Московский физико-технический институт
Физтех-школа прикладной математики и информатики

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ (БИЛЕТЫ К ЭКЗАМЕНУ)

II СЕМЕСТР

Лектор: *Степанов Илья Даниилович*

h\nu

Авторы:
Николай Спицын
Артемий Клячин
Полина Подзорова
Дмитрий Савичев
Полина Чубенко
Ирина Климанова

Даниил Дрябин
Проект на Github

весна 2021

Содержание

18 Операции над множествами (масками): объединение, пересечение, разность. Реализация в программе. Проверка, что одна маска является подмножеством другой. Проверка, что число является степенью двойки. **2**

19 Задача о самом дешёвом гамильтоновом пути: решение за $O(2^n n^2)$ **3**

20 Задача о максимальной клике: решения за $O(2^n n^2)$, $O(2^n n)$, $O(2^n)$ **4**

21 Подсчёт всех значений $b(mask) = \max_{s \subset mask} a(s)$ для данного набора значений $a(0), a(1), ..., a(2^n - 1)$ за $O(2^n n)$ **5**

22 Задача о максимальной клике: решение за $O(2^{\frac{n}{2}} n)$ **6**

23 Симпатичные узоры: количество раскрасок таблицы n × m в два цвета без одноцветных квадратиков 2 × 2. Прямой профиль: решения за $O(4^n(n+m))$ и $O(8^n log m)$ **9**

24 Симпатичные узоры. Изломанный профиль: идея решения за $O(2^n mn)$ **10**

25 Определения неориентированного и ориентированного графов, пути, (вершинно) простого пути, рёберно простого пути. Связь вершинной и рёберной простоты. Определение цикла, рёберно простого цикла, (вершинно) простого цикла. Определение достижимости между вершинами, простота пути. Определение связности. **12**

26 Три способа хранения графа в памяти компьютера, их преимущества и недостатки. **13**

27 Поиск в глубину: алгоритм dfs на ориентированном графе. Лемма о белых путях **14**

28 Поиск в глубину: множество посещаемых вершин, поиск цикла, достижимого из s, проверка на ацикличность **15**

29 Топологическая сортировка ориентированного ациклического графа: определение и алгоритм поиска (с доказательством корректности). **17**

30 Отношение сильной связности между вершинами. Компоненты сильной связности. Сильно связный граф. **17**

31 Алгоритм Косарайю. Корректность и время работы. **18**

32 Конденсация ориентированного графа, ацикличность 19

33 Постановка и решение задачи 2SAT (применение алгоритма выделения компонент сильной связности). 20

34 Алгоритм dfs на неориентированном графе. Дерево обхода dfs. Классификация рёбер на древесные и обратные. Проверка связности и ацикличности. Компоненты связности 21

35 Мосты, точки сочленения. Введение функции ret. Критерий того, что ребро является мостом. 22

36 Насчёт ret в неориентированном графе, нахождение мостов 23

37 Нахождение точек сочленения в неориентированном графе. 24

38 Понятие рёберной двусвязности. Отношение эквивалентности. 26

39 Выделение компонент рёберной двусвязности в неориентированном графе. Древесность графа со сжатыми компонентами рёберной двусвязности. 26

40 Определение эйлерова цикла. Критерий наличия эйлерова цикла в неориентированном графе. 27

41 Реализация алгоритма поиска эйлерова цикла. 28

42 Критерий наличия эйлерова пути в неориентированном графе. 29

43 Критерий наличия эйлерова цикла в ориентированном графе. 29

44 Определение кратчайшего расстояния в невзвешенном/взвешенном графе. 30

45 Поиск в ширину: алгоритм bfs с доказательством корректности. 30

46 Алгоритм $0-K-bfs$. 31

47 Двусторонний bfs 32

48 Алгоритм Дейкстры. Условия применимости, доказательство корректности. Реализации за $O(n^2), O(m \log n)$, $O(m + n \log n)$. 32

49 Двусторонний алгоритм Дейкстры. Завершение алгоритма: почему достаточно реализовать алгоритм, почему нельзя обойтись меньшим числом действий (пример). **33**

50 Алгоритм A^*, определение функций f, g, h; реализация **34**

51 Вырожденные случаи в алгоритме A^*: $h \equiv 0$, $h(v) = dist(v, t)$ **35**

52 Допустимые и монотонные эвристики в алгоритме A^*. Примеры монотонных эвристик на разных сетках. **35**

53 Формулировка работоспособности (корректность и время работы) алгоритма A^* в случае монотонной, допустимой или произвольной эвристики. Доказательство для монотонного случая. **36**

54 Алгоритм Флойда: поиск попарных кратчайших расстояний в графе без отрицательных циклов. Реализация, асимптотика. **36**

55 Восстановление ответа (пути) в алгоритме Флойда. **37**

56 Алгоритм Форда—Беллмана: поиск кратчайших расстояний от одной вершины до всех. Реализация, асимптотика (в случае отсутствия отрицательных циклов). **37**

57 Алгоритм Форда—Беллмана: нахождение кратчайших расстояний от одной вершины до всех в случае наличия отрицательных циклов. **38**

58 Остовный подграф, остовное дерево. Минимальный остов. Лемма о безопасном ребре. **39**

59 Алгоритм Прима: доказательство корректности и реализации за $O(n^2), O(m\ log\ n), O(m + n\ log\ n)$. **39**

60 Система непересекающихся множеств (СНМ). Виды запросов. Эвристика по рангу, эвристика сжатия путей. Асимптотика ответа на запрос при использовании обеих эвристик (б/д). **40**

61 Асимптотика ответа на запрос в СНМ при использовании только эвристики по рангу. **41**

62 Алгоритм Крускала: корректность, реализация, асимптотика. **41**

63 Алгоритм Борувки: выбор минимального ребра из нескольких, корректность, реализация, асимптотика. ... **42**

64 Определение паросочетания в произвольном графе, двудольного графа, увеличивающего пути. ... **43**

65 Лемма об устройстве неориентированного графа, в котором степени всех вершин не превосходят двух. ... **44**

66 Теорема Бержа. ... **44**

67 Алгоритм Куна. Корректность, реализация, асимптотика. ... **45**

68 Лемма об отсутствии увеличивающих путей из вершины при отсутствии таких путей относительного меньшего паросочетания. ... **46**

69 Определения независимого множества, вершинного покрытия. Связь определений. ... **47**

70 Алгоритм поиска максимального независимого множества и минимального вершинного покрытия в двудольном графе с помощью разбиения на доли $L^-; L^+; R^-; R^+$ (с доказательством). ... **47**

71 Алгоритм поиска максимального независимого множества и минимального вершинного покрытия в двудольном графе с помощью задачи 2SAT. ... **49**

72 Определения сети, потока, величины потока, остаточной сети. Пример, почему нельзя обойтись без обратных рёбер. ... **49**

73 Определения разреза, величины разреза, величины потока через разрез. Лемма о равенстве величины потока и величины потока через разрез. ... **50**

74 Лемма о связи величины произвольного потока и величины произвольного разреза. ... **51**

75 Теорема Форда—Фалкерсона. ... **51**

76 Алгоритм Форда—Фалкерсона. Корректность, асимптотика. Пример сверхполиномиального (от размера входа) времени работы. ... **52**

77 Алгоритм Эдмондса—Карпа. Корректность. ... **52**

78 Лемма о возрастании $dist(s, v)$ между последовательными итерациями алгоритма Эдмондса-Карпа. ... **52**

79 Лемма о числе насыщений ребра в алгоритме Эдмондса-Карпа. Асимптотика этого алгоритма. ... **53**

80 Задача о разбиении коллектива на две группы с минимизацией суммарного недовольства. ... **53**

81 Алгоритм Эдмондса-Карпа с масштабированием, асимптотика. ... **54**

82 Определение слоистой сети, блокирующего потока. Алгоритм Диница, доказательство корректности. ... **55**

83 Реализация алгоритма Диница. Асимптотика. ... **55**

84 Первая теорема Карзанова о числе итераций алгоритма Диница. ... **56**

85 Эффективность алгоритма Диница в единичных сетях. ... **57**

86 Алгоритм Хопкрофта—Карпа поиска максимального паросочетания в двудольном графе. Корректность и асимптотика. ... **58**

87 Алгоритм Штор—Вагнера поиска минимального глобального разреза. ... **58**

88 Min cost flow: постановка задачи. Алгоритм поиска потока величины k минимальной стоимости (б/д). Асимптотика. ... **60**

89 Критерий минимальности стоимости потока величины k. ... **60**

90 Корректность алгоритма поиска min cost k-flow в отсутствие отрицательных циклов. ... **61**

91 Потенциалы Джонсона. Поиск min cost k-flow с помощью алгоритма Дейкстры. ... **61**

92 Задача о назначениях. Решение сведением к потоковой задаче. ... **62**

93 Определение дерева, его свойства (б/д). Определение диаметра дерева. Алгоритм поиска диаметра в дереве. ... **63**

94 Определение центроида в дереве. Алгоритм поиска центроида в дереве. Лемма о количестве центроидов (б/д). ... **65**

95 Определение изоморфизма графов. Алгоритм проверки изоморфности двух ориентированных или неориентированных деревьев за O(n log n). **65**

96 Задача LCA. Постановка, решение с помощью двоичных подъёмов. **66**

97 Задача LCA. Решение с помощью эйлерова обхода. **68**

98 Задача LCA. Алгоритм Фарах-Колтона и Бендера..... **68**

99 Задача RMQ. Постановка, решение за O(n) предподсчёта и O(1) на запрос. **69**

100 Heavy-light decomposition. Тяжёлые и лёгкие рёбра. Лемма о числе лёгких рёбер на пути между двумя вершинами. Решение задачи обновления на ребре и суммы на пути за $O(log^2(n))$ на запрос. **70**

101 Центроидная декомпозиция. Подсчёт числа объектов, обладающих заданным свойством. **71**

102 Центроидная декомпозиция. Задача о перекрашивании синих вершин в красный цвет и поиска расстояния до ближайшей красной вершины. **71**

18 Операции над множествами (масками): объединение, пересечение, разность. Реализация в программе. Проверка, что одна маска является подмножеством другой. Проверка, что число является степенью двойки.

Идея ДП по маскам: пусть n какое-то небольшое число ($\leqslant$ 30-60), тогда мы можем эффективно закодировать все подмножества множества $\{0,...,n-1\}$. Для этого рассматриваем маску. Она будет состоять из n символов, каждый из которых - единица или ноль. Единица на i-м месте в маске означает, что мы берем число i в подмножество, 0 - не берем в подмножество

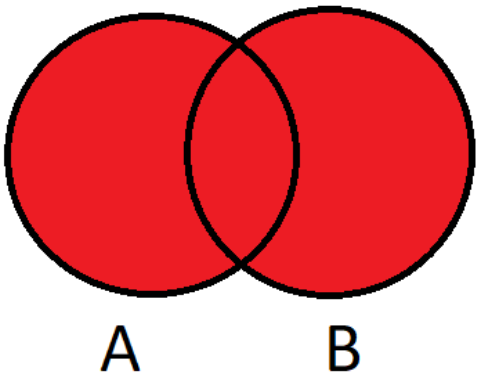
Операции над масками

Пусть даны два множества A и B с соответсвующими им масками $mask_A$ и $mask_B$ ($\oplus$ = xor)

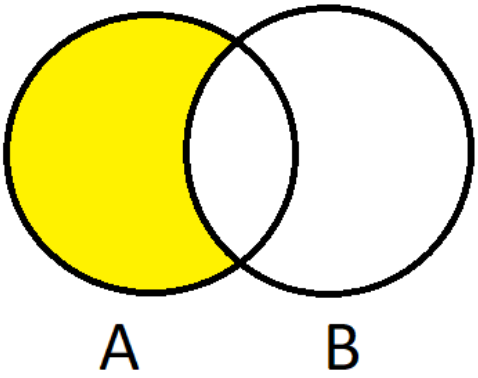
Объединение
$A \cup B = mask_A \mid mask_B$

Пересечение
$A \cap B = mask_A \ \& \ mask_B$

Разность $A \backslash B = (mask_A \mid mask_B) \ \oplus \ mask_B = mask_A \& (\sim mask_B)$.



$mask_A \mid mask_B$



$(mask_A \mid mask_B) \ \oplus \ mask_B$

Проверка, что одна маска является подмножеством другой

Пусть даны две маски A и B. Утверждается, что $B \subset A \iff (A \& B) = B$

▲

$\rightarrow$ Пусть $B \subset A$, тогда если в маске B на месте i стоит единица, то и в маске A на месте i стоит единица, а значит, их побитовое "и"даст в точности B.

$\leftarrow$ Если $(A \& B) = B$, то в маске A единицы стоят как минимум на тех местах, что и в маске B, иначе их побитовое "и"не дало бы единицу на том же месте, что и в маске B. Это значит, что $B \subset A$ ■

Проверка, что число является степенью двойки.

Утверждается, что число a является степенью двойки $\iff a\&(a-1)=0$

▲

$\rightarrow$ Рассмотрим двоичную запись числа a. Если a $= 2^n$, то a $= 1\underbrace{0...0}_{n} \implies a-1 = \underbrace{1...1}_{n-1} \implies$ $a\&(a-1)=0$

$\leftarrow$ Пусть $a \neq 1\underbrace{0...0}_{n}$, тогда $a = \underset{n}{1...1}\underbrace{...}_{i-1}$, т.е. существует единица на i-м месте в записи числа a, тогда $a-1 = \underset{n}{1}\underbrace{...}_{n-1}$, т.е. старший бит не обнуляется, а значит, побитовое "и"не может равняться нулю. Противоречие. ■

19 Задача о самом дешёвом гамильтоновом пути: решение за $O(2^n n^2)$

Запишем важную и полезную функцию, которая позволяет извлекать i-й элемент маски:

```
bool bit(long long mask, int pos) {
  return (mask >> pos) & 1;
}
```

Формулировка задачи

Дан полный, неориентированный, взвешанный граф.

Гамильтонов путь в графе - это путь, который начинается в какой-то вершине графа, проходит по всем вершинам и посещает все вершины графа ровно по одному разу

Среди всех таких путей нас интересует путь минимальной стоимости

Пусть у нас есть какой-то путь, проходящий по каким-то вершинам графа ровно один раз и заканчивающийся в вершинке v. Тогда для того, чтобы продолжить этот путь, нам необходимо знать, в какой вершинке мы находимся и маску посещенный вершин. Отсюда возникает dp:

1. dp[v][mask] - это стоимость минимального пути, который где-то начинается, посещает все вершины маски ровно по одному разу и заканчивается в v
2. База: Пути длины 0. Когда мы стоим в какой-то вершинке и еще никуда не пошли. dp[v][2^v] = 0, все остальные значения положим $+\infty$
3. Переход: перебираем все ребра, исходящие из v, которые ведут в вершины, еще не посещенные (в маске на месте соответствующей вершины будет стоять 0), и смотрим, куда можем пойти.

(1«u)|mask - взять побитовое "или"числа 2^u и mask. Нетрудно заметить, что такая процедура просто ставит единичку на u-й бит маски

```
for (int mask = 0; mask < 2^n; ++mask){
    for(int v = 0; v < n; ++v){
        for(int u = 0; u < n; ++u){
            if(bit(mask,u)) continue;
            int newmask = (1<<u)|mask;
```

```
            dp[u][newmask] = max(dp[u][newmask], dp[v][mask] + cost[v][u]);
        }
    }
}
```

4. Ответ: Минимальное $dp[u][2^n - 1]$. То есть нас интересуют все пути, которые посещают все вершины, выбираем из них тот, что имеет минимальную стоимость

Асимптотика
За счет циклов $O(2^n n^2)$

20 Задача о максимальной клике: решения за $O(2^n n^2)$, $O(2^n n)$, $O(2^n)$

Формулировка задачи
Дан неориентированный, взвешанный граф.
Кликой в неориентированном графе называется подмножество вершин, каждые две из которых соединены ребром графа.

Хотим найти максимальную по мощности клику в данном графе

$O(2^n n^2)$

Перебираем все подмножества n и за n^2 перебором проверяем, что это клика

$O(2^n n)$

1. Положим $dp[mask] = \begin{cases} true & mask - klika \\ false & else \end{cases}$
2. Будем делать дп назад
3. $dp[mask]$ - клика $\iff$ для любой вершины v из этой маски выполнено:

1 $dp[mask \oplus 2^v] = true$. $mask \oplus 2^v$ - на v-ое место маски ставит false. Здесь мы проверяем, что маска без этой вершинки v - клика

2 v соединена со всеми вершинами маски

4. База: dp[0] = true
5. Ответ: максимальная по мощности маска, для которой dp[mask] = true

Асимптотика
Перебор всех масок: 2^n. Для каждой маски необходимо за линейное время определить какую-нибудь вершинку, которая входит в эту маску (просто пройтись по n битам и найти тот, для которого bit(mask, i) = true). Далее идем по маске и за линейное время определяем, связана ли вершинка, которую мы выбрали, со всеми остальными. Итого получаем $O(2^n n)$

$O(2^n)$

Новый алгоритм - это оптимизация предыдущего.

1 Для каждой вершинки u посчитаем маску ее соседей. Пусть neigh[u] - маска соседей u. Сделаем это на этапе ввода графа.

2 Теперь бит v выбираем не произвольным образом. Это будем старший включенный бит маски. Почему старший? Его будет удобно насчитывать, что мы поймем дальше

Тогда проверка пункта 3.2(см. предыдущий алгоритм) будет проверяться за O(1): dp[$mask \oplus 2^v$] $\subset neigh[v]$. Из 18 билета мы знаем, что $a \subset b \iff (a \ \& \ b) = a$. Значит, проверка:
$(dp[mask \oplus 2^v] \ \& \ neigh[v]) = dp[mask \oplus 2^v]$

Теперь поймем, как хранить старший бит

```
int oldest_bit = -1;
for(int mask = 1; mask < 2^n; ++mask){
   if(mask & (mask - 1) == 0) ++oldest_bit;
   ...
}

```

Сначала отметим старший бит равным -1. Очевидно, что старший бит маски будет меняться только в том случае, если мы переходим через степень двойки ($mask \ \&(mask-1) == 0$ - проверка на то, что mask - степень двойки, см. 18 билет)

Асимптотика
Таким образом, мы научились делать переход за O(1). Значит, итоговая асимптотика $O(2^n)$

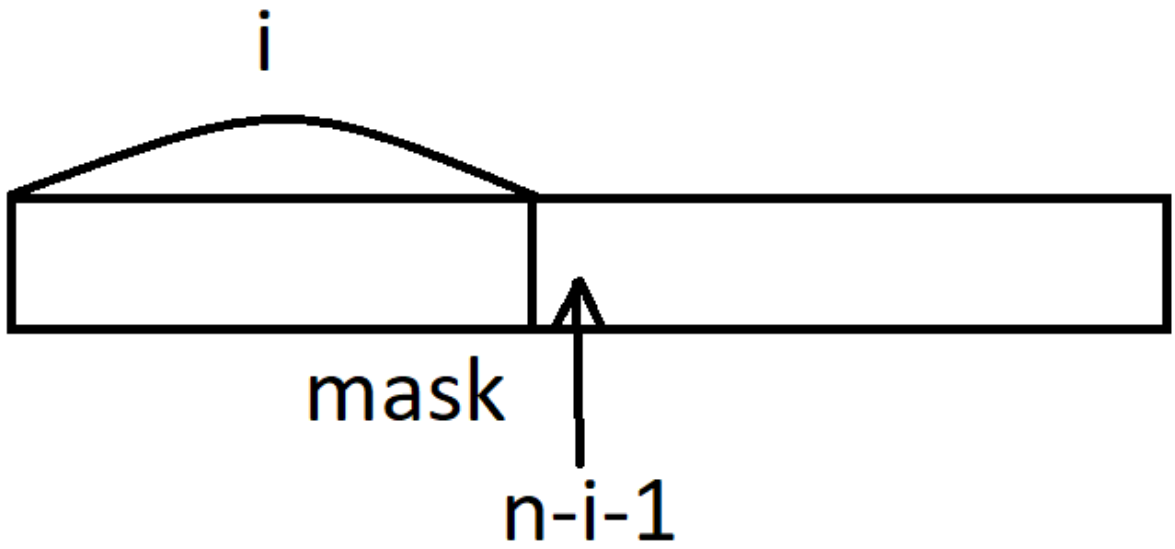
21 Подсчёт всех значений $b(mask) = \max_{s \subset mask} a(s)$ для данного набора значений $a(0), a(1), ..., a(2^n - 1)$ за $O(2^n n)$

Мы хотим для каждой из масок $0...2^n - 1$ посчитать максимальное по мощности подмножество маски, являющееся кликой Пусть a(V) =

0, если V - не клика

|V|, если V - клика

Тоогда мы хотим найти b(mask) = $\max\limits_{V \subset mask} a(V)$. Будем делать это с помощью дп
1. dp[k][mask] = $\max\limits_{V \subset mask} a(V)$, где V - такое, что первые k битов V и mask совпадают
2. База dp[n][mask] = a(mask)
3. Переход. Будем ходить по подмаскам. Пусть нам известно dp[i+1][mask]. Научимся преходить к dp[i][mask]. Зафиксируем старшие i битиков, тогла номер у следующего, если считать от начала маски, n-i-1



1 Если !bit(mask, n - i -1), то при переходе к подмаске мы не убрали ни одного элемента (т.к. при переходе к подмаске 0 не может стать единицей, значит, остается только 0), а значит, максимальная мощность подмножества, являющегося кликой, не изменилась, dp[i][mask] = dp[i+1][mask]

2 Если bit(mask, n -i-1), то у нас есть выбор, при переходе к подмаске оставить единицу на этом месте или превратить ее в ноль. Так как мы ищем максимальную мощность, то dp[i][mask] = max(dp[i+1][mask], dp[i][$mask \oplus 2^{n-i-1}$]

4. Ответ: b(mask) = dp[0][mask]

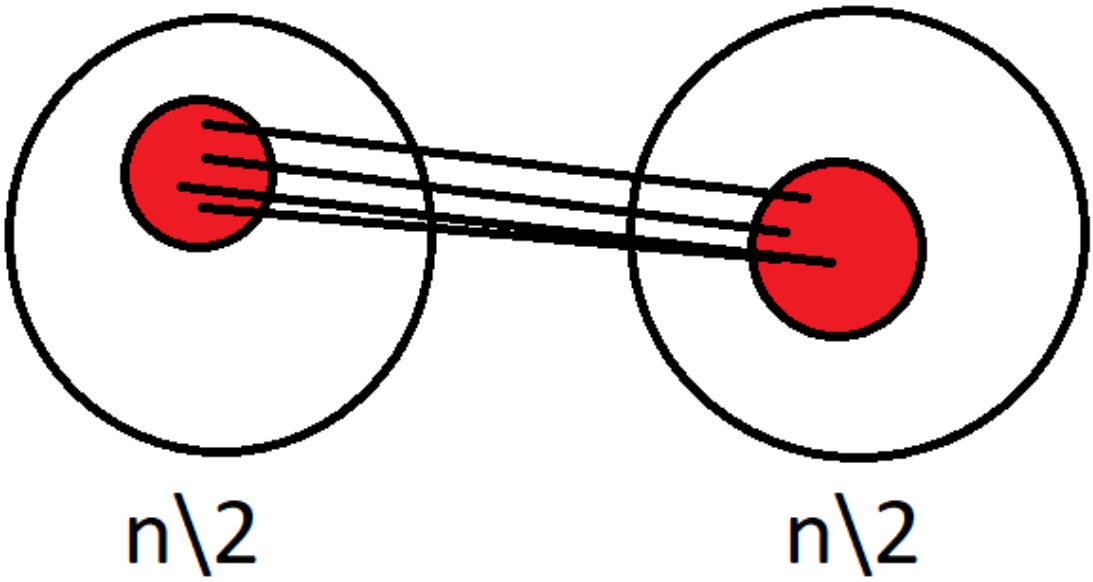
Асимптотика

1. Базу можно насчитать за O(2^n)
2. Перебираем все i = n-1...0, перебираем все маски, делаем переход за O(1)

Итог: O($2^n n$)

22 Задача о максимальной клике: решение за O($2^{\frac{n}{2}} n$).

Здесь будет использоваться идея meet-in-the-middle. Она заключается в том, что мы делим n пополам и получаем две маски длины $\frac{n}{2}$



Очевидно, что если маска является кликой, то, если мы распилим ее пополам, новые маски останутся тоже кликами.
Если мы нашли какую-то клику в одном из множеств, то, если мы дополним ее нулями, получится клика во множестве масок длины n.

Отсюда получаем, что все маски длины n можно разделить на 3 группы

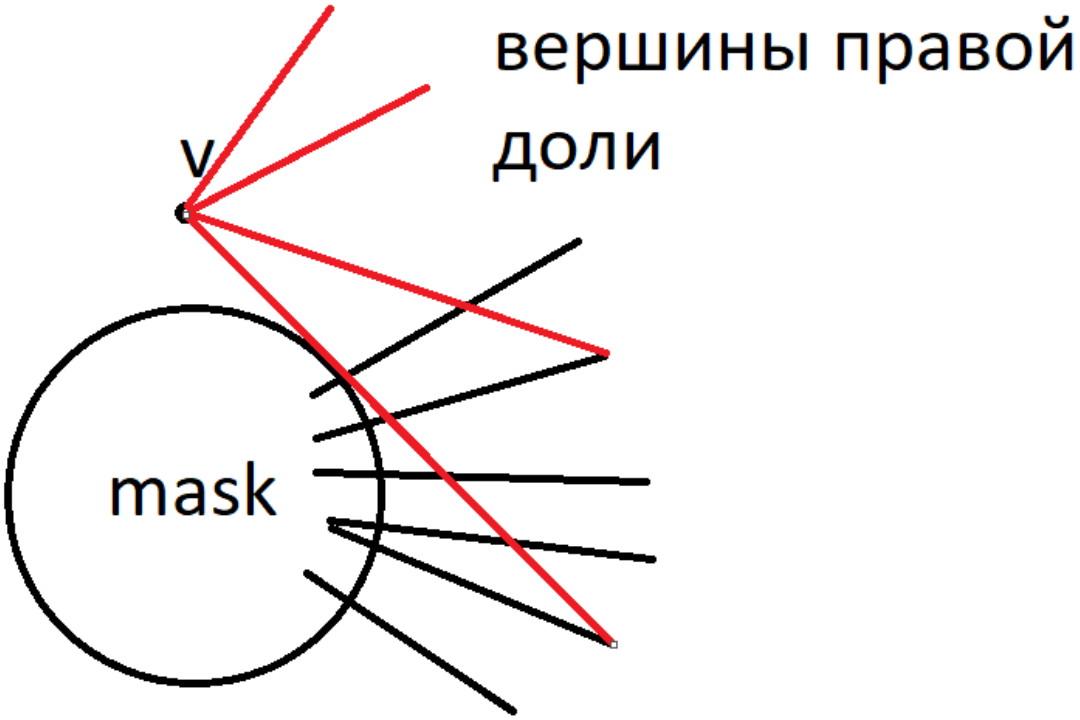
1 Правая часть - нули, левая - клика в левом множестве

2 Левая часть - нули, правая - клика в правом множестве

3 u (левая часть маски длины n) - клика в левом множестве, v (правая часть маски длины n) - клика в правом множестве

Алгоритм
Рассмотрим маску вершин левой доли mask. И запишем маску mask_right вершин правой доли, которые соединены со всеми вершинами из mask. (То есть перебираем все вершины правой доли, и если какая-то вершина соединена со всеми из mask, добавляем ее в маску mask_right)

Пусть dp1[mask] = mask_right, neigh'[v] - маска соседей v в правой доле (вершина v из левой доли), посчитать все neigh'[i] можно за квадрат простым перебором
1. dp1[0] - вся правая доля
2. Хотим добавить в маску вершинку v. dp1[mask|2^v] = (dp1[mask]& neigh'[v])



В маску dp1[mask|2^v] входит персечение соседей v и соседей всех вершин mask. По определению пересечение всех соседей mask - это dp1[mask]

Зафиксируем множество U, являющееся кликой в левой доле. Тогда дополнение V множества U до клики во всем множестве - это некоторое подмножество dp1[U]. Поскольку U - клика, и из U проведены все ребра в вершины правой доли, то V - это максимальное подмножество dp1[U], которое является кликой (так как все остальные вершины соединены между собой, осталось найти множество вершин из dp1[U], которые соединены между собой).

Для любой маски mask правой доли найдем максимальное по мощности множество $V \subset mask$, являющееся кликой. Будем это делать, используя алгоритм из билета 21, за $2^{\frac{n}{2}}n$

Асимптотика

1. Насчитать dp1[mask] - перебор всех масок за $2^{\frac{n}{2}}$, для каждой маски проходимся по всем вершинам левой доли от $0...\frac{n}{2}$, проверяем, входит ли вершина в mask O(1), если не входит, то делаем переход по формуле за O(1). Тогда этот шаг за $O(2^{\frac{n}{2}}n)$
2. Насчитать для каждой маски mask правой доли максимальную клику a[mask], являющуюся подмаской за $O(2^{\frac{n}{2}}n)$
3. Перебираем все маски mask левого множества, являющиеся кликами(можно воспользоваться алгоритмом за 2^n или $2^n n$, там будет насчитано нужное dp), для каждого dp1[mask] максимальная клика в правом множестве - a[dp1[mask]], тогда искомое множество - объединение mask и a[dp1[mask]]. Мощность объединения можно посчитать за O(n), насчитывая количество единиц в двоичной записи масок. Третий шаг за $O(2^{\frac{n}{2}}n)$

Итоговая асимптотика: $O(2^{\frac{n}{2}}n)$

23 Симпатичные узоры: количество раскрасок таблицы n × m в два цвета без одноцветных квадратиков 2 × 2. Прямой профиль: решения за O($4^n(n+m)$) и O($8^n logm$)

O($4^n(n+m)$)

Формулировка
Дана таблица nxm и два цвета

1 - черный

2 - белый

Мы не хотим, чтобы в таблице были квадратики 2x2, расскрашенные в один цвет
Сколько есть таких раскрасок?

Предположим, мы уже раскрасили j столбцов. Что нам нужно знать для продолжения раскраски? Очевидно, нам нужно знать только цвета последнего столбца. Отсюда возникает динамика:

1. dp[j][mask] - сколько есть раскрасок j столбцов так, чтобы последний из них был в точности покрашен в mask
2. Переход: перебираем все маски, соответствующие расскраске следующего слолбца и смотрим, не нарушается ли симпатичность узора

Предподсчитаем корректность перехода между масками. Для этого перебираем все комбинации масок и проверяем, сравнивая их за O(n), корректен ли переход. асимптотика предподсчета O($4^n n$)
3.База: dp[1][mask] = 1
4.Ответ: сумма по всем i dp[m][i]

```
for (int j = 1; j <= m - 1; ++j) {
  for (long long mask = 0; mask < 2^n - 1; ++mask) {
    for (long long mask1 = 0; mask1 <  2^n - 1; ++mask1) {
      if (ok[mask1][mask])
        dp[j + 1][mask1] += dp[j][mask];
    }
  }
}
```

Асимптотика
За счет циклов O($4^n(m)$) + предподсчет O($4^n(m)$). Итого O($4^n(m+n)$)

O$(8^n logm)$

Заметим, что значение dp[i][mask] $= \sum_{mask1=0}^{2^n-1} ok[mask][mask1] * dp[i-1][mask1]$, значит,

$$\begin{pmatrix} dp[i][0] \\ dp[i][1] \\ ... \\ dp[i][2^n-1] \end{pmatrix} = \begin{pmatrix} ok[0][0] & ok[0][1] & ... & ok[0][2^n-1] \\ ok[1][0] & ok[1][1] & ... & ok[1][2^n-1] \\ ... & ... & ... & ... \\ ok[2^n-1][0] & ok[2^n-1][1] & ... & ok[2^n-1][2^n-1] \end{pmatrix} * \begin{pmatrix} dp[i-1][0] \\ dp[i-1][1] \\ ... \\ dp[i-1][2^n-1] \end{pmatrix} =$$

$$= ok * \begin{pmatrix} dp[i-1][0] \\ dp[i-1][1] \\ ... \\ dp[i-1][2^n-1] \end{pmatrix}$$

Тогда

$$\begin{pmatrix} dp[m][0] \\ dp[m][1] \\ ... \\ dp[m][2^n-1] \end{pmatrix} = ok^{m-1} * \begin{pmatrix} dp[1][0] \\ dp[1][1] \\ ... \\ dp[1][2^n-1] \end{pmatrix}$$

Асимптотика

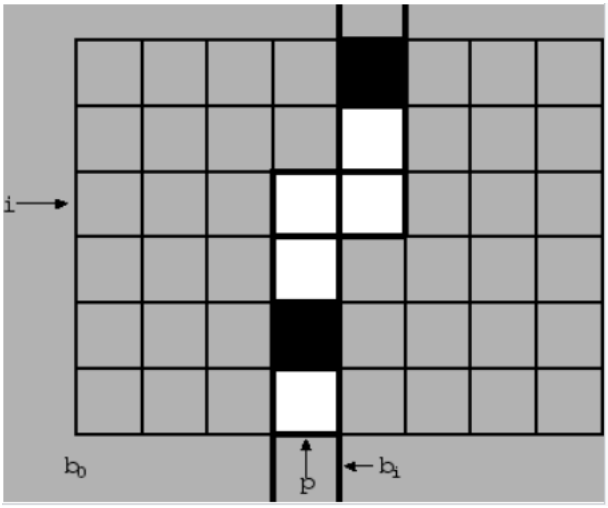
1. Матричное умножение O$((2^n)^3 log\ m)$ = O$(8^n log\ m)$

Это хорошая асимптотика, если n $\leqslant$ 6, m$\leqslant 10^{100}$

24 Симпатичные узоры. Изломанный профиль: идея решения за O$(2^n mn)$

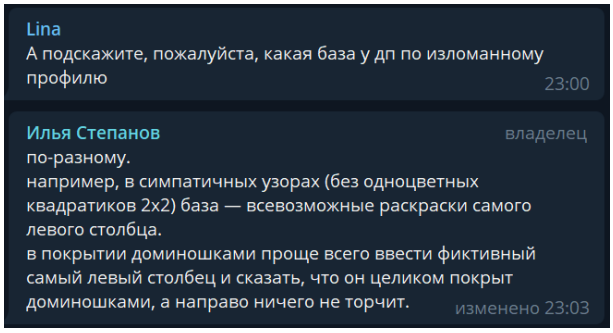
Формулировка задачи та же самая

Теперь мы режем нашу табличку не по прямой, а по ломанной прямой. Берем некоторое количество элементов в начале рассматриваемого столбца, а остальные - из предыдущего

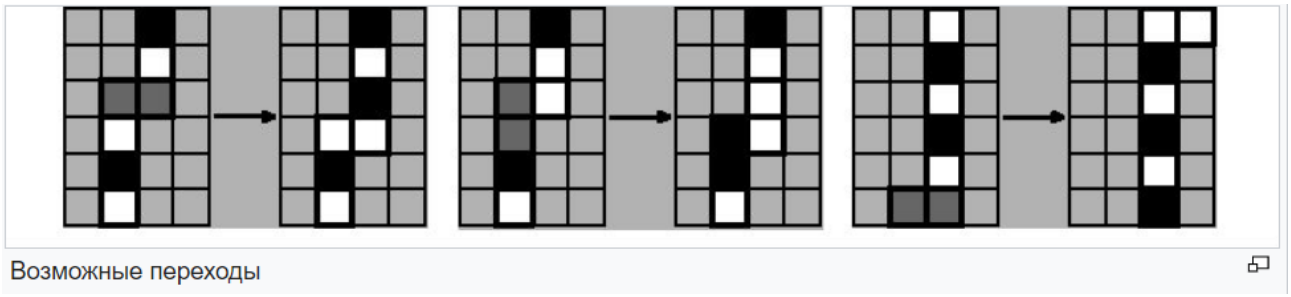


Хотим сделать то же самое: предполагая, что все левее разреза покрашено корректно, знать, как будем продолжать красить

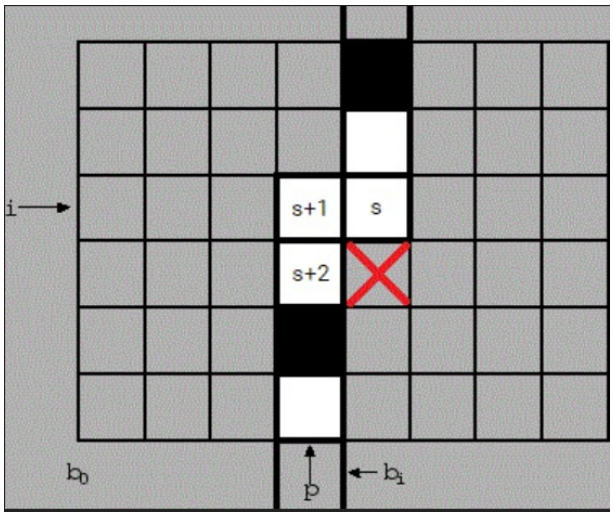
1. dp[j][i][mask] = сколько способов дойти до j столбца и i ячейки в нем, считая сверху, с маской mask у нас есть. (i+1,j) - излом, mask - соответственная маска вдоль этого излома (на картинке - раскрашенная полоска)
2. База:



, все остальные - 0
3. Переход: Просчитываем, как можно изменить квадратик, стоящий на изломе (то есть он может быть 0 или 1). Если преход корректен,красим его в этот цвет, переходим к dp[j][i+1][newmask], где новая маска пересчитывается в соответствии с тем, как мы покрасили клетку. Заметим, что хотябы один переход возможен всегда


Возможные переходы

Чуть-чуть подробнее про переходы масок. Ячейки в маске нумеруются s-ками. Проталкивая разрез, меняем в маске s+1 ячейку на выбранную раскраску излома, новым изломом становится клеточка под крестиком



Алгоритм:

1 Цикл по j

2 Цикл по mask

3 Цикл по i

4 Просчитываем квадратик излома

- Если можно поставить единичку или ноль, ставим, получаем newmask, dp[j][i+1][newmask] += dp[j][i][mask] (если дошли до конца столбца, переходим в dp[j+1][1][newmask]

Ответ: сумма по всем mask dp[m][n][mask]

25 Определения неориентированного и ориентированного графов, пути, (вершинно) простого пути, рёберно простого пути. Связь вершинной и рёберной простоты. Определение цикла, рёберно простого цикла, (вершинно) простого цикла. Определение достижимости между вершинами, простота пути. Определение связности.

Определение *Ориентированным графом* G называется пара G=(V,E), где V — множество вершин , а $E \subset V \times V$ — множество рёбер.

Определение *Неориентированным графом* G называется пара G=(V,E), где V — множество вершин, а E $\subset$ {{v,u}:v,u$\in$V} — множество рёбер. (Под {v,u} понимается неупорядоченная пара)

Определение *Путём* в графе называется последовательность вида $v_0v_1...v_k$, где $e_i \in E$, $e_i = (v_{i-1},v_i)$

Определение Путь называется *реберно-простым*, если в нем нет повторяющихся пар ребер

Определение Путь называется *вершинно-простым*(простым), если в нем нет повторяющихся вершин

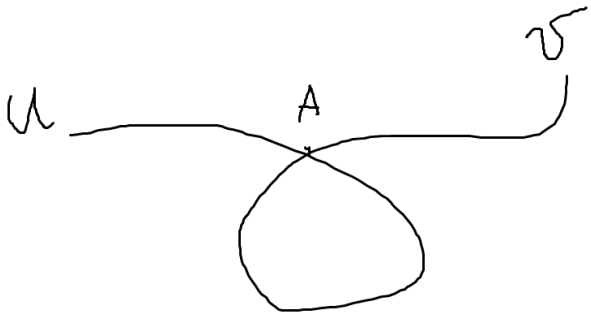
Замечание
Если путь вершинно-простой, то он и реберно-простой
▲ Если путь вершинно-простой, то в нем нет повторяющихся вершин, значит, и повторяющегося ребра возникнуть не может, так как для того, чтобы возникло повторяющееся ребро, необходима пара повторяющихся вершин ■

Определение Цикл называется *вершинно-простым* (простым), если в нем нет повторяющихся вершин (кроме начальной и конечной)

Определение Цикл называется *реберно-простым*, если в нем нет повторяющихся пар ребер

Определение Пусть u, v $\in V(G)$, тогда говорим, что из *u достижима v*, если из u есть путь в v

Замечание Если из u есть путь в v,то из u есть простой путь в v
▲

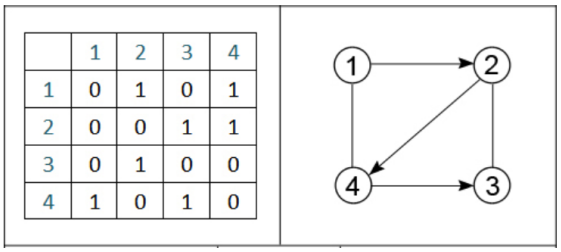


Если пути пересекаются в какой-то точке A, можем игнорировать тот участок пути, который проходит от первого попадания в A на пути до последнего попадания в A, и пойти дальше до v. Таким образом можно избавиться от всех повторяющихся в пути вершин и сделать путь простым ■

Определение Неориентированный граф называется *связным*, если между любыми двумя вершинами графа есть путь

26 Три способа хранения графа в памяти компьютера, их преимущества и недостатки.

1 **Матрица смежности**



	1	2	3	4
1	0	1	0	1
2	0	0	1	1
3	0	1	0	0
4	1	0	1	0

- это матрица nxn, где n = |V|. На пересечении i строки и j столбца матрицы стоит 1 тогда и только тогда, когда в графе G есть ребро между вершиной i и вершиной j

+ За O(1) узнать, есть ли ребро между вершинами в графе
- Занимает n^2 памяти

2 **Список ребер** - просто хранить список ребер, как пары

+ Более естественный способ задания
- Неудобно работать. Не можем быстро проверить ни наличие ребра, ни узнать степень вершины и т.д.

3 **Список смежности** - храним массив из n элементов l_1,....,l_n, где l_i - список всех соседей вершины i

+ Хранится в памяти за O(n + m), так как всего у нас n списков, в которых суммарно не более 2m элементов
+ Знаем список соседей каждой вершины
- Нельзя за O(1) узнать, есть ли ребро в графе (но можно использовать FixedSet, который занимает столько же памяти и позволяет узнать, есть ли ребро в графе. То есть этот минус можно обойти, использовав такую структурку)

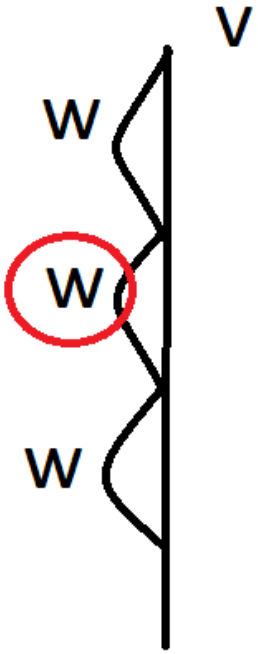
27 Поиск в глубину: алгоритм dfs на ориентированном графе. Лемма о белых путях

```
vector<vector<int>> g; \\ просто наш граф
vector<int> tin, tout; \\ время входа и выхода для каждой вершинки
int timer = 0;
vector<string> color(n, "white"); \\ будем красить вершины по мере посещения в три цвета
vector<int> parent;



void dfs(int v, int p = -1){
tin[v] = timer++; \\ заходим в в вершину, увеличиваем таймер
parent[v] = p; \\ проставляем родителя посещенной вершины
color[v] = "gray"; \\ красим вершину в цвет 1, помечая, что начинаем ее
                      использовать
for(int to: g[v]){ \\ выбираем, куда можно пойти из текущей вершины
    if(color[to]!= "white") continue; \\ если вершина, которую мы хотим посетить,
                                         уже начинала использоваться, ее цвет
                                         не белый, то есть в нее мы пойти не
                                         можем
    dfs(to,v); \\ если вершина не использована, мы в нее идем
}
tout[v] = timer++; \\ перебрали все вершины, в которые можно попасть из v,
                      записываем время выхода
color[v] = "black"; \\ помечаем, что вершина использована
}
```

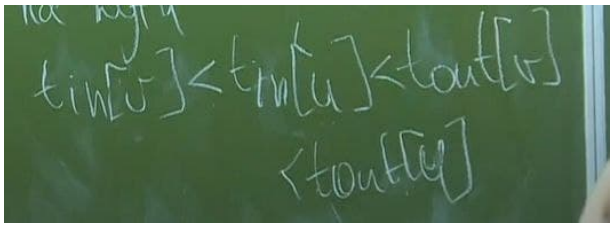
Лемма о белых путях
Все пути, бывшие белыми в момент tin[v] и начинающиеся в v, станут черными в момент tout[v]
▲ Индукция в порядке убывания tin[v].

1 База. tin[v] = max
Если tin[v] = max, то из вершины v в принципе нет белых путей. В противном случае мы пошли бы в какую-то вершинку впервые, и время входа в нее было бы больше, чем tin[v], что противоречит условию. Значит, к моменту tout[v] все пути из v будут покрашены в черный

2 Переход. Пусть есть какая-то вершина v и белый путь из нее к моменту времени tin[v].



Пусть не все вершины из v к коменту времени tout[v] покрасилось в черный. Рассмотрим вершинку, которая будет самой верхней на пути из v, не покрашенной в черный. Пусть это вершина u. Рассмотрим вершину, которая выше u. Она покрашена в черный, значит, dfs перебрал все вершины, исходящие из нее, и в том числе он сходил в u, значит, u может быть только серой вершиной. Но и такого не может быть, поскольку для того, чтобы выйти из рекурсии на более высоком уровне(в вершине, из которой мы сходили в u) dfs должен выйти из рекурсии на более низком уровне, то есть, u должна покраситься в черный. Противоречие



■

28 Поиск в глубину: множество посещаемых вершин, поиск цикла, достижимого из s, проверка на ацикличность

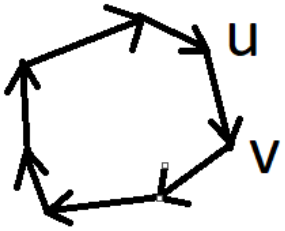
Следствие Из леммы о белых путях следует, что если вызвать dfs(s) из main, то все вершины, достижимые из s, посетятся

▲ Это очевидно, поскольку изначально все пути белые, а значит, все пути, которые только можно пройти, будут покрашены в черный, то есть все достижимые вершины посетятся ■

Следствие В графе есть цикл, достижимый из s $\iff$ dfs(s) когда-нибудь найдет

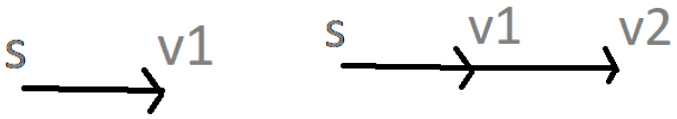
ребро в color[to] = "gray"
▲

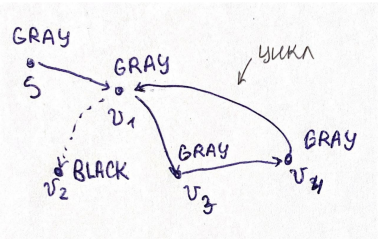
→ Пусть C - цикл, достижимый из s. Тогда при запуске dfs(s) мы обязательно посетим этот цикл. Пусть v - первая вершина цикла, которую мы посетим при обходе. Тогда к времени tin[v] весь остальной цикл еще белый, а значит, к моменту tout[v] мы пройдем весь цикл. В частности, посетится вершина u, являющаясь предыдущей для v в этом цикле. Значит, искомое ребро в серую вершину - (u,v).



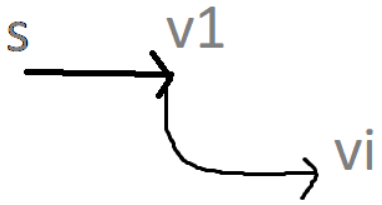
← Стек рекурсии - это путь из серых вершин. Заметим, что наш dfs работает так:

– Запускаемся из s, красим ее в серый, перебираем ребра из нее. Выбираем какую-то вершинку, красим ее в серый и так далее





– Если мы походили-походили, не наткнулись на серую вершину и возвращаемся назад, вынимаем вершину из стека рекурсии и ищем новое ребро



Отсюда следует, что если мы в какой-то момент наткнулись на серую вершину, то образуется цикл

■

Замечание

Сам цикл можно восстановить, переходя по родителям до того момента, как мы окажемся в вершинке, откуда начался цикл

Определение Граф называется *ациклическим*(DAG - directed acyclic graph), если в нем нет циклов

Проверка графа на ацикличность совершается следующим образом. Произведём серию поисков в глубину в графе. Т.е. из каждой вершины, в которую мы ещё ни разу не приходили, запустим поиск в глубину, который будет искать цикл. Если цикл найден, возвращаем false. Если все вершины покрашены в черный, а цикл не найден - true

29 Топологическая сортировка ориентированного ациклического графа: определение и алгоритм поиска (с доказательством корректности).

Топологическая сортировка ориентированного ациклического графа $G(V,E)$ представляет собой упорядочивание вершин таким образом, что для любого ребра $(u,v) \in E$ номер вершины u меньше номера вершины v.

Предположим, что граф ацикличен, т.е. решение существует. Что делает обход в глубину? При запуске из какой-то вершины v он пытается запуститься вдоль всех рёбер, исходящих из v. Вдоль тех рёбер, концы которых уже были посещены ранее, он не проходит, а вдоль всех остальных — проходит и вызывает себя от их концов.

Таким образом, к моменту выхода из вызова $DFS(v)$ все вершины, достижимые из v как непосредственно (по одному ребру), так и косвенно (по пути) — все такие вершины уже посещены обходом. Следовательно, если мы будем в момент выхода из $DFS(v)$ добавлять нашу вершину в начало некоего списка, то в конце концов в этом списке получится топологическая сортировка.

```
for (int i = 0; i < n; ++i)
    if (color[v] == "white")
        dfs(i);
// вывести вершины в порядке убывания tout
```

Корректность: ▲ Покажем, что если $(u,v) \in E$, то u выведется раньше v. u выведется раньше чем $v \Leftrightarrow tout[v] < tout[u]$ Рассмотрим 2 случая

1. v была замечена DFSом раньше чем u. Тогда к моменту выхода из v вершина u все еще не посещена, иначе нашелся бы цикл $\Rightarrow tout[v] < tout[u]$

2. u была замечена раньше чем v. Тогда мы перейдем по ребру (u,v) в v, а до выхода из u необходимо выйти из всех вершин, в которые мы попали из нее (по лемме о белых путях) $\Rightarrow tout[v] < tout[u]$ ■

Асимптотика: $O(n+m)$ (dfs проходит каждому ребру и вершине по 1 разу. Сортировку tout можно сделать прям в dfs идею см. в билете 31)

30 Отношение сильной связности между вершинами. Компоненты сильной связности. Сильно связный граф.

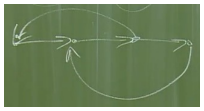
Определение: В ориентированном графе вершины u,v - *сильно связаны*, если из u есть путь в v и из v есть путь в u.

Утверждение: сильная связность является отношением эквивалентности

▲ Рефлексивность и симметричность очевидны из определения. Транзитивность получается из того что просто склеиваем пути ■

Определение: Классы эквивалентности относительно отношения сильной связности, на которые разбивается граф, называются *компонентами сильной связности*.

Определение: Ориентированный граф *сильно связен*, если для каждой вершины все остальные вершины достижимы из нее.

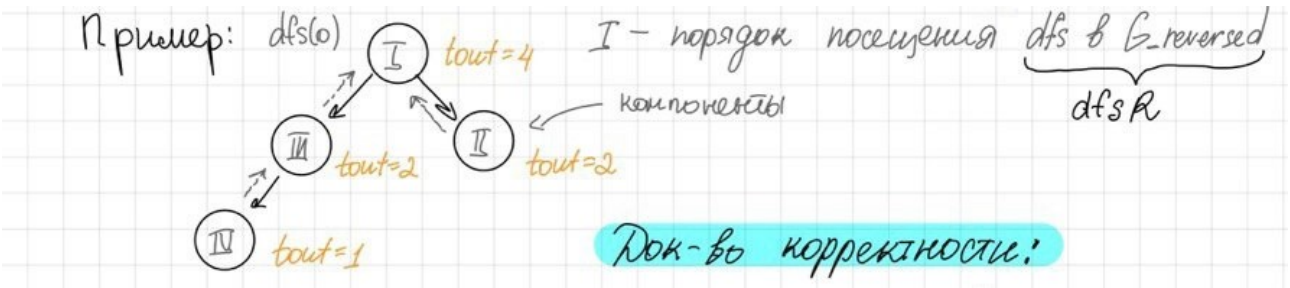


Пример: компонента сильной связности (сама является сильно связным графом)

31 Алгоритм Косарайю. Корректность и время работы.

1. Выполним DFS, вычисляющий для каждой вершины время выхода DFS из нее (этот массив обозначим за p).
2. Строим граф H с инвертированными ребрами
3. Выполняем DFS на H, перебирая вершины в порядке убывания $p(u)$.

```
used.assign(n, false);
for (int s = 0; s < n; ++s)
    if (!used[s])
        dfs(s);
// p - список вершин в порядке убывания tout
// можно получить его например добавив в конец dfs строчку
// p.pushback(v)
// (только потом надо реверснуть весь массив)
used.assign(n, false);
for (int v: p)
    if (!used[s])
        reversed_dfs(s); // dfs в графе, где все ребра инвертированы
// каждое дерево обхода обратного dfs будет являться КСС
```



▲ 1. Если v - вершина с максимальным tout в своей КСС, то $reversed\_dfs(v)$ посетит как минимум всю эту КСС (так как все остальные вершины из нее еще не used) $\Rightarrow$ целиком содержит эту КСС, так как $reversed\_dfs$ проходит по всем вершинам, из которых достижимо v в исходном графе

2. Покажем, что не посетили больше чем одну КСС. Пусть u,v лежат в разных КСС. Тогда имеет место один из двух случаев

– нет путей из u в v и из v в u

Очевидно, что в этом случае ни одна из вершин не достижима из другой ни по прямым, ни по инвертированным ребрам $\Rightarrow$ алгоритм причислит их к разным КСС

– Есть путь из u в v, но нет пути из v в u (симметричный случай доказывается аналогично)

Тогда $tout[u] > tout[v]$ (доказательство аналогично доказательству корректности в топологической сортировке, билет 29). Значит $reversed\_dfs(u)$ запустится раньше чем от v. Так как по предположению из v в u нет пути, то $reversed\_dfs(u)$ не попадет в v. Но тогда $reversed\_dfs(v)$ уже не попадет в u, так как она $used \Rightarrow$ алгоритм причислит их к разным КСС ■

Асимптотика: $O(n+m)$ (обходим все вершины и ребра по одному разу dfsом, потом по одному разу reversed_dfsом)

32 Конденсация ориентированного графа, ацикличность

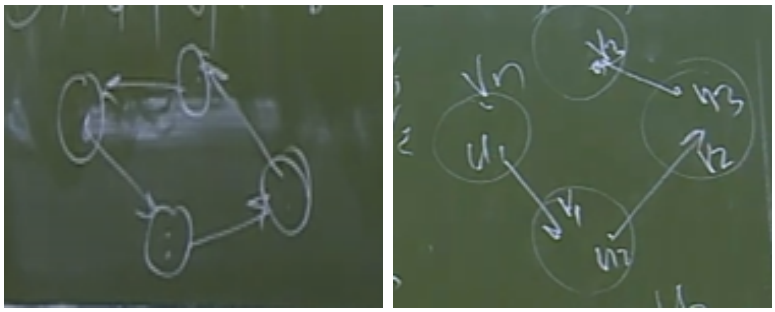
Определение *Кондексацией* ориентированного графа называется граф, построенный таким образом:

1 Выделяем компоненты сильной связности графа

2 Сжимаем компоненты сильной связности до вершин

3 Оставляем ребра между разными КСС, убираем кратные ребра

Утверждение
Конденсированный граф ацикличен.
▲ Предположим, это не так, тогда существует цикл между разными КСС



Пусть это цикл
$u_1 \to v_1$
$u_2 \to v_2$
...
$u_n \to v_n$,
Тогда, так как эти ребра соединяют компоненты сильной связности, из первой компоненты цикла можно добраться до любой компоненты оставшейся части цикла, а значит, и любой вершины из компонент сильной связности, входящих в цикл. Аналогично для второй, третьей и т.д. компоненты из цикла. То есть компоненты, образующие цикл, являются КСС, а значит, разбиение на КСС было некорректным. Противоречие ■

33 Постановка и решение задачи 2SAT (применение алгоритма выделения компонент сильной связности).

Пусть дана 2-КНФ, формула вида $(\overline{p} \vee q) \wedge (p \vee y) \wedge (y \vee \overline{z})$. Будем считать, что дизъюнктов вида $(p \wedge \overline{p}), (p \wedge p)$ нет.
Вопрос: выполнима ли эта формула?
Поступим следующим образом:

1 Для каждой переменной, входящей в КНФ, запишем саму переменную и ее отрицание

2 Для каждой скобки вида $(x \vee y)$ проведем ориентированное ребро между $\overline{x}$ и y, $\overline{y}$ и x. В логическом смысле каждая такая скобка будет эквиваленнтна $(\overline{x} \to y) \wedge (\overline{y} \to x)$.

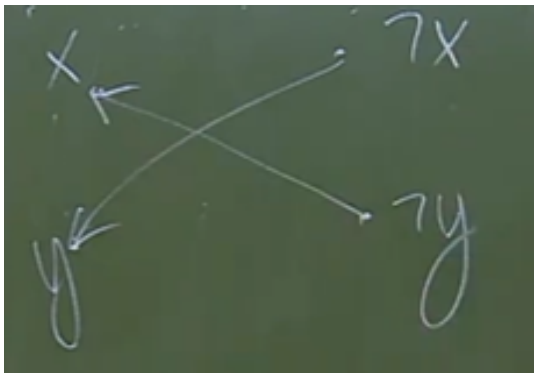
Утверждается, что формула выполнима тогда и только тогда, когда $\forall p$, p и ее отрицание не лежат в одной компоненте сильной связности. Почему это так?

$\to$ Пусть они лежат в одной компоненте сильной связности. Тогда какое значение может принимать p? Пусть единица. Тогда по построению все элементы КСС должны быть равны 1, тогда и $\overline{p} = 1$. Противоречие. Аналогично p не может принимать значение 0.

$\leftarrow$ Пусть выполнено условие. Покажем, как построить выполняющий набор для формулы. Вспомним алгоритм Косарайю. Он выделяет КСС в каком-то порядке. Пусть C(v) - номер КСС вершины v в этом порядке.
Набор:

- p $= 1 \iff C(p) > C(\overline{p})$
- p $= 0 \iff C(p) < C(\overline{p})$

Пусть этот набор не выполним, тогда существует ложный дизъюнкт $(x \vee y) = 0$, а значит, x $= 0$, y $= 0 \iff C(x) < C(\overline{x})$,$C(y) < C(\overline{y})$. Если у нас есть такой дизъюнкт, то в графе есть ребра
$\overline{x} \to y$
$\overline{y} \to x$

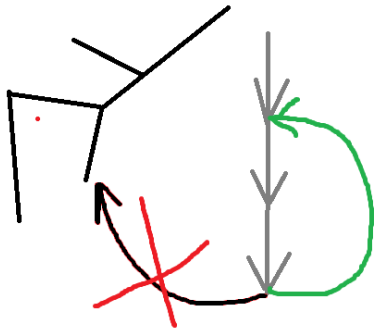


$tout[\overline{y}] \geqslant tout[x]$, а значит $C(\overline{y}) \leqslant C(x) < C(\overline{x}) \leqslant C(y)$ - противоречие

Асимптотика
O(n + m).
1. Строим граф с 2n вершинами и 2m ребрами
2. Запускаем алгоритм Косарайю
3. Узнаем номера КСС для всех пар (переменная, ее отрицание), проверяем, что они различны
4. Строим набор по праввилу, описанному в доказательстве выше

34 Алгоритм dfs на неориентированном графе. Дерево обхода dfs. Классификация рёбер на древесные и обратные. Проверка связности и ацикличности. Компоненты связности

В отличие от ориентированного графа, в неориентированном не будет ребер в черные вершины, поскольку на графе нет ориентации. Когда мы обходим граф, у нас могут быть только ребра в серые вершины.



Это - дерево dfs. Ребра, идущие в порядке обхода dfs будем называть *древесные ребра*, а те, что не были посещены dfs(они вели в серые вершины) - *обратные*

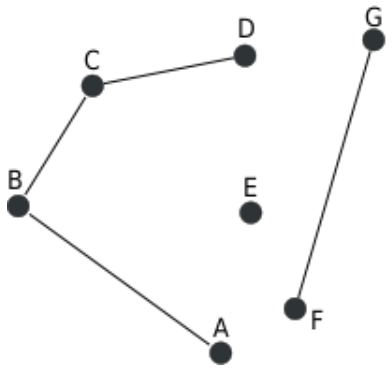
```
vector<vector<int>> g;
vector<bool> used;

void dfs(int v, int p = -1){
    used[v] = true;
    for(int to: g[v]){
        if(!used[to])
            dfs(to,v)
    }
}
```

Проверка на связность
Выберем какую-нибудь вершину в графе, запустим dfs из нее и проверим, что dfs обошел все вершины графа

Проверка на ацикличность
Будем искать цикл почти так же, как и в ориентированном случае, но теперь нас интересует ребро не в серую вершину, а в любую использованную, не являющуюся

непосредственным родителем текущей

Введем отношение $\sim$. u $\sim$ v, если между вершинами u и v есть путь. Очевидно, это отношение является отношением эквивалентности
Тогда граф распадается на классы эквивалентности, называемые *компонентами связности*
Определение *Компонента связности графа* G (или просто компонента графа G) — максимальный (по включению) связный подграф графа G.



Здесь 3 компоненты связности

35 Мосты, точки сочленения. Введение функции ret. Критерий того, что ребро является мостом.

Пусть G - связный граф
Определение Ребро $e \in E(G)$ называется *мостом*, если G-e (граф без ребра e) - несвязен

Определение Вершина v называется *точкой сочленения*, если G-v - несвязен

Введем функцию ret[v] = min(tin[v], tin[u]). Что такое u? Пусть дана вершина v, из которой мы спускаемся по древесным ребрам в вершину w. Тогда какая-то вершинка, в которую мы прыгнем по обратному ребру - вершина u

Для чего нам это надо. Рассмотрим ret[v]. Что значит, что ребро (u,v) - мост? Значит, мы не можем прыгнуть из области, куда мы спустились по этому ребру, куда-то выше. То есть ret[v] = tin[v]

Заметим, что если ребро не древесное, то оно точно не является мостом, так как мы просто удалили какое-то ребро из вершины в предка

Критерий e - мост $\Longleftrightarrow$ ret[v] $\geqslant$ tin[v]

▲

$\rightarrow$ Если ret[v] $<$ tin[v], то нашлась вершинка, в которую можно вернуться по обратным ребрам, если мы спустились ниже ребра (u,v) в дереве dfs, а значит, если убрать это ребро, найдется путь в вершинки ниже этого ребра из вершинок выше этого ребра, то есть связность не нарушится, тогда (u,v) - не мост

$\leftarrow$ Если ret[v] $\geqslant$ tin[v], то из вершин ниже ребра (u,v) нельзя вернуться в вершины, выше (u,v), а значит, удалив ребро (u,v), мы потеряем связность. Таким образом, (u,v) - мост ■

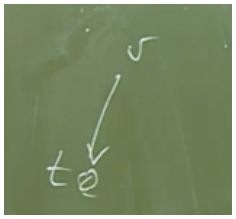
36 Насчёт ret в неориентированном графе, нахождение мостов

```
void dfs(int v, int p=-1){
    tin[v] =timer++;
    ret[v] = tin[v];
    used[v] = true;
    for(int to: g[v]){
        if(to == p) continue;
        if(used[to]){
            ret[v] = min(ret[v], tin[to]);
        }else{
            dfs(to,v);
             ret[v] = min(ret[v], ret[to]);
             if(ret[to] >=tin[to]) (v, to) - мост
```

```
        }
    }

}
```

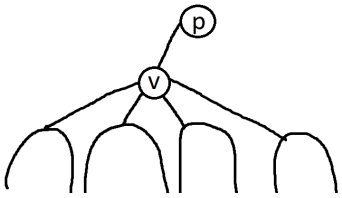
37 Нахождение точек сочленения в неориентированном графе.

Пусть v - вершина графа, **не являющаяся корнем**, (v, to) - какое-то древесное ребро из вершинки v.
v - точка сочленения $\iff$ ret[to] $\geqslant$ tin[v]



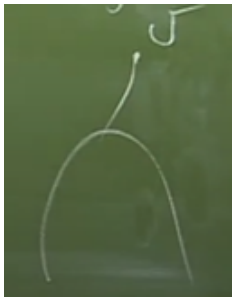
▲ $\leftarrow$ Это условие понятно интуитивно. Если при спуске ниже v лучшее, чего мы можем добиться - вернуться в v, то если мы исключим вершину v, не сможем вернуться из поддерева в предков v, а значит, связность нарушится

$\rightarrow$ предположим, ни для какого сына неравенство не выполянется. Тогда ret[to] $<$ tin[v], а значит из каждого поддерева можно вернуться выше вершинки v, то есть v - не точка сочленения



■
Теперь пусть v - вершина графа, **являющаяся корнем**.

Тогда v - точка сочленения $\iff$ из v выходит хотябы два древесных ребра
▲ $\leftarrow$ Между различными поддеревьями нет ребер, так как если бы ребро было, то эти два поддерева склеились бы в одно. Значит, после удаления v поддеревья станут независимыми компонентами связности графа, а значит, v - точка сочленения.
$\rightarrow$ Пусть это не так, и у вершины всего один сын.



Значит, удаление вершины v оставляет поддерево связной компонентой. (dfs запустился, как-то походил-походил и обошел все дерево, задействуя только одно ребро из v). Тогда v - не точка сочленения. Противоречие ■

```
void dfs(int v, int p=-1){
    tin[v] = timer++;
    ret[v] = tin[v];
    used[v] = true;
    int cld = 0;

    for(int to:g[v]){
        if(to == p) continue;

        if(used[to]){
            ret[v] = min(ret[v], tin[to]);
        }else{
            cld++;
            dfs(to,v);
            ret[v] = min(ret[v], ret[to]);
            if(p != -1){
                if(ret[to] >= tin[v]) v - точка сочленения
            }else{
                if(cld >=2) v - точка сочленения
            }
        }
    }
}
```

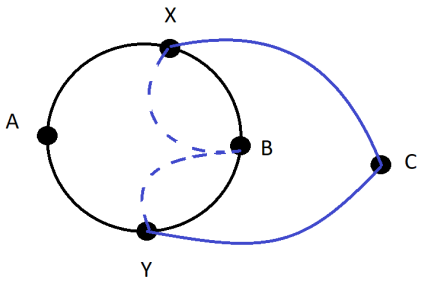
38 Понятие рёберной двусвязности. Отношение эквивалентности.

Определение. Две вершины реберно двусвязны, если между ними существуют два пути без общих ребер.

Нетрудно показать, что отношение реберной двусвязности является отношением эквивалентности. Действительно, рефлексивность и симметричность очевидны из определения.

Докажем транзитивность $a \sim b, b \sim c \Rightarrow a \sim c$ **:**

Доказательство. Рассмотрим два пути из c в b. Найдём их места первого пересечения с циклом $a \to b \to a$, получили вершины x и y. Тогда есть рёберно не пересекающиеся пути $a \to x \to c$, $a \to y \to c$.



□

В этом случае введем понятие ниже.

Определение. Компонентами рёберной двусвязности графа называют его подграфы, множества вершин которых — классы эквивалентности рёберной двусвязности, а множества рёбер — множества ребер из соответствующих классов эквивалентности.

Определение. Мост — ребро, при удалении которого число компонент связности увеличивается.

39 Выделение компонент рёберной двусвязности в неориентированном графе. Древесность графа со сжатыми компонентами рёберной двусвязности.

Теорема 39.1. *Заметим, что если удалить из графа все мосты, то компоненты связности в полученном графе будут соответствовать компонентам реберной двусвязности в исходном.*

Доказательство. 1) Если на пути между какими-то двумя вершинами в исходном графе есть мост, то их придется отнести к разным компонентам реберной двусвязности (иначе между ними есть два пути без общих ребер, но тогда между ними есть путь, который не проходит через наш мост, а отсюда следует, что наш мост не является мостом).

2) Рассмотрим древесное ребро $(v; to)$. То есть такое, что в процессе обхода графа алгоритмом dfs вершина v является родителем вершины to (в первый раз, когда мы увидели to). Если ребро $(v; to)$ не является мостом, то v и to реберно двусвязны.

Докажем это. Т.к. ребро не является мостом, то верно следующее неравенство: $ret[to] \leqslant tin[v]$. То есть мы можем прыгнуть из поддерева to в v или куда-то выше. Тогда между v и to есть два реберно непересекающихся пути. (Один – $v \to to$, второй – спустимся в поддерево to, совершим прыжок в v или его предка, потом спустимся в v)

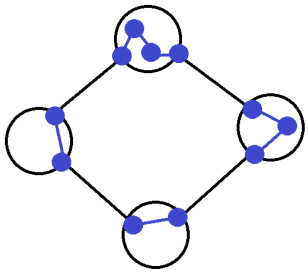
А, значит, все компоненты связности в полученном графе являются компонентами реберной двусвязности в исходном. □

Как следствие этой теоремы получаем эквивалентное определение моста:

Определение. Мост — ребро, соединяющее две компоненты реберной двусвязности.

Теорема 39.2. *Если сжать все комоненты реберной двусвязности, то получится граф без циклов. А если граф изначально связен, то это дерево.*

Доказательство. Пусть между компонентам реберной двусвязности есть простой цикл. Тогда можно построить и цикл, проходящий по вершинам исходного графа и содержащий ребра взятого нами цикла.



Но тогда все эти вершины лежат в одной компоненте реберной двусвязности. Получаем противоречие. □

40 Определение эйлерова цикла. Критерий наличия эйлерова цикла в неориентированном графе.

Определение. Вершина называется изолированной, если ее степень равна 0. В случае неориентированного графа это равносильно тому, что из нее не выходит ни одного ребра. В случае ориентированного графа это равносильно тому, что количество входящих ребер и количество исходящих ребер равно 0.

Определение. Эйлеров путь — путь, проходящий по всем рёбрам графа и притом только по одному разу.

Определение. Эйлеров цикл — цикл, проходящий через каждое ребро графа ровно по одному разу.

Определение. Граф эйлеров, если в нем есть эйлеров цикл.

Теорема 40.1 (Критерий эйлеровости неориентированного графа)**.** *Неориентированный граф, который становится связным после удаления всех изолированных вершин, является эйлеровым тогда и только тогда, когда все его вершины имеют четную степень.*

Доказательство. $\Rightarrow$

Рассмотрим эйлеров обход графа. Заметим, что при попадании в вершину и при выходе из нее мы уменьшаем ее степень на два (помечаем уже пройденые ребра). Кроме того, для

стартовой вершины мы уменьшаем ее степень на один в начале обхода эйлерова цикла, и на один при завершении. Следовательно вершин с нечетной степенью быть не может.

$\Leftarrow$

Необходимость мы доказали ранее. Докажем достаточность, используя индукцию по числу вершин n.

База индукции: $n = 0$ цикл существует.

Предположим что граф, имеющий менее n вершин, степени вершин которого четны, содержит эйлеров цикл.

Рассмотрим граф G с $n > 0$ вершинами, степени которых четны. Удалим изолированные вершины. Если такие были, воспользуемся утверждением идукции. В противном случае у нас все еще n вершины, однако теперь мы можем гарантировать, что граф связен.

Пусть v_1 — вершина графа. Начнем идти из этой вершины по ребрам графа до тех пор, пока можем (проходим по каждому ребру не более 1 раза). Если в какой-то момент мы не можем пойти дальше, выведем нашу вершину.

```
void euler(int v) {
  while из( v есть хотя бы одно неиспользованное ребро){
    пусть (v;u) - ребро, пометим его использованным
    euler(u)
  }
  print(v)
}
```

Заметим, что у вершин, которые встречаются в процессе обхода, степень каждый раз уменьшается на 2. Ведь если мы вошли в вершину, и она не стартовая, то в этот момент из нее ведет нечетное число непосещенных ребер. Тогда из нее можно выйти.

Поэтому первая выведенная нами вершина будет стартовая, то есть v_1. Кроме того, мы получим замкнутый путь (цикл), который начинается и заканчивается в вершине v_1.

Назовем этот цикл C_1. Если C_1 является эйлеровым циклом для G, тогда доказательство закончено. Если нет, то пусть G_2 — подграф графа G, полученный удалением всех рёбер, принадлежащих C_1. Поскольку C_1 содержит чётное число рёбер, инцидентных каждой вершине, то каждая вершина подграфа G_2 имеет чётную степень. Этот граф разбивается на некоторое количество компонент связности.

Рассмотрим какую-либо компоненту связности G_2 (не состоящую из изолированной вершины). Поскольку рассматриваемая компонента связности G_2 имеет менее, чем n вершин (как минимум, туда не входит v_1, ведь все ее ребра были удалены), а у каждой вершины графа G_2 чётная степень, то у каждой компоненты связности G_2 существует эйлеров цикл. Пусть для рассматриваемой компоненты связноти это цикл C_2. У C_1 и C_2 имеется общая вершина a, так как G связен. Давайте объединим C_1 и C_2 в новый цикл. Для этого нужно, начиная с вершины a, обойти C_1, вернуться в a, затем пройти по C_2 и вернуться в a. Если новый эйлеров цикл не является эйлеровым циклом для G, продолжаем использовать этот процесс, расширяя наш цикл, пока, в конце концов, не получим эйлеров цикл для G.

□

41 Реализация алгоритма поиска эйлерова цикла.

```
struct edge{
  int from, to;
}

vector <edge> edges;
\\edges[2k] и edges[2k + 1] -- две половинки одного неориентированного ребра
\\ удобно получать вторую половинку ребра, беря ind^1
vector <bool> used \\used[e] -- использовано ли ребро
vector <int> ptr; \\"указатель" на следующее неиспользованное ребро
vector <vector <int> > g; наш\\ граф в виде списков смежности, но уже с
    ориентированными ребрами

void euler(int v) {
  while(ptr[v] != g[v].size()) {
    int e = g[v][ptr[v]]; \\g[v] -- список номеров ребер
    if (used[e]){++ptr[v]; continue;}
    int u = edges[e].to;
    used[e] = used[e^1] = true;
    ++ptr[v];
    euler(u);
  }
  cout << v << " ";
}
```

Асимптотика $O(n+m)$

42 Критерий наличия эйлерова пути в неориентированном графе.

Теорема 42.1. *Граф G (связный, если убрать изолированные вершины) содержит эйлеров путь тогда и только тогда, когда количество вершин с нечетной степенью меньше или равно двум.*

Доказательство. В одну сторону, очевидно, верно. Если у нас есть путь, то у вершин, отличных от концов, степень четна, а у начальной и конечной вершины степень нечетна.

В другую сторону. Если выполняется условие на степени вершин, то добавим ребро между двумя вершинами с нечетной степенью. В полученном графе есть эйлеров цикл, удаление из которого добавленного ребра даст эйлеров путь.

В этом доказательстве мы не рассмотрели случай, когда есть ровно одна вершина с нечетной степенью. Однако такой случай невозможен. По лемме о рукопожатиях в любом графе четное число вершин с нечетной степенью. А один – нечетное число. □

Теорема 42.2 (Лемма о рукопожатиях)**.** *В любом неориентированном графе число вершин с нечетной степенью четно.*

Доказательство. Действительно. Сложим степени всех вершин. Получим некоторое число. Заметим, что оно равно удвоенному числу ребер, ведь каждое ребро в нашей сумме учитывалось два раза – по одному от каждого из его концов. Отсюда следует, что количество вершин с нечетной степенью четно. □

43 Критерий наличия эйлерова цикла в ориентированном графе.

Теорема 43.1 (Критерий эйлеровости ориентированного графа)**.** *Ориентированный граф G (который становится сильно связным при удалении изолированных вершин) эйлеров*

тогда и только тогда, когда для каждой вершины верно, что входная степень равна выходной.

Доказательство. Абсолютно аналогично доказательству для неориентированного графа. □

44 Определение кратчайшего расстояния в невзвешенном/взвешенном графе.

Определение. Если $G = (V, E)$ – невзвешенный граф, то кратчайшее расстояние между u и v:

$$dist(u, v) = \begin{cases} min \text{ число ребер в пути от } u \text{ до } v & \text{если между этими вершинами есть путь} \\ +\infty & \text{если пути между вершинами нет} \end{cases}$$

Определение. Взвешенный граф $G = (V, E, w)$, где $w : E \to \mathbb{R}$ – весовая функция.

Определение. G – взвешенный граф. Тогда $dist(u, v) = min$ сумма весов на пути от u до v (и снова $+\infty$, если пути нет).

45 Поиск в ширину: алгоритм bfs с доказательством корректности.

Этот алгоритм находит кратчайшие расстояния от заданной вершины s до всех вершин в невзвешенном графе.

```
vector <vector <int> > g;
vector <int> dist;
dist.assign(n, inf);
dist[s] = 0;
queue <int> q;
q.push(s);

while(!q.empty()) {
  int v = q.front();
  q.pop();
  for(int to : g[v]) {
    if (dist[to] != inf) continue;
    dist[to] = dist[v] + 1;
    q.push(to);
  }
}
```

Асимптотика $O(n + m)$ т.к. каждая вершина побывает в очереди $\leqslant 1$ раз.

Будем доказывать по индукции чуть более сильное утверждение (чем корректность), состоящее из трех частей.

Теорема 45.1. *Пусть* $dist[q.front()] = k$

1) Тогда q *имеет вид* $k, \ldots, k, k+1, \ldots, k+1$

2) Все $v : dist(s, v) \leqslant k$ *либо лежат в очереди, либо уже удалены из нее*

3) Если алгоритм выполнил присваивание $dist[w] = m$, *то* $dist(s, w) = m$

Доказательство. База очевидна. Поэтому докажем сразу переход.

Пусть очередь имеет нужный вид. Достаем первую вершину из очереди – v с $dist[v] = k$. Тогда в конец очереди мы добавим вершины с $dist = k + 1$ просто исходя из работы алгоритма. Значит, очередь будет по-прежнему иметь нужный вид. Мы доказали переход для 1 утв.

Докажем 3. Заметим, что для всех добавленных вершин мы посчитали расстояние верно.

Действительно, если мы рассматриваем ребро $v \to to$, то $dist[to] \leqslant k + 1$. Докажем теперь, что меньше, чем $k+1$ быть не может. Пусть это не так. Тогда по второму предположению индукции вершина to уже была обработана, а, значит, не могла быть добавлена в очередь на текущем шаге (потому что уже лежала там или даже была удалена).

Докажем 2. Заметим, что если предыдущая удаленная вершина, как и текущая, имела $dist = k$, то утверждение 2 остается верным и для текущей вершины. Поэтому единственный осмысленный случай – у предыдущей вершины $dist = k$, а у текущей $k + 1$.

Докажем, что все вершины с $dist = k+1$ лежат в очереди. Пусть это не так. Тогда есть вершина u с $dist(s, u) = k+1$. Пусть соседняя с ней вершина в этом пути – это вершина p. Тогда $dist(s, p) = k$, и по предположению индукции, p в какой-то момент была удалена из очереди (или она была удалена последней, или до этого). А, значит, u должна была быть добавлена в очередь. Противоречие. Значит, наше предположение неверно. □

46 Алгоритм $0 - K - bfs$.

Считаем, что наша весовая функция $w : E \to \{0, \dots, k\}$.

Если k не очень большое, то можно модифицировать наш bfs.

Заметим, что $\forall v : dist(s, v) \leqslant k \cdot (n - 1) \leqslant kn$, где n – число вершин.

Заводим массив из $kn + 1$ очереди. $q[d]$ – очередь вершин, находящихся на расстоянии d от s. А дальше проходимся по нашим очередям в порядке увеличения расстояния и действуем аналогично обычному bfs.

```
vector <int> dist(n, inf)
vector <bool> used(n, false)
dist[s] = 0
q[0].push(s)
for (int d = 0; d <= kn; ++d){
  while(!q[d].empty()){
    int v = q[d].front();
    q[d].pop(); if (used[v]) continue;
    used[v] = true;
    for (edge &e : g[v]) {
      int to = e.to;
      if (dist[to] <= dist[v] + e.w) continue;
      dist[to] = dist[v] + e.w;
      q[dist[to]].push(to);
    }
  }
}
```

Асимптотика $O(kn + m)$.

47 Двусторонний bfs

Ищем $dist(s, t)$ для заранее заданных s и t.

Идея: с помощью bfs ищем все вершины на расстоянии 1 от s, с помощью bfs ищем вершины на расстоянии 1 от t (на графе с "обратными"ребрами), ищем вершины на расстоянии 2 от s и вершины на расстоянии 2 от t. Будем так продолжать, пока не возникнет какая-то вершина, которая встретилась и для s, и для t. По сути, мы как будто параллельно запускаем bfs в s и t.

Пусть mid – первая точка, которая будет найдена с обеих сторон. Докажем, что через нее проходит кратчайший путь. Пусть это не так. Тогда есть какой-то путь между s и t длины меньше, чем $dist(s, mid) + dist(mid, t)$. Заметим также, что $dist(s, mid)$ и $dist(mid, t)$ отличаются не более, чем на 1 в силу нашего алгоритма. Возьмем вершину v на расстоянии $dist(s, mid) - 1$ от s. Тогда $dist(v, t) \leqslant dist(mid, t)$. Но тогда mid – не первая вершина, которая встретилась и у s, и у t. Противоречие. Значит, наше предположение неверно.

В общем случае асимптотика как у обычного bfs, но в частных случаях он выгоднее. Например, когда количество вершин на каждом уровне растет экспоненциально (например, из каждой вершины исходит по 3 ребра, или что-то в этом духе).

48 Алгоритм Дейкстры. Условия применимости, доказательство корректности. Реализации за $O(n^2), O(m \log n)$, $O(m + n \log n)$.

Алгоритм Дейкстры ищет кратчайшие пути в графе от заданной вершины до всех, если веса всех ребер неотрицательны.

В алгоритме поддерживается множество вершин U (использованных вершин), для которых уже вычислены длины кратчайших путей до них из s. На каждой итерации основного цикла выбирается вершина $u \notin U$, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина u добавляется в множество U и производится релаксация по всем исходящим из неё рёбрам. Релаксация — процесс, в котором мы обновляем расстояния до вершин, смежных с u и не принадлежащих U.

```
dist.assign(n, inf)
dist[s] = 0
while( есть неиспользованные вершины) {
  v - неисп. вершина с min dist[v]
  раскрываем v : dist[to] = min(dist[to], dist[v] + cost(v, to))
  помечаем v использованной
}
```

По сути, $0 - k - bfs$ работает точно так же.

В такой наивной реализации асимптотика $O(n^2)$, если искать минимальную неиспользованную вершину за линию.

Можно добиться $O(m \log n)$, используя двоичную кучу. А что нам, в сущности, нужно? Искать минимум на неиспользованных вершинах и уменьшать значения (extract min и decrease key). Куча такое как раз умеет.

С фиб. кучей асимптотика будет даже лучше. Будет $O(m + n \log n)$, поскольку в фиб куче decrease key выполняется амортизированно за 1.

Теорема 48.1 (Корректность)**.** *Докажем, что если v раскрываемая вершина, то $dist[v] = dist(s, v)$. Отсюда будет следовать корректность.*

Доказательство. Пусть это не так. Найдем первую вершину v, для которой это не выполняется. Это может быть только в случае, если $dist[v] > dist(s, v)$. Тогда исходя из алгоритма $dist[u] \geqslant dist[v]$ для всех неиспользованных u.

Как может выглядеть кратчайший путь от s до v? Он выглядит так. Мы сначала ходим по использованным вершинам, а потом прыгаем в некоторую вершину u из неиспользованных. А дальше идем в v. Пусть $u = v$. Тогда в этом случае такой путь должен был быть учтен в $dist[v]$, когда мы раскрывали вершину, из которой и совершается прыжок. Поэтому получаем, что кратчайший путь до v так выглядеть не может.

Значит, $u \neq v$. Тогда (исходя из предыдущих рассуждений) $dist(s, u) = dist[u] \geqslant dist[v] > dist(s, v) > dist(s, u)$. (Ведь $dist(s, v) = dist(s, u) + dist(u, v)$). Получаем противоречие. Ура. □

49 Двусторонний алгоритм Дейкстры. Завершение алгоритма: почему достаточно реализовать алгоритм, почему нельзя обойтись меньшим числом действий (пример).

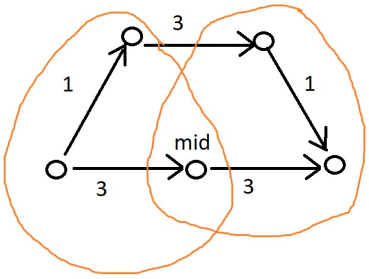
Нужно найти кратчайшее расстояние между s и t. Идея такая же, как в двустороннем bfs. Одновременно идем алгоритмом из вершины s и из вершины t (только в случае вершины t снова рассматриваем обратные ребра). Как только встретится общаа вершина, радуемся.

Собственно, для этих целей нам нужно завести две кучи – для s и для t. Каждый раз будем выбирать минимум из двух куч. Если минимум в куче вершины s, то делаем шаг алгоритма Дейкстры для нее, иначе – для t. Останавливаемся, когда какая-то вершина будет удалена из обеих куч.

Тонкость этого алгоритма заключается в том, что кратчайший путь $s \to t$ не обязательно пройдёт через вершину mid (которая будет удалена из обееих куч). Поэтому после остановки двунаправленного поиска, нам необходимо перебрать некоторое количество ребер. И ответ в этом случае будет $min\ dist_s[u] + cost(u, v) + dist_t[v]$ по ребрам $(u, v) \in E$, где имеет смысл перебирать только ребра между теми вершинами, которые были посещены во время нашего обхода (u посещалась алгоритмом для s, а v – для t).

На практике, такой двунаправленный поиск быстрее обычного алгоритма Дейкстры примерно в два раза.

Пример. Почему через mid не всегда проходит нужный путь.



Почему достаточно перебрать такие ребра.

Пусть $s \to u \to v \to t$ – путь, который нашел наш алгоритм. Где u – из "облачка"вершины s, а v – из "облачка"t. (Облачко – посещенные вершины).

mid – пересечение облачков.

Пусть мы не учли какой-то путь. Тогда он выглядится так. Мы пошли по вершинам из облачка s, дальше походили по вершинам, не принадлежащим никакому облачку, потом походили как-то еще, а потом прошлись по вершинам из облачка t. Пусть x и y – первая и последняя вершны не из облачков.

Тогда $dist(s,x) \geqslant dist(s,mid)$, $dist(y,t) \geqslant dist(mid,t)$. Значит, на самом деле, такой путь нам неинтересен.

50 Алгоритм A^*, определение функций f, g, h; реализация

Применение: нахождение минимального пути от вершины s до вершины t в графе с неотрицательными рёбрами.

A^* работает по принципу алгоритма Дейкстры.
$g(v)$ – текущее кратчайшее расстояние от s до v (из алгоритма Дейкстры)

$h(v)$ – некоторая оценка на $dist(v,t)$. ($h(v)$ называют *эвристикой*)
$f(v) = g(v) + h(v)$

Смысл алгоритма: следующую вершину выбираем не по $g(v)$ (как в алгоритме Дейкстры), а по $f(v)$.

Реализация (псевдокод):

q – очередь, сравнение элементов по функции f. $g[s] = 0$ – задали базу для функции g
gr – граф, заданный списком рёбер.

```
1
2  q.insert(s); // добавляем стартовую вершину
3
4  while(!q.empty()) {
5  v = q.top();
6  q.pop();
7
8  for (edge e : gr[v]) {
9      c = g[v] + e.cost + h(to);
10     if (c < f[to]) {
11         g[e.to] = g[v] + e.cost;
12         if (to ∈ q) {
13             q.decreaseKey(...); // уменьшить у вершины e.to значение f до
       g[e.to] + h(e.to)
14         } else {
15             q.insert(...); // добавить вершину e.to со значением f = g[e.to] + h(e.to)
16         }
17     }
```

```
18 }
```

51 Вырожденные случаи в алгоритме A^*: $h \equiv 0$, $h(v) = dist(v, t)$

1) $h \equiv 0$. Получается, что сравнение идёт только по $g(v)$, а значит алгоритм A^* выраждается в алгоритм Дейкстры.

2) $h(v) = dist(v, t)$ – эвристика всегда каким-то образом знает точное расстояние от v до t.
Тогда A^* рассматривает почти только оптимальный путь (почти только означает, что алгоритм рассматривает не только вершины, принадлежащие кратчайшему пути, но и их соседей).

Доказательство. Докажем сначала, что если h – монотонна, то значения f в куче не убывают (у извлекаемых элементов).
Пусть мы раскрываем вершину v. Рассмотрим ребро (v, u).
$g(u) = g(v) + cost(v, u)$
С другой стороны, так как h – монотонна, то $h(v) \leqslant h(u) + cost(v, u)$, то есть $h(u) \geqslant h(v) - cost(v, u)$. Сложим равенство и неравенство, получим:

$$g(u) + h(u) = f(u) \geqslant g(v) + h(v) = f(v)$$

Доказано.
Тогда получается, что каждая вершина раскроется не более одного раза. (Предположим противное, тогда получаем, что во второй раз мы её раскрыли со значением f меньшим, чем в первый раз. Противоречие.

□

Наша эвристика такая, что значения f одинаковы для всех вершин, а значит каждая вершина раскроется не более одного раза. По этому утверждению и принципу работы алгоритма понятно, что он рассматривается почти только оптимальный путь.

52 Допустимые и монотонные эвристики в алгоритме A^*. Примеры монотонных эвристик на разных сетках.

Определение 52.1. Эвристика $h(v)$ называется *допустимой*, если $\forall v\ h(v) \leqslant dist(v, t)$.

Следствие. *Если $h(v)$ – допустимая эвристика, то $h(t) = 0$.*

Замечание (Дополнительно)**.** Если эвристика $h(v)$ – недопустимая, то A^* находит неточный ответ, на практике с помощью недопустимых эвристик можно добиться, чтобы алгоритм работал быстро, при этом ответ не сильно отличался от правильного.

Определение 52.2. Эвристика $h(v)$ называется *монотонной*, если:
1) $h(t) = 0$ 2) $\forall (u, v) \in E\ h(u) \leqslant h(v) + cost(u, v)$, где E – множество рёбер (неравенство треугольника).

Следствие. *Монотонная эвристика является допустимой.*

Примеры эвристик

1) Если граф представляет собой неполную сетку (то есть из каждой точки потенциально есть 4 ребра: вверх, вниз, вправо, влево), при этом некоротых рёбер может не быть (то есть не у всех точек есть ровно 4 ребра), то в качестве эвристики можно использовать Манхэттенское расстояние: $h(v) = |v.x - t.x| + |v.y - t.y|$ (каждую вершину можно задать двумя координатами на плоскости).

2) Если в графе также можно ходить по диагонали, то в качестве эвристики можно использовать расстояние Чебышёва: $h(v) = max\{|v.x - t.x|, |v.y = t.y|\}$

3) Если можно ходить по плоскости куда угодно, то в качестве эвристики можно использовать Евклидово расстояние: $h(v) = \sqrt{(v.x - t.x)^2 + (v.y - t.y)^2}$.

53 Формулировка работоспособности (корректность и время работы) алгоритма A^* в случае монотонной, допустимой или произвольной эвристики. Доказательство для монотонного случая.

В случае произвольной эвристики A^* может довольно быстро найти хорошее приближение.

В случае монотонной и допустимой эвристики алгоритм находит точный минимальный путь, при этом в случае монотонной эвристики алгоритм раскрывает каждую вершину не более 1 раза, в случае допустимой эвристики может работать довольно долго.

Доказательство (для монотонной эвристики). Пусть h – монотонная эвристика. Алгоритм A^* на каждом шаге раскрывает вершину с минимальным значением f, при этом. Когда алгоритм дойдет до вершины t, он извлечет её из кучи со значением $f(t)$, но $f(t) = g(t) + h(t) = g(t)$ ($h(t) = 0$ по определению монотонной эвристики). Осталось доказать, что $g(t)$ – не только оценка сверху для минимального пути, но и в точности равна ей. Пусть оптимальный путь $OPT < f(t)$. Тогда бы он извлёкся из кучи раньше, так как значения f у извлекаемых элементов не убывают (доказано в пункте 51). Противоречие. □

54 Алгоритм Флойда: поиск попарных кратчайших расстояний в графе без отрицательных циклов. Реализация, асимптотика.

Применение: поиск попарных кратчайших расстояний.

Требования к графу: допускаются отрицательные рёбра, но нет отрицательных циклов.

Описание алгоритма:

Введём трехмерную динамику:
$dp[i][j][k]$ – минимальная длина пути между i и j, такая, что все промежуточные вершины имеют номера не больше, чем k.

База:
$dp[i][j][0]$ = вес ребра из i в j ($+\infty$, если ребра нет)

Переход:
Пусть посчитаны первые k слоёв. Построим $k+1$ слой.
$dp[i][j][k+1] = min(dp[i][j][k], dp[i][k+1][k] + dp[k+1][j][k])$, то есть либо не берём $k+1$ вершину, либо берем.

Ответ: $dp[i][j][n]$, где n – количество вершин.

Асимптотика: $O(n^3)$
Память: $O(n^3)$

Память можно оптимизировать до $O(n^2)$, если в качестве dp использовать матрицу смежности графа:

Пусть g – матрица смежности графа. Тогда алгоритм Флойда можно написать так:

```
for (k = 1 ... n) {
    for (i = 1 ... n) {
        for (j = 1 ... n) {
            g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
        }
    }
}

```

55 Восстановление ответа (пути) в алгоритме Флойда.

Заведем массив $p[i][j]$, заполним его каким-то нейтральным элементом (например, -1). Если на каком-то шаге выгоднее идти через вершину k, то есть $dp[i][j][k+1] == dp[i][k+1][k] + dp[k+1][j][k]$, то сохраняем это: $p[i][j] = k$.

Теперь найти путь между i и j можно рекурсивно:
Если на какой-то этапе $dp[i][j] == -1$, то кратчайший путь – ребро между i и j.
Если $dp[i][j] = k \neq -1$, то путь из i в j – это путь из i в k и из k в j (их находим рекурсивно).

56 Алгоритм Форда—Беллмана: поиск кратчайших расстояний от одной вершины до всех. Реализация, асимптотика (в случае отсутствия отрицательных циклов).

Применение: нахождение минимального пути от вершины s до вершины t (в графе могут быть и отрицательные рёбра, и отрицательные циклы).

Описание алгоритма:

Введём двумерную динамику:
$dp[v][k]$ – кратчайшее расстояние от s до v при использовании не более, чем k рёбер.

База: $dp[v][0] = \begin{cases} 0, v = s \\ +\infty, v \neq s \end{cases}$

Переход:
Пусть посчитаны первые k слоёв. Тогда $dp[u][k+1] = \min(dp[u][k], \min\limits_{(v,u)} dp[v][k] + cost(v,u))$, то есть либо оставляем ответ, достижимый за k рёбер, либо перебираем все входящие в u рёбра, и выбираем из них минимальное.

Ответ:
dp[v][n - 1], если нет отрицательных циклов

Асимптотика: $O(nm)$, n – количество слоёв в dp, m – переход между слоями.

57 Алгоритм Форда—Беллмана: нахождение кратчайших расстояний от одной вершины до всех в случае наличия отрицательных циклов.

Для обработки отрицательных циклов найдём дополнительно n-ый слой $dp[v][n]$.

Утверждение 57.1. *Пусть C – отрицательный цикл, достижимый из s. Тогда $\exists v \in C : dp[v][n] < dp[v][n-1]$.*

Доказательство. Пусть $c_1, ..., c_k$ – веса рёбер в цикле C. Предположим противное: $\forall v \in C dp[v][n] = dp[v][n-1]$ (больше оно быть не может по переходу динамики).
Пусть c_i – вес ребра, соединяющего вершины v_i и v_{i+1}. Тогда $dp[v_{i+1}][n] \leqslant dp[v_i][n-1] + c_i$. Сложим эти неравенства по всем i.

$$\sum_{v_i \in C} dp[v_i][n] \leqslant \sum_{v_i \in C} do[v_i][n-1] + \sum_i c_i$$

Первая и вторая суммы равны по предположению, третья сумма меньше нуля (так как цикл отрицательный). Получается, что 0 меньше или равен чего-то отрицательного. Противоречие. □

Тогда за $O(nm)$ можем найти хотя бы по одной вершине на каждом отрицательном цикле.

$$dist(s,x) = \begin{cases} -\infty, \text{если } x \text{ достижим из одной из вершин отрицательного цикла} \\ dp[x][n-1], \text{иначе} \end{cases}$$

На практике лучше запустить DFS из всех вершин, у которых $dp[v][n] < dp[v][n-1]$, и пометить все достижимые из них вершины $dist = -\infty$.

58 Остовный подграф, остовное дерево. Минимальный остов. Лемма о безопасном ребре.

Определение. $T \subset G$ — **остовное дерево**, если T содержит все вершины G и является деревом

Определение. Остовный подграф — подграф, содержащий все вершины.

Определение. Минимальным остовным деревом во взвешенном неориентированном графе называется остовное дерево минимального веса.

Определение. Ребро $(u, v) \notin G'$ называется безопасным, если при добавлении его в G', $G' \cup \{(u, v)\}$ также является подграфом некоторого минимального остовного дерева графа G.

Определение. (S,T) — разрез, если $S \cup T = V, S \cap T = \varnothing$

Определение. (u,v) пересекает разрез (S,T), если u и v — в разных частях разреза.

Теорема 58.1 (О безопасном ребре)**.** *Рассмотрим связный неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \to \mathbb{R}$. Пусть $G' = (V, E')$ — подграф некоторого минимального остовного дерева G, $\langle S, T\rangle$ — разрез G, такой, что ни одно ребро из E' не пересекает разрез, а (u, v) — ребро минимального веса среди всех ребер, пересекающих разрез $\langle S, T\rangle$. Тогда ребро $e = (u, v)$ является безопасным для G'.*

Доказательство. Достроим E' до некоторого минимального остовного дерева, обозначим его T_{min}. Если ребро $e \in T_{min}$, то лемма доказана, поэтому рассмотрим случай, когда ребро $e \notin T_{min}$. Рассмотрим путь в T_{min} от вершины u до вершины v. Так как эти вершины принадлежат разным долям разреза, то хотя бы одно ребро пути пересекает разрез, назовем его e'. По условию леммы $w(e) \leqslant w(e')$. Заменим ребро e' в T_{min} на ребро e. Полученное дерево также является минимальным остовным деревом графа G, поскольку все вершины G по-прежнему связаны и вес дерева не увеличился. Следовательно $E' \cup \{e\}$ можно дополнить до минимального остовного дерева в графе G, то есть ребро e — безопасное. □

59 Алгоритм Прима: доказательство корректности и реализации за $O(n^2), O(m\ log\ n), O(m + n\ log\ n)$.

Алгоритм Прима – алгоритм поиска минимального остовного дерева.

Идея: пока можем, добавляем к имеющемуся подграфу (C) минимального остовного дерева самое легкое ребро из всех, пересекающих разрез (соединяющих наш подграф с оставшейся частью графа).

В таком виде асимптотика алгоритма $O(nm)$, однако, ее несложно улучшить до $O(n^2)$. А именно для каждой вершины будем хранить минимальное ребро, которое соединяет ее с вершиной из C.

Тогда шагов будет $n - 1$ (в дереве $n - 1$ ребро), на каждом шаге $O(n)$ действий по поиску следующего ребра. Обновление происходит так: для текущей добавленной вершины

достаточно обновить минимум по всем ее соседям, не входящим в C. Так что асимптотика будет $O(n^2 + m) = O(n^2)$

Заметим, что есть сходство с алгоритмом Дейкстры. Нам нужно делать extract min, а потом decrease key, причем, O(n) раз. Так что мы можем, используя, например, кучу, добиться лучшей асимптотики.

- ▷ Выбираем произвольную стартовую вершину – начало строящегося дерева. Строим кучу с вершинами, до которых есть ребро из стартовой.
- ▷ Итеративно добавляем к дереву вершину из кучи с минимальным по весу ребром до дерева. Саму вершину удаляем из кучи. Добавляем новые вершины, доступные из добавленной вершины по ребрам графа, в кучу. А еще обновляем расстояние до вершины из куче $\Big($удаляем пару (старое расстояние; номер вершины), добавляем пару (новое расстояние; номер вершины)$\Big)$, если от добавленной вершины к ней ведет более легкое ребро.

Теорема 59.1. *Алгоритм Прима работает за $O(m\ log\ n)$ с очередью на куче и за $O(n\ log\ n + m)$ при очереди на Фибонначиевой куче.*

Доказательство.

- ▷ Извлечение вершины из кучи – $O(log\ n)$
- ▷ Вставка в кучу $O(log\ n)$

Пункт 1) выполнится n раз, пункт 2) m раз.

□

60 Система непересекающихся множеств (СНМ). Виды запросов. Эвристика по рангу, эвристика сжатия путей. Асимптотика ответа на запрос при использовании обеих эвристик (б/д).

Определение. DSU (CHM) — система непересекающихся множеств

- ▷ Create(u) — создать множество с одним элементом u.
- ▷ Find(u) — найти множество по элементу u, чтобы можно было сравнить их (Find(u) == Find(v)). Для удобства возвращает конкретного представителя множества как своеобразный идентификатор.
- ▷ Union(u, v) — объединить два множества, одно из которых содержит элемент u, а другое — элемент v.

Удобно воспринимать множества как подвешенные деревья. Корень – представитель или, другими словами, лидер. Соответственно, поиск представителя осуществляется как подъем по дереву к корню. Поэтому чем короче путь, тем лучше. Массив *parent* хранит родителей вершин в наших графах. Значения *parent* для представителей равно -1.

Определение. В операции Union будем присоединять дерево с меньшим рангом к дереву с большим рангом. Это называется ранговая эвристика. Есть два варианта ранговой

эвристики: в одном варианте рангом дерева называется количество вершин в нём, в другом — глубина дерева (точнее, верхняя граница на глубину дерева, поскольку при совместном применении эвристики сжатия путей реальная глубина дерева может уменьшаться).

Определение. Эвристика сжатия пути заключается в следующем: когда после вызова Find(v) мы найдём искомого лидера p множества, то запомним, что у вершины v и всех пройденных по пути вершин — именно этот лидер p. Проще всего это сделать, перенаправив их parent[] на эту вершину p.

Совместное использование эвристик дает асимптотику $O(\alpha(n))$, где $\alpha(n)$ – обратная функция Аккермана, которая растёт очень медленно, настолько медленно, что для всех разумных ограничений n она не превосходит 4 (по крайней мере, для $n \leqslant 10^{600}$ точно).

61 Асимптотика ответа на запрос в СНМ при использовании только эвристики по рангу.

Теорема 61.1. *Ранговая эвристика дает $O(\log n))$ на каждый запрос.*

Доказательство. Рассмотрим ранговую эвристику по глубине дерева. Покажем, что если ранг дерева равен k, то это дерево содержит как минимум 2^k вершин (отсюда будет автоматически следовать, что ранг, а, значит, и глубина дерева, есть величина $O(\log n))$. Доказывать будем по индукции: для $k = 0$ это очевидно. Ранг дерева увеличивается с $k-1$ до k, когда к нему присоединяется дерево ранга $k-1$; применяя к этим двум деревьям размера $k-1$ предположение индукции, получаем, что новое дерево ранга k действительно будет иметь как минимум 2^k вершин. □

62 Алгоритм Крускала: корректность, реализация, асимптотика.

Алгоритм Крускала – еще один алгоритм поиска минимального остовного дерева.

- ▷ Сортируем все ребра графа по весу.
- ▷ Инициализируем лес деревьев. Изначально каждая вершина — дерево.
- ▷ Последовательно рассматриваем ребра графа в порядке возрастания веса. Если очередное ребро соединяет два разных дерева из леса, то объединяем эти два дерева этим ребром в одно дерево. Если очередное ребро соединяет две вершины одного дерева из леса, то пропускаем такое ребро.
- ▷ Повторяем 3), пока в лесу не останется одно дерево.

Теорема 62.1. *Алгоритм Крускала корректен*

Доказательство. Рассмотрим шаг алгоритма. Пусть текущее ребро при добавлении не создает цикл. Тогда оно соединяет два разных поддерева в MST, то есть соединяет разрез. У него минимальный вес, значит оно безопасно. □

Теорема 62.2. *Алгоритм Крускала работает за $O(m \log m)$*

Доказательство. Сортировка ребер займет $O(m \log m)$. Работа с СНМ займет $O(m\alpha(n))$, где α — обратная функция Аккермана, которая не превосходит 4 во всех практических приложениях и которую можно принять за константу. Алгоритм работает за $O(m(\log m + \alpha(n))) = O(m \log m)$. □

```
void make_set(int v) {
  parent[v] = v;
  rank[v] = 0;
}

int find_set(int v) {
  if (parent[v] == -1) return v;
  return parent[v] = find_set(parent[v]);
}

void union_sets(int a, int b) {
  a = find_set(a);
  b = find_set(b);
  if (a != b) {
    if (rank[a] < rank[b])
      swap (a, b);
    parent[b] = a;
    if (rank[a] == rank[b]) ++rank[a];
  }
}

sort(edges.begin(), edges.end(), cmp)
\\edges -- массив ребер, сортируем его в порядке неубывания длин ребер
for (auto e : edges) {
  if (find_set(e.u) != find_set(e.v)) {
    union_sets(e.u, e.v);
  }
}
```

63 Алгоритм Борувки: выбор минимального ребра из нескольких, корректность, реализация, асимптотика.

Алгоритм Борувки - алгоритм поиска минимального остового (неориентированного) дерева на связанном неориентированном графе.

Занумеруем рёбра графа начиная с единицы.

Алгоритм.

Начало:

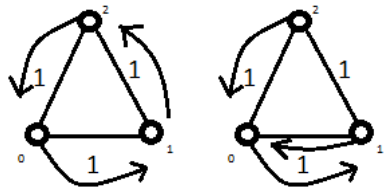
1) Изначально в лесу каждое дерево – одна вершина. Каждая вершина — отдельное дерево.

Итерация:

1) Для каждого дерева T_i из леса T выбираем самое легкое ребро, соединяющее T_i с остальным графом. Добавляем найденные ребра в лес, объединяя связанные им деревья.

Замечание:

Важно аккуратно выбирать минимальные рёбра для каждого дерева, чтобы в случае, когда из вершин исходит несколько рёбер наименьшего веса, не возникла ситуация, как на первой картинке (появился цикл). Избежать проблемы возможно, если из наименьших рёбер выбирать ребро с минимальным номером.



Теорема. Алгоритм Борувки корректен.
Доказательство.

▷ Будет построено дерево. Предположим, что это не так. (Будем проводить каждое неориентированное ребро от дерева, в котором оно выбрано, ко второй вершине.)

1) Предположим на некотором шаге будет построен простой цикл (хотя бы на 3-х вершинах), рёбра в котором были построены в одном направлении цикла (как пример - первая картинка выше). Если в цикле рёбра разной длины, то найдётся дерево, в котором было выбрано не минимальное по длине ребро. Если в цикле рёбра одной длины, то найдётся дерево, для которого было выбрано ребро большего номера, чем то, которое в него пришло. Противоречие.

2) Предположим результат алгоритма будет несвязным. Так как на каждом шаге алгоритма добавляется количество ребер равное n - количеству деревьев в лесу. Рёбра не образуют циклов, а мест, где произошли совпадения рёбер не более $\frac{n}{2}$. (Если было проведено 2 ребра между двумя компонентами, то эти рёбра совпадают) Следовательно, на каждом шаге было проведено не менее $\frac{n}{2}$ различных рёбер, значит количество деревьев в лесу на каждом шаге уменьшалось как минимум в два раза. Следовательно, в итоге мы получим связный граф.

Доказали, что будет построено дерево.

▷ Будет построено дерево минимального веса. От противного. По аналогии с теоремой о разрезе рассмотреть первое добавленное ребро, не принадлежащее мин.остову.

Утверждение. Алгоритм Борувки работает за $O(ElogV)$
Доказательство. На каждом шаге количество деревьев уменьшается не менее чем в 2 раза. Следовательно, количество итераций не больше $logV$. Один шаг выполняется за $O(E)$. Общее время работы — $O(ElogV)$. □

64 Определение паросочетания в произвольном графе, двудольного графа, увеличивающего пути.

Определение. Двудольный граф — это граф, множество вершин которого можно разбить на две части таким образом, что каждое ребро графа соединяет каждую вершину из одной части с какой-то вершиной другой части, то есть не существует рёбер между вершинами одной и той же части графа.

Определение. Паросочетание в (обычном неориентированном) графе - произвольное множество ребер, такое что никакие два ребра не имеют общей вершины.

Определение. Паросочетание M в двудольном графе - произвольное множество рёбер двудольного графа, такое что никакие два ребра не имеют общей вершины.

Определение. Вершина v насыщена (инцидентна, покрыта) ребрами из паросочетания M, если v лежит в одном из рёбер M. Остальные вершины называются ненасщенными (свободными).

Определение. Рёберное покрытие графа — множество рёбер, в котором каждая вершина графа инцидентна по меньшей мере одному ребру покрытия.

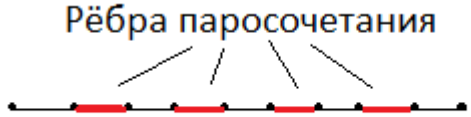
Определение. *Максимальное* паросочетание — это паросочетание в графе G, содержащее максимальное количество ребер

Определение. Паросочетание называется совершенным(полным), если оно покрывает все вершины графа

Определение. Цепь (путь) - последовательность вершин, в которой каждая вершина соединена со следующей ребром.

Определение. Чередующаяся цепь (путь) - цепь в двудольном графе, для любых двух соседних рёбер которой верно, что одно из них принадлежит паросочетанию, а другое нет

Определение. Увеличивающая (дополняющая) цепь (путь) в G относительно паросочетания M - чередующаяся цепь, у которой оба конца свободные вершины.



65 Лемма об устройстве неориентированного графа, в котором степени всех вершин не превосходят двух.

Лемма. Неориентированный граф, в котором степени всех вершин не превосходят двух, состоит из цепей и циклов.

Доказывается проходом по графу:

Если есть вершины степени 1, то начинаем идти по рёбрам начиная с неё, когда-то придём к концу (мы рассматриваем лишь конечные графы), получили цепь. Если все вершины степени 2, то начинаем идти по рёбрам начиная с некоторой вершины, так как наш проход не может закончится ни на одной другой вершине, ведь их степень 2 и граф конечный, то проход замкнётся на той вершине, с которой мы начинали. Получили цикл.

В итоге получим, что граф состоит из простых путей и циклов.

66 Теорема Бержа.

Теорема.(Бержа) Паросочетание M в двудольном графе G является максимальным тогда и только тогда, когда в G нет дополняющей цепи.

Доказательство.

$\Rightarrow$

Пусть в двудольном графе G с максимальным паросочетанием M существует дополняющая цепь. Тогда пройдя по ней и заменив вдоль неё все рёбра, входящие в паросочетание, на не входящие и наоборот, мы получим большее паросочетание. То есть M не являлось максимальным. Противоречие.

$\Leftarrow$

От противного. Пусть M – не максимальное. Пусть M' – другое паросочетание, $|M'| > |M|$. Рассмотрим их симметрическую разницу. В таком графе все вершины имеют степень не больше 2 (т.е. все компоненты будут цепями или циклами). Найдется компонента, в которой ребер M' больше. Это будет увеличивающей цепью для M, а не циклом, так как в цикле всегда одинаковое количество рёбер из M и M'. (Потому что любое ребро цикла принадлежит либо M, либо M' и в цикле чётное число рёбер) А это противоречие к условию отсутствия для M увеличивающих цепей.

67 Алгоритм Куна. Корректность, реализация, асимптотика.

Алгоритм Куна.

Задан граф $G\langle V, E\rangle$, про который известно, что он двудольный, но разбиение не задано явно. Требуется найти наибольшее паросочетание в нём

Алгоритм можно описать так: сначала возьмём пустое паросочетание, а потом — пока в графе удаётся найти увеличивающую цепь, — будем выполнять чередование паросочетания вдоль этой цепи, и повторять процесс поиска увеличивающей цепи. Как только такую цепь найти не удалось — процесс останавливаем, — текущее паросочетание и есть максимальное.

В массиве `matching` хранятся ребра паросочетания: $(v, \texttt{matching}[v])$ (v - вершина левой доли, $\texttt{matching}[v]$ - вершина правой доли) (Если паросочетания с вершиной v не существует, то $\texttt{matching}[v] = -1$). А $used$ — обычный массив "посещённостей"вершин в обходе в глубину (он нужен, чтобы обход в глубину не заходил в одну вершину дважды). Функция dfs возвращает $true$, если ей удалось найти увеличивающую цепь из вершины v, при этом считается, что эта функция уже произвела чередование паросочетания вдоль найденной цепи.

Внутри функции $dfs(v)$, где v - вершина левой доли, просматриваются все рёбра, исходящие из вершины v, и затем проверяется: если это ребро ведёт в ненасыщенную вершину to, то возвращаем true. Если эта вершина to насыщена, но удаётся найти увеличивающую цепь рекурсивным запуском из $\texttt{matching}[to]$ (т.е. $dfs(\texttt{matching}[to]) = true$), то мы говорим, что мы нашли увеличивающую цепь. Производим чередование в текущем ребре: перенаправляем ребро, смежное с to, в вершину v, возвращаем $true$.

В основной программе сначала указывается, что текущее паросочетание — пустое (массив `matching` заполняется числами -1). Затем перебирается все вершины левой доли, и из них запускается обход в глубину dfs, предварительно обнулив массив $used$.

Немного корректности:

Из теоремы в следующем билете следует, что если из вершины x не существует увеличивающей цепи относительно паросочетания M, тогда из x не существует увеличивающей цепи в M' (паросочетание M' получается из M изменением вдоль увеличивающей цепи).

Следовательно, если от вершины v запускался хоть раз dfs, то из v больше никогда нельзя будет провести удлиняющую цепь.

Так как любая удлиняющая цепь имеет концы в разных долях, то после запуска dfs из всех вершин левой доли в графе не останется удлиняющих цепей. Следовательно, мы получим максимальное паросочетание.

Реализация:

```
bool dfs(int v) {
    if (used[v]) return false;
    used[v] = true;
```

```
    for (to : g[v]) {
        if (matching[to] == -1 || dfs(matching[to])) {
            matching[to] = v;
            return true;
        }
    }
    return false;
}

int main() {
    matching.assign(n, -1); \\ где n - количество вершин левой доли.
    for (int i = 0; i < n; ++i) {
        used.assign(n, false);
        dfs(i);
    }
    for (int i = 0; i < n; ++i) {
        if (matching[i] != -1) {
            cout << i << " " << matching[i] << endl;
        }
    }
}
```

Асимптотика:

Можно явно задано разбиение графа размера n на две доли размером n_1 и n_2. Алгоритм Куна можно представить как серию из n_1 запусков обхода в глубину на всём графе. Следовательно, всего этот алгоритм исполняется за время $O(n_1 m)$ (или $O(nm)$), где m — количество рёбер.

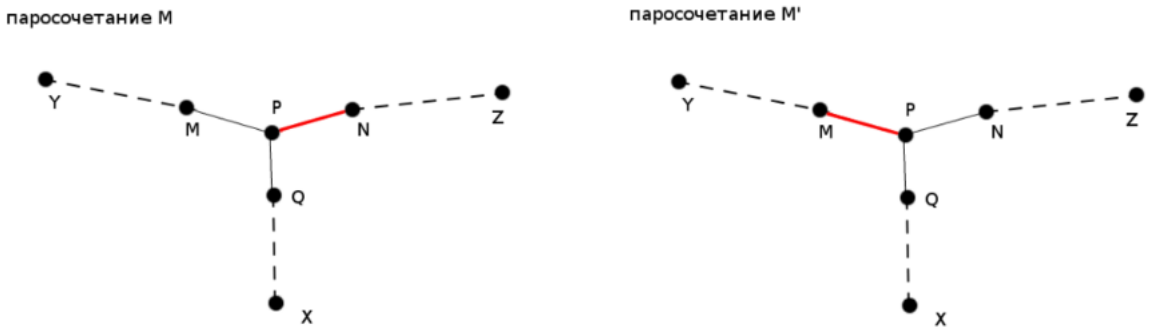
68 Лемма об отсутствии увеличивающих путей из вершины при отсутствии таких путей относительного меньшего паросочетания.

Теорема. Если из вершины x не существует увеличивающей цепи относительно паросочетания M, тогда из x не существует увеличивающей цепи в M' (паросочетание M' получается из M изменением вдоль увеличивающей цепи).

Доказательство.

Доказательство от противного.

Допустим в паросочетание внесли изменения вдоль дополняющей цепи ($y \rightsquigarrow z$) (цепь из y в z) и из вершины x появилась дополняющая цепь. Заметим, что эта дополняющая цепь должна вершинно пересекаться с той цепью, вдоль которой вносились изменения, иначе такая же дополняющая цепь из x существовала и в исходном паросочетании.



Пусть p – ближайшая к x вершина, которая принадлежит и новой дополняющей цепи и цепи $(y \rightsquigarrow z)$. Тогда MP – последнее ребро на отрезке $(y \rightsquigarrow p)$ цепи $(y \rightsquigarrow z)$, NP – последнее ребро на отрезке $(z \rightsquigarrow p)$ цепи $(y \rightsquigarrow z)$, QP - последнее ребро лежащее на отрезке $(x \rightsquigarrow p)$ новой дополняющей цепи.

Допустим MP принадлежит паросочетанию M', тогда NP ему не принадлежит. (Случай, когда NP принадлежит паросочетанию M' полностью симметричен.)

Поскольку паросочетание M' получается из M изменением вдоль дополняющей цепи $(y \rightsquigarrow z)$, в паросочетание M входило ребро NP, а ребро MP нет. Кроме того, ребро QP не лежит ни в исходном паросочетании M, ни в паросочетании M', в противном случае оказалось бы, что вершина p инцидентна нескольким рёбрам из паросочетания, что противоречит определению паросочетания.

Тогда заметим, что цепь $(x \rightsquigarrow z)$, полученная объединением цепей $(x \rightsquigarrow p)$ и $(p \rightsquigarrow z)$, по определению будет дополняющей в паросочетании M, что приводит к противоречию, поскольку в паросочетании M из вершины x не существует дополняющей цепи. (по условию задачи)

69 Определения независимого множества, вершинного покрытия. Связь определений.

Определение. Подмножество вершин графа G называется независимым множеством, если не существует ребра графа соединяющего какие-либо две из них.

Определение. Подмножество вершин графа G называется вершинным покрытием G, если любое ребро графа содержит вершину из этого множества.

Связь. Подмножество вершин является независимым множеством тогда, и только тогда когда его дополнение является вершинным покрытием.

(лемма следует из определения)

Утверждение. Подмножество вершин является максимальным независимым множеством тогда, и только тогда когда его дополнение является минимальным вершинным покрытием.

70 Алгоритм поиска максимального независимого множества и минимального вершинного покрытия в двудольном графе с помощью разбиения на доли $L^-; L^+; R^-; R^+$ (с доказательством).

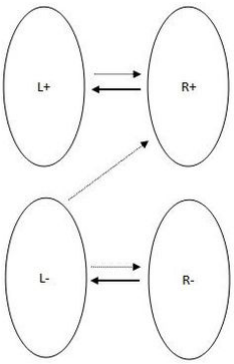
Пусть G - двудольный неориентированный граф, в нём M - максимальное паросочетание. Строим D - граф G, в котором рёбра из M ориентированы справа налево, а остальные

слева направо. Пусть L, R - множество вершин левой и правой доли соответственно. Запустим обход из всех ненасыщенных вершин левой доли. Пусть L^- - посещённые обходом вершины левой доли, L^+ - не посещённые вершины левой доли. Для правой доли аналогично. Тогда

$$L^- \cup R^+ - \text{минимальное вершинное покрытие}$$

$$L^+ \cup R^- - \text{максимально независимое множество}$$

Корректность:



Понятно, что не могут быть рёбра из L^+ в R^- и из R^+ в L^-. Также нет рёбер из R^- в L^+. Предположим противное и существует такое ребро из u в v. Это ребро ведёт справа налево, следовательно, оно из M, тогда v насыщена и из неё не запускали обход, но v была посещена ($v \in L^+$). Следовательно, существует путь до v из какой-нибудь ненасыщенной вершины левой доли w. Но единственный способ попасть в вершину v из правой доли это ребро (u, v). При этом вершина v была посещена обходом, а u нет ($u \in R^-$). Противоречие. Следовательно, нет рёбер из R^- в L^+

По картинке видно, что $L^+ \cup R^-$ – независимое множество, а $L^- \cup R^+$ – вершинное покрытие.

Осталось доказать, что $L^- \cup R^+$ – минимальное вершинное покрытие. Покажем, что $|L^- \cup R^+| \leqslant |M|$:

1) В L^- лежат только насыщенные паросочетанием M вершины.

2) В R^+ лежат только насыщенные вершины (иначе есть увеличивающий путь)

3) $\forall (u, v) \in M : u \notin L^-$ или $v \notin R^+$.

Показали.

Покажем, что мощность вершинного покрытия не меньше мощности максимального паросочетания:

Понятно, что каждое ребро M будет покрыто хотя бы одной вершиной из вершинного покрытия.

Показали оценку снизу, показали оценку сверху. Следовательно, $|VC| \geqslant |M| \geqslant |L^- \cup R^+| \geqslant |VC|$, где VC - минимальное вершинное покрытие. Следовательно,

$$L^- \cup R^+ - \text{минимальное вершинное покрытие}$$

$$L^+ \cup R^- - \text{максимально независимое множество}$$

71 Алгоритм поиска максимального независимого множества и минимального вершинного покрытия в двудольном графе с помощью задачи 2SAT.

В предыдущем доказательстве мы показали, что мощность минимального вершинного покрытия (МВП) равна мощности максимального паросочетания (МП), т.е. $|\text{МВП}| = |\text{МП}|$. Следовательно, каждая вершина из МВП лежит на ребре максимального паросочетания и ровно одна на каждом ребре. Зададим на каждом ребре порядок: одна вершина ребра - левая, другая - правая. Значит нам необходимо решить задачу: какую вершину каждого ребра МП нужно выбрать для МВП. Пронумеруем рёбра паросочетания. Введём обозначения: переменная $x_1 = 1$, если вершина МВП может быть правой вершиной первого ребра максимального паросочетания, и $x_1 = 0$, если вершина МВП может быть левой вершине ребра. Для остальных рёбер максимального паросочетания вводим аналогичные переменные. Далее для каждого ребра графа выписываем логическую формулу, так чтобы в любой ситуации удовлетворяющей формуле хотя бы одна вершина МВП покрыла данное ребро:

1) Например, если ребро соединяет две насыщенные вершины одна из которых правая вершина ребра i-го ребра МП, а другая левая вершина ребра j-го ребра МП, то формула будет выглядеть как: $x_i \vee \overline{x_j}$

2) Если ребро имеет только одну насыщенную вершину - правую вершину i-го ребра МП. Формула будет выглядеть как: x_i

3) Если у ребра нет насыщенных вершин, то её можно включить в паросочетание. Противоречие с максимальностью паросочетания.

Получили систему логических формул, решив которую получим искомое минимальное вершинное покрытие. Систему можно решить с помощью алгоритма 2SAT. Свели.

72 Определения сети, потока, величины потока, остаточной сети. Пример, почему нельзя обойтись без обратных рёбер.

Определение: *Сеть* - это (G, s, t, c), где $G = (V, E)$ - ориентированный граф (без петель и кратных ребер), s, t - различные вершины из V (s - исток (source), t - сток (target)), $c : E \to \mathbb{Z}_+$ - пропускные способности (capacity) на каждом ребре.

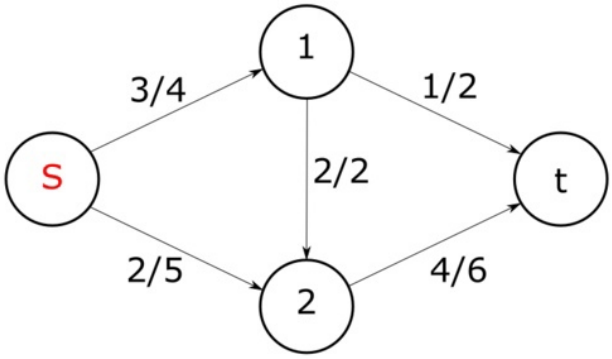
Определение: $f : V \times V \to \mathbb{Z}$ называется *потоком* в сети G, если выполняются следующие условия

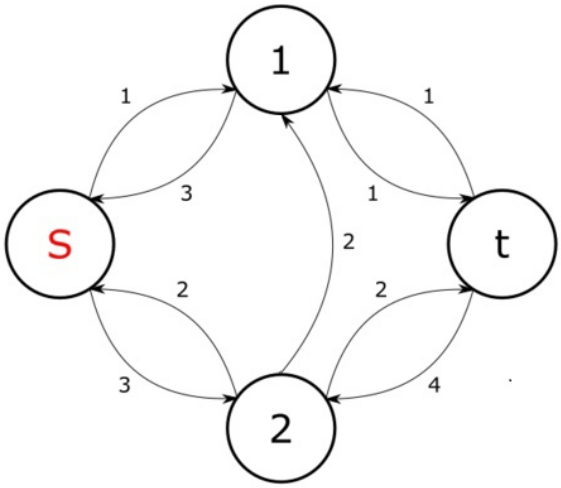
1. $\forall u \forall v\ f(u,v) \leqslant c(u,v)$ (если $(u,v) \notin E$, то $c(u,v) = 0$)
2. Сохранение потока (сколько втекло в вершину столько и вытечет): $\forall v \in V \setminus \{s,t\} \hookrightarrow \sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$
3. Антисимметричность: $f(u,v) = -f(v,u)$

Определение: *Величиной потока* называется число $|f| = \sum_{v\in V} f(s,v)$

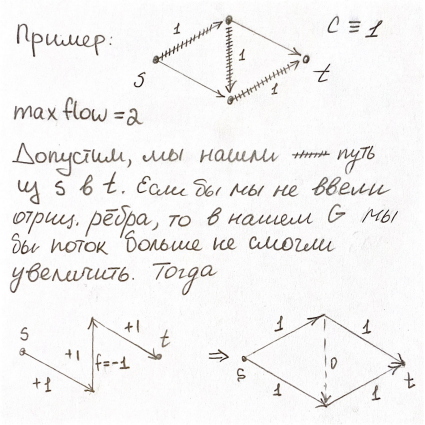
Определение: Пусть G - сеть, f - поток в ней. Тогда *остаточной сетью* G_f называется сеть с $c_f(u,v) = c(u,v) - f(u,v)$

Пример: Слева граф, справа - его остаточная сеть.





Пример: Зачем нужны отрицательные ребра? Ответ: чтобы отменять действия, которые нам не нравятся



73 Определения разреза, величины разреза, величины потока через разрез. Лемма о равенстве величины потока и величины потока через разрез.

Определение: G - сеть. (S,T) - *разрез*, если $s \in S$, $t \in T$, $S \sqcup T = V$.

$$c(S,T) = \sum_{u \in s, v \in T} c(u,v) \text{ - величина разреза}$$

$$f(S,T) = \sum_{u \in s, v \in T} f(u,v) \text{ - величина потока через разрез}$$

Лемма: $\forall (S,T)$ - разрез выполнено $f(S,T) = |f|$
▲ Доказательство индукцией по величине S

1. $S = \{s\} \Rightarrow (\{s\}, V \setminus \{s\})$ - разрез

$$f(\{s\}, V \setminus \{s\}) = \sum_{v \in V \setminus \{s\}} f(s,v) = |f| \text{ (так как } f(s,s) = 0)$$

2. $S = \{s, v_1, \ldots, v_{i+1}\}$. Пусть $U = \{s, v_1, \ldots, v_i\}$ и $f(U, V \setminus U) = |f|$. Тогда при добавлении v_{i+1} часть ребер перестанут вносить вклад в величину разреза (из $u \in U$ в v_{i+1}), а

часть - начнут (из v_{i+1} в $w \in V \setminus U \Leftrightarrow w \notin U$). Тогда

$$f(S, V\backslash S) = f(U, V\backslash U) - \sum_{u\in U} f(u, v_{i+1}) + \sum_{w\notin U} f(v_{i+1}, w) = |f| - \sum_{u\in U} f(u, v_{i+1}) + \sum_{w\notin U} f(v_{i+1}, w)$$

Покажем, что $\sum_{u\in U} f(u, v_{i+1}) = \sum_{w\notin U} f(v_{i+1}, w)$, тогда все будет доказано. Для этого распишем $\sum_{x\in V} f(v_{i+1}, x)$ двумя способами

$$\sum_{x\in V} f(v_{i+1}, x) = \sum_{x\in U} f(v_{i+1}, x) + \sum_{x\notin U} f(v_{i+1}, x) = \underbrace{-\sum_{x\in U} f(x, v_{i+1})}_{\text{антисимметричность}} + \sum_{x\notin U} f(v_{i+1}, x)$$

$$\underbrace{\sum_{x\in V} f(v_{i+1}, x) = \sum_{y\in V} f(y, v_{i+1})}_{\text{из сохранения потока}} = \sum_{y\in U} f(y, v_{i+1}) + \sum_{y\notin U} f(y, v_{i+1}) = \sum_{y\in U} f(y, v_{i+1}) \underbrace{- \sum_{y\notin U} f(v_{i+1}, y)}_{\text{антисимметричность}}$$

Тогда верно, что

$$-\sum_{x\in U} f(x, v_{i+1}) + \sum_{x\notin U} f(v_{i+1}, x) = \sum_{y\in U} f(y, v_{i+1}) - \sum_{y\notin U} f(v_{i+1}, y) \Rightarrow \sum_{x\in U} f(v_{i+1}, x) = \sum_{x\notin U} f(v_{i+1}, x) \blacksquare$$

74 Лемма о связи величины произвольного потока и величины произвольного разреза.

Лемма: Величина произвольного потока не превосходит величины произвольного разреза

▲

$$\underbrace{f(S', T') = |f| = f(S, T)}_{\text{см. билет 73}} = \sum_{u\in S, v\in T} f(u, v) \leqslant \sum_{u\in S, v\in T} c(u, v) = c(S, T) \blacksquare$$

75 Теорема Форда—Фалкерсона.

Теорема: Следующие утверждения эквивалентны

1. f - максимальный поток
2. В G_f нет пути из s в t (то есть относительно f нет увеличивающего пути)
3. $\exists (S, T)$ - разрез, такой что $|f| = c(S, T)$

▲ $1 \Rightarrow 2$: От противного: пусть в G_f есть путь из s в t. Рассмотрим минимальную capacity на этом пути: такую величину потока можно по нему протолкнуть $\Rightarrow$ мы увеличили поток - противоречие с максимакльностью

$2 \Rightarrow 3$: Пусть S - множество достижимых из s вершин в G_f, $t \notin S$ (так как иначе был бы путь из s в t). $T = V \setminus S$

Рассмотрим произвольное ребро e между S и T. Так как оно не лежит полностью ни там, ни там, то его нет в остаточной сети $\Rightarrow$ поток пропущенный по нему равен его capacity. Просуммируем все такие ребра и получим, что

$$c(S,T) = \sum_{(u,v)\in S\times T} c(u,v) = \underbrace{\sum_{(u,v)\in S\times T} f(u,v) = |f|}_{\text{лемма из билета 73}}$$

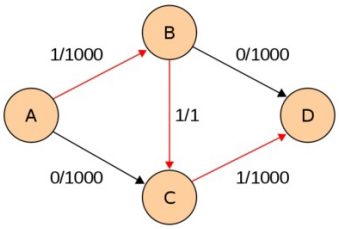
$3 \Rightarrow 1$: По лемме из билета 74 любой поток меньше или равен любого разреза. Так как мы достигли равенства, увеличить его мы уже не сможем $\Rightarrow$ это максимальный поток ■

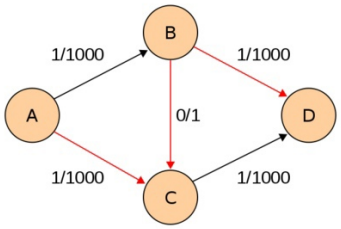
76 Алгоритм Форда—Фалкерсона. Корректность, асимптотика. Пример сверх-полиномиального (от размера входа) времени работы.

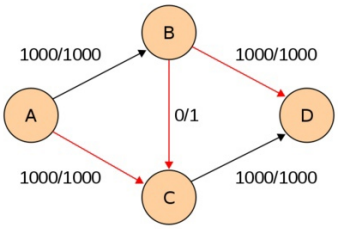
Алгоритм Форда-Фалкерсона: пока в G_f есть путь из s в t находим минимальную capacity на этом пути и проталкиваем такой поток по этому пути. После этого перестраиваем G_f

Корректность: очевидно следует из теоремы Форда-Фалкерсона (см. билет 75)

Пример: Посмотрим на рисунки приведенные ниже. Алгоритм может найти путь ABCD и пустить по нему единичку потока. После этого в остаточной сети появится ребро CB величины 1. Теперь алгоритм может найти путь ACBD и пустить по нему едиинчку потока. В итоге нам придется сделать примерно 2000 таких итераций







Асимптотика: $O(ans \cdot (n+m))$ - размер ответа (может быть что на каждой итерации проталкиваем по единице потока) умноженный на асимптотику поиска пути (DFS)

77 Алгоритм Эдмондса—Карпа. Корректность.

Алгоритм Эдмондса-Карпа: На каждой итерации алгоритма Форда-Фалкерсона находим путь, кратчайший по количеству ребер в нем.

Корректность: очевидно следует из теоремы Форда-Фалкерсона (см. билет 75)

78 Лемма о возрастании $dist(s,v)$ между последовательными итерациями алгоритма Эдмондса-Карпа.

Лемма: Пусть f и f' два последовательных потока в алгоритме. Пусть $d(v) = dist(s,v)$ в G_f, $d'(v) = dist(s,v)$ в $G_{f'}$. Тогда $\forall v \in V \hookrightarrow d'(v) \geqslant d(v)$

▲ Если $v = s$, то $d'(v) = 0 = d(v)$ и все доказано.

Пусть $v \neq s$. Среди всех v таких что выполнено $d'(v) < d(v)$ найдём v такую что $d'(v)$ минимально.

Пусть u - предпоследняя вершина на кратчайшем пути из s в v в $G_{f'}$. Тогда $d'(u) + 1 = d'(v)$ (1) $\Rightarrow d'(u) < d'(v) \Rightarrow$ так как $d'(v)$ - минимальное такое что $d'(v) < d(v)$, а $d'(u) < d'(v)$, то $d'(u) \geqslant d(u)$ (2).

Как ребро (u,v) оказалось в $G_{f'}$? Рассмотрим 2 случая

1. (u,v) было в G_f. Тогда $d(v) \leqslant d(u) + 1$ (кратчайший путь до v не больше чем путь до v через u) $\overset{(2)}{\leqslant} d'(u) + 1 \overset{(1)}{=} d'(v)$ - противоречие с выбором v.

2. (u,v) появилось только в $G_{f'}$. Значит оно появилось как обратное к ребру (v,u), по которому был пропущен поток, то есть оно лежало на кратчайшем пути из s в t в G_f $\Rightarrow d(v)+1 = d(u) \overset{(2)}{\leqslant} d'(u) \overset{(1)}{=} d'(v) - 1 \Rightarrow d(v)+2 \leqslant d'(v) \Rightarrow d'(v) > d(v)$ - противоречие с выбором v

Так как в обоих случаях получили противоречие, то таких v не существует, а значит $\forall v \in V \hookrightarrow d'(v) \geqslant d(v)$ ■

79 Лемма о числе насыщений ребра в алгоритме Эдмондса-Карпа. Асимптотика этого алгоритма.

Лемма: Говорим, что ребро насыщается, если $f(u,v)$ становится равным $c(u,v)$. Тогда каждое ребро насыщается $O(V)$ раз.

▲ Пусть ребро (u,v) насытилось (назовем этот момент 1). Чтобы оно насытилось еще раз, его надо «разнасытить», то есть кратчайший путь из s в t должен проходить через (v,u) (обозначим этот момент как 2)

Пусть d - кратчайшее расстояние в момент 1, а d' - в момент 2. Тогда $d(v) = d(u) + 1$ (так как в момент 1 (u,v) лежало на кратчайшем пути из s в t), $d'(v) \geqslant d(v)$ (по прошлой лемме), $d'(u) = d'(v) + 1$ (так как ребро (v,u) в момент 2 лежало на кратчайшем пути из s в t). Получаем $d'(u) = d'(v) + 1 \geqslant d(v) + 1 = d(u) + 2 \Rightarrow$ чтобы (u,v) разнасытилось, $d(u)$ должно вырасти хотя бы на 2, $d(u) \leqslant V - 1 \Rightarrow$ это происходит $\leqslant \frac{V}{2}$ раз ■

Асимптотика: $O(VE^2)$ - всего $O(VE)$ итераций (ребер E, каждое насыщается $O(V)$ раз) на каждой итерации ищем кратчайший путь через BFS $(O(E))$

80 Задача о разбиении коллектива на две группы с минимизацией суммарного недовольства.

Формулировка задачи: Есть n людей, нужно разбить их на 2 группы: занимающихся математикой и программированием. Каждый из них имеет некоторое недовольство к математике (a_i) и к программированию (b_i). Также между некоторыми из них есть дружеские связи, при разрыве которых суммарное недовольство увеличивается (на c_{ij}). Необходимо найти такое разбиение, чтобы суммарное недовольство было минимальным.

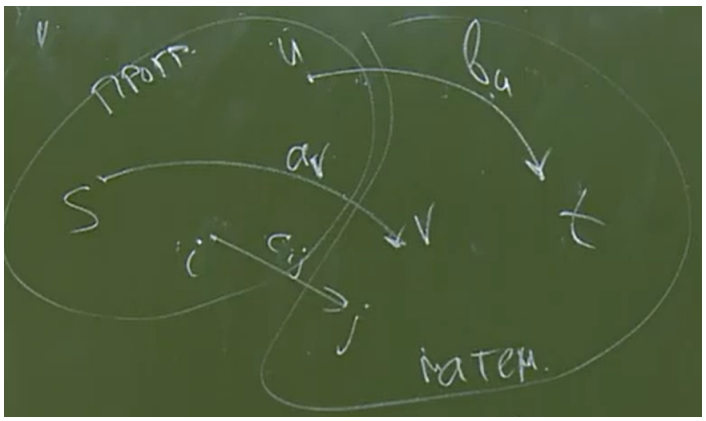
Решение: Создадим по одной вершине для каждого человека и две вершины s и t. Для кадого i

1. Проведем ребро (s,i) с capacity a_i

2. Проведем ребро (i,t) с capacity b_i

3. Для каждого друга j проведем два ребра (i,j) и (j,i) с capacity c_{ij}

Искомому разбиению с минимальным недовольством будет соответствовать минимальный разрез в построенном графе (S - программисты, T - математики)



Из картинки видно почему это правда: для каждой вершины будет учтено недовольство своей текущей профессией, а также разорванные дружбы (будут учтены только один раз по определению величины разреза).

81 Алгоритм Эдмондса-Карпа с масштабированием, асимптотика.

Пусть C - верхнее ограничение на все capacity. Перебираем $k = \lceil \log_2 C \rceil \dots 0$, на k-ом шаге алгоритма рассматриваем сеть с $c'(e) = \lfloor \frac{c_f(e)}{2^k} \rfloor \cdot 2^k$, например

$$c_f(e) < 2^k \Rightarrow c'(e) = 0$$

$$2^k \leqslant c_f(e) < 2^{k+1} \Rightarrow c'(e) = 2^k$$

Алгоритм: Перебираем $k = \lceil \log_2 C \rceil \dots 0$, находим и проталкиваем максимальный поток в сети $c'(e)$ (находим его Эдмондсом-Карпом, может быть проталкиваем по нескольким путям если получится).

Асимптотика: $O(E^2 \log C)$

▲ Понятно, что $\log C$ - это количество итераций. Покажем, что на каждой итерации мы находим $O(E)$ путей (из $O(E)$ на поиск одного пути как раз получится искомая асимптотика.

Пусть после рассмотрения k пропустили поток F_k. Рассмотрим разрез: в одной части все вершины достижимые из S по ребрам $\geqslant 2^k$, в другой - остальные. Тогда величина разреза $\leqslant 2^k E$ (все ребра между частями $< 2^k$) следовательно $F - F_k \leqslant 2^k E$, так как оставшийся поток не превосходит разреза.

Рассмотрим k-ый шаг. Заметим, что каждый раз когда мы увеличиваем поток, он увеличивается хотя бы на 2^k. Так как наш поток отличается от максимального на $\leqslant 2^k E$, то увеличить поток мы сможем $O(E)$ раз. Итог: на каждом шаге делаем DFS $O(E)$ раз. Таким образом, получаем асимптотику $O(E^2 \log C)$ ■

82 Определение слоистой сети, блокирующего потока. Алгоритм Диница, доказательство корректности.

Определение: G - сеть, $V_i = \{v | dist(s,v) = i\}$ - слои. Тогда *слоистая сеть* построенная по G - сеть, в которой оставлены только ребра из меньших слоев в большие (ребра из больших слоев в меньшие и внутри слоев игнорируются)

Определение: Пусть G - сеть. Тогда *блокирующим потоком* в ней называется такой поток, который нельзя увеличить без введения обратных ребер.

Алгоритм Диница: Пока из s в t есть путь в G_f строим слоистую сеть и ищем в ней блокирующий поток.

Корректность: остановимся только когда нет пути из s в t в остаточной сети $\Rightarrow$ алгоритм корректен по теореме Форда-Фалкерсона

83 Реализация алгоритма Диница. Асимптотика.

Слоистая сеть строится просто через 1 BFS (запускаем и оставляем только те ребра, которые идут из расстояния i в $i+1$)

Как же искать блокирующий поток? Рассмотрим одну вершину v каждое ребро выходящее из нее имеет смысл рассматривать только один раз: если мы нашли путь, то просто проталктваем туда поток, а если не нашли, то в будущем мы уже и не найдем, поэтому считаем его "неинтересным". Введем $ptr[v]$ - номер первого интересного ребра исходящего из v.

```
int dfs(int v, int flow) {
    if (v == t) return flow; // если уже дошли до стока - возвращаем ответ
    while (ptr[v] != g[v].size()) { // пока есть интересные ребра
        Edge e = g[ptr[v]]; // ptr[v]-ое ребро из v
        if (level[v] + 1 != level[e.to]) { // если это не ребро между слоями
            ++ptr[v]; // отмечаем ребро неинтересным
            continue; // переходим к следующей итерации
        }
        if (e.capacity == e.flow) { // если ребро уже насыщено
            ++ptr[v]; // отмечаем ребро неинтересным
            continue; // перейти к следующей итерации
        }
        int x = dfs(e.to, min(flow, e.capacity - e.flow)); // рекурсивно
    запускаемся от конца интересного ребра
        if (x > 0) { // если мы смогли протолкнуть ненулевой поток
            e.flow += x; // добавляем поток по e
            reverse_e.flow -= x; // отнимаем поток из обратного ребра
            return x; // возвращаем поток который смогли протолкнуть
        }
        ++ptr[v]; // иначе ребро неинтересно и переходим к следующему
    }
    return 0; // если ничего не получилось вернуть 0
}
```

Весь алгоритм Диница:

```
int findMaxFlow() {
    int maxFlow = 0;
    while (true) {
        bfs(); // строим слоистую сеть
        if (noPath(s, t)) break; // если t недостижимо из s то выходим из алгоритма
```

```
        flow = dfs(s, inf);
        while (flow > 0) { // иначе пока поток проталкивается
            maxFlow += flow; // добавляем его к общему потоку
            flow = dfs(s, inf); // пытаемся протолкнуть поток по другому пути
        }
    }
    return maxFlow;
}
```

Асимптотика: $O(V^2E)$

▲ Рассмотрим асимптотику операции $dfs(s, \infty)$. Если за время ее выполнения указатели интересных ребер сместились на k, то ее асимптотика - это $O(V + k)$. Действительно, все что мы делаем - это проходим по одному пути (максимальной длины V) и переключаем k указателей.

Тогда одна итерация алгоритма Диница работает за $O(V \cdot$ (количество найденных путей) $+$ (суммарное изменение всех указателей)). Количество найденных путей не превосходит E, так как каждый новый путь насыщает хотя бы одно ребро. Суммарное изменение всех указателей также не превосходит E (так как всего ребер E) $\Rightarrow$ асимптотика одной итерации алгоритма Диница равна $O(VE)$.

Покажем, что после каждой итерации $dist'(s,t) > dist(s,t)$. После пропускания блокирующего потока в остаточной сети могли добавиться только ребра из i-го слоя в $i-1$-й, (так как поток пускаем только по ребрам из i-го в $i+1$-й). Очевидно, что такие ребра не могут сократить расстояние между s и t лежащими в 0 и l-ом слоях. Пусть $dist(s,t)$ остался таким же. Заметим, что у нас всего $dist(s,t)+1$ слоев и есть ребра либо вперед на один слой, либо назад на один слой. Очевидно, что чтобы получился путь длины $dist(s,t)$ мы можем прыгать только вперед на 1 слой. Следовательно, этот путь полностью лежит в нашей слоистой сети, а значит мы можем пустить по нему поток - противоречие с тем, что нашли блокирующий поток

Таким образом, так как расстояние между s и t постоянно растет в алгоритме Диница всего $O(V)$ итераций $\Rightarrow$ общая асимптотика равна $O(V^2E)$ ■

84 Первая теорема Карзанова о числе итераций алгоритма Диница.

Обозначения: для любой вершины v отличной от s и t введем следующие величины:

$$C_{in}(v) = \sum_{u \in V} c(u,v);\ C_{out}(v) = \sum_{w \in V} c(v,w) \text{ - входящая и исходящая capacity}$$

$$p(v) = \min(C_{in}(v), C_{out}(v)) \text{ - потенциал вершины}$$

$$P = \sum_{v \in V \setminus \{s,t\}} p(v) \text{ - потенциал сети}$$

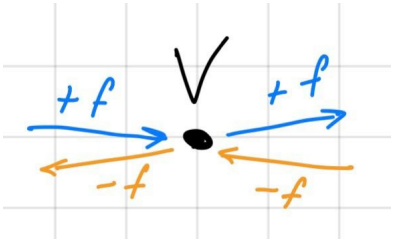
Замечание: Через v может протекать не больше чем $p(v)$ потока, так как потенциал вершины - это минимум из входящей и исходящей capacity и выполнен закон сохранения потока.

Лемма: Пусть $L = dist(s,t), F$ - максимальный поток. Тогда $L \leqslant \frac{P}{F} + 1$

▲ Так как $L = dist(s,t)$ в слоистой сети есть $L+1$ слой включая слои содержащие s и t (слоев без s и t - $L-1$). Обозначим $P_i = \sum_{v \in V_i} p(v)$.

Через V_i протекает $\leqslant P_i$ потока, так как P_i - это сумма потенциалов всех вершин в слое $\Rightarrow F \leqslant P_i \Rightarrow (L-1)F \leqslant P_1 + \ldots + P_{l-1} \leqslant P \Rightarrow L \leqslant \frac{P}{F} + 1$ ■

Лемма: P не изменяется при проталкиваниях потока при переходе к остаточной сети.



▲ Пусть v лежит на пути, по которому протолкнули поток f в остаточной сети. Тогда capacity ребер лежащих на этом пути уменьшилась на f, а обратных - увеличилась на f. После изображения такого изменения на картинке вижно, что C_{in} и C_{out} не поменялись, а значит и $p(v)$ не поменялся $\Rightarrow$ потенциал всей сети также не поменялся ■

Теорема (первая теорема Карзанова): Число итераций алгоритма Диница равно $O(\sqrt{P})$.

▲ Сделаем $\sqrt{P}$ итераций алгоритма Диница. Тогда $dist(s,t) = l_{new} \geqslant \sqrt{P}$ (так как доказали, что на каждой итерации $dist(s,t)$ увеличивается в билете 83). По лемме $\sqrt{P} \leqslant l_{new} \leqslant \frac{P}{F_{\text{ост}}} + 1 \Rightarrow \sqrt{P} - 1 \leqslant \frac{P}{F_{\text{ост}}} \Rightarrow F_{\text{ост}} \leqslant \frac{P}{\sqrt{P}-1} = O(\sqrt{P})$. Так как осталось пропустить $O(\sqrt{P})$ потока, то осталось найти $O(\sqrt{P})$ путей $\Rightarrow$ осталось $O(\sqrt{P})$ итераций. Так как до этого мы сделали ровно $\sqrt{P}$ итераций, то всего у нас тоже получается $O(\sqrt{P})$ итераций ■

Теорема (вторая теорема Карзанова): Число итераций алгоритма Диница равно $O(C^{1/3}V^{2/3})$, где C - ограничение сверху на все capacity (нет в программе, для общего развития).

85 Эффективность алгоритма Диница в единичных сетях.

Определение: *Единичная сеть* - сеть, в которой capacity принимают только значения 0 и 1

Замечание 1: В единичной сети $P \leqslant O(E)$

▲ Так как сеть единичная, то $C_{in}(v) = deg_{in}(v)$; $C_{out}(v) = deg_{out}(v)$, где deg_{in}, deg_{out} - входящая и исходящая степени вершины соответственно (то есть количество ребер входящих и выходящих из вершины). Тогда по определению

$$p(v) = \min(deg_{in}(v), deg_{out}(v)) \leqslant deg_{in}(v) + deg_{out}(v)$$

Откуда получаем, что

$$P = \sum_{v \in V \setminus \{s,t\}} p(v) \leqslant \sum_{v \in V} deg_{in}(v) + \sum_{v \in V} deg_{out}(v) = 2E \text{ ■}$$

Замечание 2: В единичной сети одна итерация алгоритма Диница работает за $O(E)$ так как за все dfsы ребро рассматривается максимум 1 раз (либо протолкнем по нему поток, либо выкинем из рассмотрения как неинтересное).

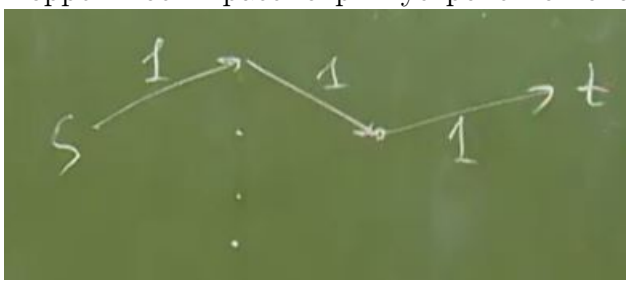
Вывод: В единичной сети алгоритм Диница работает за $O(E\sqrt{E})$

86 Алгоритм Хопкрофта—Карпа поиска максимального паросочетания в двудольном графе. Корректность и асимптотика.

Алгоритм Хопкрофта—Карпа - поиск максимального паросочетания в двудольном графе. Эта задача сводится к потоковой задаче: вводим искусственно вершинки s и t, а дальше делаем сеть: из всех вершин левой доли проводим в s рёбра капасити 1, из всех вершин правой доли - в t рёбра капасити 1, все рёбра исходного графа ориентируем слева направо и ставим капасити 1. MaxFlow = MaxMatching.

Асимптотика: используем алгоритм Диница, где все капасити - 1, потенциал всей сети $P = V$ (В левой доли у всех вершин $C_{in} = 1$, значит, потенциал 1, в правой $C_{out} = 1$, значит, потенциал 1. Значит, суммарный потенциал V), $\Rightarrow$ Диниц работает за $O(E\sqrt{V})$.

Корректность: рассмотрим устройство потока в такой сети.



Все пути имеют именно такой вид: одно ребро от s до левой доли, от левой доли до правой, от правой доли до t. Все такие пути не пересекаются $\Rightarrow$ ну а тогда мы получили из центральных рёбер паросочетания. Отсюда MaxFlow $\leqslant$ MaxMatching. Но и обратное так же верно: по максимальному паросочетанию можно достроить поток, так что MaxMatching $\leqslant$ MaxFlow. Отсюда следует это равенство.

87 Алгоритм Штор—Вагнера поиска минимального глобального разреза.

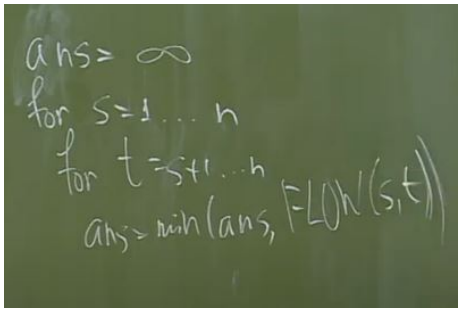
Глобальный разрез - разбиение множества V на два подмножества A и B, что $A, B \subset V; A, B \neq \varnothing; A \cap B = \varnothing; A \cup B = V$

Величина (вес) разреза: $c(S,T) = \sum_{u \in S, v \in T} c(u,v)$, где c(u, v) - capacity ребра из u в v (если нет такого ребра, то это ноль).

Цель: найти максимальный (или минимальный) глобальный разрез в G.

Поиск максимального глобального разреза в G - np-трудная. (Аналогично поиску максимальной клики/поиску Гамильтонова пути/цикла).

Поиск минимального разреза во взвешенном неориентированном графе: наивный алгоритм.



(s - вершина, которая лежит в S, t - вершина, которая лежит в T). Для неориентированных графов можно считать, что вершина 1 всегда лежит в s, тогда s = 1, for t = 2, ..., n).

Асимптотика: n раз запустить поток, где $n = |V|$; поток за Диница считается за V^2E. Итого: V^3E.

Замечание. Если граф невзвешенный, то алгоритм Диница работает быстрее.

Утверждение 1. Пусть $a_1 = 1$, $A_i = \{a_1, \dots, a_i\}$. Тогда a_{i+1} - вершина $\notin A_i$ с максимальным значением суммы рёбер до вершин из A_i: $c(\{v\}, A_i)$. $A_n = \{1, 2, \dots, n\}$. Тогда min разрез между a_n и a_{n-1} есть $(\{a_n\}, V \backslash \{a_n\})$.

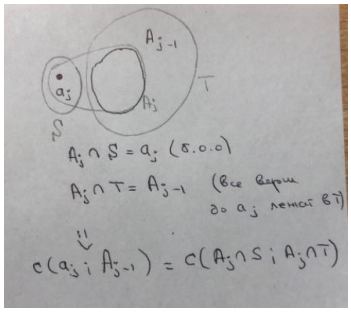
Алгоритм Штор-Вагнера.

Тогда min разрез из утверждения выше либо искомый ответ, либо a_n, a_{n-1} можно положить в одну долю ("склеить": суммировать расстояния до оставшихся вершин), тогда рано или поздно найдём нужный разрез такими склеиваниями и проверками, **то есть, по сути, запускаем n-1 раз, склеивая вершины**.

Асимптотика: $O(V^3)$, потому что n раз запускаем эту процедуру поиска. (Что-то похожее на алгоритм Прима). Нахождение вершины с наибольшей w за O(n), n-1 фаза по n-1 итерации в каждой. В итоге имеем $O(n^3)$. Применяя фибоначчиевы или двоичные кучи, можно добиться асимптотик $O(nm+n^2logn)$ или $O(nmlogn+n^2)$ соответственно. Ссылочка

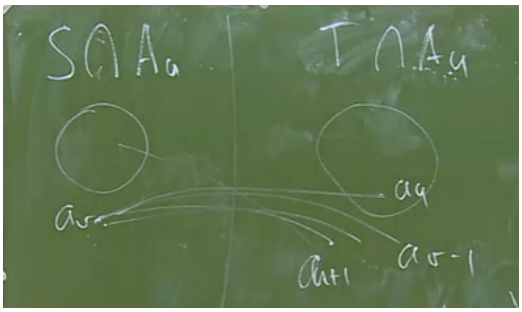
Докажем утверждение 1: пусть это не так. Тогда рассмотрим C - настоящий минимальный разрез, $C = (S, T)$. Назовём a_i - активной, если a_{i-1} лежит в другой части вершин (вместо S - T, вместо T - S). Тогда для каждой активной a_i $c(\{a_i\}, A_{i-1}) \leqslant c(S \cap A_i, T \cap A_i)$.

Пусть a_j - первая активная вершина. Тогда все предыдущие лежат в одной половине, a_j - в другой, т.е. $c(\{a_j\}, A_{j-1})$ - сумма рёбер из a_j в другие рёбра; $c(S \cap A_j, T \cap A_j)$ - то же самое. База тривиальна, выполняется равенство: смотри картинку ниже.



По индукции: пусть верно для предыдущих активных вершин, a_u, a_v - две последовательные активные вершины, т.е. вершины $[a_u, a_v)$ - в одной доле. Тогда $c(\{a_v\}, A_{v-1}) = c(\{a_v\}, A_{u-1}) + c(\{a_v\}, A_{v-1} \backslash A_{u-1}) \leqslant c(\{a_u\}, A_{u-1}) + c(\{a_v\}, A_{v-1} \backslash A_{u-1}) \leqslant c(S \cap A_u, T \cap A_u) + (c(S \cap A_v, T \cap A_v) - c(S \cap A_u, T \cap A_u)) = c(S \cap A_v, T \cap A_v)$ (последнее неравенство следует из определения, где лежат вершины между a_u и a_v: в одном из множеств).

Последняя вершина обязательно активная, так как она и предыдущая обязательно лежат в разных $\Rightarrow c(\{a_n\}, A_{n-1}) \leqslant c(S \cap A_n, T \cap A_n) = c(S, T)$. - смотри картинку ниже. А значит, c(S, T) не меньше того, что мы нашли. Победа!



88 Min cost flow: постановка задачи. Алгоритм поиска потока величины k минимальной стоимости (б/д). Асимптотика.

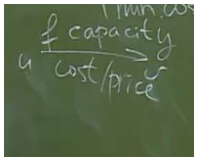


Рис. 1: Пример, что такое стоимость

Рис. 2: Что такое P, C

Таким образом, у нас появляются кроме вместимости потоков стоимости, т.е. сколько нам надо заплатить за пропуск одной единицы потока. Цель: либо фиксированная величина потока, либо максимально возможная при минимальной стоимости (первое: min cost k-flow, второе: min cost max-flow); таким образом, первая задача более общая, чем вторая.

Алгоритм поиска min-cost k-flow: k раз найдём в остаточной сети путь минимальной стоимости, протолкнём единицу потока сквозь него на каждом шаге. Тут dfs не поможет, нужна уже Дейкстра или что-то типа того. Но тут уже возникают рёбра отрицательного веса, из-за чего Дейкстру сходу применить не получится, поэтому по умолчанию применяем Форда-Беллмана + боремся с отрицательными циклами.

Считаем, что в исходной сети нет отрицательных циклов. Но доказывается, что добавление вот этой единички потока никогда не создаст отрицательных циклов (утв. 2).

Асимптотика: $O(k \cdot Ford - Bellman) = O(kVE)$

89 Критерий минимальности стоимости потока величины k.

Утверждение 2. Пусть f - поток величины k. Тогда f - min cost k flow тогда и только тогда, когда в остаточной сети G_f нет отрицательных циклов.

▲ $\Rightarrow$: Очевидно, если в G_f есть отрицательный цикл, то по нему можно пустить часть потока, стоимость уменьшится, величина потока - нет, противоречие.

$\Leftarrow$: Предположим обратное, f - не min cost k flow, тогда пусть f' - настоящий min cost k flow. Тогда рассмотрим новый поток $g = f' - f$, т.е. на каждом ребре мы рассматриваем разность: $g(e) = f'(e) - f(e)$. Тогда g - поток, притом поток величины ноль; по сути это несколько циклов, где постоянно ходит товар (циркуляция), его стоимость отрицательна, но тогда хотя бы один из циклов отрицательный: противоречие. ■

90 Корректность алгоритма поиска min cost k-flow в отсутствие отрицательных циклов.

Утверждение. Алгоритм поиска min cost k flow после i итераций находит min cost i flow, если в исходном графе G не было отрицательных циклов.

▲ f_0 - min cost 0 flow. База доказана. Тогда рассмотрим переход от f_i к f_{i+1}: мы просто берём старый поток и остаточную сеть, проталкиваем вдоль неё по потоку поток размера 1, делаем переход. Тогда осталось доказать, что после этого шага мы не создадим отрицательных циклов + воспользуемся критерием минимальности.

Предположим, что в $G_{f_{i+1}}$ появился отрицательный цикл. Он может появиться только если новый поток использует рёбра, обратные к добавленному потоку. Тогда P + C - путь из s в t в G_{f_i} веса меньше чем P (смотри рисунок 2). $cost(C) < 0 \Rightarrow cost(P+C) < cost(P)$. Таким образом, мы либо нашли путь из s в t меньшей стоимости, либо у нас есть путь + отрицательный цикл, что противоречит либо тому, что P - не кратчайший, либо, что был отрицательный цикл. Противоречие, значит, отрицательных циклов в $G_{f_{i+1}}$ нет. ■

91 Потенциалы Джонсона. Поиск min cost k-flow с помощью алгоритма Дейкстры.

Идея: хотим оптимизировать всё, что было выше, используя Дейкстру.

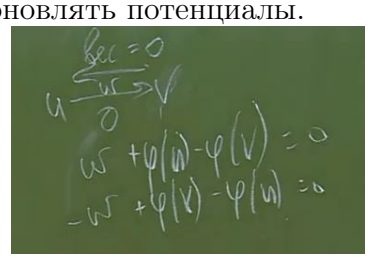
Потенциалы Джонсона (алгоритм Дейкстры-Джонсона) Пусть $\varphi : V \to \mathbb{Z}$ - потенциальная функция (некоторая функция из множество вершин в множество целых чисел). Тогда мы можем переопределить стоимости всех рёбер: $cost_\varphi(u,v) = cost(u,v) + \varphi(u) - \varphi(v)$. Это преобразование не меняет кратчайших путей: так как мы рассматривам путь из s в t, то складывая все новые косты, слагаемые взаимоуничтожаются, и значит, новый путь - это старый $+\varphi(s) - \varphi(t)$, что фиксированно для всех путей из s в t.

Теперь мы хотим при помощи этой потенциальной функции добиться того, чтобы все веса стали неотрицательными: $cost_\varphi(u,v) \geqslant 0 \ \forall u, v \in G_f$.

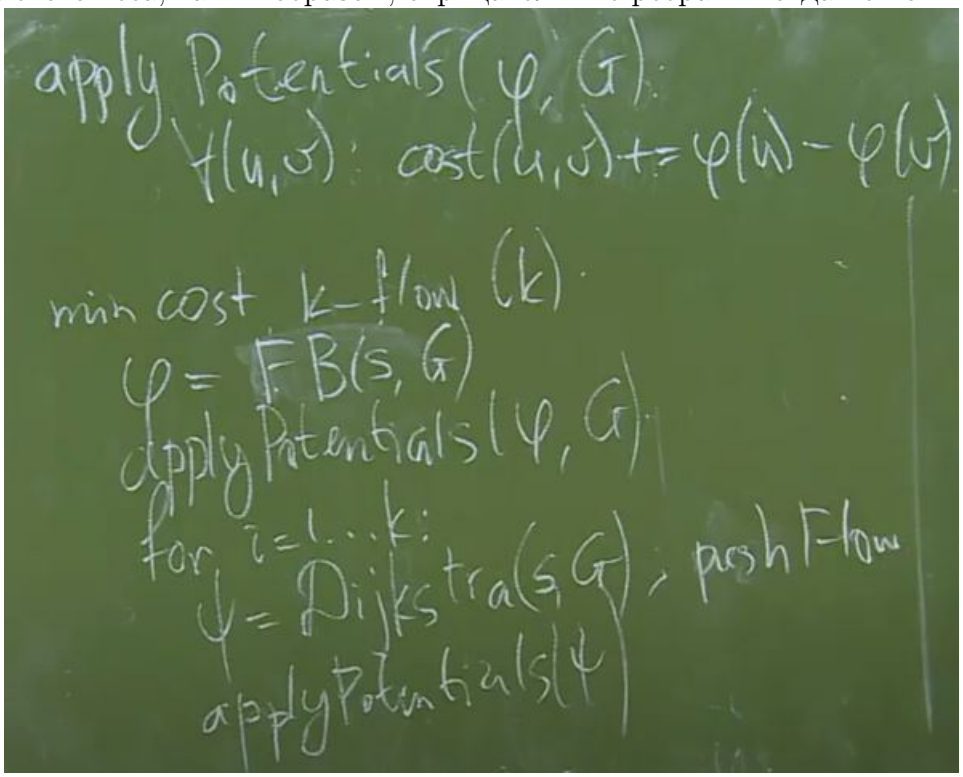
Форд-Беллмана запускаем в самом начале: FB из s, $\varphi(v) = dist(s,v)$ - потенциальная функция. Тогда $cost_\varphi(u,v) \geqslant 0 \ \forall u, v \in G$. Это то же самое, что $cost(u,v) + dist(s,u) - dist(s,v) \geqslant 0 \Leftrightarrow dist(s,v) \leqslant cost(u,v) + dist(s,u)$. Но это уже очевидно из того, что dist(s, v) - кратчайший путь (он либо вот такой же - путь от s до u и потом от u до v, либо где-то есть ещё более короткий путь).

Запустим Дейкстру на G с весовой функцией $cost_\varphi$. Кратчайший путь от s до t имеет вес 0 в этом графе + проходит только по нулевым рёбрам. Это верно, т.к. если (u, v) - ребро на кратчайшем пути из s в t в $cost_\varphi$, то оно так же лежало на кратчайшем пути из s в t на $cost$. Но тогда $dist(s,u) + cost(u,v) = dist(s,v) \Rightarrow cost_\varphi(u,v) = 0$.

Тогда проталкиваем по этому пути единичку. Появляются/пропадают некоторые рёбра, но если слева направо вес ноль, то справа налево (у обратного ребра) вес тоже ноль. Тогда, получается, отрицательных рёбер не появилось, когда мы протолкнули единичку. Но это верно только для первой итерации, где нули. Но дальше так не проканает: придётся обновлять потенциалы.



Пусть $\psi(v) = dist_{cost_\varphi}(s, v)$ - расстояние, которое находит Дейкстра, в сети с $cost_\varphi$. Тогда обновляем потенциалы. Тогда находим с помощью Дейкстры новые потенциалы, и работаем уже с ними. При переходе к новой сети на проталкивании снова будут рёбра нулевого веса, таким образом, отрицательные рёбра никогда не появятся.



Обновление: $\varphi(v) += \psi(v)$

Асимптотика: VE (Форд-Беллман) $+kV^2$ (или $Elog(V)$, в зависимости от плотности графа) - Дейкстра. Если же в исходном графе не было отрицательных рёбер, то первый Форд-Беллман так же может быть Дейкстрой.

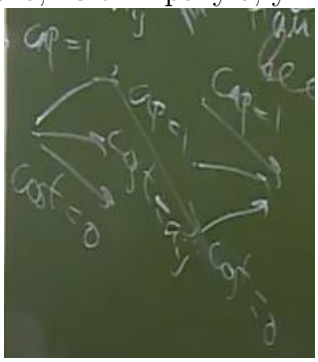
Замечание. Для min cost max flow берём $k = \infty$, если Дейкстра не находит пути, то делает break.

92 Задача о назначениях. Решение сведением к потоковой задаче.

Условие. Есть взвешенный полный двудольный граф (на самом деле, может быть и не полным), на каждом ребре по весу. Надо найти совершенное паросочетание минимального веса (в долях по n вершин, размер паросочетания - n). По сути - найти биекцию из левой

доли в правую. Название возникло из того, что есть n работников, ресурсы, которые они тратят, n задач, которые они должны выполнить.

Идея аналогична алгоритму Хопкрофта-Карпа: проведём такие же рёбра из s в левую долю, из t в правую, у них capacity 1, косты нулевые.



Вот мы и свели. Тогда через Дейкстру это будет кубическая асимптотика, так как мы n раз запускаем Дейкстру (в случае полноты графа лучше писать Дейкстру за n^2)

93 Определение дерева, его свойства (б/д). Определение диаметра дерева. Алгоритм поиска диаметра в дереве.

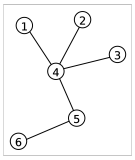


Рис. 3: Пример дерева

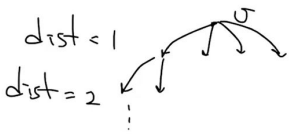


Рис. 4: DFS в дереве

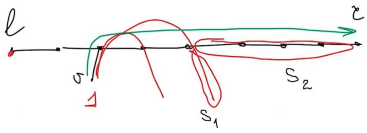


Рис. 5: Корректность поиска диаметра

G - **дерево**, если G - неориентированный связный граф без циклов.
E(G) - рёбра графа/дерева (edges), V(G) - вершины дерева.
Свойства деревьев:

1. $|E(G)| = |V(G)| - 1$; количество рёбер на один меньше количества вершин дерева.
2. $\forall u, v \exists!$ простой путь из u в v.

Пусть G - неориентированный невзвешенный граф. $dist(s,t)$ - кратчайшее расстояние между вершинами s и t. **Диаметр G** - такой путь между вершинами u и v, что $dist(u,v) = max_{s,t} dist(s,t)$. Обозначение: diam(G).

Тезис: многие задачи на графах являются очень трудными/np-трудными, а на деревьях решаются быстро. Пример 1: Поиск максимальной клики. Пример 2: Нахождение диаметра. Произвольный граф: Флойд - за $O(n^3)$. В дереве ищется за $O(n)$.

Алгоритм поиска диаметра в дереве:

Поиск самой далёкой вершины: "подвесим"дерево за v, дальше запускаем dfs. В силу вида дерева (см. рис. 2) мы не можем подняться на другой слой/попасть в вершину, которую мы уже посещали. Победа! - x - вершина с самого низкого уровня.

1) Пусть v - произвольная вершина. Найдём l - самую далёкую вершину от v.
2) Теперь r - самая далёкая вершина от l. Тогда (l, r) - это диаметр.

Асимптотика: $O(n)$

Корректность:

Посмотрим на рисунок три. Мы выбрали максимальный путь. По сути мы к каждой вершине подвесили своё дерево. Однако максимальные глубины подвешенных поддеревьев меняются в следующей последовательности: $0, 1, 2, \ldots, 2, 1, 0$ (так как иначе можно найти более длинный путь). Получается, для нашей произвольной v $s_1 \leqslant s_2$. Тогда самый длинный путь из v имеет ту же длину, что путь от v до r (или до l, надо выбрать максимум). Но тогда найденная вершина находится от l на таком же расстоянии, как и r. Таким образом, мы нашли какой-то диаметр (не обязательно именно тот, который мы выбрали в начале, когда подвешивали деревья к вершинам).

Альтернативный алгоритм: подвесим дерево за произвольную v, среди всех деревьев-детей v выбираем самые глубокие, вершины на низших слоях этих поддеревьев будут искомыми концами диаметра.

94 Определение центроида в дереве. Алгоритм поиска центроида в дереве. Лемма о количестве центроидов (б/д).

Пусть G - дерево, $v \in V(G)$. Тогда v - **центроид**, если после удаления v дерево распадается на несколько компонент, размером не более $\frac{n}{2}$

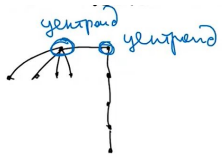


Рис. 6: Пример центроидов

Рис. 7: Алгоритм поиска центроида: шаг 1

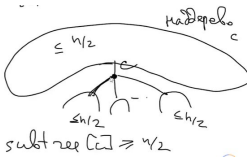


Рис. 8: Наддерево: шаг 2

Утверждение 1. В любом дереве есть центроид.

Утверждение 2 (б/д). В любом дереве либо один центроид, либо центроидов два и они соединены ребром (см. рис. 4)

Алгоритм поиска центроида за $O(n)$.

1. Пусть r - произвольный корень; subtree[v] = размер поддерева вершины v, включая v - это обычная динамика, dfs; subtree[r] = n.

1.5. Цель: найти такую вершину s, размеры всех поддеревьев детей которой не больше n/2, но subtree[s] $\geqslant \frac{n}{2}$. (Наддерево также станет одним из деревьев, образованных после удаления вершины; поддерево + наддерево = всё дерево)

P.S. Дабы избавиться от бед с округлениями, лучше сравнивать не так, как записано выше, а 2subtree[s] $\geqslant n$.

2. Спускаемся от корня вниз к ребёнку с размером поддерева subtree·2 $\geqslant n$, пока можем. Как только такого ребёнка нет - вершина, где мы закончились, - центроид.

95 Определение изоморфизма графов. Алгоритм проверки изоморфности двух ориентированных или неориентированных деревьев за O(n log n).

Рис. 9: Пример изоморфизма.

Рис. 10: Изоморфизм ориентированных деревьев.

Пусть G и H - два неориентированных графа. Функция $\varphi : V(G) \to V(H)$ - **изоморфизм**, если:

1) φ - биекция

2) $(u, v) \in E(G) \Leftrightarrow (\varphi(u), \varphi(v)) \in E(V)$

Замечание. Для ориентированных аналогично.

Пункт 1. Проверка ориентированных (подвешенных/корневых) деревьев на изоморфность.

0. См. рис. 2 - совсем тривиальный алгоритм, сравнивающий по количеству сыновей, не совсем работает, валится на совпадениях по количеству.

1. Пусть G - корневое дерево с корнем в r. Тогда для всех поддеревьев найдём "номер класса эквивалентности": мы разбиваем вершины на классы эквивалентности по количеству детей в каждом из поддеревьев, которые являются их сыновьями. Получаются вектора размеров поддеревьев, и по их равенству (без учёта порядка элементов в векторе, скажем, эл-ты отсортированы) вводятся классы эквивалентности.

```
map<vector<int>, int> num;
vector<int> classes; // вектор классов экв. по изоморфизму для вершин деревьев
ll c = 0; // количество классов

void dfs (int v) {
  vector<int> ans;
  for(int u: сын v) {
    dfs(u);
    ans.push_back(classes[u]);
  }
  sort(ans.begin(), ans.end());
  if (!num.count(ans)) {
    num[ans] = c++;
  }
  classes[v] = num[ans];
}
```

3. Для двух деревьев - запускаем dfs$[r_1]$, dfs$[r_2]$, после чего сравниваем classes.

Де-факто это быстро, хотя точную асимптотику сказать сложно (мап работает за логарифм длины сравнения, а там сравниваются вектора; но с хешированием векторов можно ускорить, если надо, и тогда будет $O(n)$ - время работы хеш-таблицы.)

Пункт 2. Проверка неориентированных деревьев на изоморфность: раньше у нас были фиксированные пары корней, которые мы сопоставляли: в биекции корень всегда сопоставляется корню - тут же мы берём пару центроидов (все возможные пары центроидов), потому что аналогично, центроид в первом графе останется центроидом во втором, вот наша гарантированная пара, а дальше - пункт 1.

96 Задача LCA. Постановка, решение с помощью двоичных подъёмов.

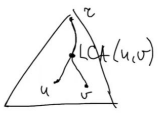


Рис. 11: Пример задачи LCA.

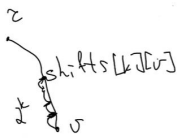


Рис. 12: Бинарные подъёмы.

Задача LCA - Least (lowest) Common Ancestor. Предком назовём путь от корня к данному узлу, тогда возникает задача найти самого низко расположенного общего предка для двух вершин в фиксированном дереве, подвешенном за корень r.

1. Разберёмся, как проверять, является ли одна вершина предком другой (u является предком самой себя).

```
// процедура проверки, что u - предок v
bool ancestor(int u, int v) {
  // вспомним, как мы это находили при помощи DFS
  // и порядка, в котором мы обрабатывали вершины:
  return (tin[u] <= tin[v]) and (tout[u] >= tout[v]);
}
```

2. Бинарные подъёмы: shifts[k][v] - предок в поколении 2^k для вершины v (или корень, если k слишком большой) - см. рис. 2.

shift[0][v] = parent[v]; Пусть shifts[k][v] = p, тогда shifts[k+1][v] = shifts[k][p].

Тогда вся таблица обсчитывается за $O(nlog(n))$, так как $k \leqslant log(n)$.

3. Пишем алгоритм поиска LCA:

```
int lca(int u, int v) {
  if (ancestor(u, v)) return u;
  for(int k = kmax - 1; k>=0; --k) {
    // пытаемся прыгнуть вверх, не перескочив общего предка
    if (!ancestor(shifts[k][u], v)) u = shifts[k][u];
  }
  // мы всегда находимся ниже, чем LCA, но так как последний k = 0, то
  // мы находимся ниже только на один слой, поэтому достаточно вернуть parent'a
  return parent[u];
}
```

Асимптотика: предпосчёт $O(nlog(n)$, шаг 3 - $O(log(n))$

97 Задача LCA. Решение с помощью эйлерова обхода.

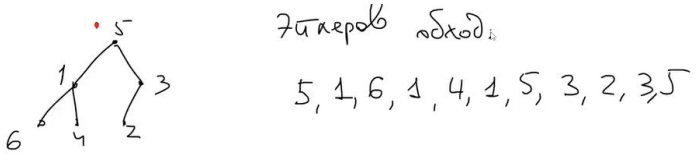


Рис. 13: Пример эйлерова обхода.

Также задачу LCA можно решить с помощью Sparse Table на эйлеровом обходе.

Эйлеров обход - печатает вершины в том порядке, в котором мы встречаем их в dfs. Тезис: между первыми вхождениями u и v в эйлеров обход встретится lca, при том какой-то более высокий общий предок не встретится.

Вывод: осталось найти вершину с минимальной глубиной из тех, которые перечислены между вхождениями u и v.

Длина эйлерова обхода: $O(n)$. Тогда на нём можно построить SparseTable на глубинах - предпосчёт за $O(nlog(n))$, ответ на запрос за $O(1)$ (стало чуть лучше, чем предыдущий алгоритм, в силу скорости ответа на запрос).

98 Задача LCA. Алгоритм Фарах-Колтона и Бендера.

Решает задачу LCA: предпосчёт за $O(n)$, запрос за $O(1)$.

Пользуемся предыдущим билетом: свели задачу LCA к поиску минимума на отрезке *(Range-Minimum Query, RMQ)*. Но этот RMQ специфический в силу того, что у нас соседние элементы отличаются на единицу.

Алгоритм.

0. Пусть есть массив длины n. Фиксируем $k = \frac{1}{2}log_2(n)$ + разобьём массив на блоки длины k. Тогда можно будет брать минимум по блокам и/или их префиксам/суффиксам.

Возьмём блок длины k, который начинается с нуля - **нормализованный**. Всего возможных нормализованных блоков - $2^{k-1} = O(\sqrt{n})$, так как на каждом следующем из k-1 элементов мы либо делаем +1, либо делаем -1 относительно предыдущего элемента, а первый у нас фиксирован.

1. Делаем все блоки нормализованными (вычитаем первый элемент), запоминаем, что вычли.

2. Кодируем каждый нормализованный блок int'ом или long long'ом.

3. Для всех нормализованных блоков находим минимумы на всех префиксах/суффиксах. Время: $O(\sqrt{(}n)log(n))$

4. Строим Sparse Table на $\frac{n}{k}$ блоков, где вместо блока пишем минимум на этом блоке. Время: $O(\frac{n}{k}log(\frac{n}{k})) = O(\frac{n}{log(n)}log(\frac{n}{log(n)})) = O(n - \frac{nlog(log(n))}{log(n)}) = O(n)$.

Ответ считаем за O(1): у нас закодированы наши нормализованные блоки, по коду мы находим минимумы на префиксах/суффиксах + на каждом из внутренних блоков (которые входят в отрезок полностью), выбираем самый минимальный минимум на всех (на внутренних - поиск минимума по отрезкам SparseTable, работает за O(1), +2 минимума на концах (суффикс + префикс)).

Предпосчёт: смотрим 3 и 4 пункты, слагаемое из 4 пункта превалирует, получается $O(n)$.

99 Задача RMQ. Постановка, решение за O(n) предподсчёта и O(1) на запрос.

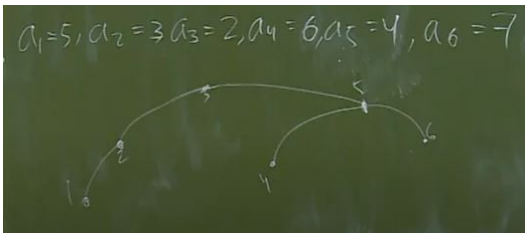


Рис. 14: Пример графа, который мы строим

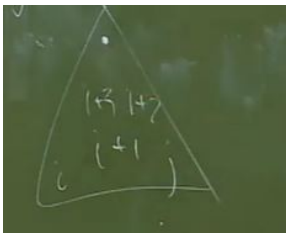


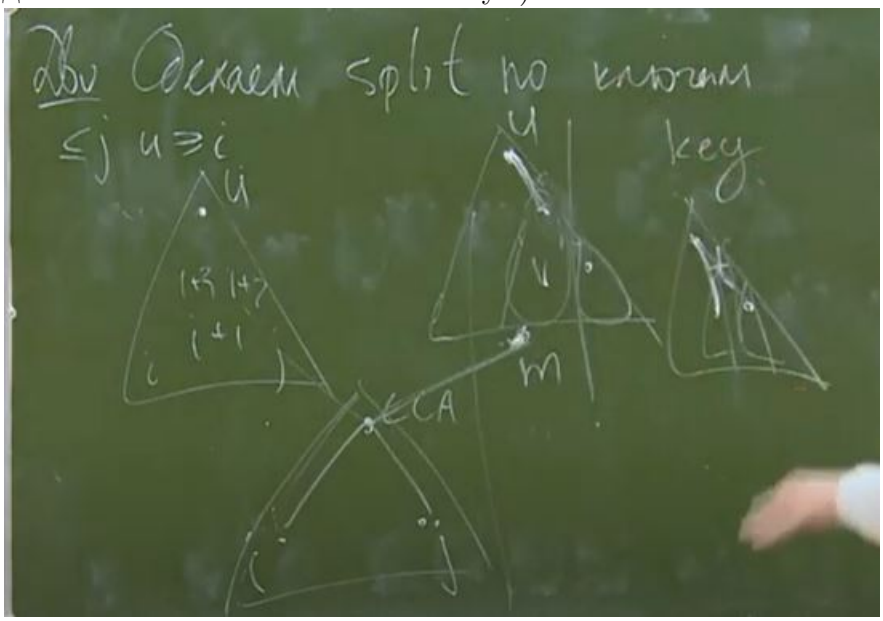
Рис. 15: Сплит по i, j

RMQ (Range-Minimum Query) - задача о поиске минимума на подотрезке. **Попробуем решить произвольный RMQ за то же время, сведя RMQ к LCA и воспользовавшись алгоритмом Фарах-Колтона.**

Пусть у нас массив $a_1 a_2 \dots a_n$, и в нём надо научиться находить минимум на отрезке. Рассмотрим декартово дерево на наборе точек: $(1, a_1), (2, a_2), \dots, (n, a_n)$ (т.е. ключ - порядок, а приоритет - наши элементы, см. рис. 4). Чтобы найти минимум на отрезке с i по j, надо найти LCA в таком декартовом дереве, и оно будет соответствовать минимальному среди этих значений.

Утверждение. Если (k, a_k) - LCA точек (i, a_i) и (j, a_j), то $a_k = min(a_i, a_{i+1}, \dots, a_j)$.

▲ Сделаем split по ключам $\leqslant j$ и $\geqslant i$ - это все ключи в отрезке от i до j. (см. рис. 5 - выделенная точка - искомый минимум).



Вспоминаем, как работает split: идея с переподвешиваниями. В терминах рёбер меняется только "перекладывание"слева направо, т.е. выделенная вершина как была предком v, так и осталась. Тогда, по сути, в контексте split новые рёбра не появляются, они только пропадают (вместе с вершинами), поэтому не может быть такого, что какая-то вершина станет предком своего бывшего предка. Корень в поддереве в исходном дереве был минимумом (и предком i, j), и в новом дереве он всё ещё общий предок i, j.

Осталось доказать, что в сплит не попадают вершины, которые являются более старшими предками, чем LCA, т.е., что равносильно, i и j лежат в разных поддеревьях отно-

сительно корня. ■

Split мы не делаем в декартовом дереве, мы исключительно ищем LCA, сплит нужен для доказательства, так что нам осталось лишь научиться строить декартово дерево за линию.

Пусть построено дерево для элементов с ключами от 1 до k. К нам пришёл новый ключ, значит, он располагается правее, дальше сравниваем его положение по вертикали и перепривязываем элементы, как на рисунке.

Таким образом, достаточно хранить правую ветку в виде стека: удаляем несколько элементов, пока не дойдём до нужного, и там перепривязываем (+ добавляем в стек).

100 Heavy-light decomposition. Тяжёлые и лёгкие рёбра. Лемма о числе лёгких рёбер на пути между двумя вершинами. Решение задачи обновления на ребре и суммы на пути за $O(log^2(n))$ на запрос.

Heavy-light decomposition. Пулл задач: как на ДО, только в роли отрезков - пути в дереве. Числа написаны на каждом ребре.

Подвесим дерево за произвольную вершину r. Насчитаем размеры всех поддеревьев: subtree (см. предыдущие билеты на центроиды). Ребро из u в v - **тяжёлое**, если subtree[v] $\geqslant \frac{1}{2}$subtree[u]. Замечание: из каждой вершины выходит не более одного тяжёлого ребра. Вот эти тяжёлые рёбра можно склеить в пути; все остальные рёбра лёгкие.

Идея: разбили дерево на тяжёлые пути, между ними лёгкие рёбра. Тогда если мы хотим обновлять значение в ребре и находить сумму на пути между двумя вершинами, то мы хотим выстроить на тяжёлых путях деревья отрезков (котировать их как массивы, всё находить), а количество лёгких путей, которые надо будет доставить, будет немного.

Лемма (о числе лёгких рёбер на пути между двумя вершинами). На пути от u до v ($\forall u, v$ количество лёгких рёбер равно $O(log(n))$).

▲ Как устроен путь от u до v? Это путь от u до LCA и путь от LCA до v. Лёгкие рёбра, если мы проходим снизу вверх, увеличивают размер поддерева не менее, чем в два раза - в силу определения тяжёлого ребра, если бы это было неправдой, то мы только что прошли по тяжёлому ребру; отсюда на первом пути максимум логарифм лёгких рёбер; аналогичные рассуждения о пути вниз. ■

Следствие: путь от u до v пересекает $O(log(n))$ тяжёлых путей.

Тогда обновление значения в ребре либо изменяет ДО на тяжёлом пути (если ребро тяжёлое), либо ребро лёгкое и ничего не меняется; первое делается за $O(log(n))$, второе - за $O(1)$.

Сумма ищется за $O(log(n) \cdot log(n) + 1 \cdot log(n))$, где первое слагаемое - это из работы с тяжёлыми путями (тяжёлых путей логарифм, на каждом из них сумма ищется за логи-

рифм), правая - работа с лёгкими путями (лёгких путей логарифм, работаем с ними за единицу). Тогда итоговая асимптотика - это лишь первое слагаемое, $O(log^2(n))$

101 Центроидная декомпозиция. Подсчёт числа объектов, обладающих заданным свойством.

Задача: найти число объектов со свойством α. Например, количество троек вершин, которые находятся на одном и том же расстоянии x друг от друга. Или количество путей, на которых сумма хорошая. Или на рёбрах написаны скобки, найти количество путей, образующих ПСП.

Решение в общем виде: Пусть C - центроид дерева T.

1.Находим его за линейное время. Подвесим дерево за центроид, все размеры не больше половины исходного. Найдём количество объектов, для которых свойство α пропадёт, если удалить вершинку C. Как именно это сделать? Зависит от задачи.

2. Если $T_1, \ldots, T_k$ - поддеревья, которые остались после удаления центроида - запускаемся рекурсивно от них (с удалённой вершинкой C). Таким образом, при запуске рекурсии мы минимум в два раза уменьшаем количество вершин в дереве на каждом шаге, значит, глубина рекурсии не превышает логарифм. Тогда если, например, пункт 1 реализован за $O(n)$, то суммарная сложность - $O(nlog(n))$

Ответ - рекурсия, количество искомых свойств, на которых влияет удаление центроида плюс количество искомых свойств, на которых влияет удаление центроида из поддеревьев-детей предыдущего центроида и так далее. По сути мы просто разбили все искомые объекты так, чтобы их было проще считать.

102 Центроидная декомпозиция. Задача о перекрашивании синих вершин в красный цвет и поиска расстояния до ближайшей красной вершины.

Задача. Дано дерево. Каждая вершина либо синяя, либо красная. Дано два типа запросов:

- перекрасить синюю v в красный цвет.
- для данной вершины u сообщить расстояние до ближайшей красной.

Дерево центроидов - корень дерева - исходный центроид, его дети - центроиды в поддеревьях, на которые мы разбились при рекурсии. Глубина дерева - максимум логарифм.

Запрос типа 2: искомая вершина лежит в одном из бОльших поддеревьев (наддеревьев). Она там лежит, так как по сути мы просто поднимаемся в более верхнее дерево, которое было до этого разбито на поддеревья, и когда-то поднимемся до всего дерева. Значит,

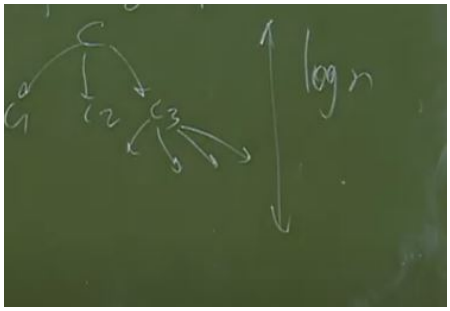


Рис. 16: Изображение дерева центроидов - 1.

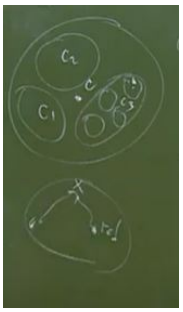
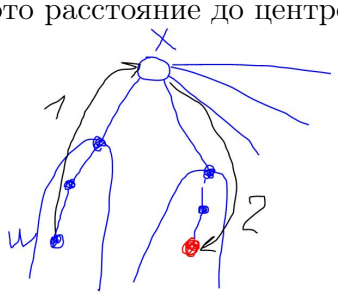


Рис. 17: Дерево центроидов 2 + поиск красной вершины.

в каком-то из наддеревьев лежит эта вершина. Соотвественно, на произвольном шаге, где центроид x, хочется найти ближайшую к x красную вершину и сказать, что она же ближайшая к u, просто надо сначала дойти до x, потом до u. (см. рис. 2)

Доказательство: рассматриваем дерево, в котором лежат обе вершины (u и ближайшая к ней красная), а в следующих поддеревьях - не лежат, но тогда расстояние от u до красной - это расстояние до центроида + расстояние от центроида до красной.



Пусть T_v - поддерево с центроидом v. Для каждой v находим ближайшую красную в T_v: closest[v] = $min_{r \in T_V} dist(v, r)$, где r - красная. Тогда ответ на запрос 2 типа - min по всем v - предки в деревне центроидов от $dist(u, v) + closest[v]$. Первое слагаемое вычисляется за логарифм (потому что мы умеем искать LCA за логарифм), значит, всё в итоге делается за $O(log^2(n))$.

Для запроса первого типа надо обновить closest для тех вершин v, для которых r стало лежать в T_v: в точности все предки вершины r. По всем v - предкам r в дереве центроидов берём новый closest - минимум из старого и нового. Время: $O(log^2(n))$, т.к. для каждого предка за логарифм обновляем расстояния до красной точки.