0 – 1 BFS, поиск в ширину

10056. Поиск в ширину 0 – 1
10048. Обернуть граф
4033. Трамвай
7403. Включить лампу
6426. Сломать тюрьму
5471. Граф 1, 1/2, 1/3, 1/4

10056. Поиск в ширину 0 – 1

Задан неориентированный граф. Вес его ребер может принимать только значения 0 или 1. Найдите кратчайшее расстояние между вершинами s и d.

Вход. Первая строка содержит четыре целых числа: количество вершин n, количество ребер m (n, $m \le 10^5$) и номера вершин s и d ($1 \le s, d \le n$). Каждая из следующих m строк содержит три целых числа a, b и w задающих неориентированное ребро (a, b) весом w ($0 \le w \le 1$).

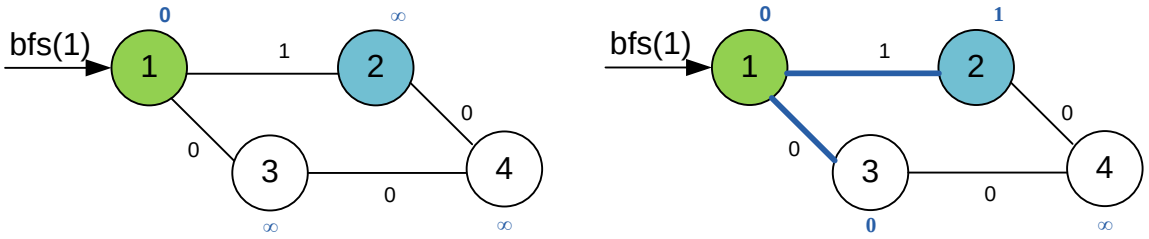
Выход. Выведите кратчайший путь между вершинами s и d.

Пример входа	**Пример выхода**
5 5 1 3 1 2 0 2 3 1 3 4 0 4 5 1 1 5 1	1

Кратчайший путь на 0 – 1 графе ищем поиском в ширину. При проходе по ребру (u, v), когда вычисляется кратчайшее расстояние dist[v] = dist[u] + 1, вершину v заносим:

- в начало очереди если вес ребра (u, v) равен 0;
- в конец очереди если вес ребра (u, v) равен 1;

В случае 0 – 1 графа при поиске в ширину следует релаксировать ребра. Присвоим изначально dist[i] = ∞ ($2 \le i \le 4$), dist[1] = 0, queue = (1) (очередь содержит стартовую вершину). Рассматриваем ребра, исходящие из вершины 1. Будет установлено dist[2] = 1, dist[3] = 0. Очередь примет вид queue = (3, 2), вершина 3 будет занесена в начало очереди, а вершина 2 в конец.

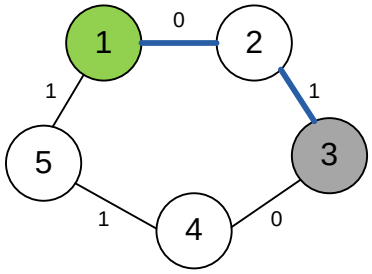


Однако значение dist[2] = 1 не является окончательным. Пройдем по ребру 3 – 4 установим dist[4] = 0, queue = (4, 2). Затем рассмотрим ребро 4 – 2 и значение dist[2] установится равным 0.

Таким образом dist[2] принимало два различных значения (сначала 1, потом 0).

Пример

Граф, приведенный в примере, имеет вид:



Реализация алгоритма

Объявим константу бесконечность.

```
#define INF 0x3F3F3F3F
```

Объявим массив кратчайших расстояний dist и список смежности графа g.

```
vector<int> dist;
vector<vector<pair<int, int> > > g;
```

Функция ***bfs*** реализует поиск в ширину из вершины *start*.

```
void bfs(int start)
{
  dist = vector<int>(n + 1, INF);
  dist[start] = 0;

  deque<int> q;
  q.push_back(start);

  while (!q.empty())
  {
    int v = q.front(); q.pop_front();
    for (int i = 0; i < g[v].size(); i++)
    {
      int to = g[v][i].first;
      int w = g[v][i].second;

      if (dist[to] > dist[v] + w)
```

```
      {
        dist[to] = dist[v] + w;
        if (w == 1)
          q.push_back(to);
        else
          q.push_front(to);
      }
    }
  }
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d %d %d", &n, &m, &s, &d);
g.resize(n + 1);
```

Читаем граф.

```
for (i = 0; i < m; i++)
{
  scanf("%d %d %d", &a, &b, &w);
  g[a].push_back(make_pair(b, w));
  g[b].push_back(make_pair(a, w));
}
```

Запускаем поиск в ширину со стартовой вершины *s*.

```
bfs(s);
```

Выводим кратчайшее расстояние до вершины *d*.

```
printf("%d\n", dist[d]);
```

10048. Обернуть граф

Задан ориентированный граф с n вершинами и m ребрами. Вершины графа пронумерованы от 1 до n. Найдите наименьшее количество ребер, которое следует обернуть, чтобы существовал хотя бы один путь от вершины 1 до вершины n.

Вход. Первая строка содержит два целых числа n и m ($1 \le n, m \le 10^5$) – количество вершин и ребер. i-ая строка из следующих m строк содержит два целых числа x_i и y_i ($1 \le x_i, y_i \le n$), означающих что i-ое ориентированное ребро идет от вершины x_i до вершины y_i.

Выход. Выведите наименьшее количество ребер, которое следует обернуть. Если невозможно получить ни одного пути от 1 до n, выведите -1.

Пример входа

```
7 7
1 2
3 2
3 4
7 4
6 2
5 6
7 5
```

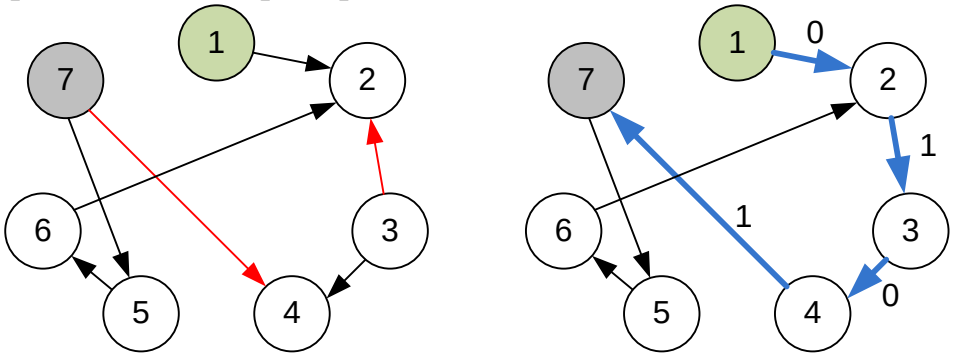
Пример выхода

```
2
```

Построим 0-1 граф. Имеющимся ребрам присвоим вес 0, а им обратным 1. Запустим поиск в ширину. Величина кратчайшего пути из вершины 1 в вершину *n* равна наименьшему количеству ребер, которое следует обернуть.

Пример

Граф, приведенный в примере, имеет вид:



Реализация алгоритма

Определим константу бесконечность.

```
#define INF 0x3F3F3F3F
```

Объявим массив кратчайших расстояний *dist* и список смежности графа *g*. Для каждого ребра вместе со смежной вершиной храним вес ребра (0 или 1).

```
vector<int> dist;
vector<vector<pair<int, int> > > g;
```

Функция ***bfs*** запускает поиск в ширину из вершины *start*.

```
void bfs(int start)
{
  dist = vector<int>(n + 1, INF);
  dist[start] = 0;

  deque<int> q;
  q.push_back(start);

  while (!q.empty())
  {
    int v = q.front(); q.pop_front();
```

```
      for (int i = 0; i < g[v].size(); i++)
      {
        int to = g[v][i].first;
        int w = g[v][i].second;

        if (dist[to] > dist[v] + w)
        {
          dist[to] = dist[v] + w;
```

Если вес ребра 1, то новую вершину кладем в конец очереди. Если вес 0, то в начало.

```
          if (w == 1)
            q.push_back(to);
          else
            q.push_front(to);
        }
      }
    }
}
```

Основная часть программы. Читаем входной граф.

```
scanf("%d %d", &n, &m);
g.resize(n + 1);
for (i = 0; i < m; i++)
{
  scanf("%d %d", &a, &b);
```

Ориентированному ребру $a \to b$ присваиваем вес 0, обратному ребру присваиваем вес 1.

```
  g[a].push_back(make_pair(b, 0));
  g[b].push_back(make_pair(a, 1));
}
```

Запускаем поиск в ширину из вершины 1.

```
bfs(1);
```

Выводим ответ – кратчайшее расстояние до вершины n.

```
if (dist[n] == INF)
  printf("-1\n");
else
  printf("%d\n", dist[n]);
```

4033. Трамвай

Трамвайная сеть в Загребе состоит из множества перекрестков и рельсов, соединяющих их. На каждом перекрестке имеется переключатель, указывающий на один из рельсов, выходящих из него. Когда трамвай въезжает на перекресток, он может покинуть его только в том направлении, на которое указывает

переключатель. Если водитель хочет идти другим путем, он должен вручную изменить переключатель.

Когда водитель едет от перекрестка a к перекрестку b, он старается выбрать путь, на котором количество изменений состояний переключателей минимально.

Напишите программу, которая вычислит наименьшее число изменений переключателей, необходимых для проезда от пересечения a до пересечения b.

Вход. Первая строка содержит целые числа n, a и b ($2 \le n \le 10^5$, $1 \le a, b \le n$), где n – количество перекрестков в сети, перекрестки пронумерованы числами от 1 до n.

Каждая из следующих n строк содержит последовательность чисел, разделенных пробелом. Первое число в i-ой строке - k_i ($0 \le k_i \le n - 1$), указывающее на количество трамвайных путей, исходящих из i-го перекрестка. Следующие k_i чисел указывают номера перекрестков, непосредственно связанными с i-ым. Переключатель на i-ом перекрестке изначально указывает на перекресток, который первым указан в списке смежности.

Выход. Вывести наименьшее искомое количество переключений. Если проехать от a до b невозможно, то вывести -1.

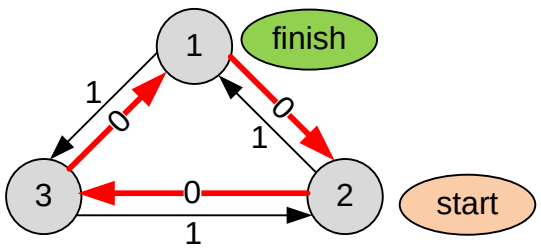
Пример входа	**Пример выхода**
3 2 1 2 2 3 2 3 1 2 1 2	0

Построим граф, в котором вершинами будут перекрестки, а ребрами – всевозможные трамвайные пути. Если на перекрестке x переключатель стоит по направлению к перекрестку y, то положим вес ребра (x, y) равным 0. Если от x можно изменить состояние переключателя к y, то вес ребра (x, y) положим равным 1.

Таким образом при движении от перекрестка a к перекрестку b мы будем минимизировать не длину пути, а количество переключений. Получили 0-1 граф. Алгоритм Дейкстры даст Time Limit. Воспользуемся поиском в ширину с релаксацией ребер:

- если релаксирует ребро (x, y) весом 1, то кладем y в конец очереди.
- если релаксирует ребро (x, y) весом 0, то кладем y в начало очереди.

Пример



Ребра с весом 0 указывают на начальное состояние направлений переключателей.

Реализация алгоритма

Входной взвешенный граф храним в списке смежности *g*. Объявим массив кратчайших расстояний *dist*. Объявим константу бесконечность INF.

```
#define INF 0x3F3F3F3F
vector<int> dist;
vector<vector<pair<int, int> > > g;
```

Функция ***bfs*** реализует поиск в ширину из вершины *start*. Положим кратчайшее расстояние от стартовой вершины до всех остальных равными INF. Расстояние от *start* до *start* равно 0.

```
void bfs(int start)
{
  dist = vector<int>(n+1,INF);
  dist[start] = 0;

  deque<int> q;
  q.push_back(start);

  while(!q.empty())
  {
    int v = q.front(); q.pop_front();
    for(int i = 0; i < g[v].size(); i++)
    {
      int to = g[v][i].first;
      int w = g[v][i].second;
```

Если ребро (*v*, *to*) релаксирует, то пересчитываем кратчайшее расстояние dist[*to*].

```
      if (dist[to] > dist[v] + w)
      {
        dist[to] = dist[v] + w;
```

В зависимости от веса релаксируемого ребра заносим вершину *to* в конец или в начало очереди.

```
        if (w == 1)
          q.push_back(to);
       else
          q.push_front(to);
      }
    }
  }
}
```

Основная часть программы. Читаем входные данные.

```
scanf("%d %d %d",&n,&a,&b);
g.resize(n+1);
```

```
for(i = 1; i <= n; i++)
{
```

Читаем количество ребер k, исходящих из вершины i.

```
  scanf("%d",&k);
  for(j = 0; j < k; j++)
  {
    scanf("%d",&to);
```

В i-ой строке после числа k_i стоит номер перекрестка, на который указывает переключатель. Вес этого ребра равен 0. Веса всех остальных исходящих ребер равны 1.

```
    w = (j == 0) ? 0 : 1;
    g[i].push_back(make_pair(to,w));
   }
}
```

Запускаем поиск в ширину из вершины a.

```
bfs(a);
```

Выводим ответ.

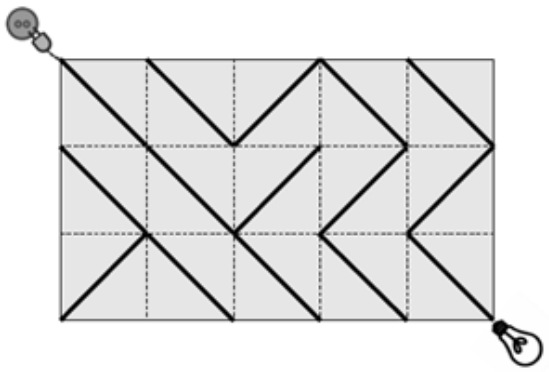
```
if (dist[b] == INF)
  printf("-1\n");
else
  printf("%d\n",dist[b]);
```

7403. Включить лампу

Степан разрабатывает электронную схему на прямоугольной сетке размером $n * m$. Всего $n * m$ квадратных плиток. Два (из четырёх) противоположных угла каждой плитки соединены проводом.

Источник питания подсоединен к левому верхнему углу сетки, лампа – к правому нижнему. Для того чтобы включить лампу можно повернуть любую плитку на 90 градусов в обоих направлениях.

На изображении лампа выключена. Если повернуть любую плитку во второй колонке справа, то лампа включится.



Напишите программу, которая найдет минимальное количество плиток, которое надо перевернуть для того, чтобы включить лампу.

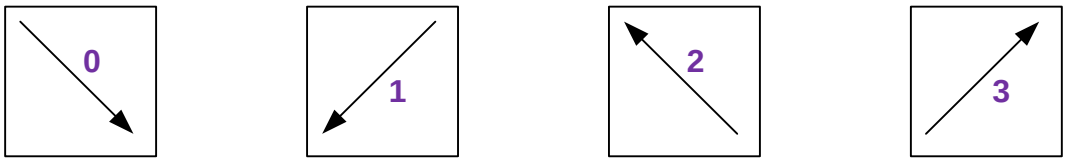
Вход. Первая строка содержит два целых числа n и m ($1 \le n, m \le 500$) – размеры сетки. Далее следуют n строк по m символов – \ или /, характеризующие направление провода на данной плитке.

Выход. Вывести ответ к задаче, или сообщение “NO SOLUTION”, если включить лампу невозможно.

Пример входа	**Пример выхода**
3 5 \\/\\ \\// /\\\\	1

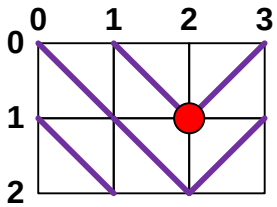
Пронумеруем горизонтальные и вертикальные линии прямоугольной сетки начиная с 0. Всего горизонтальных линий будет $n + 1$, они пронумерованы от 0 до n. Вертикальных линий будет $m + 1$, они пронумерованы от 0 до m. Левый верхний угол сетки имеет координаты (0, 0), правый нижний – координаты (n, m).

Каждому узлу сетки (i, j) поставим в соответствие вершину графа. Из каждого узла имеются 4 направления передвижения (по диагоналям). Пронумеруем направления:

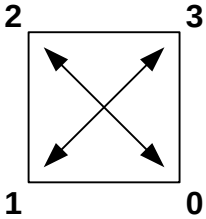


Входное состоянииe прямоугольной сетки и плиток на ней храним в массиве weight. Значение weight[i][j][dir] содержит вес ребра из вершины (i, j) по направлению dir (dir = 0, 1, 2, 3). Если из точки (i, j) по направлению dir идет провод, то установим weight[i][j][dir] = 0 (для передвижения плитку вращать не надо). Если точка (i, j) по направлению dir не соединена проводом, то положим weight[i][j][dir] = 1 (плитку следует повернуть). Таким образом построен взвешенный 0 – 1 граф, в котором минимальный путь между вершинами равен наименьшему числу плиток, которые следует повернуть для установления связи.

Например, для следующего примера имеют место соотношения:

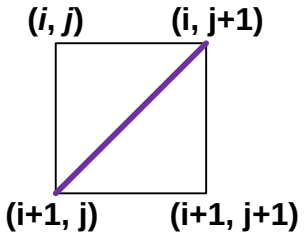


weight[1][2][0] = 1
weight[1][2][1] = 1
weight[1][2][2] = 0
weight[1][2][3] = 0



dir
направление

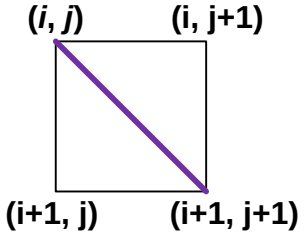
Рассмотрим процесс построения графа по входному текстовому массиву – входному состоянию сетки. Рассмотрим символ ‘/’ в клетке с левым верхом (*i*, *j*).



В этом случае:

- из точки (*i*, *j*) по направлению 0 двигаться нельзя и weight[*i*][*j*][0] = 1;
- из точки (*i* + 1, *j* + 1) по направлению 2 двигаться нельзя и weight[*i* + 1][*j* + 1][2] = 1;
- из точки (*i*, *j* + 1) можно двигаться по направлению 1, поэтому weight[*i*][*j* + 1][1] = 0;
- из точки (*i* + 1, *j*) можно двигаться по направлению 3, поэтому weight[*i* + 1][*j*][3] = 0;

Рассмотрим символ ‘\’ в клетке с левым верхом (*i*, *j*).



В этом случае:

- из точки (*i*, *j*) можно двигаться по направлению 0 и weight[*i*][*j*][0] = 0;
- из точки (*i* + 1, *j* + 1) можно двигаться по направлению 2 и weight[*i* + 1][*j* + 1][2] = 0;
- из точки (*i*, *j* + 1) по направлению 1 двигаться нельзя и weight[*i*][*j* + 1][1] = 1;
- из точки (*i* + 1, *j*) по направлению 3 двигаться нельзя и weight[*i* + 1][*j*][3] = 1;

Поиском в ширину ищем кратчайший путь на 0 – 1 графе из (0, 0) в (*n*, *m*).

Реализация алгоритма

Объявим константы и рабочие массивы. Входную сетку храним в массиве строк g.

```
#define INF 0x3F3F3F3F
#define MAX 510
int dist[MAX][MAX], weight[MAX][MAX][4];
string g[MAX];
```

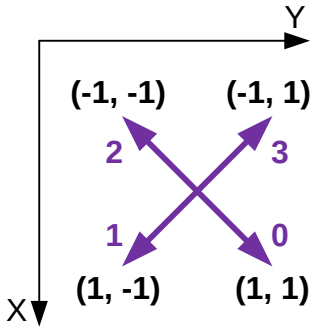
Функция ***CanGo*** проверяет, не вышла ли точка с координатами (*x*, *y*) за границы сетки.

```
int CanGo(int x, int y)
{
  return (x >= 0) && (x <= n) && (y >= 0) && (y <= m);
}
```

Функция ***bfs*** реализует поиск в ширину из точки (*sx*, *sy*).

```
void bfs(int sx, int sy)
{
```

Инициализация массивов. Пара (dx[*i*], dy[*i*]) задает направление *i*.




```
  int dx[] = { 1,1,-1,-1 };
  int dy[] = { 1,-1,-1,1 };
  memset(dist, 0x3F, sizeof(dist));
  dist[sx][sy] = 0;

  deque<pair<int, int> > q;
  q.push_back(make_pair(sx,sy));
```

Основной цикл поиск в ширину – пока очередь не пуста.

```
  while (!q.empty())
  {
```

Извлекаем координаты (*x*, *y*) вершины из головы очереди.

```
    int x = q.front().first;
    int y = q.front().second;
    q.pop_front();
```

Перебираем 4 ребра – направления, куда можно попасть из (*x*, *y*). Из точки (*x*, *y*) имеется ребро в (*x* + dx[*i*], *y* + dy[*i*]) весом weight[*x*][*y*][*i*].

```
    for (int i = 0; i < 4; i++)
    {
      int xx = x + dx[i];
      int yy = y + dy[i];
      if (!CanGo(xx, yy)) continue;
      int w = weight[x][y][i];
```

Если ребро (*x*, *y*) – (*xx*, *yy*) релаксирует, то пересчитываем кратчайшее расстояние dist[*xx*][*yy*] и кладем вершину (*xx*, *yy*) или в начало или в конец очереди в зависимости от веса ребра.

```
      if (dist[xx][yy] > dist[x][y] + w)
      {
        dist[xx][yy] = dist[x][y] + w;
        if (w == 1)
          q.push_back(make_pair(xx, yy));
        else
          q.push_front(make_pair(xx, yy));
      }
    }
  }
}
```

Основная часть программы. Читаем входные данные.

```
cin >> n >> m;
for (i = 0; i < n; i++)
  cin >> g[i];
```

Строим граф.

```
memset(weight, -1, sizeof(weight));
for (i = 0; i < n; i++)
for (j = 0; j < m; j++)
{
  if (g[i][j] == '/')
  {
    weight[i][j][0] = 1;
    weight[i + 1][j + 1][2] = 1;
    weight[i][j + 1][1] = 0;
    weight[i + 1][j][3] = 0;
  }
  else
  {
    weight[i][j][0] = 0;
    weight[i + 1][j + 1][2] = 0;
    weight[i][j + 1][1] = 1;
    weight[i + 1][j][3] = 1;
  }
}
```

Запускаем поиск в ширину из точки (0, 0),

```
bfs(0, 0);
```

Выводим кратчайшее расстояние до (*n*, *m*).

```
if (dist[n][m] == INF)
  cout << "NO SOLUTION" << endl;
else
  cout << dist[n][m] << endl;
```

6426. Сломать тюрьму

Миссия Джона состоит в том, чтобы вытащить двух человек из тюрьмы. Тюрьма представляет собой одноэтажное здание. Джон сумел достать подробный план этажа с указанием всех стен и дверей. Он также знает расположение двух людей, которых должен освободить. Тюремные охранники не проблема – он спланировал диверсию, в результате которой они покинут здание практически незаметно.

Двери являются его главной проблемой. Все двери, как правило, удаленно открываются из диспетчерской, однако Джон может открыть их с помощью других средств. После того как ему удалось открыть дверь, она остается открытой. Тем не менее открытие двери занимает много времени. Поэтому он хочет свести к минимуму количество дверей, которое он должен открыть. Сможете ли Вы помочь Джону спланировать оптимальный маршрут, по которому он сможет добраться до двух заключенных?

Вход. Первая строка содержит количество тестов, не более 100. Первая строка каждого теста содержит два числа n и m ($2 \le n, m \le 100$) – ширину и высоту карты. Следующие n строк по m символов описывают здание тюрьмы:

- ‘.’ пустое место.
- ‘*’ непроницаемая стена.
- ‘#’ дверь.
- ‘$’ один из двух людей которых следует освободить.

Джон может свободно перемещаться по внешней стороне здания. Карта содержит ровно два человека. До каждого человека внутри тюрьмы существует путь из внешней стороны.

Выход. Для каждого теста вывести в отдельной строке наименьшее количество дверей, которое Джон должен открыть для освобождения обоих узников.

Пример входа

```
3
5 9
****#****
*..#.#..*
****.****
*$#.#.#$*
*********
5 11
*#*********
*$*...*...*
*$*.*.*.*.*
*...*...*.*
*********.*
```

Пример выхода

```
4
0
9
```

```
9 9
*#**#**#*
*#**#**#*
*#**#**#*
*#**.**#*
*#*#.#*#*
*$##*##$*
*#******#*
*.#.#.#.*
*********
```

Прочитаем карту $n * m$ в двумерный символьный массив так чтобы левый верхний угол карты находился в позиции (1, 1). Строки 0 и $n + 1$, а также столбцы 0 и $m + 1$ заполним символами ‘.’ (пустое место).

Каждую ячейку массива трактуем как вершину графа. Из каждой вершины можно передвигаться в четырех направлениях (вверх, вниз, вправо, влево). Передвигаться в клетку с символом ‘*’ запрещено (непроницаемая стена). Разрешенные передвижения – это ребра графа. Положим веса ребер равными нулю кроме тех, которые идут в ячейку ‘#’ (дверь). Ребрам, идущим в дверь, положим вес равный 1. Таким образом нами построен 0 – 1 граф.

Запустим три поиска в ширину на 0 – 1 графе: из вершин (0, 0), (a_1, b_1) и (a_2, b_2), где (a_1, b_1) и (a_2, b_2) – координаты заключенных. Результатом поиска в ширину будут соответственно три массива a, b, c. Например, a[i][j] хранит кратчайшее расстояние (наименьшее количество дверей которое следует открыть) от (0, 0) до (i, j).

Имеется два варианта освобождения узников:

- добраться из (0, 0) в (a_1, b_1), вернуться назад в (0, 0) и опять пойти из (0, 0) в (a_2, b_2). Число открытых дверей будет равно b[0][0] + c[0][0].
- пойти из (0, 0) в некоторую позицию (i, j) в которой находится дверь, затем пойти из (i, j) в (a_1, b_1), вернуться в (i, j) и снова пойти из (i, j) в (a_2, b_2). Число открытых дверей равно a[i][j] + b[i][j] + c[i][j] – 2. Каждое слагаемое учитывает открытие двери в ячейке (i, j). Однако дверь следует открывать не 3, а 1 раз. Поэтому из общей суммы следует вычесть 2. Перебираем все двери (i, j) и находим минимум среди значений a[i][j] + b[i][j] + c[i][j] – 2.

Реализация алгоритма

Объявим константы и тип матрица *matrix*. Входную карту храним в массиве g.

```
#define INF 0x3F3F3F3F
#define MAX 110
string g[MAX];
typedef vector<vector<int> > matrix;
matrix a, b, c;
```

Функция ***CanGo*** проверяет, не вышла ли точка с координатами (x, y) за границы сетки. В позиции (x, y) также не должна находиться непроницаемая стена (символ ‘*’).

```
int CanGo(int x, int y)
{
  return (x >= 0) && (x <= n + 1) &&
         (y >= 0) && (y <= m + 1) && g[x][y] != '*';
}
```

Функция ***bfs*** реализует поиск в ширину из точки (sx, sy). Возвращает двумерный массив, который в позиции (i, j) содержит наименьшее количество дверей которое следует открыть чтобы попасть из (sx, sy) в (i, j).

```
matrix bfs(int sx, int sy)
{
  matrix dist(n + 2, vector<int>(m + 2, INF));
  dist[sx][sy] = 0;
```

Инициализируем очередь. Заносим в нее стартовую вершину – пару (sx, sy).

```
  deque<pair<int, int> > q;
  q.push_back(make_pair(sx, sy));
```

Инициализируем массивы dx и dy. Пара (dx[i], dy[i]) содержит направление, в котором разрешено передвигаться по комнатам тюрьмы.

```
  int dx[] = { 0, 0, -1, 1 };
  int dy[] = { -1, 1, 0, 0 };
```

Основной цикл поиск в ширину.

```
  while (!q.empty())
  {
```

Извлекаем координаты (x, y) вершины из головы очереди.

```
    int x = q.front().first;
    int y = q.front().second;
    q.pop_front();
```

Перебираем 4 ребра – направления, куда можно попасть из (x, y). Из точки (x, y) имеется ребро в (x + dx[i], y + dy[i]).

```
    for (int i = 0; i < 4; i++)
    {
      int xx = x + dx[i];
      int yy = y + dy[i];
      if (!CanGo(xx, yy)) continue;
```

Если ребро идет в дверь, то его вес равен 1. Иначе вес ребра равен 0.

```
      int w = (g[xx][yy] == '#');
```

Если ребро (x, y) – (xx, yy) релаксирует, то пересчитываем кратчайшее расстояние dist[xx][yy] и кладем вершину (xx, yy) или в начало или в конец очереди в зависимости от веса ребра.

```
        if (dist[xx][yy] > dist[x][y] + w)
        {
          dist[xx][yy] = dist[x][y] + w;
          if (w == 1)
            q.push_back(make_pair(xx, yy));
          else
            q.push_front(make_pair(xx, yy));
        }
      }
    }
    return dist;
}
```

Основная часть программы. Читаем входные данные.

```
cin >> tests;
while (tests--)
{
```

Карта расположена в подмассиве [1 .. n] * [1 .. m].

```
  cin >> n >> m;
```

Читаем карту тюрьмы. Строки 0 и n + 1, а также столбцы 0 и m + 1 заполним символами ‘.’ (пустое место).

```
  g[0].resize(m + 1, '.');
  a1 = -1;
  for (i = 1; i <= n; i++)
  {
    cin >> g[i];
    g[i] = " " + g[i] + " ";
    for (j = 1; j <= m; j++)
```

Позиции заключенных читаем в (a_1, b_1) и (a_2, b_2).

```
        if (g[i][j] == '$')
        {
          if (a1 == -1)
          {
            a1 = i; b1 = j;
          }
          else
          {
            a2 = i; b2 = j;
          }
```

Символ ‘$’ меняем на ‘.’.

```
          g[i][j] = '.';
```

```
    }
  }
  g[n + 1].resize(m + 1, '.');
```

Запускаем поиск в ширину из позиций (0, 0), (a_1, b_1) и (a_2, b_2). Матрицы a, b и c содержат соответственно информацию о кратчайших расстояниях.

```
  a = bfs(0, 0);
  b = bfs(a1, b1);
  c = bfs(a2, b2);
```

Вычисляем наименьшее количество дверей *res*, которое Джон должен открыть для освобождения обоих узников. Инициализируем *res* значением b[0][0] + c[0][0], когда каждого узника выводим независимо друг от друга.

```
  int res = b[0][0] + c[0][0];
```

Проходим по всем ячейкам тюрьмы, находим двери.

```
  for (i = 1; i <= n; i++)
  for (j = 1; j <= m; j++)
```

Среди всех позиций дверей (i, j) ищем наименьшее значение ответа.

```
    if (g[i][j] == '#')
      res = min(res, a[i][j] + b[i][j] + c[i][j] - 2);
```

Выводим ответ.

```
  cout << res << endl;
}
```

5471. Граф 1, 1/2, 1/3, 1/4

Дан связный, взвешенный неориентированный граф, ребра которого имеют веса 1, 1/2, 1/3, 1/4. Найдите кратчайший путь от вершины 1 до всех остальных.

Вход. В первой строке записаны два натуральных числа n и m ($1 \le n \le 5 * 10^5$, $1 \le m \le 8 * 10^5$) – количество вершин и ребер графа соответственно. Далее записаны ребра на отдельных строках. Ребра задаются тремя натуральными числами: u, v и w ($1 \le u, v \le n$, $u \le v$, $1 \le w \le 4$), которые обозначают наличие ребра из u в v веса $1 / w$.

Выход. Для каждой вершины от 2 до n выведите одно число – длину кратчайшего пути от вершины 1 до нее, с точностью не менее 8 знаков после запятой.

Пример входа

```
4 4
1 2 1
2 3 2
3 4 4
4 1 3
```

Пример выхода

```
1.00000000
0.58333333
0.33333333
```

Сделаем веса ребер целочисленными. Для этого поделим 12 на w (то же самое что вес ребра умножить на 12). Например, вершины 2 и 3 соединяет ребро весом 1/2 (w = 2). Имеем: 12 / w = 12 / 2 = 6. Или 1/2 * 12 = 6, что есть то же самое.

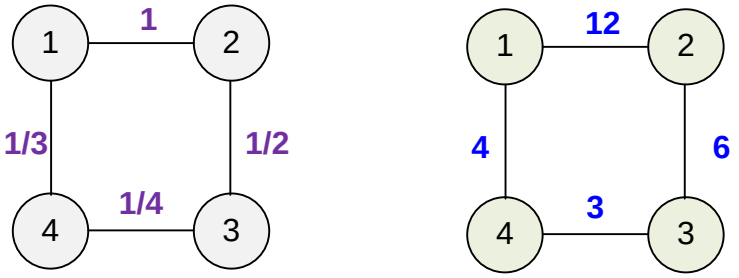
Имеем граф с целочисленными ребрами 3, 4, 6, 12. Для нахождения кратчайших расстояний запустим 1-k поиск в ширину для k = 12.

1-k поиск в ширину известен как алгоритм Дайала – модификация Дейкстры для малых целочисленных весов ребер.

https://www.geeksforgeeks.org/dials-algorithm-optimized-dijkstra-for-small-range-weights/

Пример

Рассмотрим граф из примера. Домножим веса всех ребер на 12 чтобы они стали целочисленными.



Реализация алгоритма

Константа MAX равна наибольшему возможному числу вершин в графе. Объявим массивы кратчайших расстояний dist и пройденных вершин used.

```
#define MAX 500010
int dist[MAX], used[MAX];
```

Граф храним в списке смежности g. g[i] содержит информацию о смежных вершинах в виде пар. Пара чисел содержит вершину, куда направлено ребро и его вес.

```
vector<vector<pair<int, int>>> g; // (to, dist)
```

Для реализации 1-k BFS объявим как минимум k = 13 очередей (очереди нумеруются от 0 до 12).

```
queue<int> q[14];
```

Функция ***bfs*** реализует 1-*k* поиск в ширину из вершины *start*.

```
void bfs(int start, int k)
{
```

Переменная pos содержит номер текущей обрабатываемой очереди.

```
  int pos = 0; // current queue
```

Инициализируем массив кратчайших расстояний dist.

```
  memset(dist, 0x3F, sizeof(dist));
  dist[start] = 0;
```

Заносим начальную вершину в очередь 0. Кратчайшее расстояние от *start* до *start* равно 0.

```
  q[0].push(start);
```

Переменная *cnt* хранит общее количество вершин, находящихся в очередях.

```
  int cnt = 1; // number of vertices in the queues
```

Основной цикл поиска в ширину. Продолжаем его пока хотя бы одна очередь содержит хотя бы одну вершину.

```
  while (cnt > 0)
  {
```

Ищем непуствую очередь *pos*.

```
    while (q[pos % (k + 1)].empty()) pos++;
```

Извлекаем вершину *from* из головы очереди *pos*.

```
    int from = q[pos % (k + 1)].front();
    q[pos % (k + 1)].pop();
```

Одна вершина из очереди удалена, уменьшаем *cnt* на 1.

```
    cnt--; // one vertex popped
```

Если кратчайшее расстояние до вершины *from* уже посчитано (вершина была обработана в других очередях), то пропускаем ее.

```
    if (used[from]) continue;  // from can be processed earlier
    used[from] = 1;
```

Перебираем вершины, смежные с *from*.

```
    for (int i = 0; i < g[from].size(); i++)
    {
```

Обрабатываем ребро (*from*, *to*) весом *d*.

```
      int to = g[from][i].first; // to
      int d = g[from][i].second; // dist;
```

Если ребро релаксирует, то заносим в очередь номер dist[*from*] + *d* вершину *to*. Пересчитываем значение dist[*to*]. Новая вершина добавлена в очередь, увеличим *cnt* на 1.

```
      if (dist[to] > dist[from] + d)
      {
        q[(dist[from] + d) % (k + 1)].push(to);
        dist[to] = dist[from] + d;
        cnt++;
      }
    }
  }
}
```

Основная часть программы. Читаем входные данные. Строим граф. Пересчитываем веса ребер, делая их целочисленными.

```
scanf("%d %d", &n, &m);
g.resize(n + 1);
while (m--)
{
  scanf("%d %d %d", &from, &to, &d);
  g[from].push_back(make_pair(to, 12 / d));
  g[to].push_back(make_pair(from, 12 / d));
}
```

Запускаем 1-*k* поиск в ширину с *k* = 12.

```
bfs(1, 12);
```

Выводим ответ – кратчайшие расстояния от первой вершины до всех остальных.

```
for (i = 2; i <= n; i++)
  printf("%.8lf\n", dist[i] / 12.0);
```

Реализация алгоритма – Дейкстра

Рассмотрим решение задачи алгоритмом Дейкстры.

```
#include <cstdio>
#include <vector>
#include <cstring>
#include <queue>
#define MAX 500010
using namespace std;

int i, j, n, m, a, b, from, temp;
int d[MAX];
```

```
struct Vertex
{
  int to, dist;
} v;

vector<vector<Vertex> > g;
queue<int> q;

void Dijkstra(int start)
{
  memset(d, 0x3F, sizeof(d));
  d[start] = 0;

  q.push(start);
  while (!q.empty())
  {
    int v = q.front();
    q.pop();
    for (int i = 0; i < g[v].size(); i++)
    {
      int to = g[v][i].to;
      int dist = g[v][i].dist;
      if (d[to] > d[v] + dist)
      {
        d[to] = d[v] + dist;
        q.push(to);
      }
    }
  }
}

int main(void)
{
  scanf("%d %d", &n, &m);
  g.resize(n + 1);
  while (m--)
  {
    scanf("%d %d %d", &from, &v.to, &v.dist);
    v.dist = 12 / v.dist;
    g[from].push_back(v);
    swap(from, v.to);
    g[from].push_back(v);
  }

  Dijkstra(1);

  for (i = 2; i <= n; i++)
    printf("%.8lf\n", d[i] / 12.0);
  return 0;
}
```