
Стандартный вывод
`cb758673616c757465827765`

Решение

Функция RLP-кодирования может принимать в качестве аргумента элементы, которые мы назовем 'item', где item определен следующим образом:

1. Строка (т.е. массив байтов) является item-ом
2. Список item-ов является item-ом

Теперь рассмотрим каким образом происходит само кодирование:

- Для одного байта, значение которого находится на отрезке [0, 127] (или [0x00, 0x7f]), данный байт возвращается без каких-либо изменений.
- Для строки, длиной от 0 до 55 байт, RLP-кодирование включает байт, который определяет длину строки и вычисляется как длина строки (количество байтов в массиве) +128 (0x80). Таким образом первый байт лежит на отрезке [128, 183] ([0x80, 0xb7]). Следом за ним идет сама строка.
- Для строки, длиной свыше 55 байт, в начало добавляется первый дополнительный байт, который находится как минимально необходимое для кодирования длины строки количество байтов +183 (0xb7) и лежит на отрезке [184, 191] ([0xb8, 0xbf]). Следом за ним идет некоторое количество байт, в которых представлена длина строки с порядком байтов - "BigEndian". Затем же идет сама строка.

  Попробуем разобраться, что означает минимально необходимое для кодирования длины строки количество байтов. Предположим что наша строка состоит из 1024 байт. Тогда её длина будет равна 1024. Легко заметить, что число 1024 соответствует числу '400' в шестнадцатеричной системе счисления и для её кодирования достаточно 2 байта. Таким образом первый байт для данной строки будет равен 183+2 или 185 (0xb9), а следом за ним будут находиться 2 байта, равные 'x04' и 'x00' соответсвенно.

Для списков все выглядит очень похоже:

- Если итоговая длина списка не превышает 55 байтов, то RLP-кодирование состоит из байта, равного длине списка +192 (0xc0), следом за которым идет сам список. Соответственно, первый байт будет находиться в пределах отрезка [192, 247] ([0xc0, 0xf7]).

  Однако тут необходимо уточнить, каким образом находится итоговая длина списка. Покажем это на примере. Предположим, что мы хотим закодировать список, состоящий из двух элементов - ['cat', 'dog']. При этом и 'cat' и 'dog' должны быть уже закодированы в виде массива байтов. Сначала, мы должны преобразовать каждый элемент списка в его RLP-представлении. Тогда наш массив можно будет представить в виде [0x83, 'c', 'a', 't', 0x83, 'd', 'o', 'g']. Таким образом, его длина будет равна 8 байтам и после кодирования мы должны получить: [0xc8, 0x83, 'c', 'a', 't', 0x83, 'd', 'o', 'g'].

- Если же итоговая длина списка больше 55 байт, то RLP-кодирование состоит из байта, равного минимальной длине в байтах итоговой длины списка + 247 и лежащего на отрезке [248, 255] ([0xf8, 0xff]), следом за которым идет минимальное количество байт, которые представляют итоговую длину списка, следом за которыми идет сам список.

Теперь обратимся к условию. Заметим, что для нас не имеет значения число или строка были закодированы, так как и в том и в другом случае, нам необходимо переставить байты местами. Таким образом, по первому байту возможно сразу определить, чем являются полученные данные - строкой или списком, и каким образом необходимо переставлять их элементы. Остается лишь аккуратно разобрать все возможные случаи.

Пример программы

Ниже представлено решение на языке Python3

```python
def RLPDecode(s: str):
    result = ''
    temp = s[:2]
    if 0 <= int(temp, 16) <= 191:
        result += RLPDecodeString(s)
    else:
        result += RLPDecodeArray(s)
    return result

def RLPDecodeString(s: str):
    result = ''
    temp = s[:2]
    result += temp
    if 128 <= int(temp, 16) <= 183:
        for i in range(len(s)-2, 1, -2):
            result += s[i] + s[i+1]
    elif 184 <= int(temp, 16) <= 191:
        tsize = int(temp, 16) - 183
        for i in range(1, int(temp, 16) - 183 + 1):
            temp = s[2*i:2*i+2]
            result += temp
        for i in range(len(s) - 2, 1+tsize*2, -2):
            result += s[i] + s[i+1]
    return result

def RLPDecodeArray(s: str):
    result = ''
    elements = []
    temp = s[:2]
    result += temp
    if 192 <= int(temp, 16) <= 247:
        i = 2
        while i < len(s):
            temp = s[i:i+2]
            if 0 <= int(temp, 16) <= 191:
                tempPair = RLPDepodeStringFromArray(s[i:])
                elements.append(tempPair[0])
                i += tempPair[1]
            elif int(temp, 16) <= 255:
                tempPair = RLPDecodeArrayFromArray(s[i:])
```

```
                elements.append(tempPair[0])
                i += tempPair[1]
    elif int(temp, 16) <= 255:
        tsize = int(temp, 16) - 247
        for i in range(1, int(temp, 16) - 247 + 1):
            temp = s[2*i:2*i+2]
            result += temp
        i = tsize*2 + 2
        while i < len(s):
            temp = s[i:i+2]
            if 0 <= int(temp, 16) <= 191:
                tempPair = RLPDepodeStringFromArray(s[i:])
                elements.append(tempPair[0])
                i += tempPair[1]
            elif int(temp, 16) <= 255:
                tempPair = RLPDecodeArrayFromArray(s[i:])
                elements.append(tempPair[0])
                i += tempPair[1]
    for i in range(len(elements)-1, -1, -1):
        result += elements[i]
    return result

# Находит закодированную строку
def RLPDepodeStringFromArray(s: str):
    result = ''
    tempSize = 0
    temp = s[:2]
    if 0 <= int(temp, 16) <= 127:
        result += temp
        tempSize = 2
    elif int(temp, 16) <= 183:
        size = int(temp, 16) - 128
        result += s[:2+size*2]
        tempSize = 2+size*2
    elif int(temp, 16) <= 191:
        tsize = int(temp, 16) - 183
        temp = s[2:2*tsize+2]
        size = int(temp, 16)
        result += s[:size*2+tsize*2+2]
        tempSize = size*2+tsize*2+2
    return (result, tempSize)

# Находит закодированный массив
def RLPDecodeArrayFromArray(s: str):
    result = ''
    tempSize = 0
    temp = s[:2]
    if 192 <= int(temp, 16) <= 247:
        size = int(temp, 16) - 192
        result += s[:2+size*2]
        tempSize = 2+size*2
    elif int(temp, 16) <= 255:
        tsize = int(temp, 16) - 247
        temp = s[2:2*tsize+2]
        size = int(temp, 16)
        result += s[:size*2+tsize*2+2]
        tempSize = size*2+tsize*2+2
    return (result, tempSize)

f = open('input.txt', 'r')
```

```
out = open('output.txt', 'w')
for i in f.readlines():
    i = i.strip()
    out.write(RLPDecode(i))
    out.write('\n')
```

Задача 6.3.4. Вычисление контрольной суммы (2 балла)

Контрольная сумма используется для проверки целостности данных при их передаче или хранении. Также контрольные суммы могут использоваться для быстрого сравнения двух наборов данных на неэквивалентность: при правильном выборе алгоритма, вычисляющего контрольную сумму, с большой вероятностью различные наборы данных будут иметь неравные контрольные суммы.

Один из самых простых алгоритмов вычисления контрольной суммы - сложение по модулю 2 (XOR, $\oplus$) для всех блоков, на которые разбиваются входные данные.

$$CRC = w_1 \oplus w_2 \oplus w_3 \oplus \cdots \oplus w_n, \tag{6.2}$$

где w_1, w_2, $\ldots w_n$ - блоки одинакового размера, на которые разбиты входные данные.

Пример для контрольной суммы, размером в 16 бит: если входные данные - строка символов "0xyz", то они разбиваются на блоки "0x" и "yz". Тогда контрольная сумма будет равна 4902 в шестнадцатиричной системе счисления.

Метод продемонстрированный выше, можно также отнести к функциям хэширования (`https://ru.wikipedia.org/wiki/%D0%A5%D0%B5%D1%88%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5`).

Напишите программу для нахождения такой последовательности бит в тридцатидвухбитном слове, чтобы контрольная сумма для входных данных сконкатенированных с полученным словом совпадала с заданной.

Для вычисления контрольной суммы используйте алгоритм предложенный выше.

Формат входных данных

Входной файл содержит только 2 строки.

Первая строка содержит восемь символов - запись требуемой контрольной суммы в шестнадцатиричной системе счислениюю.

Вторая строка содержит последовательность символов, включающую цифры, латинские строчные и заглавные буквы, для которых нужно вычислить контрольную сумму. Длина строки не больше 65535 символов.

Формат выходных данных

8 символов - контрольная сумма в шестнадцатиричной системе счисления.

Примеры

Пример №1

Стандартный ввод
45a060c5 1234diGi

Стандартный вывод
10fb1498

Пример №2

Стандартный ввод
43852d88 sYmb0ls

Стандартный вывод
9fb03375

Комментарии

Пояснение к примеру №1: последовательность `'1234diGi'` можно представить в виде последовательности байт `b'\x31\x32\x33\x34\x64\x69\x47\x69'`, тогда конкатенируя ее с последовательностью `b'\x10\xfb\x14\x98'`, получится `b'\x31\x32\x33\x34\x64\x69\x47\x69\x10\xfb\x14\x98'`, по которой уже необходимо считать контрольную сумму.

Решение

Для начала заметим, что операция сложения по модулю 2 обладает таким замечательным свойством как обратимость, то есть $(a \oplus b) \oplus b = a$. Таким образом, для любого набора из трех битов, один из которых является результатом сложением двух других, не имеет значения какие из двух битов складывать. В результате мы всегда будем получать значение третьего бита.

Далее нам необходимо понять на блоки каких размеров надо делить данную на вход строку. В этом нам помогает итоговая контрольная сумма, которую надо получить. Её запись в шестнадцатеричной системе счисления состоит из 8 символов, следовательно один блок должен состоять из 4 байтов (или 4 символов).

Для начала разберемся со случаем, когда количество символов во входной строке кратно 4. Тогда разобьем данное слово на равные части и сконкатенируем их между собой. Затем воспользуемся указанным выше свойством обратимости и сконкатенируем полученный до этого результат с контрольной суммой. Полученное значение и будет ответом.

Но остается вопрос, что делать в случае, когда количество символов в строке, для которой необходимо вычислить контрольную сумму не кратно 4. Для того, чтобы ответить на него, обратимся ко второму примеру и составим таблицу, состоящую из двоичного представления полученных данных.

Все строки в данной таблице разделены и состоят из 8 ячеек, представляющих собой 1 байт данных. В последней ячейке каждой строки указано, что было закодировано в этой строке.

Исходя из этой таблицы, можно понять, как происходит сложение в случае, если строка содержит количество символов не кратное 4. В этом случае первые несколько байт в ответе, который надо вывести, необходимо конкатенировать с окончанием

0	1	1	1	0	0	1	1	0	1	0	1	1	0	0	1	0	1	1	0	1	1	0	1	0	1	1	0	0	0	1	0	"sYmb" - (1)
0	0	1	1	0	0	0	0	0	1	1	0	1	1	0	0	0	1	1	1	0	0	1	1	1	0	0	1	1	1	1	1	"0ls" + "9f" - (2)
0	1	0	0	0	0	1	1	0	0	1	1	0	1	0	1	0	0	0	1	1	1	1	0	1	1	1	1	1	1	0	1	xor((1), (2)) - (3)
1	0	0	1	1	1	1	1	1	0	1	1	0	0	0	0	0	0	1	1	0	0	1	1	0	1	1	1	0	1	0	1	"00" + "b03375" - (4)
0	1	0	0	0	0	1	1	1	0	0	0	0	1	0	1	0	0	1	0	1	1	0	1	1	0	0	0	1	0	0	0	xor((3), (4))

Рис. 6.1: Бинарное представление примера

данной строки. А их место в конечной строке должны занять нули. То есть, если остаток от деления количества символов в строке на 4 равен, к примеру, двум, то к данной строке необходимо прибавить еще первые 2 байта из ответа, а затем полученный результат необходимо сложить с последними двумя байтами ответа. Так, во втором примере, конкатенация должна происходить в следующем порядке (последняя строка выражает контрольную сумму):

73	59	6d	62
30	6c	73	9f
0	b0	33	75
43	85	2d	88

Рис. 6.2: Шестнадцатеричное представление примера

Осталось понять, как найти этот самый ответ, из которого и будут добавляться недостающие байты.

Для удобства будем называть первой строку, которая получается конкатенацией всех подстрок из 4 символов, не считая последнюю подстроку, в которой количество символов меньше 4. Второй будем называть ту самую строку, в которой не достает символов, а третьей - последнюю строку, с которой происходит сложение. Ответом будем называть результат, который мы должны вывести.

Из примера становится ясно, что, для успешного решения задачи, все тесты при сложении должны давать в первых n битах (n - остаток от деления количества символов на 4) правильный ответ сразу же, так как независимо от ответа в третьей строке на этих местах будут нули. При этом, кажется, совершенно неважно, что будет в первых байтах, которые заменят недостающие во второй строке символы, так как мы всегда можем подобрать для них последовательность байт в третьей строке, с которой они будут складываться. Однако тут важно заметить, что будет в случае, если n равно 1. Тогда ответственность за правильный второй байт во второй строке лежит именно на нас. Поэтому будем добавлять во вторую строку такие байты, чтобы при конкатенации с первой строкой она сразу давала необходимый результат, а в третьей строке в этом случае можно было оставить нули.

Чтобы получить такие байты снова воспользуемся свойством операции *XOR*. Заполним недостающее количество байт во второй строке нулями и сложим полученную строку сначала с первой строкой, а затем и с результатом, который нам необходимо получить. Полученная последовательность байт - это ответ, где первые n байт расположены в конце, а оставшиеся - в начале.

Пример программы

Ниже представлено решение на языке Python3

```python
def string_xor(s, code: str):
    result = 0
    while len(s) >= 4:
        result ^= int(s[:4].encode().hex(), 16)
        s = s[4:]
    if len(s) == 0:
        ans = hex((result ^ int(code, 16)))[2:]
        ans = '0' * (8-len(ans)) + ans
    else:
        result ^= int(code, 16)
        tempStr = (s.encode() + b'\x00'*(4 - len(s)%4)).hex()
        ans = hex(result ^ int(tempStr, 16))[2:]
        ans = '0' * (8 - len(ans)) + ans
        ans = ans[2*(len(s)%4):] + \
              ans[8-2*(len(s)%4):2*(len(s)%4)] + ans[:8-2*(len(s)%4)]

    return ans

code = input()
s = input()
print(string_xor(s, code))
```

Задача 6.3.5. Хэширование SHA-3 (2 балла)

Существует достаточно много методов хэширования данных. Какие-то из методов отличаются скоростью работы, другие - обладают свойством криптостойкости.

Хэш-функция будет считаться криптостойкой, если она - наобратима, т.е. когда с помощью преобразований результата ее работы нельзя восстановить изначальную последовательность данных. Еще одно свойство криптостойких хэш-функций - устойчивость к коллизиям, когда вычислительно невозможно найти два разных набора данных для которых хэщ-функция будет выдавать одинаковый результат.

Те криптостойкие хэш-функции, которые используются в криптографии, называют криптографическими функциями.

В блокчейн Ethereum базовой криптографической хэш-функцией является алгоритм SHA-3(*keccak*) с размером выходных значений 256 бит.

Пример: результат работы хэш-функции SHA-3 для строки *'Ethereum is a distributed database'*, записанный в шестнадцатиричном виде будет
`80529d6b03ffe0bcc4064ac573a512cb114cf4d8033913178e329b6dcda208a7`.

Как уже было сказано выше, единственный способ узнать, какие данные были поданы на вход криптографической хэш-функции, - это выполнить их полный перебор. Напишите программу, которая определяет точную дату рождения школьника, если известны результат хэш-функции SHA-3 и год, когда он родился.

Формат входных данных

Входной файл содержит 2 строки.

Первая строка целое число - год рождения школьника.

Вторая строка содержит результат хэш-функции SHA-3, записанный в шестнадцатиричной системе счисления. Значение, которое передается в хэш-функцию,

определяется как количество секунд (целое), прошедшее с 1 января 1970 Всемирного координированного времени (UTC) до 0 часов 0 минут соответствующего дня рождения.

Формат выходных данных

Строка в формате `DD.MM`, где

1. DD - двузначное число, обозначающее день месяца;
2. MM - двузначное число, обозначающее месяц.

Примеры

Пример №1

Стандартный ввод
2003 82717807fc9b67b1510f9dc37425bb2453a66ea41a9c297b0c9148816d449c6b
Стандартный вывод
21.10

Пример №2

Стандартный ввод
2004 a62bd35682c84f6ea77fbf057ab30c9a3525cb4e2ddd5f8d9211c8b3e1b95d8d
Стандартный вывод
29.02

Комментарии

Для решения задачи используйте функцию `keccak_256` из Python библиотеки `pysha3`.

Обратите внимание, что функции хэширования принимают на вход набор байт. Следовательно, перед тем как вычислить хэш-функцию от целого числа, данное число нужно представить в виде набора байт. В Ethereum целые числа конвертируются как 256битный код с порядком байт big-endian.

Решение

Для решения задачи, помимо указанной в примечании библиотеки `pysha3`, будем использовать встроенный модуль `datetime` и функцию *timestamp*, которая вычисляет количество секунд, прошедшее с 1 января 1970 Всемирного координированного времени, автоматически устанавливая временную зону +0 относительно UTC.

Однако уже для первого же теста ответ, полученный при помощи функции *timestamp* не совпадает с тем, который дан в условии. Проблема в летнем времени, которое сдвигает время на один час назад или вперед, в зависимости от даты. Для того, чтобы

обойти эту проблему, при переборе будем учитывать этот сдвиг, то есть проверять не только для 0 часов текущего дня, но и для 23 часов предыдущего, считая датой рождения в этом случае день, который наступит спустя час.

Кроме того, как было написано в задании, важно учитывать, что на вход функция *keccak_256* принимает набор из 32 байт.

Пример программы

Ниже представлено решение на языке Python3

```python
from _pysha3 import keccak_256
import datetime

def increaseTime(curr_datetime: datetime):
    if curr_datetime.hour == 23:
        curr_datetime += datetime.timedelta(hours=1)
    else:
        curr_datetime += datetime.timedelta(hours=23)
    return curr_datetime

year = int(input())
code = input()
year, month, day = year-1, 12, 31
ans = ""
dfTime = datetime.datetime(year, month, day, 23, 0, 0)

while ans != code:
    dfTime = increaseTime(dfTime)
    seconds = dfTime.timestamp()
    tempTime = int.to_bytes(int(seconds), 32, byteorder='big')
    ans = keccak_256(tempTime).hexdigest()
if dfTime.hour == 23:
    dfTime = increaseTime(dfTime)

print('{num:02d}'.format(num=dfTime.day),'{num:02d}'.format(num=dfTime.month), sep='.')
```

Задача 6.3.6. Получение публичного ключа по приватному (3 балла)

В блокчейн Ethereum для идентификации пользователей и доказательства владения используется криптография с открытым ключом, основанная на математических операциях на эллиптической кривой.

С каждым пользователем сети блокчейн ассоциирован его приватный ключ - случайная последовательность длиной 256 бит. Очевидно, что эту последовательность также можно представить в виде некоторого большого числа K.

Полные узлы сети Ethereum, майнеры, а также обычные пользователи всегда могут проверить, действительно ли какой-то конкретный пользователь является источником транзакции, зная его публичный ключ.

Публичный ключ генерируется на основе приватного. С точки зрения эллиптической криптографии, публичный ключ – координата точки на эллиптической кривой, полученная в качестве умножения некоторой базовой точки G, на значение равное приватный ключ K.

$P = G \cdot K$

Поскольку результатом данной функции являются координаты, то для представления их в виде публичного ключа, нужно выравнять их до 32байтных слов и сконкатенировать друг с другом.

Напишите программу, которая из приватного ключа генерирует публичный, используя криптографию на эллиптических кривых. Имейте в виду, что перед генерацией публичного ключа, который будет использоваться в блокчейн Ethereum, соответствующий приватный ключ нужно привести к набору из 32 байт (даже если он, как число, требует меньше разрядов).

Формат входных данных

Входная строка содержит последовательность символов – приватный ключ, записанный в шестнадцатеричном формате.

Формат выходных данных

Одна строка – публичный ключ, соответствующий приватному, записанный в шестнадцатеричном формате.

Примеры

Пример №1

Стандартный ввод
`c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`
Стандартный вывод
`a63a07e888061ec9e8b64a3dc2937805c76089af36459305920373cd98a9f4ce15` `c27dbbe60928161eb62ae19f94ea48f399ce85e6db698520f3bcd4a9257157`

Пример №2

Стандартный ввод
`10ca3eff73ebec87d2394fc58560afeab86dac7a21f5e402ea0a55e5c8a6758f`
Стандартный вывод
`765ee8b856d2ec6a1ea400c0dea57095fd096a45fe6735fe199738c4ec4909c6c2` `bda34586353edd45419d715925875ebb9f371c3eeb38ed3a3615b93e20dc1a`

Комментарии

В примере выходных данных публичный ключ представлен в виде двух разделенных строк из-за форматирования. В реальных входных данных публичный ключ должен быть выведен в одной строке.

Для решения задачи используйте модуль `secp256k1` библиотеки Python `py_ecc`.

Решение

Для решения задачи воспользуемся функцией `privtopub` из модуля `secp256k1`.

Напишем функцию, которая будет из приватного ключа получать публичный ключ, записанный в шестнадцатеричном формате. Для этого:

1. Сначала приведем приватный ключ к набору из 32 байт.
2. Затем при помощи библиотеки `py_ecc` получим необходимые координаты.
3. Выровняем полученные координаты до 32 байт и сконкатенируем их друг с другом.

Пример программы

Ниже представлено решение на языке Python3

```
from py_ecc import secp256k1

def privToPub(priv):
    priv = int(priv, 16).to_bytes(32, 'big')
    res = secp256k1.privtopub(priv)
    a = res[0].to_bytes(32, 'big')
    b = res[1].to_bytes(32, 'big')
    pub = int.from_bytes(a + b, 'big')
    return '0' * (128-len(hex(pub)[2:])) + hex(pub)[2:]

inp = open('input.txt', 'r')
out = open('output.txt', 'w')

for i in inp.readlines():
    i = i.strip()
    out.write(privToPub(i) + '\n')

inp.close()
out.close()
```

Задача 6.3.7. Получение адреса Ethereum (4 балла)

Между адресом пользователя в блокчейн Ethereum и его публичным ключом устанавливается непосредственная связь. Это сделано для того, чтобы не тратить дополнительное место (64 байта) в каждой транзакции для хранения публичного ключа.

Адрес пользователя - последовательность из 20 байт, которые получены из публичного ключа следующим преобразованием:

$addr = right(keccak256(pubkey), 20)$

Чаще всего, адрес пользователя можно увидеть записанным в виде шестнадцатеричного числа. Чтобы защитить адрес от случайных ошибок при переписывании с какого-то бумажного носителя (например, кто-то может свой адрес указать на визитке), была предложена система для интеграции кода проверки (checksum) в шестнадцатеричную запись адреса без изменения самого адреса: *EIP-55 «Mixed-case checksum address encoding»*. Суть системы в том, что в зависимости от значений

определенных битов хэш-сумме адреса некоторые строчные латинские буквы в его шестнадцатеричном отображении заменяются на заглавные.

Например,

`5aAeb6053F3E94C9b9A09f33669435E7Ef1BeAed`

Напишите программу, которая из приватного ключа генерирует адрес пользователя Ethereum и выводит позиции в шестнадцатеричной записи адреса, в которых строчная буква должна быть заменена на заглавную согласно *EIP-55*.

Формат входных данных

Входная строка содержит последовательность символов – приватный ключ, записанный в шестнадцатеричном формате.

Формат выходных данных

Две строки, первая - адрес без кода проверки, вторая – последовательность чисел, разделенных пробелом. Самый левый знак в шестнадцатеричной записи имеет позицию 0.

Примеры

Пример №1

Стандартный ввод
`c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470`
Стандартный вывод
`9cce34f7ab185c7aba1b7c8140d620b4bda941d6` `6 9 15 16 17 21 30 32 33 34`

Пример №2

Стандартный ввод
`10ca3eff73ebec87d2394fc58560afeab86dac7a21f5e402ea0a55e5c8a6758f`
Стандартный вывод
`eaee08c91b13d8d7a82850b714c7c68826e9cfae` `0 1 9 16 26 34 36 37 38`

Решение

Для получения адреса из публичного ключа воспользуемся модулем `keccak_256` из библиотеки `sha3`. Получение публичного ключа из приватного уже было разобрано в соответствующей задаче.

Во-первых, необходимо заметить, что на вход *keccak_256* получает последовательность байт. Кроме того, после кодирования данная функция возвращает массив из 32 байтов. Таким образом, для получения адреса надо обрезать первые 12 байт,

полученных в результате кодирования публичного ключа, что соответствует 24 символам в шестнадцатеричной записи.

Теперь рассмотрим алгоритм кода проверки *EIP-55*. Основная идея заключается в том, что адрес в шестнадцатеричной записи сравнивается с двоичным представлением результата применения функции *keccak_256* к данному адресу. То есть, если в шестнадцатеричной записи адреса стоит буква, а в двоичной записи закодированного адреса этой букве соответствует *1*, то данная буква должна быть заменена на заглавную. Если же данной букве соответствует *0* бит, то она не изменяется. Точно так же остаются без изменений и цифры, независимо от того, какой бит им соответствует. Кроме того, один символ в шестнадцатеричной системе счисления соответствует 4 битам в двоичном представлении, однако нам нужен самый первый из них.

Пример программы

Ниже представлено решение на языке Python3

```python
from py_ecc import secp256k1
from sha3 import keccak_256

def privToPub(priv):
    priv = int(priv, 16).to_bytes(32, 'big')
    res = secp256k1.privtopub(priv)
    x = res[0].to_bytes(32, 'big')
    y = res[1].to_bytes(32, 'big')
    return x+y

def pubToAddr(pub):
    return keccak_256(pub).hexdigest()[24:]

def privToAddr(priv):
    return pubToAddr(privToPub(priv))

def checkAddr(addr):
    hashAddr = bin(int(keccak_256(addr.encode()).hexdigest(), 16))[2:]
    hashAddr = '0' * (256 - len(hashAddr)) + hashAddr
    res = ''
    for i in range(len(addr)):
        if addr[i].isalpha() and hashAddr[4*i] == '1':
            res += str(i) + ' '
    return res

inp = open('input.txt', 'r')
out = open('output.txt', 'w')

for i in inp.readlines():
    i = i.strip()
    addr = privToAddr(i)
    out.write(addr + '\n')
    res = checkAddr(addr)
    out.write(res + '\n')

inp.close()
out.close()
```

Задача 6.3.8. Красивый адрес (5 баллов)

В сообществе криптовалют сформировалось несколько стартапов, которые предлагают пользователям сгенерировать "красивые"адреса.

Например, сколько может стоить вот такой Ethereum адрес?

`faceb00cab185c7aba1b7c8140d620b4bda941d6`

Генерация таких адресов без раскрытия своего приватного ключа стала возможна благодаря криптографии на эллиптических кривых.

Если зависимость адреса от приватного ключа можно выразить в виде:

$P = G \cdot K$

$addr = right(keccak256(P), 20)$

то можно написать также

$P_{faceb00c} = G \cdot (K + K_{faceb00c}) = G \cdot K + G \cdot K_{faceb00c} = P + G \cdot K_{faceb00c}$

Следовательно, пользователь, который хочет красивый адрес, не обязательно должен раскрывать свой приватный ключ. Вместо этого он может послать стартапу только свой публичный ключ и получить от него специальное дополнение, позволяющее получать требуемый красивый адрес.

Напишите программу, которая для известного публичного ключа формирует дополнительный приватный ключ, позволяющий получить красивый адрес.

Формат входных данных

В качестве входных данных будут переданы две строки. Первая строка содержит последовательность символов – публичный ключ, записанный в шестнадцатеричном формате. Вторая строка содержит четыре шестнадцатеричных знака - эти знаки должны стоять в красивом адресе в первых четырех символах.

Формат выходных данных

Одна строка - дополнительный приватный ключ, записанный в шестнадцатеричном формате.

Примеры

Пример №1

Стандартный ввод

`580e9e8c13aa4da3040962f57273ed5f1759c4edb32c6705310362bbf52f2c36`
`80c811172f177b2c3ed1d53ac7c521b8b1813284fbe0279557401bfe20d256da`
`face`

Стандартный вывод
`000000000000000000000000000000000000000000000000000000000000331c`

Пример №2

Стандартный ввод
45b23c86707f54d7f168f70d62248a0f4ca463cc9c57f7b2865759a4b6a8f9c7 cc2bb0a90ea6c8a0eeace75281c5fc09470996e93eed17cfc3e21a1a061ff45d b00c

Стандартный вывод
00000000000000000000000000000000000000000000000000000000000007e1

Комментарии

В примере входных данных публичный ключ представлен в виде двух разделенных строк из-за форматирования. В реальных входных данных публичный ключ будет записан в одной строке.

Для решения задачи можно скопировать содержимое модуля `secp256k1` Python библиотеки `py_ecc` в код своей программы.

Решение

Пожалуй, наиболее трудным в этой задаче было понять, каким образом необходимо складывать публичные ключи. Обычное сложение тут не подходит, как не подходит и конкатенация массивов байт. Дело в том, что для получения публичного ключа используются эллиптические кривые, и для решения задачи необходимо было использовать функционал, реализующий математику на этих кривых.

В самом условии дана подсказка, в которой советуется использовать модуль `secp256k1` Python библиотеки `py_ecc`. К сожалению, документация к данной библиотеке не представлена, но если зайти в нее, то можно найти целый набор функций, которые реализуют различные операции на эллиптической кривой, используемой в блокчейн ethereum.

Для данной задачи, в частности, можно использовать функцию *add*, которая принимает на вход две упорядоченные пары с хранящимися в них целочисленными координатами двух точек, которые необходимо сложить. На выходе же она возвращает координаты точки, получившейся в результате сложения.

Таким образом, для использования данной функции необходимо перевести публичный ключ в упорядоченную пару из двух целых чисел, которые представляют собой координаты для соответствующей точки на эллиптической кривой. Для хранения координат можно использовать структуру данных - кортеж.

Осталось только подобрать такой приватный ключ, чтобы при сложении с публичным ключом мы получали адрес, начинающийся с искомых символов. Единственный способ сделать это - это полный перебор всех 2^{256} вариантов. К счастью, ответ всегда достаточно близок, чтобы его можно было вычислить.

Пример программы

Ниже представлено решение на языке Python3

```
1 from py_ecc import secp256k1
2 from sha3 import keccak_256
3 from py_ecc.secp256k1.secp256k1 import add
```

```
4
5  # return public key in byteArray
6  def privToPub(priv):
7      priv = int(priv, 16).to_bytes(32, 'big')
8      res = secp256k1.privtopub(priv)
9      x = res[0].to_bytes(32, 'big')
10     y = res[1].to_bytes(32, 'big')
11     return x+y
12
13 # return address in hex
14 def pubToAddr(pub):
15     return keccak_256(pub).hexdigest()[24:]
16
17 # param s: 64 length byteArray
18 def prepareToAdd(s):
19     return (int.from_bytes(s[:32], 'big'), int.from_bytes(s[32:], 'big'))
20
21 # return public key in byteArray
22 def vanityAdd(x, y):
23     res = add(x, y)
24     return res[0].to_bytes(32, 'big') + res[1].to_bytes(32, 'big')
25
26 pub = input()
27 key = input()
28
29 pub = int(pub, 16).to_bytes(64, 'big')
30 pub = prepareToAdd(pub)
31
32 for i in range(2**256):
33     i = hex(i)[2:]
34     privAdd = privToPub(i)
35     privAdd = prepareToAdd(privAdd)
36     a = pubToAddr(vanityAdd(pub, privAdd))
37     if a[:4] == key:
38         print('0' * (64-len(i)) + i)
39         break
```

Задача 6.3.9. Цифровая подпись документа (2 балла)

Цифровая подпись – это цифровой аналог рукописной подписи. У цифровой подписи должны быть два свойства, присущи подписи рукописной:

- Во-первых, только конкретный человек может сделать свою подпись и все могут проверить, является ли подпись действительной;
- Во-вторых, подпись, оставленная на документе, не может быть использована для того, чтобы подписать другой документ.

Криптография с публичным ключом на математике эллиптических кривых позволяет реализовать цифровую подпись для доказательства владения данными.

В общем случае, говорят о двух функциях:

$sig = sign(K, message)$ - возвращает цифровую подпись для данного сообщения.

и

$isValid = verify(P, message, sign)$ - проверяет, что данное сообщение было подписано таким ключом, которому соответствует данный публичный ключ.

Видно, что для создания подписи нужен приватный ключ, а для проверки подписи достаточно лишь публичного ключа.

Следовательно:

- подделка цифровой подписи для конкретного сообщения невозможна, поскольку использование публичного ключ, соответствующего данному приватному ключу, продемонстрирует несоответствие подписи
- подмена сообщения невозможно, поскольку текущая цифровая подпись не будет соответствовать новому сообщению

В криптографии эллиптических кривых функция получения цифровой подписи возвращает два числа: r, s. В блокчейн Ethereum к цифровой подписи также добавляется еще одно число v. Оно позволяет упаковать цифровую подпись в структуру размером 65 байт, а также несет информацию о том, на каких узлах сети Ethereum данная цифровая подпись может проверяться.

Обратите внимание, что в функциях генерации и проверки цифровой подписи используется не само сообщение, а его хэш. В блокчейн Ethereum для получения хэша используется алгоритм *keccak*.

Напишите программу, которая для приватного адреса и заданного сообщения генерировала бы цифровую подпись. Для получения хэша сообщения используйте алгоритм *keccak*.

Формат входных данных

В качестве входных данных будут переданы две строки. Первая строка содержит последовательность символов – приватный ключ, записанный в шестнадцатеричном формате. Вторая строка содержит набор символов длиной не больше 255.

Формат выходных данных

Три строки. В первой v-компонента цифровой подписи, записанная в шестнадцатеричном формате (два символа). Во второй и третьей строках r- и s-компоненты, записанные в шестнадцатеричном формате.

Примеры

Пример №1

Стандартный ввод
`f56226b5d46a4c3307ab0283932095716ca5f1ba8d88b19f8ccf4227c09a1d9e` `Ethereum uses public key cryptography for authentication.`

Стандартный вывод

`1b`
`eec36bb56f15d2954d573ffad9e7a9e751479310971ed641787986f8aed7b5b8`
`40505b53f3f976f114cf585139dbf67be79538eb006f6bb7d2ca4bf10cd4b6f2`

Пример №2

Стандартный ввод
e1b791559d56f9a081fb45c8b1a1bdc36c10c0673e00d95497c13ec48b9a5ab8 Digital signatures employ asymmetric cryptography.

Стандартный вывод
1b 6cb8c7d270d8d254cd864e185a54ae90cb84669d012fcf9b4e39bfe245e40acc 67d7630b02c5a672eb5b071b296429ae8a7835b3e9fe022a82974493a6e891db

Решение

Для решения будем использовать функцию *ecdsa_ raw_ sign* из модуля `secp256k1` библиотеки `py_ecc`.

На вход данная функция принимает два набора байт: данные, закодированные при помощи алгоритма *keccak*, и приватный ключ, - и возвращает три целых числа: v, s и r.

Однако тут важно учесть сколько байт должны быть каждое из данных компонентов.

Согласно спецификации ethereum (`https://ethereum.github.io/yellowpaper/paper.pdf`, *Appendix F*), компонента v является 1-байтным элементом, а компоненты s и r - 32-байтными. Таким образом, перед выводом ответа, обязательно нужно добавить недостающее количество нулей в шестнадцатеричную запись компонентов.

Пример программы

Ниже представлено решение на языке Python3

```
from py_ecc.secp256k1 import ecdsa_raw_sign
from sha3 import keccak_256

inp = open('input.txt', 'r')
out = open('output.txt', 'w')

n = 5

inp = inp.read()
inp = inp.split('\n')

for i in range(0, n*2, 2):
    priv = int(inp[i], 16).to_bytes(32, 'big')
    data = keccak_256(inp[i+1].encode()).digest()
    ans = ecdsa_raw_sign(data, priv)
    res = ''
    for j in range(3):
        if j > 0:
            res += '0' * (64 - len(hex(ans[j])[2:])) + hex(ans[j])[2:] + '\n'
        else:
            res += '0' * (2 - len(hex(ans[j])[2:])) + hex(ans[j])[2:] + '\n'
    out.write(res)
```

Задача 6.3.10. Проверка подписи в транзакции (5 баллов)

Перед тем, как транзакция Ethereum будет отправлена в сеть, она подписывается на узле с использованием одного из приватных ключей, доступных на данном узле.

Транзакция подписывается не вся, а только следующие данные в ней:

```
nonce
gasprice
gas
to
value
data
```

Перед тем, как быть подписаны, эти данные сериализуются - представляются в виде набора байт с помощью RLP (Recursive Length Prefix) кодирования.

Напишите программу, которая бы проверяла транзакцию на предмет подделки.

Формат входных данных

В качестве входных данных будут передано 8 транзакции в виде JSON-записей их содержимого.

Формат выходных данных

Одна строка - число от 0 до 255, которое, если его записать в двоичном виде, кодировало бы состояние транзакций: если в нулевом бите — 0, то первая в списке транзакция подделана, если — 1, то верификация первой транзакции прошла без ошибок; если в первом бите — 0, то вторая в списке транзакция подделана, если — 1, то верификация второй транзакции прошла без ошибок и т.д.

Примеры

Пример правильной транзакции в формате JSON:


```
{"blockHash": "0x00", "blockNumber": 0,
"from": "0x58f6ab42b6345907fa20aeca0274548dee6e2e32",
 "gas": 90000, "gasPrice": "28000000000", "hash": "0x00", "input": "0x", "nonce": 1,
 "r": "0xcc8350cb3a4390832f6c1af92a8354438d6c0d7029666c414bc88f44bfd2caba",
 "s": "0x7a00af5fc114415494e16a9a657801e1813e5f4193b226b409a05d87dd613de7",
 "to": "0x9a1cdf391b4241da2aca61af6b746e273b24fad9", "transactionIndex": 0,
 "v": "0x26", "value": "1236096691978933262"}
```


Пример некорректной транзакции в формате JSON:


```
{"blockHash": "0x00", "blockNumber": 0,
"from": "0x2ed42f423ee2f457ab6968203ba08a7e564ac0e5",
 "gas": 90000, "gasPrice": "53000000000", "hash": "0x00", "input": "0x", "nonce": 0,
 "r": "0x518eeb19f4b25ddadfab228cc4bdd45b585056544effc78d804b4821da648b87",
 "s": "0x348138d4db546f6f488cb5cd99c3940966ab52627a80fd60c4490f857ab76f99",
 "to": "0x4dc922e93e2e1f1059d72bc7e7e0415c2b22335a", "transactionIndex": 0,
 "v": "0x26", "value": "5944533907010808614"}
```

Комментарии

Для решения задачи используйте модуль `transactions` из Python библиотеки `pyethereum`.

Решение

Для решения задачи воспользуемся указанным в примечании модулем `transactions` из библиотеки `pyethereum`.

Входные данные в том виде, в котором они даются в задаче, будем считывать из файла "transactions.txt".

Создадим новый экземпляр класса Transaction с данной транзакцией. При этом стоит заметить, что при инициализации объекта значения v, r и s должны быть приведены к типу *big_ endian_ int*, а все префиксы для обозначения системы счисления - удалены.

Далее для каждой транзакции необходимо найти адрес её отправителя, для чего можно воспользоваться функцией `sender`, и сравнить его с адресом, записанным в JSON-записи. При их совпадении можно считать транзакцию правильной. Кроме того, если значения v, r или s будут некорректными функция *sender* может поднимать исключение. Это будет означать, что транзакция была подделана, и обрабатывать этот случай необходимо отдельно.

Пример программы

Ниже представлено решение на языке Python3

```
from ethereum.transactions import Transaction
from ethereum import utils

res = ''
f = open('transactions.txt', 'r')
for i in f.readlines():
    in_tx = eval(i)
    t = Transaction(in_tx['nonce'], int(in_tx['gasPrice']), in_tx['gas'], in_tx['to'],
        int(in_tx['value']), in_tx['input'][2:], int(in_tx['v'], 16),
        int(in_tx['r'], 16), int(in_tx['s'], 16))
    try:
        sender = utils.encode_hex(t.sender)
        if sender == in_tx['from'][2:]:
            res += '1'
        else:
            res += '0'
    except:
        res += '0'
print(int(res[::-1], 2))
```

Задача 6.3.11. Построение Merkle Tree (2 балла)

Одним из базовых методов, обеспечивающих обеспечение целостности данных в блокчейн, является построение дерева Меркле (*Merkle Tree*, `https://en.wikipedia.`

org/wiki/Merkle_tree). Поскольку не всегда именно строится классическое дерево, иногда этот метод называют даже хэш-функцией.

Данный метод позволяет решить две проблемы:

- определить хэш какого-то набора данных и использовать этот хэш для проверки их целостности;
- проверить наличие какого-то конкретного набора данных в дереве, используя *merkle path*.

Самым первым дерево Меркле стало использоваться в блокчейн Bitcoin для получения хэша всех транзакций (*merkle root*), включенных майнером в блок.

Напишите программу, которая бы находила *merkle root* для транзакций в блоке, сформированном в блокчейн Bitcoin.

Формат входных данных

Первая входная строка N содержит число ($2 \leqslant N \leqslant 2000$), определяющее количество транзакций, для которых нужно получить *merkle root*. Последующие N строк содержат хэши транзакций, записанные в шестнадцатиричном формате.

Формат выходных данных

Одна строка – итоговый хэш - вершину дерева *merkle root*.

Примеры

Пример №1

Стандартный ввод
4 8c14f0db3df150123e6f3dbbf30f8b955a8249b62ac1d1ff16284aefa3d06d87 fff2525b8931402dd09222c50775608f75787bd2b87e56995a7bdd30f79702c4 6359f0868171b1d194cbee1af2f16ea598ae8fad666d9b012c8ed2b79a236ec4 e9a66845e05d5abc0ad04ec80f774a7e585c6e8db975962d069a522137b80c1d
Стандартный вывод
f3e94742aca4b5ef85488dc37c06c3282295ffec960994b2c0d5ac2a25a95766

Пример №2

Стандартный ввод
9 ef1d870d24c85b89d92ad50f4631026f585d6a34e972eaf427475e5d60acf3a3 f9fc751cb7dc372406a9f8d738d5e6f8f63bab71986a39cf36ee70ee17036d07 db60fb93d736894ed0b86cb92548920a3fe8310dd19b0da7ad97e48725e1e12e 220ebc64e21abece964927322cba69180ed853bb187fbc6923bac7d010b9d87a 71b3dbaca67e9f9189dad3617138c19725ab541ef0b49c05a94913e9f28e3f4e fe305e1ed08212d76161d853222048eea1f34af42ea0e197896a269fbf8dc2e0 21d2eb195736af2a40d42107e6abd59c97eb6cffd4a5a7a7709e86590ae61987 dd1fd2a6fc16404faf339881a90adbde7f4f728691ac62e8f168809cdfae1053 74d681e0e03bafa802c8aa084379aa98d9fcd632ddc2ed9782b586ec87451f20

Стандартный вывод
2fda58e5959b0ee53c5253da9b9f3c0c739422ae04946966991cf55895287552

Комментарии

В качестве хэш функции в Bitcoin используется стандартный *sha256()* (например, из *Python* библиотеки `hashlib`).

Также еще одной особенностью вычисления хэша двух элементов в дереве Меркле блоков Bitcoin, является двойное хэширование:

$H_{AB} = sha256(sha256(H_A + H_B))$

Обратите внимание, что для хранения числа в Bitcoin используется порядок байт little-endian.

Решение

Для начала рассмотрим, что представляет из себя дерево Меркле. В документации к блокчейну Bitcoin можно найти следующий рисунок (`https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch09.asciidoc`):

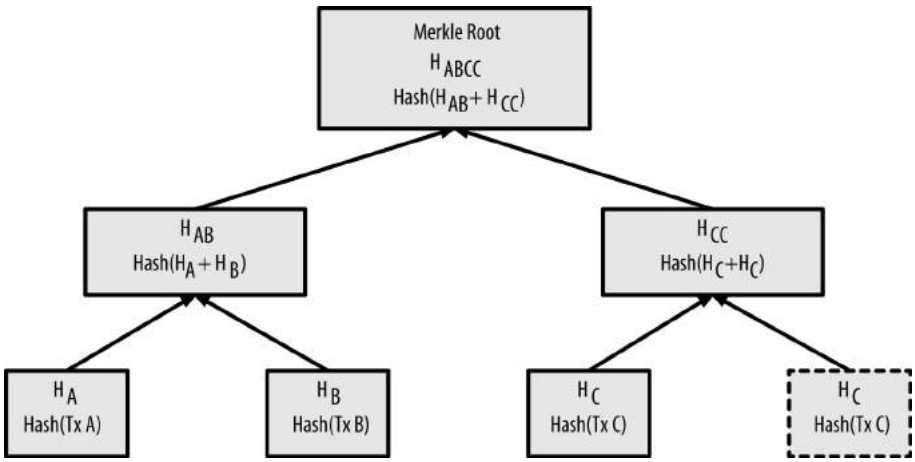

Рис. 6.3: Дерево Меркле

При помощи данного рисунка попробуем разобрать принцип получения *merkle root* в блокчейне Bitcoin. Так, в листьях дерева находятся хэши транзакций, которые хранятся в блоке. Каждая верхняя нода получается при хэшировании результата

конкатенации двух своих детей. Заметим также, что если на каком-либо уровне количество нод нечетное, то мы дублируем последнюю ноду и конкатенируем её с ней же.

Далее, все там же мы можем найти, что представляет из себя функция хэширования в *Bitcoin*. Её результатом будет двойное применение алгоритма *sha256*. То есть, $H_A = sha256(sha256(A))$

И наконец надо вспомнить, что *Bitcoin* использует для кодирования последовательность little-endian. То есть перед тем как применять хэш-функцию к какому-либо значению, предварительно её необходимо представить в виде последовательности байтов, упорядоченных от младшего байта к старшему.

Зная все это, мы можем написать рекурсивный алгоритм по вычислению *merkle root*, который принимает на вход массив из транзакций и возвращает единственное значение - корень дерева Меркле.

Пример программы

Ниже представлено решение на языке Python3

```python
import hashlib, binascii

def hash2(a, b):
    a1 = binascii.a2b_hex(a)[::-1]
    b1 = binascii.a2b_hex(b)[::-1]
    h = hashlib.sha256(hashlib.sha256(a1+b1).digest()).digest()
    return h[::-1].hex()

def merkle(hashList):
    if len(hashList) == 1:
        return hashList[0]
    newHashList = []
    for i in range(0, len(hashList)-1, 2):
        newHashList.append(hash2(hashList[i], hashList[i+1]))
    if len(hashList) % 2 == 1:
        newHashList.append(hash2(hashList[-1], hashList[-1]))
    return merkle(newHashList)

a = []

n = int(input())

for i in range(n):
    a.append(input())

print(merkle(a))
```

Задача 6.3.12. Использование Merkle Patricia Tree (3 балла)

В блокчейн Ethereum вместо вычисления хэша списка транзакций по методу дерева Меркле используется комбинированный подход, который получил название Merkle Patricia Tree.

В основе этого подхода лежит префиксное дерево (*Radix tree*, `https://en.wikipedia.org/wiki/Trie`), которое очень требовательно к потреблению памяти, поэтому для

решения этой проблемы в блокчейн Ethereum было предложена оптимизация, так появился метод построения дерева *Patricia Tree*. Чтобы сохранить возможность подсчитывать итоговую хэш сумму всех конечных элементов дерева в алгоритм были введены механизмы, использующиеся при построении дерево Меркле, и конечное название метода вычисления хэш суммы для какого-то списка данных стал называться *Merkle Patricia Tree*:

- `https://blog.ethereum.org/2015/11/15/merkling-in-ethereum/`;
- `https://github.com/ethereum/wiki/wiki/Patricia-Tree`;
- `https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/`

Для построения дерева используются пары *ключ*: *значение*. *Ключ* определяет путь в дереве, *значение* помещается в конечный элемент ("лист") дерева. Промежуточные узлы дерева также представляются в виде пары *ключ*: *значение*, обозначая промежуточный путь и промежуточный хэш соответственно.

В случае списка транзакций в качестве *ключа* используется порядковый номер транзакции, выданный ей при включении в блок: $rlp(transactionIndex)$. В качестве *значения* используется хэш транзакции.

Напишите программу для определения количества непустых конечных элементов во всех промежуточных узлах (*extensions*), сформированных при построении *Merkle Patricia Tree* для списка транзакций в блоке Ethereum.

Формат входных данных

Три строки, каждая из которых содержит число N ($256 \leqslant N \leqslant 4370$), определяющее количество транзакций, включенных в блок.

Формат выходных данных

Каждой входной строке сопоставляются три выходных строки, в каждой из которых по два числа, разделенных пробелом.

В каждой строке первое число - количество непустых элементов, принадлежащих непосредственно промежуточным (*extensions*) узлам, полученных при построении дерева транзакций. Второе - количество всех дочерних конечных, но не являющихся поддеревьями, элементов в промежуточных узлах. Выведите 0 0, если промежуточные узлы не были сформированы при построении дерева.

Примеры

Пример №1

Стандартный ввод
256 4099 330
Стандартный вывод
0 0 3 3 5 74

Комментарии

Примечание к примеру: для 4099 транзакций при построении дерева сформировался 1 промежуточный узел, который содержит только 3 элемента, а для 330 транзакций при построении дерева сформировался 1 промежуточный узел, который содержит только 5 поддеревьев, в которых содержится 74 элемента.

Решение

Для построения дерева воспользуемся модулем `trie` библиотеки `ethereum`. При помощи данного модуля мы можем построить дерево для списка транзакций. Как было указано в задании в этом случае в качестве ключа используется порядковый номер транзакции. Заметим, что нам не особо важно что использовать в качестве значений, так как это не должно влиять на само дерево.

Таким образом, самым первым шагом необходимо построить дерево из указанного количества транзакций, а затем, используя построенное дерево, надо посчитать количество непустых элементов, принадлежащих непосредственно промежуточным узлам, и количество всех дочерних конечных, но не являющихся поддеревьями, элементов в промежуточных узлах.

Однако для того чтобы полностью обойти дерево и найти количество необходимых узлов и листьев, необходимо прежде всего понять, что же эти узлы значат и каким образом они располагаются в дереве.

На приведенном ниже рисунке схематично изображено Merkle Patricia Tree. Все значения кодируются в шестнадцатеричной системе счисления, поэтому максимальное количество детей у узла равно 16 (по одному на каждый возможный вариант следующего символа ключа). Существует 4 разновидности узлов:

1. Blank (пустой) node - представленный как пустой набор байт.
2. Leaf (конечный) node - узел, представляющий собой обычную пару [ключ, значение].
3. Extension (промежуточный) node - узел, которых также представляет собой пару [ключ, значение], однако в данном узле. значение является хэшом другого узла.
4. Branch (ветвления) node - массив, состоящий из 17 узлов, каждый из которых представляет собой один из 16 возможных следующих символов в ключе и значение для данного ключа, если оно существует.

Как можно заметить, за каждым промежуточным узлом следует узел ветвления, а вот из узел ветвления может вести как в промежуточный узел, так и в конечный.

Наша задача - посчитать сколько узлов в узле ветвления являются не пустыми, и сколько листьев можно достичь пройдя через промежуточный узел.

Для этого напишем рекурсивную функцию обхода всего дерева, которая будет принимать на вход само дерево, текущий узел, количество непустых элементов, принадлежащих промежуточным узлам, переменную, хранящую значение, был ли на данном пути пройден хотя бы один промежуточный узел, и количество конечных элементов в промежуточных узлах.

Пример программы

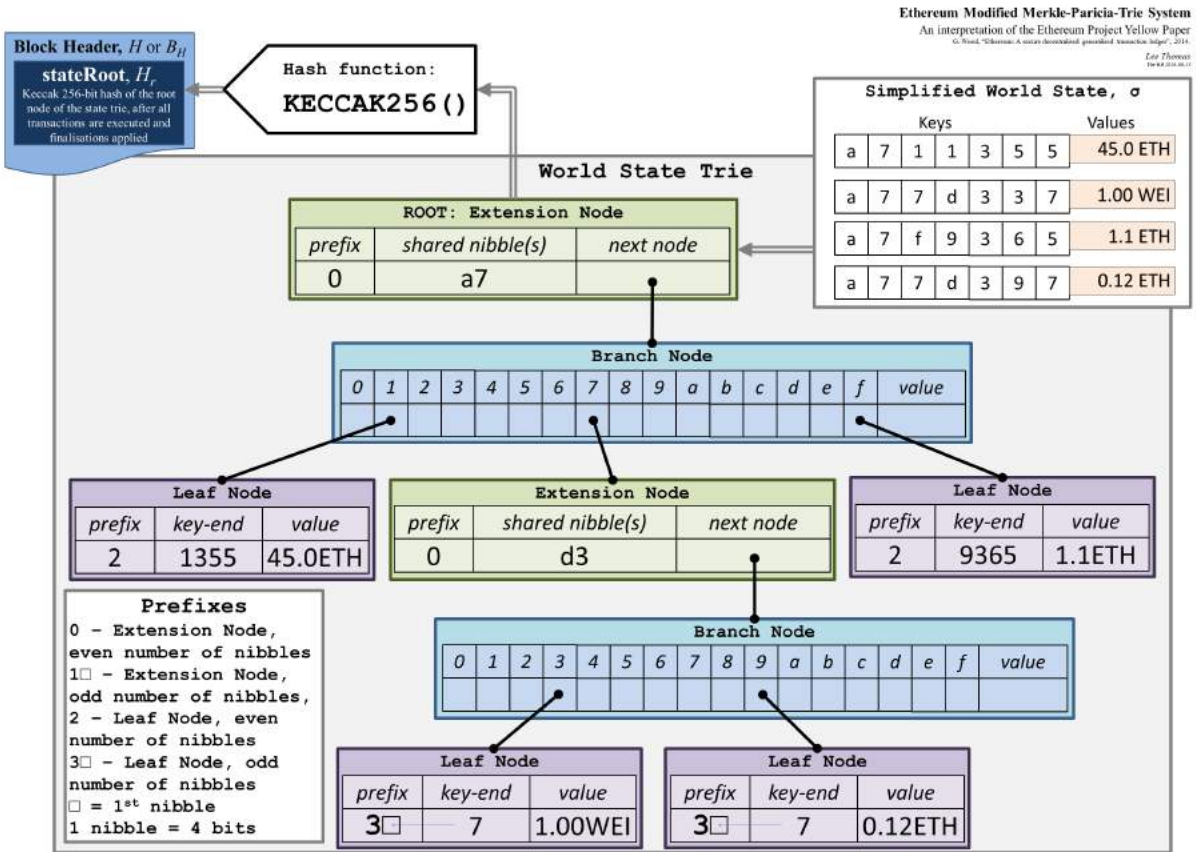

Рис. 6.4: Patricia Trie

Ниже представлено решение на языке Python3

```
from ethereum import db
from ethereum import trie
from ethereum import utils
import rlp

def countExtensionNodesWithLeaves(state, node, ext_size, flag, leaves_num):
    if state._get_node_type(node) == trie.NODE_TYPE_LEAF and flag:
        leaves_num += 1
    if state._get_node_type(node) == trie.NODE_TYPE_EXTENSION:
        temp = state._decode_to_node(node[1])
        for i in state._decode_to_node(temp):
            if i != b'':
                ext_size += 1
        flag = True
        ext_size, leaves_num = count_extension_nodes(state, state._decode_to_node(temp),
            ext_size, flag, leaves_num)
    elif state._get_node_type(node) == trie.NODE_TYPE_BRANCH:
        for i in node:
            if i != b'':
                temp = state._decode_to_node(i)
                ext_size, leaves_num = count_extension_nodes(state, temp,
                    ext_size, flag, leaves_num)
    return ext_size, leaves_num

res = []

for i in range(3):
    n = int(input())
```

```
29
30      state = trie.Trie(db.EphemDB(), trie.BLANK_ROOT)
31      for i in range(0, n):
32          state.update(rlp.encode(i), b'')
33
34      res.append(countExtensionNodesWithLeaves(state, state.root_node, 0, False, 0))
35
36  for i in res:
37      print(*i)
```

Задача 6.3.13. Дерево состояний (7 баллов)

Одним из нововведений, которое было сделано в блокчейн Ethereum, является введение понятия «состояния». Его смысл позаимствован из *Теории автоматов*. Теперь, когда говорится о выпуске нового блок одним из майнеров, то это значит, что блокчейн переходит в новое состояние. Набор транзакций, входящих в блок, определяет каким именно это новое состояние является: какое стало состояние баланса на адресах пользователей, как изменилось внутреннее состояние умных контрактов.

Например, пусть блокчейн был в некотором состоянии S_1, когда баланс адреса $A_{S_1} = 50\ ether$, а баланс адреса $B_{S_1} = 10\ ether$. Тогда, при появлении нового блока с транзакцией перевода 15 $ether$ с адреса A на адрес B и распространении этого блока на все узлы сети, блокчейн перейдет в состояние S_2, которое характеризуется тем, что баланс адреса $A_{S_2} = 35\ ether$, а баланс адреса $B_{S_1} = 25\ ether$.

Для того, чтобы отслеживать соответствие состояния на всех узлах сети Ethereum, в каждый блок майнеры включают хэш новой вершины *Merkle Patricia Tree*, которое хранит информацию о состоянии каждого конкретного адреса и поэтому называется дерево состояния блокчейна (*World State Tree*).

Иными словами, в простом случае, алгоритм изменения состояния блокчейн Ethereum можно описать следующим образом:

1. Выбрать из списка неподтвержденных транзакций те, которые планируется включить в новый блок.
2. Проверить корректность каждой транзакции:
   (a) Адрес, получаемый от проверки цифровой подписи, не нулевой;
   (b) Номер транзакции (nonce), совпадает с ожидаемым номером, хранимым в записи соответствующего адреса в дереве состояний блокчейна, обозначающим сколько транзакций отправитель уже послал;
   (c) Ограничению по использованию газа (газ - условная единица, обозначающая сколько потребуется Ethereum Virtual Machine ресурсов, таких как память и процессорное время, для обработки транзакции), указанное отправителем в транзакции, не меньше, чем количество газа, требуемое для хранения информации, переданной в транзакции;
   (d) В дереве состояния блокчейна на балансе адреса, полученного от проверки цифровой подписи, достаточно количества средств для списания.
3. Для каждой транзакции в дереве состояния блокчейна изменить баланс соответствующих адресов: уменьшить на том адресе, который получился после проверки подписи, и увеличить на адресе-получателе. А также увеличить

номер ожидаемой транзакции у соответствующих адресов.

4. Увеличить баланс адреса, выставленного как coinbase на данном узле.
5. Включить все транзакции в новый блок и отправить его соседним узлам для дальнейшего распространения по сети.

Как только новый блок достигает других участников сети, каждый узел сети Ethereum выполняет следующие операции:

1. Проверка блока на корректность (здесь не будет рассматриваться подробно).
2. Для каждой транзакции включенной в блок проверяется корректность, выполняя шаги 2.a - 2.d, описанные выше.
3. Для каждой транзакции в дереве состояния блокчейна изменить баланс соответствующих адресов: уменьшить на том адресе, который получился после проверки подписи, и увеличить на адресе-получателе. А также увеличить номер ожидаемой транзакции у соответствующих адресов.
4. Увеличить баланс адреса, выставленного как coinbase в блоке.

Если в какой-то момент на шаге 2 будет обнаружена некорректная транзакция, то блок будет отброшен, а значит, ни одна транзакция, включенная в этот блок, не будет использована для изменения баланса адресов. В дальнейшем, если на данный узел не получит другой блок, но уже корректный, с таким же номером, то узел перестанет быть сихнронизированным с остальной сетью.

При этом, поскольку одни и те же действия по изменению баланса адресов выполняются на всех узлах, то дерево состояния блокчейна (*World State Tree*) на всех узлах одно и тоже. Все узлы должны обновлять это дерево с при получении новых блоков для того, чтобы иметь возможность создавать новые блоки, которые будут приняты другими участниками сети. Следовательно, участники сети будут признавать право на получение награды за майнинг и позволять распоряжаться полученными в качестве награды средствами.

Напишите программу для проверки на корректность транзакций в блоках. Под корректной транзакцией в данном задании будет подразумеваться такое изменение баланса адреса Ethereum, которое приводит к значению большему, либо равному нулю.

При подсчете баланса не учитывать оплату газа на обработку транзакции.

Формат входных данных

Первая входная строка содержит число N ($5 \leqslant N \leqslant 1000$), определяющее количество блоков. Следующие N наборов строк описывают транзакции, включенные в блоки следующим образом:

- Первая строка в наборе состоит из двух значений - количества транзакций в блоке и адреса, получающего награду за данный блок, записанного в виде 40 шестнадцатеричных символов. Величина награды - $5 \cdot 10^{18}$ условных единиц криптовалюты.
- Оставшиеся строки в наборе - транзакции в формате Ethereum, закодированные с помощью RLP и записанные в шестнадцатеричном виде.

Формат выходных данных

Вывести сумму на балансе пяти адресов с самым большим остатком на момент обнаружения некорректной транзакции. В случае, если некорректных транзакций нет, вывести сумму на балансе пяти адресов с самым большим остатком на момент применения самого последнего блока.

Примеры

Пример №1

Стандартный ввод
6 0 5230b4da9c28bfb089b8dd7bd6898a1594dba960 3 f75c80053674ee58a99cef9ee3b0ff19f112c795 f86d0186246139ca800082520894c6c4571388ecc770a25c4aeb30f9bbe16459371a882c68af0bb 14000008026a0da384535815daa0e910354d74f3a7b35dcd427baa31366595b6f6dc6baa96341a0 18a7ab431c67bd795ea16a8431eb6c5cff5d440c9699752b6e27770df9fed86b f86d028612309ce5400082520894272ac4bccc024692903951996b7e4c8b3c19ffc28806f05b59d 3b200008025a09dd726bb23005f20910a4f749a5113e2eba40e35ced62d64bdedcbe681bc1a1fa0 6a4ec0d1f81e1aab839a19a35780af2946ab93f455fe71c509338a2ed76d6fe9 f86d01863205af7670008252089466d3144d6ee8be6a340bea305b13ad202fe6c66f880429d0691 89e00008026a094e3551725a9cd9693a8e923a56155e02ea55b2ece0cd2be15fee9c6c908024aa0 4db3e2303cecb5aed9358b14cfc5fec7be271eb145c499f57a6b74f00304d88d 1 bf0b5a4099f0bf6c8bc4252ebec548bae95602ea f86d018615d3ef79800082520894abd46c781f71a836c941e6fc4d245cb31b0b60da8802c68af0b b1400008025a015194a147749c6f49a40ba8e25089ad71ea084f5c95c036b564a8df664b3b36da0 28c0bf37923672dabbccf030bfcc5533c00729f4bb194d8cb65bf5541543298c 2 d55aba8b51669eff3b91995c7a545a9d0da3b69e f86d018615d3ef79800082520894ffaf33a6f1f7061b75ce510fbc2ee1a443befb1188058d15e17 62800008025a005ff35ad86a69e28609ae696b2509ea6be3225cf8a9c73bddb041bbdff381366a0 304f44088648e8a4e4c5af9fdba1995241e3ffee13f1bd618c273750b5df0a53 f86d0186246139ca80008252089485bf78837156632e92a80ebe160f64b228994a24880c7d713b4 9da00008025a04bf24fa514dd5264091afff040c0f7533991cd2a31faa36fd63ae583e4cb1e7ba0 46bee665e7319af5da61955931584cabb85cfd8ea91b20fdeeca117ae3f11b8b 5 53a7522e00e530f1076c4056a1cbb47229a39dd6 f86d01861b48eb57e0008252089402bd7d6559bb2597f6515390fae4035ed3a4186e880853a0d23 13c00008025a0bc7eff3689ff290e9b7675508229c57a8ac5d9a044eb884ec92f4085b6e8a6a4a0 240116d133f2e5cced21cb0ea6abc98a942e2700135367b283c07928c3c8d60f f86d01862d79883d2000825208949702014 40c7bc5689554107ba3f67be0395176f4880c7d713b4 9da00008026a082a236df87e79a872e8a8caaff7c1e4952b15a63e995ce54745b23d106df6b71a0 1a223c0404a7bed357c6523920bbe3fa8c5252ff31d540a97972516ec411b751 f86d02861147c84030008252089416ac683d6c7504ad0b31e808b2e991575cc7e99e88016345785 d8a00008025a013b02f8013baf3b53059b5c951b073b04478fe16557b438e45b8553a91e5afb6a0 23ad05f9d8475bdde280808e8bc55015631cd17395a430dad2dfedd3af1fea13 f86d01861402462f6000825208944d7000adbfac757f9e1337f3033bd33eaeb969078802c68af0b b1400008026a0ecfeabff10c6cc8d2783359d11c01a4ee29eb52763cfd726eb28cca55f05e441a0 08b68423c874fe23a959fc7d157fc1f85174396896906d782f2a59eafbf7301b f86d038632ee841b800082520894abd46c781f71a836c941e6fc4d245cb31b0b60da880853a0d23 13c00008026a03f79c560dc049eb19ad7e9b169b65fc07d801f889733dbe78e3d53f5a92e17a3a0 24630de8f523261b36634b13d31fd81e48ccba621c47aaafa26d15ce99036bf9 1 7d050326fdf9c1d94495f22dd3ab8807422f1a96 f86d018617a598c3a00082520894bf0b5a4099f0bf6c8bc4252ebec548bae95602ea88016345785 d8a00008026a03742e9ebfb32039c8104c6853ce01ed505f2a7220e5414aaecdc5a68d0b654a3a0 4be302869f0219688c263b66205ec3b0e0e0191a64335c14dcbc026c9404cde8

Стандартный вывод
18300000000000000000

Пример №2

Стандартный ввод
7 0 db6c6e77162e0e023eb9abd972621265202772e3 0 f3e63cff4f7f3202abe9126658e9b3359a24243a 0 4943b96530bab4678d68fa093a8d7bc003cadd52 1 f75c80053674ee58a99cef9ee3b0ff19f112c795 f86d01861a6016b2d00082520894bca0bc749270a42ab2d747ff62a949ddeba93dae88136dcc951 d8c00008025a0290cd60459f616232a71eae9ca05885e9aca557eb6dfcf953b0b886a09c8fb8ca0 5afb526545a5b9434835994950272d191d389cad02dfa04fe2728175a823d157 0 c948603e1b4b553c3e1fc3d99e95d5260d667fca 0 53a7522e00e530f1076c4056a1cbb47229a39dd6 3 02178117a34fe7e25b8454b0d55cb00c91b1c6a4 f86d0186105ef39b2000825208947abf7cde5d983137d9b09487b5091a3033953461880b1a2bc2e c5000008025a0145ccdf0f330596629dcda12c31d785fd4a9ff0a34f9899a699bceef755317e6a0 6847afb566c391afc3c166b908bc19f0843016fb4d70136ecfbe6a1df8bf1758 f86d01861a6016b2d000825208940839702e3f61f7fed6e058f4a5f08f3bd23be74f88016345785 d8a00008026a0043bce12b091e17858852623889c798ca61106a65dc8dfc420938ce4250bc211a0 255ab21e6be35d7bfdbed86831ff621b41e1cd67ff49392e2b1e6038cf056e5b f86d018628ed6103d0008252089485bf78837156632e92a80ebe160f64b228994a2488016345785 d8a00008025a061d3d728042f6051a42a91c3416b3e18cffc7e7d190b5d8fc1c0978f0d167108a0 703e094e029dcbf8d5022f23e02bd077ea35ba217a59829f04b7046836855baf

Стандартный вывод
25000000000000000000

Решение

Для решения воспользуемся тем функционалом, который предоставляют нам библиотеки `ethereum`, а в частности модулями `transactions`, `utils` и `rlp`.

Так как транзакции закодированы в шестнадцатеричном формате, то перед тем, как декодировать их при помощи rlp, нам необходимо конвертировать их в массив байт. Для этого воспользуемся функцией *decode_hex* модуля *utils*.

Полученный массив байт необходимо десериализовать, используя rlp. Заметим, что модуль *rlp* позволяет сериализовывать свои структуры данных, в частности, мы можем сериализовать объект класса Transaction, конструктор которого находится в модуле *transactions* библиотеки *ethereum*. Соответственно, с таким же успехом мы можем десериализовать объект данного класса. Таким образом, мы получаем возможность работать с данными транзакциями, используя все преимущества этой структуры данных.

Далее, необходимо заметить, что майнер получит награду только в том случае, если выпущенный им блок окажется правильным. То есть, в начале необходимо проверять правильность блока и лишь затем присваивать майнеру награду. Более того, если какая-либо транзакция в блоке оказалась некорректной, то все транзакции в блоке являются недействительными и никак не влияют на состояние сети.

Принимая во внимание описанные выше ограничения, напишем функцию, которая будет возвращать состояние аккаунтов в сети к моменту обнаружения некорректной транзакции. Входные данные будем считывать из файла, а изменения на балансах будем производить только после проверки правильности конкретного блока.

Пример программы

Ниже представлено решение на языке Python3

```python
from ethereum import transactions
from ethereum import utils
import rlp

# Возвращает состояние балансов на момент обнаружения некорректной транзакции
def get_state(f):
    balances = dict()
    n = int(f.readline())
    for i in range(n):
        t_balances = balances.copy()
        tx_num, miner = f.readline().strip().split()
        for j in range(int(tx_num)):
            tx_hash = f.readline().strip()
            tx = (rlp.decode(utils.decode_hex(tx_hash), transactions.Transaction)).to_dict()
            sender, to, value = tx['sender'][2:], tx['to'][2:], int(tx['value'])
            if t_balances.get(sender, 0) - value < 0:
                return balances
            t_balances[sender] -= value
            t_balances[to] = t_balances.get(to, 0) + value
        balances = t_balances.copy()
        balances[miner] = balances.get(miner, 0) + 5*10**18
    return balances

f = open('input.txt', 'r')
balances = get_state(f)

sorted_balances = sorted(list(balances.values()), reverse=True)

res = 0
n = len(sorted_balances)
for i in range(min(n, 5)):
    res += sorted_balances[i]

print(res)
```

Задача 6.3.14. Доказательство работы (5 баллов)

Принято считать, что алгоритмы доказательства работы (Proof-of-Work, PoW) представляют из себя некую математическую ”головоломку”, которую должен решить узел перед тем, как выпустить новый блок. ”Головоломка” (англ. *puzzle*) заключается в том, что узел должен найти число или последовательность байт, удовлетворяющих определенным условиям. Эти условия, чаще всего, зависят от конкретного момента времени, а точнее данных, которые доступны только в данный момент времени, поэтому ”головоломку” нельзя решить заранее. Также ответ к ”головоломке” не поддается непосредственному вычислению: единственный способ - угадать его или подобрать, перебрав из нескольких возможных вариантов, в то время как проверить правильный ли это ответ для конкретных данных другие участники сети могут относительно просто и без использования дополнительного оборудования.

Узел нашедший ответ, перебрав его из нескольких вариантов, как бы заявляет: «Я отправляю информацию, приложив к ней доказательство того, что я в полном праве эту информацию отправлять, поскольку это право я приобрел, выполнив значительный объем работы для поиска данного доказательства.»

Использование алгоритма доказательства работы в блокчейн преследует несколько целей:

1. Выполнение долгого по времени перебора для поиска подходящего ответа позволяет защитить информацию, записанную в блокчейн от изменения. Каждый, кто захочет изменить данные "задним числом" должен подобрать новое число для измененной записи. А поскольку между записями существует связь, когда последующая запись ссылается на предыдущую, то нужно новое подобрать новое число для каждой последующей записи. Чем их больше, тем дольше придется подбирать новые числа, а за это время могут появиться новые записи, что снова добавит работы злоумышленнику.

2. Поскольку заранее невозможно предугадать сколько займет времени подбор нового числа, то эта неопределенность позволяет осуществить выбор узла, который выпустит новый блок транзакций. Разная вычислительная производительность систем, разное количество транзакций, накопившихся у каждого элемента сети, делают процесс выбора узла вероятностным. С точки зрения стороннего наблюдателя кажется, что выбор узла, который выпустил новый блок, происходит случайным образом.

3. За счет перебора узел не может посылать блоки очень часто, что защищает сеть блокчейн от "наводнения" (*flooding*) фальшивыми блоками. Как было сказано выше, проверка выполненной работы осуществляется очень быстро, поэтому если блок приходящий в сеть не отвечает условиям проверки, то узлы начинают его игнорировать и перестают пересылать дальше соседним узлам.

В блокчейн Ethereum для доказательства используется алгоритм *Ethash* (`https://github.com/ethereum/wiki/wiki/Ethash`), взявший свое начало от алгоритма *Dagger-Hashimoto*. Основным свойством данного алгоритма является то, что он не столько требователен к вычислительной мощности, сколько к объему памяти: узел, желающий выпускать новые блоки с высокой частотой, так чтобы конкурировать с другими узлами также занимающимися майнингом, должен иметь не меньше 2 гигабайт оперативной памяти.

Данное свойство делает неоправданно дорогим выпуск и продажу ASIC (*Application-specific integrated circuit*, `https://en.wikipedia.org/wiki/Application-specific_integrated_circuit`), что позволяет владельцам относительно недорогого оборудования (например, графических карт) участвовать в выпуске новых блоков, а значит не дает сконцентрировать вычислительные мощности, направленные на майнинг, в одной или нескольких "богатых" организациях, а значит не дает возможность им единолично принимать решения о том какие блоки должны быть выпущены и включены в цепочку блоков.

Алгоритм Ethash полагается на огромный ($\geqslant 2Gb$) массив данных, которые сгенерированы псевдослучайным образом на базе текущего момента времени. Если быть более точным то один и тот же массив данных действителен в течение 30000 блоков, после чего он перестраивается заново для следующих 30000 блоков. Причем изменяется не только содержимое массива, но и его длина: каждую итерацию она растет на $\approx 8Mb$. Время, в течение которого существует один и то же набор данных, называется эпохой.

При формировании массива используется направленный ациклический граф (Directed acyclic graph, DAG, `https://en.wikipedia.org/wiki/Directed_acyclic_graph`):

- Сначала строится небольшой ($\approx 32Mb$) кэш, его наполнение зависит от номера эпохи - он используется для генерации псевдослучайных последовательностей. Кэш состоит из элементов по 64 байта каждый.
- Затем, строится большой массив данных из элементов по 64 байта каждый.

При вычислении каждого элемента используется 257 элементов кэша, которые выбираются псевдослучайным образом, управляемым номером элемента в массиве данных.

В дальнейшем построенный массив данных используется для алгоритма доказательства работы, а именно ищется такая последовательность из 8 байт, называемая *nonce*, которая при объединении с хэшированными данными из заголовка блока и 128 элементами массива данных даст значение определенного свойства. Причем псевдослучайная выборка элементов массива обуславливается как *nonce*, так и данными из заголовка блока (`https://www.vijaypradeep.com/blog/2017-04-28-ethereums-memory-hardness-explained/`). Свойство, которому должно удовлетворять полученное значение, определяется величиной *difficulty*. Значение *difficulty* разное от одного блока к другому и зависит от частоты выпуска блоков таким образом, что позволяет удерживать периодичность появления новых блоков примерно на одном уровне, даже если вычислительная мощность сети возрастает. Изменение *difficulty* позволяет выдерживать примерно одинаковое время задержки между появлением в сети двух соседних блоков, что важно для задания "тактов" синхронизации узлов в сети блокчейн и обеспечивает целостность данных.

В итоге, поскольку майнеру, узлу подготавливающий новый блок, во время перебора *nonce* приходится выполнять операции обращения к элементам массива данных очень много и часто, а скорость обращения к массиву критична для победы в гонке за награду за новый блок, то ему необходимо держать этот массив в памяти постоянно.

Узлы, которые только получают новые блоки и не выпускают новые, могут хранить только кэш. Для того, чтобы проверить, что новый пришедший блок содержит правильное доказательство работы, можно пробежаться по $128 * 257$ элементам кэша и убедиться, что полученное значение соответствует заданному *difficulty*.

Напишите программу, которая бы находила последовательность байт (*mixHash*), получающуюся в ходе процесса выпуска нового блока для заданного *nonce* и *difficulty*.

Формат входных данных

Описание полей заголовка блока в JSON-формате.

Формат выходных данных

Необходимо вывести *mixHash* - последовательность из 32 байт в шестнадцатеричном формате.

Примеры

Пример №1

Стандартный ввод


```
{
    "difficulty": 175807,
    "extraData": "0xd583010702846765746885676f312e39856c696e7578",
    "gasLimit": 4712388,
    "gasUsed": 21772,
    "logsBloom": 0,
    "miner": "0xd6c76b3ca5c36ced890f0fa70d884b42de96d0cb",
    "nonce": "0x5a0ce998ca547aa5",
    "number": 1912,
    "parentHash": "0xfe6ffd6e80e878367f63b4454a8d834b5561242241df76b4dfbfe83e55
177806",
    "receiptsRoot": "0x590e8d789b4dff06cbf8e561d13df39dd23542874cc1b5b296a368b4
f71a7105",
    "sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40
d49347",
    "stateRoot": "0x7b43df1701d2a7709285b75fa9024f3e4db3b1a598fec63eb44cdeee1a3
bc561",
    "timestamp": 1508783678,
    "transactionsRoot": "0xaa639781610199b925119e24d868d32b689eb2676d0e1e4e1dd3
13e95db6e1a6"
}
```


Стандартный вывод


```
5fa4a4dffaa1111990fb02ee453626d24f6f7db9bc467044f127c071bc894119
```


Решение

Для решения этой задачи будем использовать модули `block` и `utils` библиотеки `ethereum`, а также библиотеку `pyethash`. Входные данные в том виде, в котором они даются в задаче, будем считывать из файла 'ethash.txt'.

При помощи конструктора из модуля *block* создадим новый заголовок блока, включив туда значения из входных данных. Это делается для удобного получения заголовка блока, необходимого для майнинга.

Кэш можно получить при помощи функции *mkcache_bytes*, которая на вход принимает номер блока, для которого нужно посчитать кэш, а возвращает набор байт, необходимый для генерации данных.

Так как мы знаем значение nonce, то для получения mixHash мы можем воспользоваться функцией *hashimoto_light*, которая использует кэш для создания только тех значений из датасета, которые необходимы, и таким образом позволяет не генерировать полный набор данных. На вход функция получает номер блока, сгенерированный кэш заголовок блока и nonce, а возвращает значения mixHash и result. В процессе майнинга result используется для определения валидности значения nonce при данном difficulty. Так, result должно быть меньше чем $\frac{2^{256}}{difficulty}$. А значение mixHash используется для проверки на то, что во время процесса майнинга был использован правильный DAG.

Пример программы

Ниже представлено решение на языке Python3

```
1 from ethereum.utils import encode_hex, decode_hex, big_endian_to_int
2 from ethereum.block import BlockHeader
```

```
import pyethash

with open('ethash.txt', 'r') as f:
    desc = eval(f.read())

blockHeader = BlockHeader(decode_hex(desc['parentHash'][2:]),
    decode_hex(desc['sha3Uncles'][2:]),
    desc['miner'][2:], decode_hex(desc['stateRoot'][2:]),
    decode_hex(desc['transactionsRoot'][2:]),
    decode_hex(desc['receiptsRoot'][2:]), desc['logsBloom'],
    desc['difficulty'], desc['number'],
    desc['gasLimit'], desc['gasUsed'], desc['timestamp'],
    decode_hex(desc['extraData'][2:]),
    nonce=desc['nonce'])

b_number = blockHeader.number
b_hash = blockHeader.mining_hash
b_nonce = decode_hex(blockHeader.nonce[2:])
cache = pyethash.mkcache_bytes(b_number)

ans = pyethash.hashimoto_light(b_number, cache, b_hash,
                               big_endian_to_int(b_nonce))[b'mix digest']

print(encode_hex(ans))
```

Задача 6.3.15. Отправка транзакций (3 балла)

Базовой операцией, которая может изменять состояние блокчейн Ethereum, является транзакция. Самое простое действие для изменения состояния - отправка средств с адреса на адрес. Для этого в транзакции указывается адрес получателя и сумма, которая пересылается. Если транзакция отправляется интерактивно через JavaScript консоль или в приложении непосредственно помощью библиотек `web3` для Python или JavaScript, то также указывается адрес отправителя.

Узел, на котором в итоге формируется транзакция для отправки в сеть, пытается получить доступ к приватному ключу отправителя и подписать им транзакцию. Для того, чтобы узел успешно смог выполнить операцию, чаще всего, необходимо предварительно разблокировать соответствующий адрес на нем с использованием пароля доступа к приватному ключу.

Помимо суммы, указываемой в поле ”value”, можно также передавать данные через поле ”data”. Эти данные обычно не используются программным обеспечением кошельков, и поэтому к ним сложно получить доступ - для этого нужно вызывать метод `getTransaction` модуля `eth` в консоли или в приложении, использующем `web3` библиотеку.

Обработка каждой транзакции на узле, включающем их в новый блок (при условии, что потом блок будет принят остальной частью сети Ethereum), потребляет газ. Газ - понятие, введенное в блокчейн Ethereum, чтобы контролировать ресурсы потребляемые виртуальной машиной Ethereum (Ethereum Virtual Machine, EVM). Например, обработка обычной транзакции перевода средств с адреса на адрес потребляет 21000 газа.

Предполагаемое количество газа (поле ”gas”), которое может потратиться в ходе обработки транзакции умножается на цену за единицу газа (”gasPrice”) и вычитается с баланса адреса, отправившего транзакцию. Если выясняется, что у адреса на ба-

лансе не хватает средств для обработки транзакции, то она не включается в блок и не обрабатывается.

Газ тратится не только на обработку информации, но также и на ее хранение, поэтому он потребляется за каждый байт, переданный как ”data” в теле транзакции.

Итоговое количество газа, потраченное в ходе включения транзакции в блок, можно узнать в поле ”gasUsed” при вызове метода `getTransactionReceipt` модуля `eth`.

Отправьте в сеть Ethereum две транзакции с разными данными в поле ”data” и определите, на сколько в одной транзакции потреблялось больше газа, чем в другой.

Формат входных данных

Во входном файле - две строки, определяющие данные, которые должны быть отправлены. Первая строка для первой транзакции, вторая строка - для второй. Данные указаны в шестнадцатеричной системе счисления.

Формат выходных данных

Одно число – разница потребленного газа между двумя транзакциями. При указании ответа нужно использовать модуль числа.

Примеры

Пример №1

Стандартный ввод
0000000022957a0000000000000006e0000b3495300 f10019070000b9bb66b717004e000056d9
Стандартный вывод
236

Решение

Настроим и подключимся к узлу Ethereum по RPC, как это указано в инструкции.

Адреса аккаунтов хранятся в отдельном файле. При этом каждый адрес хранится в шестнадцатеричном представлении с префиксом ’0x’ (Например, 0xf6e1fde85ac7a238cf905e5e8f69a28a34e72904). Web3 может выдавать исключение с описанием, что адрес имеет неверную *EIP* проверочную сумму. Это может происходить из-за того, что буквенные значения адреса, которые должны быть в верхнем регистре, написаны в нижнем. В этом случае имеет смысл явно привести адрес к EIP-55.

Данные, которые необходимо отправить, также должны быть в шестнадцатеричном формате с префиксом ’0x’.

Для отправки транзакции воспользуемся методом *sendTransaction* из модуля `eth` библиотеки `web3`, которая возвращает хэш транзакции.

Осталось найти разницу между затраченным количеством газа, для чего можно воспользоваться описанной в задании функцией *getTransaction*, и вывести результат.

Пример программы

Ниже представлено решение на языке Python3

```
from web3 import Web3, HTTPProvider

web3 = Web3(HTTPProvider('http://localhost:48002'))
eth = web3.eth
personal = web3.personal

accounts = []
f = open('accounts.txt', 'r')
for i in f:
    accounts.append(i.strip())
f.close()

with open('input.txt') as f:
    data1 = '0x' + f.readline().strip()
    data2 = '0x' + f.readline().strip()

personal.unlockAccount(accounts[0])

transaction1 = eth.sendTransaction({'from': accounts[0], 'to': accounts[1],
                                    'value': 1, 'data': data1})
transaction2 = eth.sendTransaction({'from': accounts[0], 'to': accounts[1],
                                    'value': 1, 'data': data2})

print(abs(eth.getTransaction(transaction2)['gas'] - eth.getTransaction(transaction1)['gas']))
```

Задача 6.3.16. Получение баланса (5 баллов)

Транзакции формируют блоки, а затем между двумя соседними блоками образуется связь по принципу ”предыдущий” – ”следующий”. Каждый блок фиксирует набор изменений, выполняемых транзакциями, поэтому блок можно контрольной точкой, определяющей "моментальный снимок"данных в блокчейн.

Таким образом, в памяти блокчейн хранятся все "моментальные снимки"и можно проследить историю изменения любых данных, например, баланса адресов.

Баланс адреса в блокчейн Ethereum измеряется в *wei* - самая маленькая единица измерения, которая составляет 10^{-18} от одного *ether*. Вот список всех единиц измерения:

Единица измерения	Значение
wei	1
kwei	1.000
ada	1.000
mwei	1.000.000
babbage	1.000.000
gwei	1.000.000.000
shannon	1.000.000.000
szabo	1.000.000.000.000
finney	1.000.000.000.000.000
ether	1.000.000.000.000.000.000

Когда отправляется транзакция для перечисления средств с одного счета на другой, то сумма для перевода указывается в *wei*. Это достаточно не удобно для человека. Но для виртуальной машины Ethereum так проще - не нужно иметь дело с дробными значениями, и, следовательно, с проблемами округления.

Напишите программу, которая бы позволяла бы получать баланс адресов на момент выпуска определенного блока.

Формат входных данных

В первой строке входного файла - одно число - номер блока, после обработки которого сформировался новый баланс на адресе.

Вторая строка - единицы измерения (ether, finney, shannon, ..., wei), в которых нужно предоставить ответ.

Дальше идет 20 строк, в каждой из которых - Ethereum адрес в тестовой сети Ropsten, записанный в шестнадцатеричном виде. Длина адреса - 40 символов.

Формат выходных данных

Одно целое число – максимальный баланс среди адресов, указанных во входных данных, сформировавшийся в World State тестовой сети Ethereum (Ropsten) после выпуска соответствующего блока. При переводе из одних единиц измерения в другие округление нужно производить отбрасыванием дробной части. Баланс сравнивать до перевода единиц измерения.

Примеры

Пример №1

Стандартный ввод
2173756 gwei 0xcabcb63c7dccc50c8bd715ebd6ddd9c3cc6055ab 0x20b094933ae12b443093bae4c948ab4976064fbb 0x90d0e1a5ca91a6c4c2390ce996e74c3d711c7f0e 0xf5a997a020b80cbbe4abacbe303628a19cc7df54 0xc9bbc295ec70274ebf96e886cea78998672fb04f 0x22d519accb88f4a30589b89baeb19d58419d682b 0x00f605655ab76cd3e195811234dd1b4b2b9b393b 0x54697fb47b8328de48a78e7b0136fb2b7a8fffcf 0x48f15f9bba01df85a2ab73891e603ae7c0f778ea 0x05b515ef047376ad74a6c00e8cf14b9c977b4f9d 0x159881171ae3cba81fecc23f724c0ba018cdfff7 0x11b52232b64ac629a451e8167c9a48299d299014 0x3ee3e6e5f01206eea239d74ba165719d940b84b4 0x24c29cfe4d95661e0178b340f1f0e60d6bccce89 0x4f59a86775e89f4d504d5393ab03ece01d583ee1 0x270aa72779584fd0b4f645c348b39af4a1492f2c 0x39847a89c6da9a625b3e4da9a1bae1544c3c93ee 0x1ad895f8fa2578590257b45216b0cffca912e2dc 0x4454bc5e7609bb4b2860711ae34530bb0a1f41a3 0x496f829701f68869d806e6fa6b458af132847c82

Стандартный вывод
6177000000

Решение

Так как для решения задачи не нужен аккаунт в тестовой сети, мы можем подключиться к сети Ropsten по удаленной ноде расположенной по адресу `https://ropsten.infura.io/`. Это позволит не скачивать всю историю сети, без чего нельзя было бы обойтись, так как для решения задачи необходим доступ к истории изменений балансов, которую не сможет обеспечить "ускоренная" синхронизация узла с блокчейн сетью.

Для решения воспользуемся функцией *getBalance*, находящуюся в модуле *eth*. С её помощью мы можем получить баланс для каждого аккаунта на момент выпуска данного блока. Найдя максимальный из них, можно получить ответ, который необходимо преобразовать в нужную единицу измерения.

В примере, данном ниже, входные данные считываются из файла и представлены в том виде, в котором даются во входных данных. Функция *get_max* находит максимальный баланс, а функция *convert* преобразовывает его в ответ.

Пример программы

Ниже представлено решение на языке Python3

```
1  def get_max(addr: list, block_n: int):
2      maxeth = -1
3      for i in addr:
```

```
        if eth.getBalance(i, block_n) > maxeth:
            maxeth = eth.getBalance(i, block_n)
    return maxeth

def convert(unit: str, bal):
    if unit == 'wei':
        return bal
    if unit == 'kwei' or unit == 'ada':
        return int(bal / 10**3)
    if unit == 'mwei' or unit == 'babbage':
        return int(bal / 10**6)
    if unit == 'gwei' or unit == 'shannon':
        return int(bal / 10**9)
    if unit == 'szabo':
        return int(bal / 10**12)
    if unit == 'finney':
        return int(bal / 10**15)
    return int(bal / 10**18)

from web3 import Web3

web3 = Web3(Web3.HTTPProvider('https://ropsten.infura.io/'))
eth = web3.eth

n = 20
addr = []
f = open('input.txt', 'r')
block_n = int(f.readline().strip())
unit = f.readline().strip()
for i in f.readlines():
    addr.append(web3.toChecksumAddress(i.strip()))

ans = convert(unit, get_max(addr, block_n))
print(ans)
```

Задача 6.3.17. Определение адреса контракта (2 балла)

Поскольку блокчейн с технологической точки зрения - это децентрализованная база данных, то как и большинство современных баз данных она позволяет хранить не только данные, но и исполняемый код.

Изначально подразумевалось, что исполняемый код в блокчейн должен регулировать денежные потоки, так как основная цель блокчейн, как базы данных, подразумевалась для хранения записей о перемещении условных единиц, эквивалентных деньгам. Например, код мог определять появление какой-то суммы на балансе счета, предназначенного для накопления полученной предприятием прибыли, после чего он автоматически мог бы делить данную прибыль между акционерами данного предприятия. Это было бы похоже на то, как исполняются обязательства по договорам (англ. *contracts*). Именно поэтому код, управляющий денежными потоками в блокчейн, согласились называть смарт-контрактами.

И блокчейн Bitcoin, и блокчейн Ethereum позволяют пользователями заносить в блокчейн свой собственный исполняемый код. Отличия только в том, где этот код в итоге храниться, и насколько универсальные алгоритмы могут быть реализованы с использованием тех команд, которые доступны для программирования.

В блокчейн Ethereum набор доступных команд является полным по Тьюрин-

гу (`https://ethclassic.ru/2016/10/21/turing-completeness-reality/`), что позволяет составлять довольно сложные смарт-контракты, в которых будут и циклы, и условные переходы, и процедуры (части программы, выделенные в отдельный блок из-за частого использования). Пользователь Ethereum может подготовить смарт-контракт и, с помощью специальной транзакции, поместить его в блокчейн. Процесс загрузки смарт-контракта называется деплойментом (*deployment*). По-русски его иногда называют регистрацией.

Транзакция для регистрации смарт-контракта отправляется без адреса получателя. В поле данных (*data*) у такой транзакции находится набор байт - скомпилированный в виде байт-кода смарт-контракт. При этом байт-код состоит из двух частей: *заголовка* и *байт-кода для хранения*:

- В заголовке хранится набор инструкций, которые выполнятся только один раз при регистрации контракта. Здесь может быть изначальная инициализация каких-то переменных, выполнение "конструктора" контракта.
- В байт-коде для хранения располагается тело самого смарт-контракта, которое содержит инструкции, исполняющиеся во время дальнейших обращений к этому смарт-контракту.

Дальнейшая судьба транзакции, регистрирующей смарт-контракт такова:

1. Транзакция включается в новый блок одним из майнеров сети Ethereum. При этом майнер
   (a) Инициализирует новое хранилище (*storage*), в котором будут храниться данные смарт-контракта (глобальные переменные, массивы, словари и т.п.). Данное хранилище выстраивается с использованием структуры данных Patricia Merkle Tree, поэтому оно называется *Storage tree*.
   (b) Выполняет все действия из заголовка байт-кода, что приводит к начальному заполнению *Storage tree*.
   (c) Формирует у в своей копии дерева состояния блокчейна (*World State Tree*) новую вершину, которая используется для хранения данных о смарт-контракте: *(balance, nonce, bytecode, storage tree root)*, где *balance* - сумма на счету контракта, по умолчанию - ноль, но может быть другая, если в траназкции, регистрируеющей контракт, выставлено значение *value*; *nonce* - количество дочерних смарт-контрактов, который данный контракт создал, если в ходе инициализации смарт-контракт создает другие контракты, то после регистрации здесь возможно значение отличное от нуля; *bytecode* - байт-код для хранения; *storage tree root* - хэш - корень *Storage Tree*, позволяющий контролировать синхронизацию изменений данных смарт-контракта на всех узлах в сети блокчейн.

   С вершиной *World State Tree* ассоциируется адрес, который называется адресом смарт-контракта. Поскольку, у смарт-контракта нет собственного приватного ключа, для назначения адреса смарт-контракта используется адрес аккунта, который отправляет транзакцию на регистрацию в сеть, а также номер этой транзакции относительно других транзакций, отправленных с этого адреса: $right(keccak256(rlp([account, nonce])), 20)$.

   Из описанного выше видно, что регистрация контракта изменяет дерево состояния блокчейн (*World State Tree*), следовательно измениться его *State*

Root, который будет отражен в новом блоке.

2. Блок распространяется по всем узлам сети.
3. При получении нового блока узел сети обрабатывает транзакцию на регистрацию смарт-контракта ровно таким же образом, как это делал майнер, включивший ее в блок.

   При этом в узле измениться сначала *Storage Root*, а соответственно и *State Root*. Вычисленный *State Root* должен совпадать с соответствующим полем в принятом блоке.

Таким образом, можно отметить, что несмотря на то, что код смарт-контракта хранится изначально в транзакции, как часть процесса майнинга блока он перемещается в *World State Tree* сначала только на одном узле, а потом на всех устройствах в сети блокчейн. Также выполнение инициализации (заголовка байт-кода) происходит на всех узлах сети блокчейн, что обеспечивает децентрализованное хранение не только кода, но и данных смарт-контракта.

Напишите программу, которая бы определяла бы адрес смарт-контракта по приватному ключу отправителя транзакции на регистрацию данного смарт-контракта.

Формат входных данных

Первая строка входного файла определяет приватный ключ, записанный в шестнадцатеричном формате. Длина строки - 64 символа.

Остальные строки в файле - текст смарт-контракта на языке Solidity, который необходимо сохранить в **приватной сети** блокчейн Ethereum.

Формат выходных данных

Одна строка - адрес смарт-контракта в блокчейн Ethereum.

Примеры

Пример №1

Стандартный ввод

```
227afa8360dacd29e170f0da77ea730e432aec597231d0e96d77360a126205b2
//Code of the smart contract starts below
contract CheckHashing {
    function check(uint8 n) view public returns(uint8) {
        bytes32 s = keccak256(msg.sender, block.number, n);

        return uint8(s[1]);
    }
}
```

Стандартный вывод

```
228343b58a53e4d0ec674e7aeb719f2920c8e291
```

Комментарии

Для решения задачи нужно использовать только что созданную приватную сеть блокчейн Ethereum. Т.е. не должно быть транзакций, отправленных с адреса, формируемого приватным ключом, передающимся в качестве входных данных.

Решение

Заметим, что для получения адреса контракта нам достаточно знать только адрес отправителя и количество транзакций, проведенных с этого адреса.

Так как по условию задачи с адреса, с которого отправляется смарт-контракт, на момент отправки не должно быть ни одной транзакции, можно заключить, что он находится в начальном состоянии(initial state). В момент создания аккаунта его начальное состояние устанавливается равным 1, а nonce вычисляется как разница между состоянием аккаунта и 1.

Таким образом, для того, чтобы узнать адрес смарт-контракта, нам необходимо использовать метод, описанный в условии задачи (*right(keccak256(rlp([account, nonce])), 20)*), где nonce устанавливается равным 0 (state - 1), а адрес равен адресу, полученному из приватного ключа.

Пример программы

Ниже представлено решение на языке Python3

```
from sha3 import keccak_256
from py_ecc.secp256k1 import secp256k1
import rlp

def privToPub(priv):
    priv = int(priv, 16).to_bytes(32, 'big')
    res = secp256k1.privtopub(priv)
    x = res[0].to_bytes(32, 'big')
    y = res[1].to_bytes(32, 'big')
    return x+y

def pubToAddr(pub):
    return keccak_256(pub).hexdigest()[24:]

def privToAddr(priv):
    return pubToAddr(privToPub(priv))

with open('input.txt') as f:
    priv = f.readline().strip()

sender_addr = privToAddr(priv)
contract_addr = pubToAddr(rlp.encode([int(sender_addr, 16), 0]))
print(contract_addr)
```

Задача 6.3.18. Вызов метода контракта (3 балла)

Для исполнения кода смарт-контракта используется виртуальная машина Ethereum (*Ethereum Virtual Machine*, EVM). EVM является частью практически любого клиента сети Ethereum. Виртуальной ее называют, поскольку она реализована на уровне программного обеспечения, что позволяет абстрагироваться от аппаратного уровня

компьютера, а значит гарантирует, что на каком бы реальном железе ни запускался бы код смарт-контракта, результат выполнения будет один и тот же без необходимости перекомпилировать код (`https://en.wikipedia.org/wiki/Virtual_machine`).

Когда происходит запуск смарт-контракта на выполнение, в память EVM передается байт-код данного контракта, а также обеспечивается доступ к его хранилищу (*storage*). Определенные наборы байт в байт-кода интерпретируются в команды виртуальной машины и операнды к этим командам.

Пример части смарт-контракта, представленного в виде команд и операндов:

```
SHA3
SWAP4
POP
PUSH1 0x35
PUSH1 0x01
DUP7
PUSH2 0xffff
AND
MULMOD
```

Если посмотреть спецификацию Ethereum, то можно увидеть, что каждая команда EVM потребляет какое-то количество ресурсов, которые было принято измерять с использованием понятия "газ" (*gas*). Газ - это обобщенный подход к потреблению ресурсов, выражающий одновременно потребление процессорного времени и памяти. Концепция газа была введена, чтобы решить несколько проблем:

- Поскольку подразумевается, что смарт-контракты будут выполнять майнеры во время выпуска новых блоков, то несмотря на различные возможные аппаратные платформы у майнеров, необходимо ввести какой-то универсальный механизм измерения ресурсов, потраченных на выполнение смарт-контракта. Таким образом, заказчик выполнения смарт-контракта (отправитель транзакции) сможет построить свои ожидания относительно того, в каком объеме работа оборудования майнера должна быть компенсирована.
- Ограниченные вычислительные ресурсы майнера требуют возможности оценки сколько выполнение смарт-контракта может занять еще до того, как смарт-контракт будет запущен на выполнение. Благодаря тому, что отправитель транзакции указывает максимальное количество газа (*gas, сокр. от gasLimit*), которое он готов оплатить, и которое, в свою очередь, характеризует количество газа потребляемое смарт-контрактом, майнер может решать - возьмется он за выполнение данного смарт-контракта

Разные команды потребляют разное количество газа. Изначально, потребление газа в командах было ассоциировано с предположением того, каких ресурсов будет требовать от майнера выполнение этой команды - чем больше ресурсов для вычисления или хранения нужно, тем больше газа будет потребляться командой. Так, например, команды сохранения данных в хранилище смарт-контракта и команда создания нового смарт-контракта являются самыми тяжеловесными с этой точки зрения, поскольку требуют от майнера выделения дополнительной памяти, а, следовательно, дополнительных ресурсов на долговременное хранение этой памяти.

Запуск смарт-контракта, который уже зарегистрирован в блокчейн может выполняться несколькими способами:

- Если в ходе выполнения код смарт-контракта будет менять данные в *storage*, то тогда требуется отслеживание данных изменений в *World State* всеми узлами сети блокчейн. Это возможно только через посылку транзакции и последующим ее включением в новый блок.
- Если код контракта будет только что-то вычислять, базируясь на переданных ему параметрах или данных из *storage*, или, иными словами, не меняет *World State* то в этом случае можно использовать локальный вызов смарт-контракта. Данный способ не требует оплаты потребленного газа, но требует доступ к web3 модулю на узле Ethereum.

И в том и в другом случае, в итоге, произойдет загрузка целиком всего байт-кода контракта в виртуальную машину, после чего EVM начнет его исполнение. При этом важной деталью является то, что поле *data* транзакции, вызывающей код контракта, передается на вход коду, загруженному в EVM.

Большинство современных смарт-контрактов в блокчейн Ethereum написано на языке Solidity - высокоуровневом языке программирования. Байт-код для EVM получается в результате компиляции исходного кода на Solidity. Процесс компиляции автоматически подготавливает набор команд для выполнения EVM способный обработать входные данные, пришедшие через поле *data* транзакции. В частности, за счет этого набора команд реализуется возможность иметь в смарт-контракте несколько точек вызова - отдельных методов (функций) одного и того же контракта. Пошаговый анализ исполнения контракта виртуальной машиной показывает, что в первые команды кода по-факту определяют к какой из точек вызова должно перейти управление.

EVM ожидает, что описание точки вызова находится в первых четырех байтах поля *data* транзакции, отправленной для запуска смарт-контракта. Код смарт-контракта начинает сравнение этих четырех байт с теми, что у него прописаны. Если найдено совпадение, то происходит переход к соответствующей части кода:

Псевдокод, описывающий определение точки вызова в смарт-контракте

```
селектор = извлечь первые четыре байта поля data транзакции;
если (селектор == селектор_1) тогда
    переход к метка_1;
иначе если (селектор == селектор_2) тогда
    переход к метка_2;
. . .
иначе если (селектор == селектор_N) тогда
    переход к метка_N;
иначе
    вызвать исключение;
конец если;
метка_1:
    <<команды первого метода контракта>>;
метка_2:
    <<команды второго метода контракта>>;
    . . .
метка_N:
    <<команды N-го метода контракта>>;
```

Чаще всего, первые четыре байта поля *data* транзакции, вызывающей код смарт-

контракта, генерируются автоматически, на основании установленного стандарта. Для этого берется имя метода контракта (функции), который нужно вызвать, а также типы и последовательность аргументов, которые передаются в нанный метод. Эта информация представляется в виде строки, и от нее берется хэш $keccak256$. Например, для функции "get_balance", которая принимает Ethereum адрес и номер блока, преобразование будет выглядеть следующим образом:

$$desc = keccak256('get\_balance(address, uint32)')[:4]$$

Имя функции и аргументов, берутся из исходного кода контракта. Но есть ситуации, когда не всегда целесообразно использовать код целиком или когда вообще нет возможности его использовать. Вместо этого используется описание интерфейсов смарт-контракта (*Application Binary Interface*, ABI), представляющий из себя JSON-структуру, где перечислены те сущности контракта на языке Solidity, к которым можно обращаться в конкретном смарт-контракте.

Пример ABI - описания интерфейса смарт-контракта

```
[
    {
        "constant": false,
        "inputs": [
            {
                "name": "n",
                "type": "uint16"
            }
        ],
        "name": "checkin",
        "outputs": [],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    }
]
```

Большинство современных приложений, предоставляющих графический интерфейс пользователя для операций с блокчейн Ethereum умеют генерировать ABI из исходного кода смарт-контракта. А практически любая программная библиотека, работающая с Ethereum, позволяет с использованием известного ABI отправлять транзакции для вызова методов смарт-контракта.

Напишите программу, которая бы позволяла вызвать описанный выше метод *checkin* у смарт-контракта, расположенного по заданному адресу.

Формат входных данных

Первая строка входного файла определяет Ethereum адрес в **тестовой сети** Ropsten, по которому располагается смарт-контракт. Адрес контракта записан в шестнадцатеричном формате. Длина строки - 40 символов.

Вторая - целое число N ($1 \leqslant N \leqslant 198$), которое нужно будет передать в метод *checkin* смарт-контракта.

Формат выходных данных

Одна строка - количество газа, которое потратилось на обработку транзакции.

Примеры

Пример №1

Стандартный ввод
a623ea48a1a6f7257f24dfd59698773d5ca354c9 162
Стандартный вывод
79441

Комментарии

Для решения задачи нужно использовать тестовую сеть Ropsten блокчейн Ethereum.

Решение

Для решения задачи воспользуемся модулем `eth.contract` из библиотеки `web3`.

Функция *estimateGas* высчитывает и возвращает количество газа, которое потребовалось бы для вызова метода данного контракта. Таким образом, мы можем посчитать необходимое количество газа без необходимости создавать аккаунт в сети. Это дает возможность подключаться к тестовой сети Ropsten, не создавая узла на локальной машине.

ABI-описание интерфейса метода *checkin*, которое можно найти в условии к задаче, для удобства хранится в отдельном файле.

Все, что необходимо - это создать экземпляр контракта с данным ABI и вывести количество газа, которое потребуется на обработку транзакции.

Пример программы

Ниже представлено решение на языке Python3

```
from web3 import Web3, HTTPProvider

web3 = Web3(Web3.HTTPProvider('https://ropsten.infura.io/'))

eth = web3.eth

# Считывание ABI метода 'checkin'
with open('scABI.txt', 'r') as f:
    scABI = f.read()

addr = input()
v = int(input())

scObj = web3.eth.contract(web3.toChecksumAddress(addr), abi=scABI)
print(scObj.functions.checkin(v).estimateGas())
```

Задача 6.3.19. Исполнение смарт-контрактов (4 балла)

Дан смарт-контракт:

```
pragma solidity ^0.4.17;

contract A {
    mapping (address => bool) checked;
    uint16 P = <some prime number>;

    function randomize(uint16 n) internal pure returns(uint8) {
        return uint8(mulmod(n, 1, 2));
    }

    function is_required(uint16 n) internal view returns(uint16) {
        return uint16(P - (n * addmod(n, randomize(n), P)));
    }

    function hash(uint256 message) internal pure returns (bytes32) {
        bytes memory prefix = "\x19Ethereum Signed Message:\n";
        return keccak256(prefix, '32', message);
    }

    function verify(uint16 n, uint8 v, bytes32 r, bytes32 s) public {
        require(is_required(n) == 0);

        bytes32 h = hash(uint256(n));

        address retAddr = ecrecover(h, v, r, s);

        assert(retAddr == msg.sender);

        checked[msg.sender] = true;
    }
}
```

Необходимо найти такие параметры для функции `verify`, при которых функция выполнится полностью. Т.е. словарь `checked` для отправителя транзакции запишется "true".

Содержимое смарт-контракта (поле контракта `P`) может изменяться для разных наборов входных данных.

Формат входных данных

Первая строка входного файла определяет приватный ключ, записанный в шестнадцатеричном формате. Длина строки - 64 символа.

В следующих строках идет код смарт-контракта, основанного на том, что представлен выше.

Формат выходных данных

В выходном файле должно быть три строки. В каждой из них содержатся r, s, v компоненты цифровой подписи, записанные в шестнадцатеричном виде, без префикса '0x'.

Первая строка длиной 64 символа - r. Вторая строка длиной 64 символа - s. Третья строка длиной 2 символа - v.

Примеры

Пример №1

Стандартный ввод

```
b10dbc23cce6ea8b29e791346c161c3c17fc24db7f4295eb8a96494d0f8fbfec
//Code of smart contract starts below
pragma solidity ^0.4.17;

contract A {
    mapping (address => bool) checked;
    uint16 P = 52837;

    function randomize(uint16 n) internal pure returns(uint8) {
        return uint8(mulmod(n, 1, 2));
    }

    function is_required(uint16 n) internal view returns(uint16) {
        return uint16(P - (n * addmod(n, randomize(n), P)));
    }

    function hash(uint256 message) internal pure returns (bytes32) {
        bytes memory prefix = "\x19Ethereum Signed Message:\n";
        return keccak256(prefix, '32', message);
    }

    function verify(uint16 n, uint8 v, bytes32 r, bytes32 s) public {
        require(is_required(n) == 0);

        bytes32 h = hash(uint256(n));

        address retAddr = ecrecover(h, v, r, s);

        assert(retAddr == msg.sender);

        checked[msg.sender] = true;
    }
}
```

Стандартный вывод

```
b215d1f3539a85437a194908c423c6a87d3f832bcf811dca9d41cf7b375af63e
71707da3071ac1b1af495d09bf1472545ce378565881a9d0f22cd3852864119c
27
```

Комментарии

Для решения задачи нужно использовать тестовую сеть Ropsten блокчейн Ethereum.

Решение

Для решения задачи воспользуемся модулями `keccak_256` из библиотеки `sha3` и `ecdsa_raw_sign` из библиотеки `py_ecc`.

Для того чтобы найти, какие значения должно выдавать наше решения, необходимо для начала понять, что же делает данный смарт-контракт и найти условия, при которых он полностью выполнится. Для этого попробуем разобраться с каждой функцией по отдельности.

1. Сначала разберемся с переменной *checked*. Согласно документации Solidity, mapping - это динамическая структура данных похожая на хэш-таблицу. То есть она ставит в соответствие некоторому ключу ячейку в памяти, где записано значение этого ключа. В нашем случае это соответствие булевой переменной для каждого адреса.
2. Переменная P имеет формат unsigned int размером в 16 бит. Она может может принимать значения в промежутке от 0 до 65535. В дальнейшем, когда в решении к этой задаче будет написано n-битное число, будет подразумеваться число, не имеющее знака, то есть всегда положительное, так как в данном контракте присутствуют только такие числа. Важно понимать, что будет происходить со значением переменной P при её переполнении, то есть если ей будет присвоено значение, выходящее за рамки данного диапазона. В этом случае лишние биты будут обрезаться. Реализовать это можно при помощи операции сравнения по модулю, в случае с 16-битным числом - по модулю 65536.
3. Функция *randomize* принимает на вход 16-битное число n, а возвращает 8-битное, полученное как остаток по модулю 2 от умножения n на 1. Легко заметить, что результат всегда будет равен либо 0, либо 1, в зависимости от того, чётное число подавалось на вход или нечётное. Поэтому при обрезании до 8 бит информация теряться не будет
4. Как мы увидим позднее, функция *is_required* является одной из самых главных для того, чтобы контракт полностью выполнился. Она принимает и возвращает 16-битное число n. Функция *addmod*, вызываемая внутри, возвращает остаток от деления суммы числа n и результата функции *randomize(n)* на число P. В большинстве случаев результат выражения в скобке будет больше числа P, поэтому велика вероятность, что при вычитании произойдет так называемый *underflow*, то есть, когда результат будет меньше минимально возможного. Однако и в этом случае нахождение остатка от деления по модулю будет давать правильный результат.
5. Для имплементации функции *hash* необходимо посмотреть как работает функция keccak256 на языке solidity. При её вызове для нескольких аргументов данные аргументы кодируются минимально необходимым количеством байт и конкатенируются. И уже к полученному набору байт применяется keccak256.
6. Главное назначение функции *verify* - проверка соответствия адреса отправителя значениям v, r и s, полученным при подписи документа.

   Функция *ecrecover* возвращает адрес пользователя, подписавшего документ. Данный адрес должен совпадать с адресом пользователя, обратившегося к контракту, иначе будет вызван метод *assert*, который откатит все изменения к предыдущему состоянию и израсходует весь предоставленный газ.

   Метод *recovery* похож на метод *assert* с той лишь разницей, что он не потребляет весь оставшийся газ, а просто откатывает состояние контракта и завершает выполнение функции.

   Таким образом, для выполнения вызванного метода значение n необходимо подобрать так, чтобы функция *is_required* от этой переменной равнялось нулю. А значения v, r и s должны быть подобраны так, чтобы адрес, полученный при помощи n, совпадал с адресом полученным из приватного ключа отправителя. Если все условия выполняются, то в соответствие данному ад-

ресу в переменной *checked* устанавливается значение true, и функцию *verify* можно считать полностью выполненной.

Для начала необходимо подобрать число n, чтобы выполнялся первый require. Для этого нужно перебрать все числа на промежутке от 0 до 65535 и выбрать то из них, которое даёт 0 при вызове функции *is_required*. На самом деле, возможны несколько таких значений, но для примера, данного в задании, подходит первое из них, соответственно будем брать также первое подходящее число n.

Значения v, r, s будем получать при помощи функции `ecdsa_raw_sign`, которая принимает на вход два набора байт: закодированные данные и приватный ключ пользователя, - а возвращает искомые значения в целых числах. Полученный ответ перед выводом необходимо перевести в шестнадцатеричный вид и привести к нужной длине.

Пример программы

Ниже представлено решение на языке Python3

```
from sha3 import keccak_256
from py_ecc.secp256k1 import ecdsa_raw_sign

def randomize(n):
    return (n*1)%2

def is_required(n, P):
    Mod = 65536
    return (P - (n * (((n + randomize(n)) % P)))) % Mod

def zpad(s, l):
    return '0' * (l-len(s)) + s

P = int(input())
priv = input()

i = 0
while is_required(i, P) != 0:
    i+=1
n = i

prefix = "\x19Ethereum Signed Message:\n".encode()
number = '32'.encode()
message = n.to_bytes(32, byteorder='big')
hash = prefix+number+message
h = keccak_256(hash).digest()

priv = int(priv, 16).to_bytes(32, 'big')
ans = ecdsa_raw_sign(h, priv)

r = zpad(hex(ans[1])[2:], 64)
s = zpad(hex(ans[2])[2:], 64)
v = zpad(str(ans[0]), 2)

print(r, s, v, sep = '\n')
```

Задача 6.3.20. Получение баланса ERC-20 token (5 баллов)

Несмотря на относительно небольшую историю существования технологии смарт-контрактов Ethereum, на текущий момент уже наработано большое число шаблонных контрактов.

Одним из самых известных шаблонов является контракт ERC-20 (`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md`), реализующий функционал, так называемого, токена. Использование измененного шаблона контракта «ERC-20 token» настолько распространено, что поэтому сложно дать точное определение слову токен.

Вот несколько примеров использования токена:

- токены, как внутренняя валюта. т.е. накапливая токены, я могу обменять их на товар, даже если товаром является валюта (подобным образом работают системы накопленных бонусов в магазинах);
- токены, как билеты. т.е. я могу обменять токен на какой-то сервис;
- токен, как акция. т.е. я могу получить часть прибыли при предъявлении токена;
- токен, как право участия. т.е. я могу принимать участие в решении каких-то вопросов, если я владею токеном.

В общем случае, получается, что токен - некая условная единица, которая может быть накоплена на счетах владельцев токенов. При этом должен предоставляться функционал получения информации о наличии и количестве токенов на определенном счету, а владельцы токенов имеют возможность передавать токены друг другу.

Стандарт «ERC-20 token» описывает несколько основных методов, которые должен предоставлять контракт, реализующий токен.

- *totalSupply* - возвращает общее количество токенов находящихся в обращении.
- *balanceOf* - возвращает количество токенов, находящихся во владении у определенного аккаунта.
- *transfer* - если транзакция с вызовом этого метода вызвана со определенного аккаунта, то данный аккаунт может сделать владельцем части своих токенов другой аккаунт. При этом сгенерируется событие *Transfer*, которое позволит отслеживать внешними сервисами списания и начисления токенов.

Один из алгоритмов обращения с токенами, используя методы, перечисленные выше, может выглядеть следующим образом:

1. Компания-владелец социальной сети в сети Интернет решает использовать токены для предоставления дополнительных услуг своим пользователям.
2. При инициализации токена на определенном выставляется, сколько токенов выпущено сразу после регистрации контракта.
3. Потенциальные пользователи токена перечисляют на счет смарт-контракта токена средства, *fallback* функция (см. документацию по Solidity) определяет количество перечисленных пользователем средств и ассоциирует с адресом пользователя определенное число токенов. Сумма полученных средств идет на разработку и поддержку функционала дополнительных услуг на сайте.

4. Позднее, пользователи могут передавать часть своих токенов сайту для получения доступа к дополнительным услугам.
5. Также пользователи могут передавать/дарить/обменивать токены друг другу. Таким образом, могут возникать экономические отношения с использованием токена, независящие от основной валюты блокчейна: ценность токена может изменяться в зависимости от качества и востребованности услуг сайта. Что при росте ценности позволит получать экономическую выгоду пользователям, купившим токен на ранних стадиях развития услуги.

Поскольку основными данными любого токена является информация о распределении токенов между его пользователями, практически любое современное программное обеспечение, выполняющее функции кошелька умеет получать эту информацию.

Подразумевая, что в сети Ropsten есть смарт-контракт, основанного на шаблоне *StandardToken* (`https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/token/StandardToken.sol`) таким образом, что никаких новых полей не было добавлено в контракт, напишите программу, которая бы получала количество токенов для заданных аккаунтов, сформированных после подтверждения блокчейн сетью определенного блока.

Формат входных данных

Первая строка входного файла определяет Ethereum адрес в **тестовой сети** Ropsten, по которому располагается смарт-контракт ERC-20 token. Адрес контракта записан в шестнадцатеричном формате. Длина строки - 40 символов.

Во второй строке целое число - номер блока, после обработки которого сформировались новое распределение токенов в смарт-контракте ERC-20 token.

Дальше идет 9 строк, в каждой из которых - Ethereum адрес владельца ERC-20 token, записанный в шестнадцатиричном виде. Длина адреса - 40 символов.

Формат выходных данных

Одно целое число – среднее медианное значение количества токенов среди адресов, указанных во входных данных. Необходимо работать с балансами сформировавшимися в Storage ERC-20 token после выпуска соответствующего блока.

Примеры

Пример №1

Стандартный ввод
63945c28ea8e15eabdf58d668e91c8518c710347 2218311 e09ba0ef60e3ec006071cc92caa1d43947fb25ff 4f59a86775e89f4d504d5393ab03ece01d583ee1 496f829701f68869d806e6fa6b458af132847c82 05b515ef047376ad74a6c00e8cf14b9c977b4f9d 59009daaefded4fd404ca7f0a484aac99cb2cfcd 84374ae93b30670fd657fc742923e10e35781efc 796a0c211aa2588f7bef4ecf94f05ccf3065ee91 90d0e1a5ca91a6c4c2390ce996e74c3d711c7f0e 5634ec3fb89c2667fdccf9a110800b62eb2ce249

Стандартный вывод
38272

Комментарии

Для решения задачи нужно использовать тестовую сеть Ropsten блокчейн Ethereum. Обратите внимание, что для решения задачи будет необходим доступ к истории изменений полей смарт-контракта, которую не сможет обеспечить "ускоренная" синхронизация узла с блокчейн сетью.

Для получения доступа к данным контракта, сформированным после применения определенного блока нужно использовать один из JSON-RPC методов, описанных на странице `https://github.com/ethereum/wiki/wiki/JSON-RPC`.

Решение

Для решения задачи воспользуемся методом `getStorageAt` модуля `web3.eth`. Входные данные будем считывать из файла "smart_contracts_4.txt".

В первом задании на смарт-контракты рассказано, как происходит регистрация смарт-контракта в сети блокчейн. Так, во время регистрации майнер инициализирует новое хранилище, в котором будут храниться данные контракта, или его состояние. Это состояние мы и можем считать при помощи метода *getStorageAt*, если знаем, в каком порядке расположены элементы в этом хранилище.

Все параметры в хранилище имеют размер 256 бит и индексируются с самого начала. Таким образом, мы можем узнать состояние первой объявленной в смарт-контракте переменной, обратившись к ней по индексу 0, второй переменной - по индексу 1 и т.д. Хочется заметить, что индекс должен быть представлен набором из 32 байт.

Однако, когда мы пытаемся прочитать состояние переменной, которая может хранить в себе несколько значений, например, mapping, возникает проблема: состояние всегда равно нулевому набору байт. Происходит это из-за того, что mapping имеет другую систему индексации, и для того, чтобы узнать его значение, необходимо знать значение ключа, состояние которого мы хотим прочитать. Это значение можно найти как результат применения *keccak_256* кодирования к сумме значения ключа и индекса (конкатенации двух наборов из 32 байт).

В смарт-контракте, основанном на шаблоне StandardToken первая переменная хранит в себе общее количество токенов, находящихся в обращении, а вторая - ко-

личество токенов для каждого адреса по отдельности. То есть необходимый индекс для поиска в хранилище равен '1'.

Таким образом, для решения задачи необходимо найти количество токенов для всех адресов, указанных во входных данных, и выбрать из них среднее медианное значение, которое в нашем случае равно количеству токенов для адреса, находящегося по этому количеству на 4 месте.

Пример программы

Ниже представлено решение на языке Python3

```
from web3 import Web3
from ethereum.utils import sha3, encode_int32, zpad, encode_hex, decode_hex

def get_balance_at_contract(c, a, bn):
    object_num = 1
    object_bytes = encode_int32(object_num)
    addr_bytes = zpad(decode_hex(a[2:]), 32)
    maddr = sha3(addr_bytes + object_bytes).hex()
    return web3.toInt(web3.eth.getStorageAt(c, int(maddr, 16), bn))

web3 = Web3(Web3.HTTPProvider('https://ropsten.infura.io/'))
eth = web3.eth

f = open('smart_contracts_4.txt', 'r')

c_address = web3.toChecksumAddress(f.readline().strip())
bn = int(f.readline())
balance = []
for i in f.readlines():
    address = web3.toChecksumAddress(i.strip())
    balance.append(get_balance_at_contract(c_address, address, bn))

print(sorted(balance)[4])
```

Критерии определения призеров и победителей второго этапа

Количество баллов, набранных при решении всех задач суммируется. Призерам второго отборочного этапа было необходимо набрать 20 баллов (для участников из 9 класса) и 30 (для участников из 10-11 класса). Победители второго отборочного этапа должны были набрать 80 баллов и выше.