Домашнее задание 5, Павливского Сергея, 873.

9 б

Решение:

Делаем проверку, что вход корректный (2 символа #). Идем по входной ленте, запоминаем на второй ленте индексы обоих # и конца входной строки (индексов будет не более чем 3, каждый не более чем n, записывать придется $O(\log n)$ символов). Дальше идем одновременно от индексов # и индекса конца строки влево:

$\Rightarrow$ на второй ленте суммируем одновременно-встреченные биты чисел a и b (идущие от первого символа # и второго влево соответственно) и бит переноса, изначально инициализируемый 0;

запоминаем бит переноса (целое от деления суммы на 2), остаток от деления суммы на 2 сравниваем с текущим битом слова c (идущего от конца строки влево);

если остаток суммы и бит числа c равны - возвращаемся в «$\Rightarrow$», иначе выходим из цикла и возвращаем «Нет!».

Если a или b заканчивается, а второе из них не заканчивается, то вместо бита закончившегося слова подставляется 0.

Смещение можно реализовать отдельно храня величину смещения относительно # = смещению относительно конца строки. Оно также логарифмично от длины входа, т. к. по значению не превосходит длины входа.

Алгоритм возвращает «Да!», если на некоторой итерации биты всех слов равны 0 (в этом случае он сразу выходит из цикла).

Память:

Мы храним ограниченное количество чисел запись каждого из которых логарифмична от длины входа, значит и вся требуемая память $O(\log n)$.

Корректность:

Числа сравниваются побитово. Если бит суммы a и b не равен биту c, то и $a+b \equiv c$. Так как в случае различия в битах $a+b$ и c алгоритм сразу терминируется, то если в какой-то момент биты суммы и бит c равны 0, значит мы прошли все биты суммы и биты c, все они

оказались равны, а значит и $a + b = c$.

Что и требовалось

9 в

Решение:

Во-первых, на второй ленте будем хранить сумму переноса (у нас при сложении в столбик в следующий разряд в отличие от сложения 2-х чисел может переноситься не одна единица; в худшем случае при перемножении числа из всех 1 на число из всех 1 длины n_1, у нас в k столбике в перенос уходят $\lfloor \frac{k}{2} \rfloor$ единиц, суммируя по всем n_1 получаем квадратичную ассимптотику от n_1, при умножении на 2 (для учета столбцов, которые левее первого столбца максимальной длины), все еще имеем квадратичную ассимптотику; т. к. n_1 логарифмически зависит от длины входа, то и объем памяти для хранения любого возможного числа единиц переноса логарифмичен). Дальше идем по числу которое меньшее из двух (проверка на неравенство логарифмична по памяти (доказывалось на семинаре); если числа равны, то по любому из них), не теряя общности пусть им оказалось a, пусть длина его двоичной записи $= w$. Заводим счетчик того, какой бит мы сейчас рассматриваем (ограничен сверху $\log w$). Прогоняем данный счетчик от 0 до $\log w$ (индекса последнего бита). Заводим переменную sum. При счетчике равном i проходимся другим циклом по переменной j от 0 до i и sum += a[i - j] · b[j] (просто эмулируем сумму столбца, который возникает при перемножении в столбик. К sum прибавляем сумму переноса. $sum\%2$ идет в записывается в результат (в начало строки результата), $sum/2$ (целая часть) перезаписывает сумму переноса. По аналогии мы можем высчитать и все остальные биты произведения, главное, что ни для какого промежуточного высиления, как было показано выше, нам не требуется более, чем $O(\log n)$ памяти. Тогда язык по определению $\in$ L.

9 ж

Решение:

Длина записи количества вершин логарифмична от количества вершин, т. е. логарифмична от длины входа. Нам известно, что UPATH $\in$ L. Тогда сэмулируем на второй ленте работу МТ для UPATH за

тем исключением, что если МТ для UPATH для получения информации есть ли ребро между вершинами смотрела на входную ленту, то здесь она будет задавать вопрос некоторой $_1$ о том есть ли ребро(по сути, вычисления с оракулом). $_2$ для UCYCLE будет композицией из этих 2-х МТ. Тогда алгоритм таков:

для каждой вершины рассматриваем все пары вида (сосед; ребро); даем указание $_1$ на все вопросы со стороны МТ о наличии ребра отвечать правду, а о рассматриваемом ребре говорить, что его нет; ищем в графе путь из вершины в ее соседа (МТ для UPATH умеет это делать на логарифмической памяти); если хоть для какой-то вершины нашелся путь в ее соседа, то цикл есть, иначе нет.

По памяти:

UPATH распознается на логарифмичной памяти, также нам нужно помнить конечное количество индексов текущей рассматриваемой вершины, соседа и ребра, которое нужно игнорировать. Суммарная память логарифмична по длине входа.

Корректность:

Если в графе есть цикл, то в нем найдется путь из одной вершины в ее соседа не проходящий через соединяющее их ребро (очевидно из определения цикла). Аналогично в обратную сторону. Собственно, эти проверки и делает алгоритм.

10 d)

Решение:

Как неоднократно показывалось в предыдущих задачах, для хранения числа, равного количеству в данном случае вершин графа, требуется логарифмическая память от длины входа. Тогда пусть вершины индексируются в порядке записи на входной ленте. Заведем переменную res, которая будет принимать значения 0 или 1, исходно инициализируется 0. Запустим цикл по переменной i, также хранящейся на логарифмической памяти в силу ограниченностью количеством вершин, от 0 до *количество вершин* - 1 (как истинные программисты индексируем вершины с 0) . Для каждого i пройдемся циклом по переменной j от 0 до $j-1$; если ни для какого j нет ребра с данным i, то $res = (res+1)\%2$. Если после обработки (*количество вершин* - 1)-й вершины, то компонент связности в графе

четное количество, и возвращаем "Да! иначе "Нет!".

По памяти:

Мы стабильно храним индексы i и j, переменную res, возможно еще некоторое, но строго конечное количество переменных для реализации проверки, что нет ребер между j-ми и i-ми вершинами, и каждая ничто из этого не занимает более чем логарифмической памяти по длине входа (для первых 3-х объктов было показано в блоке алгоритма, а для проверки можно просто завести флажок из одного бита, который будет изначально равен 0, и если в ходе проверки для i-й вершины было обнаружено ребро, то принимает значение 1, и больше не меняется). Итого, алгоритм работает на $O(\log n)$ памяти.

Корректность:

res по сути просто флажок, меняющий бинарное значение при нахождении новой компоненты связности (т. е. характеризует четность количества найденных компонент связности) .

Докажем по индукции корректность нахождения этих компонент.

База: для 1 вершины у нас res станет равной 1, т. е. алгоритм обнаружит 1 компоненту связности, что соответствует с действительностью.

Индукционный переход: мы корректно определяем количество компонент связности для первых k вершин. $k+1$-я вершина может :

- являться частью уже рассмотренной компоненты. Но тогда она имеет ребро с одной из уже рассмотренных вершин. Т. е. в случае наличия ребра между $k+1$-й вершиной и одной из первых k вершин количество найденных компонент связности не меняется, res не меняется. Что и делает алгоритм.

- не лежать ни в одной ранее рассмотренной компоненте. Но тогда она не имеет ребра ни с одной из предыдущих вершин. Т. е. в случае отсутствия ребра между $k+1$-й вершиной и хоть одной из k ранее рассмотренных вершин мы нашли новую компоненту связности, res меняет значение, характеризуя тем самым изменение четности количества компонент связности в графе. Что и делает алгоритм.

Что и требовалось

10 г)

Решение:

Аналогично 9 в) за исключением того, что теперь мы сразу говорим $_1$ говорить, что ребра с не существует, и искать путь между вершинами между которыми ребро находится.

10 д)

Решение:

Конъюнкция выполнима на наборе переменных, когда каждая ее скобка обращается в 1 на наборе переменных. xor обращается в 1 т. т. д., когда логические переменные в него входящие имеют разное значение. Будем считать нашу конъюнкцию графом с той точки зрения, что вершины графа соответствуют логическим переменным, и считается что вершины x_i и x_j соединены ребром, если есть скобка $x_i \oplus x_j$. Тогда заметим, что если граф двудолен, то конъюнкция выполнима, потому что тогда переменным из одной доли присвоим значение 1, переменным из другой доли значение 0, и т. к. все ребра есть только между вершинами из разных долей, а каждое ребро соответствует своему xor, то все xor обращаются в 1, т. е. и вся конъюнкция обращается в 1, т. е. выполнима. Если же конъюнкция выполнима, то и граф разбивается на 2 доли по значениям переменных 0 или 1, т. е. двудолен. Тогда выполнимость конъюнкции эквивалентна двудольности соответствующего графа. Тогда, т. к. нам известно, что $\mathrm{BIPARTITE} \in \mathrm{L}$, то проверяем соответствующий конъюнкции граф на двудольность, и результат проверки будет ответом на вопрос о выполнимости конъюнкции.

12

Решение:

Запишем определение класса L :

$A \in \mathrm{L} \leftrightarrow \exists$ двухленточная ДМТ, разрешающая язык за $O(\log n)$ по памяти (от длины входа), такая что на одной ленте ей подается вход, полимномиальный от длины самого себя (спасибо, кэп), на другую идет запись.

Запишем сертификатное определение для требуемого класса при разрешенном движении по s в обе стороны:

$A \in$ *класс* $\leftrightarrow \exists$ двухленточная ДМТ, разрешающая язык за $O(\log n)$

по памяти от длины входа, такая что на отдельной ленте ей подается s, полиномиальная от длины входа, на другую ленту идет запись.

Как можем видеть, определения в точности одинаковые. Значит при разрешенном движении по s в обе стороны, описание становится описанием L.

13в) Сделано при помощи neerc.ifmo

Решение:

Пусть исходная КНФ содержит n переменных $x_1, x_2, ..., x_n$. Построим по нашей 2КНФ ориентированный граф на $2n$ вершинах, содержащий вершины $x_1, ..., x_n, \overline{x_1}, ..., \overline{x_n}$. Заметим, что любой дизъюнкт $a \vee b \equiv (\overline{a} \to b) \wedge (\overline{b} \to a)$. Тогда добавим в наш граф все ребра вида из литерала a в литерал b, такими, что при расписывании некоторого дизънкта 2КНФ описанным выше способом возникнет импликация $a \to b$.

Лемма.

Исходная 2КНФ выполнима тогда и только тогда, когда в графе ни для какой x_i не существует одновременно пути из x_i в $\overline{x_i}$ и из $\overline{x_i}$ в x_i.

Доказательство:

Действительно, если 2КНФ выполнима, то такой пары путей не может существовать, потому что каждое ребро соответствует импликации, а тогда из транзитивности импликации в случае существования обоих путей следовало бы что одновременно $x_i \to \overline{x_i}$ и $\overline{x_i} \to x_i$, что не выполнимо ни для какого значения x_i.

Пусть теперь одновременно не существует пары таких путей. Для определенности пусть нет пути из x в $\overline{x}$. Тогда ни для какого x не существует одновременно пути в y и в $\overline{y}$ (так как в случае пути из x в y, т. е. импликации $x \to y$, следовала бы импликация $\overline{y} \to \overline{x}$, т. е. путь из $\overline{y}$ в $\overline{x}$; если же еще был бы путь из x в $\overline{y}$, то был бы путь из x в $\overline{x}$, которого нет по предположению). Тогда мы можем сделать истинными x и все достижимые из него литералы. Противоречий не будет, т. к. x не соединен с переменной и ее отрицанием одновременно. Если в некотором дизънкте $a \vee b$ литерал a стал ложен, значит литерал $\overline{a}$ соединен с x, а тогда из импликации $\overline{a} \to b$ существует путь из x в b, значит и b истинен, и весь дизъюнкт истинен. Значит после выбора значений переменных таких, чтобы соседние с x литералы

стали истинными, в любом дизюнкте либо один из литералов станет истинным, либо ни один не поменяется. Так, выбирая x с невыбранным для него значением и делая описанные действия, продолжаем пока все дизъюнкты не станут равными 1. Оставшиеся переменные без значения можно выбрать любыми. Выбранный набор значений переменных по построению и будет выполняющим для 2КНФ.

Лемма доказана

Тогда построим 2SAT к PATH: сводящая функция будет строить описанный выше граф. Дальше для каждого x_i будем искать пути $(x_i, \overline{x_i})$, $(\overline{x_i}, x_i)$. Если хоть для какого-то x_i существуют оба пути, то 2SAT невыполнима, иначе выполнима. Можно определить язык IPATH $=$ $\{G|$граф G состоит из n вершин x_i и n вершин $\overline{x_i}$, причем нет одновременно путей $(x_i, \overline{x_i})$, $(\overline{x_i}, x_i)\}$. IPATH $\in$ NL (так как это просто графы со специфической структурой, для распознания которых выполняется 2n вызовов на логарифмической памяти МТ, которая ищет путь для языка PATH). Тогда сводящая функция сводит 2SAT к IPATH, а из задачи 15 следует, что если язык свелся к NL, то и он сам NL.

Что и требовалось

13 г)

Решена в конспекте к семинару.

15

$\cdot A \leqslant_l A$

Сводящая функция на выходящую ленту записывает сам же x, тогда, очевидно, $x \in A \leftrightarrow f(x) = x \in A$.

Что и требовалось

$\cdot A \leqslant_l B$ и $B \leqslant_l C \rightarrow A \leqslant_l C$

Пусть $f_1(x)$ сводит A к B, $f_2(x)$ сводит B к C. Тогда возьмем сводящую функцию $f_3(x) = f_2(f_1(x))$. Так как функции f_1 и f_2 вычисляются последовательно, а каждая на своей области определения вычисляется на логарифмической памяти, то и f_3 будет вычисляться на логарифмической памяти. $x \in A \leftrightarrow f_1(x) \in B, y \in B \leftrightarrow f_2(y) \in C \Rightarrow$
$y = f_1(x) \rightarrow x \in A \leftrightarrow f_2(f_1(x)) = f_3(x) \in C$ из транзитивности эквивалентности. Все условия сводимости выполнены. Единственное, что по определению сводимости мы должны сэмулировать вторую сво-

димость на логарифмической памяти рабочей ленты итоговой МТ. Действия аналогичны описанным в следующем пункте.

Что и требовалось

$\cdot A \leqslant_l B, B \in L \to A \in L.$

По определению l-сводимости мы можем на логарифмической памяти вычислить сводящую функцию от слова из A, потом по определению класса L на логарифмической памяти определить принадлежность $f(x)$ языку B, и эта принадлежность будет эквивалентна принадлежности исходного слова языку A. Все вычисления можно уместить на логарифмическую память на ленте вывода МТ, которая по определению МТ для распознавания L имеет две ленты одну из которых нельзя трогать. Проблему того, что на выход сводящая функция выдает в общем случае не обязательно логаримическое по памяти слово, и мы его не можем целиком тогда зранить, можно решить, если например хранить количество значение длины такого слова (занимает $\log n$ по памяти), и при обращении к конкретной ячейке вызывать сводящую функцию от входного слова, делать вычисления на логарифмической памяти, и выводить только требуемую позицию вычисленного слова, делать в зависимости от ее значения необходимое действие, очищать память по эту позицию, повторять эти шаги до тех пор пока МТ распознающая B не завершит свою работу. Тогда нам необходимо хранить значение длины результата сводящей функции на входном слове, требуемый индекс по которому находится необходимая позиция результата работы сводящей функции, значение самой найденной позиции, некоторая логарифмическая память для работы распознавателя B. Суммарно логарифмическая память.

$\cdot A \leqslant_l B, B \in NL \to A \in NL$

В точности те же рассуждения, что и для предыдущего пункта: на логарифмической памяти можем посчитать сводящую функцию, потом на недетерминированной МТ распознать на логарифмической памяти язык B, тогда и суммарная МТ работает на логарифмической памяти. Обход проблемы с тем, что результат сводимости может не уместиться на логарифмическую память, взять дополнительную ленту мы не можем, а работать с ним надо, описан в предыдущем пункте.

Что и требовалось

19 а) При помощи neerc.ifmo

Принадлежность NL доказана в 13 в).

$\overline{\text{PATH}} \in \text{NLc}$, т. к. по теореме Иммермана $\overline{A} \leqslant_l \text{PATH} \leftrightarrow A \leqslant_l \overline{\text{PATH}}$.

Сведем $\overline{\text{PATH}}$ к 2SAT. Для каждой вершины графа установим свою переменную, а для каждого ребра $a \to b$ дизъюнкт $\overline{a} \vee b$. Добавим дизъюнкты s и $\overline{t}$ (для второго пункта доказательства).

Если в исходном графе отсутствует путь из s в t, то все достижимые из s литералы сделаем истинными (в том числе литерал от переменной s, а все недостижимые (т. е. из которых ребра идут в достижимые из s литералы) сделаем ложными. Тогда все ребра идут либо из истинной вершины в истинную, либо из ложной в ложную , либо из ложной в истинную. Тогда и вся КНФ выполняется.

Если КНФ выполняется, то все литералы, достижимые из s, должны обращаться в 1. Но т. к. для выполнимости КНФ дизъюнкт t равен 0, то из s нет пути в t.

Что и требовалось