Федеральное государственное бюджетное образовательное учреждение высшего образования «Нижегородский государственный архитектурно-строительный университет» (ННГАСУ)

*Факультет инженерно-экологических систем и сооружений*

*Кафедра информационных систем и технологий*

КУРСОВАЯ РАБОТА

по дисциплине: «Язык программирования Python»

## На тему: «Алгоритмы поиска пути и структурное программирование»

Выполнил студент 1 курса гр. ИС-34 Миронов Е.С

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc1)

[Задачи 3](#_Toc2)

[1.Теоретическая часть 4](#_Toc3)

[2. Реализация алгоритма 4](#_Toc4)

[Пример работы 18](#_Toc5)

[Заключение 19](#_Toc6)

[Список литературы 20](#_Toc7)

[Приложение 1 21](#_Toc8)

[Листинг программы 21](#_Toc9)

# Введение

Алгоритмы поиска пути и структурное программирование довольно часто встречаются программистам. Жадный алгоритм пытается достичь максимальной выгоды, а потом приступить к менее выгодным. Достоинства жадного алгоритма: 1)Быстрое время работы, ограниченного только скоростью сортировки. 2)Может работать с большими значениями.

**Цель работы**: Используя жадный алгоритм пройти лабиринт до ключа и от ключа до выхода, помечая путь: «.» до ключа, а «,» до выхода.

# Задачи

1. В папке \_3 запустить скрипт gen\_lab\_origin.py, подождать его работу несколько минут;
2. Полученный файл maze-for-u.txt переместить в свою папку;
3. Найти любой маршрут от начальной координаты аватара до ключа, используя жадный алгоритм.
4. Используя А\* со следующими параметрами (вес g(x), вес h(x), максимальная длина хранимого списка возможных шагов), найти оптимальный путь до ближайшего выхода;
5. Сохранить в файл 'maze-for-me-done.txt', в котором точка ключа будет указана как '\*', а сам маршрут построен точками к ключу и запятыми от него к выходу.

# 1.Теоретическая часть.

## Почему жадные алгоритмы называются жадными?

Алгоритмы называются жадными, когда они используют жадное свойство. Жадное свойство:

В каждый момент времени, что есть **лучший выбор**?

Жадные алгоритмы – **жадные**. Они не смотрят в будущее, чтобы выбрать глобальное оптимальное решение. Их интересует только лучшее решение в данный момент. Но общее оптимальное решение может отличаться от решения, которое выбирает алгоритм на каждом шаге своей работы. Так же они никогда не оглядываются назад на то, что сделали, чтобы понять, нужна ли глобальная оптимизация. В этом главное отличие жадного и [динамического программирования](https://skerritt.blog/dynamic-programming/).

## Для чего используются жадные алгоритмы?

Жадные алгоритмы очень быстрые. Намного быстрее, чем две другие альтернативы (**Разделяй и властвуй** – Divide & Conquer, и **Динамическое программирование** Dynamic Programming). Они популярные, потому что они быстрые.

Примеры популярных жадных алгоритмов:

* [Dijkstra’s Algorithm](https://www.cs.cmu.edu/~deva/papers/tracking2011.pdf) (Алгоритм Дейкстры)
* [Kruskal’s algorithm](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm) (Алгоритм Крускала)
* [Prim’s algorithm](https://en.wikipedia.org/wiki/Prim%27s_algorithm) (Алгоритм Прима)
* [Huffman trees](https://en.wikipedia.org/wiki/Huffman_tree) (Деревья Хаффмана)

Мы собираемся рассмотреть жадные алгоритмы, на известном примере – **подсчет выдачи сдачи**. Это не так сложно для этой парадигмы.

## Как мне создать жадный алгоритм?

Ваш алгоритм должен всегда отвечать на этот вопрос:

Каков лучший выбор в этот момент времени?

Вот и все. Не так уж и много. Жадные алгоритмы, как правило, легче кодировать, чем «разделяй и властвуй»([Divide & Conquer](https://dev.to/brandonskerritt/a-gentle-introduction-to-divide-and-conquer-algorithms-1ga" \t "_blank)) или динамическое программирование ([Dynamic Programming](https://skerritt.blog/dynamic-programming/)).

# Подсчет выдачи сдачи используя жадность

Представьте, что вы торговый автомат. Кто-то дает вам **£1** и покупает напиток за **£0,70**. В фунте стерлингов нет монеты 30 пенсов (30p). **Как рассчитать, сколько сдачи вернуть?**

Для справки, номинал каждой монеты в Великобритании:

1p, 2p, 5p, 10p, 20p, 50p, £1

Жадный алгоритм начинается с самого высокого номинала и работает в обратном направлении. Наш алгоритм начинается с **£1**.  
**£1** фунт больше, чем **30** пенсов, поэтому он не может его использовать. Далее смотрим на **50р**, видим то же самое. Потом **20р**. **20p<30p**, поэтому требуется 1 монета **20p**.

Теперь алгоритм должен вернуть сдачу еще на **10p**. Он снова пробует **20p**, но **20p> 10p**. Затем идет до **10р**. Он выбирает 1 монету на **10p**, и теперь возвращается 0, мы останавливаем алгоритм.

Итого мы возвращаем **1x20p** и **1x10p**.

Этот алгоритм работает довольно хорошо в реальной жизни. Давайте рассмотрим другой пример, на этот раз у нас есть номинал и количество этой монеты в автомате в формате (номинал, количество).

(1p, 10), (2p, 3), (5p, 1), (10p, 0), (20p, 1p), (50p, 19p), (100p, 16)

Алгоритм попросили снова вернуть сдачу на **30p**. **100р (£ 1)** – это нет. То же самое для **50**. Далее **20p**, мы можем сделать это. Мы выбираем **1x 20p**. Теперь нам нужно вернуть **10р**. **20p** закончилось, поэтому мы идем вниз на 1 номинал.

**10p** закончилось, поэтому мы идем далее вниз еще на 1 номинал.

У нас **5p**, поэтому мы выбираем **1x5p**. Теперь нам нужно вернуть еще **5р**. Проверяем **5p** но они закончились, поэтому мы спускаемся вниз.

**[Падение Тела 3D - Яндекс Игры](https://an.yandex.ru/count/WXmejI_zO6K1bH0011qvewqMAcd9nmK0PG8nJR5EP000000uufmlXfFitvZMxPkV0O01pfEsbkZgtP3Y0OW1fVIfsvW1a06iuxJXte20W0AO0QpZjE5Uk07gt_Uq9S010jW1cftPd07W0TI2ZPG1w048-06K1A02j8q2y0A2WDFju3UWwGVu1FUEOOW5ev1Sa0NtZc6W1RxG7QW5jDWOi0Mqs1Yu1RJO6C05-PT_o0NygY3G1Q_Y1tq_-nnfzDPpgGS74gaJgHUi7xW7W0NG2Bg8W872W8262E08seO1w0a7W0e12EWCbmAO3VMxWO20W0YO3lt-xEUHZRthSv0GzikQnyELjT_ymB0GeH41wH97KzVIgjvsiAWJoRN9hwE2zwmNq1JVtTOUu1G1w1I40g0MaFEeeWR95j0MbBhUlW6O5x_vYYke5mcu5m705xNM0Q0PNh0Pk1d_0VWPlRsT784Q___JlcHos36m6fs-wDM3_lNKtW6270rePKKwJ64pULWnSsWtg1u1i1y2o1_DpgnEgI3hLVtbq0ZeFxWW_eOka2ANpIwG8fhDBf0Yeyqka2AapIwG8ghDBjKY__z__u4ZYIFMFv0ZqkhmtvlVzCwd0PWZqVIRvQtmkRlR0H80FBe6ggL7pXOXhOWfUTjaeXV5s58piVz4LfwdqmTThYF6tZ52fs1kMEFaHgCu8p0QLJDGGqRISGDrc7x-Ll40~1?stat-id=4&test-tag=245191093049889&banner-sizes=eyI3MjA1NzYwNjc1NjUzODU4MiI6IjM4MngzMDAifQ%3D%3D&format-type=118&actual-format=13&pcodever=764350&banner-test-tags=eyI3MjA1NzYwNjc1NjUzODU4MiI6IjU3Mzk0In0%3D&pcode-active-testids=760340%2C0%2C77&width=770&height=300" \t "_blank)**

Мы выбираем 1 монету на **2p**. Теперь нам нужно вернуть **3р**. Мы выбираем еще 2 монеты. Теперь нам нужно вернуть **1р**. Мы спускаемся еще раз вниз. Тут мы выбираем **1x1p** монета.

В общем, наш алгоритм выбрал следующие монеты, чтобы вернуть сдачу:

# (значение монеты, количество монет)

(10, 1)

(5, 1)

(2, 2)

(1, 1)

Давайте теперь закодируем это. Во-первых, нам нужно определить проблему. Начнем с условий.

# типы значений монет

denominations = [1, 2, 5, 10, 20, 50, 100]

# 100p is £1

Теперь перейдем к основной функции. С учетом номиналов и суммы для внесения изменений мы хотим вернуть список того, сколько раз эта монета была возвращена.

Если наш список **denominations** такой же, как указано выше, то [6, 3, 0, 0, 0, 0, 0] представляет собой взятие 6 монет **1p** и 3 монет **2p**, но 0 всех других монет.

denominations = [1, 2, 5, 10, 20, 50, 100]

# 100p is £1

**def** returnChange(change, denominations):

toGiveBack = [0] \* len(denominations)

**for** pos, coin **in** reversed(list(enumerate(denominations))):

Создаем полный список **denominations** и отствующие значения заполним нулями.

Мы хотим пройтись от самого большого до самого маленького. **Reversed(x)**переворачивает x и позволяет нам зацикливаться в обратном направлении. **Enumerate** означает «цикл по этому списку, но сохраняйте позицию в другой переменной». В нашем примере, когда мы запускаем цикл. **coin = 100**и **pos = 6**.

Наш следующий шаг – многократный выбор монеты, пока мы можем использовать эту монету. Если нам нужно дать сдачу **change = 40**, мы хотим, чтобы наш алгоритм выбрал **20**, затем снова **20**, пока он больше не сможет использовать **20**. Мы делаем это с помощью цикла **for**.

denominations = [1, 2, 5, 10, 20, 50, 100]

# 100p is £1

def returnChange(change, denominations):

# делам список toGiveBack размером с denominations

toGiveBack = [0] \* len(denominations)

# проходим по всему denominations

# с сохраненим coin и pos.

**for** pos, coin in enumerate(reversed(denominations)):

# пока coin меньше или равно change

**while** coin <= change:

В то время как **coin** все еще меньше или равно **change**, добавим эту **coin** в наш список возврата **toGiveBack** и удалим ее из сдачи.

denominations = [1, 2, 5, 10, 20, 50, 100]

# 100p is £1

**def** returnChange(change, denominations):

toGiveBack = [0] \* len(denominations)

**for** pos, coin **in** enumerate(reversed(denominations)):

**while** coin <= change:

change = change - coin

toGiveBack[pos] += 1

**return**(toGiveBack)

print(returnChange(30, denominations))

# вернет [0, 0, 0, 1, 1, 0, 0]

# 1x 10p, 1x 20p

Сложность выполнения этого алгоритма ([О большое](https://webdevblog.ru/bolshoe-o-chto-eto-takoe-pochemu-eto-vazhno-i-pochemu-eto-ne-vazhno/)) определяется двумя циклами, поэтому оно равно O(n2).

## Является ли жадный алгоритм лучшим? Жадность всегда работает?

То что оптимально на локальном уровне, иногда не оптимально на глобальном уровне. В алгоритме подсчете сдачи мы можем определить точку, в которой он не является лучшим в глобальном масштабе.

Допустим у нас есть условие:

* пусть у нас будет монеты трех номиналов [1p, 15p, 25p].

И нас попросим выдать сдачу в 30 пенсов. Теперь посмотрим, что вернет наш алгоритм.

[5, 0, 1]

То есть он выбирает **1x25p** и **5x1p**. Хотя лучшим решением будет – **2х15р**.

Наш алгоритм потерпел неудачу, потому что он вообще не смотрел на **15p**. Он первым делом посмотрел на **25р** и подумал: «Да, это подходит. Давайте возьмем это».

Затем он посмотрел на **15p** и подумал, что «это не подходит, давайте двигаться дальше» (так как 25 + 15 > 30).

Это пример того, где жадные алгоритмы не лучший выбор.

Чтобы обойти это, вам нужно будет либо создать валюту, где это не работает, либо использовать перебор решений. Или используйте динамическое программирование.

Но жадные алгоритмы иногда могут быть лучшими на глобальном уровне то есть глобально оптимальными . Ранее мы видели, что эти алгоритмы глобально оптимальны:

* [Dijkstra’s Algorithm](https://www.cs.cmu.edu/~deva/papers/tracking2011.pdf)
* [Kruskal’s algorithm](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm)
* [Prim’s algorithm](https://en.wikipedia.org/wiki/Prim%27s_algorithm)
* [Huffman trees](https://en.wikipedia.org/wiki/Huffman_tree)

Существуют и другие глобально оптимальные решения, но жадные алгоритмы быстрее и проще в программировании, чем другие решения.

# Алгоритм Дейкстры

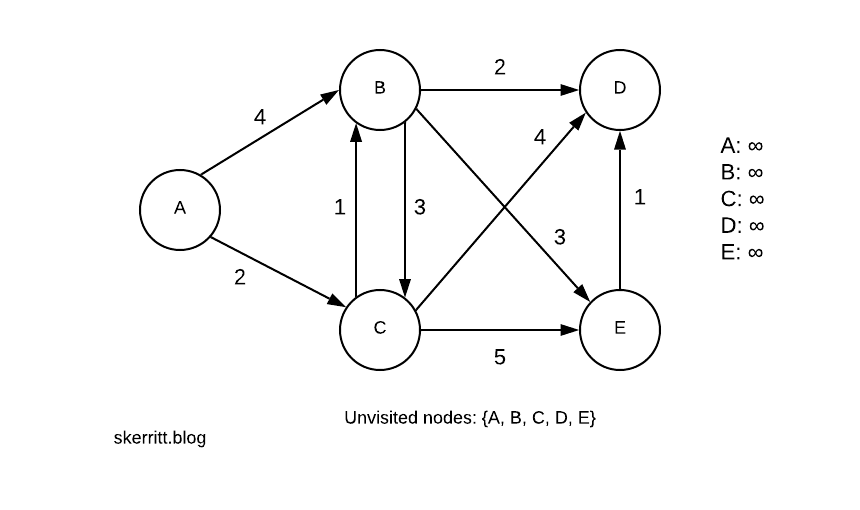
Алгоритм Дейкстры находит кратчайший путь от узла ко всем остальным узлам графа. В нашем примере мы будем использовать [взвешенный ориентированный граф](https://skerritt.blog/graph-theory/). У каждого ребра есть направление, и у каждого ребра есть вес.

Алгоритм Дейкстры имеет много применений. Он может быть очень полезен в дорожных сетях, где вам нужно найти самый быстрый маршрут к месту. Алгоритм также используется для:

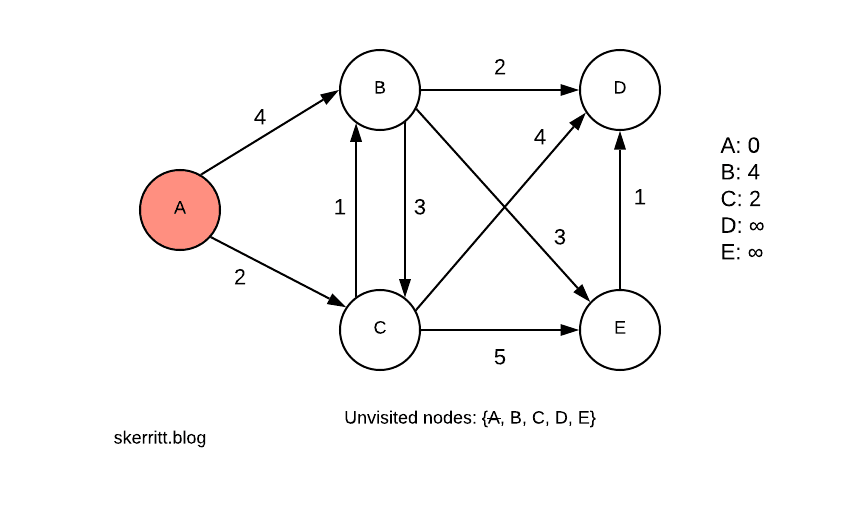
* [IP Routing](https://ieeexplore.ieee.org/document/5381955)
* [A\* Algorithm](https://dev.to/brandonskerritt/a-primer-on-search-algorithms-1768-temp-slug-3978406)
* [Telephone networks](https://www.sciencedirect.com/science/article/pii/S0020019006003498)

Алгоритм следует следующим **правилам**:

1. Каждый раз, когда мы хотим посетить новый узел, мы выберем узел с наименьшим известным расстоянием.
2. Как только мы переместились в узел, мы проверяем каждый из соседних узлов. Мы вычисляем расстояние от соседних узлов до корневых узлов, суммируя стоимость ребер, которые ведут к этому новому узлу.
3. Если расстояние до узла меньше известного расстояния, мы обновим самое короткое расстояние.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--pjCdyD5o--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-33-.png)

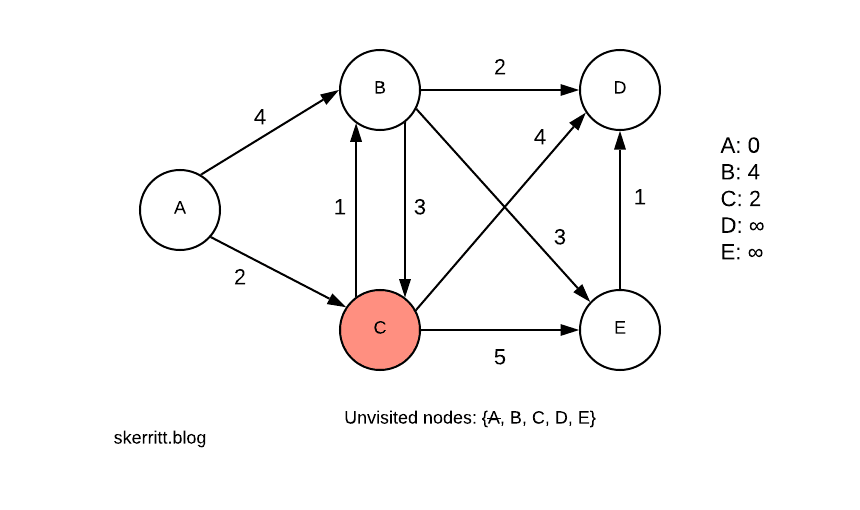
Наш **первый шаг** – выбрать начальный узел. Давайте выберем **A**. Все расстояния назначаются равными бесконечности, так как мы пока не знаем их расстояния, пока не достигнем узла, который знает расстояние.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--oXhOoYh_--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-35-.png)

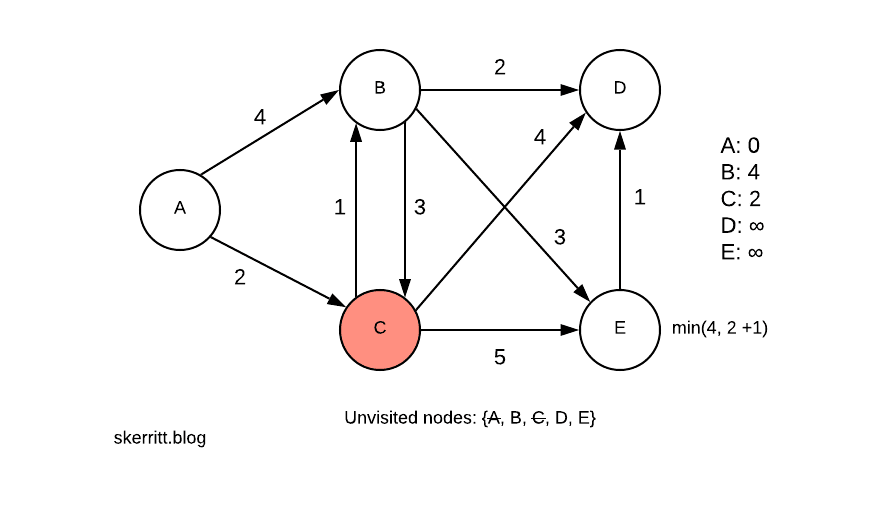
Мы отмечаем **A** в нашем списке **не посещенных узлов**. Расстояние от **A** до **A** равно 0. Расстояние от **A** до **B** равно 4. Расстояние от **A** до **C** равно 2. Наш список расстояний с правой стороны рисунка.

Затем мы выбираем наименьшее ребро, где вершина не была выбрана. Наименьшее ребро **A ->C**, и мы еще не выбрали **C**. Далее мы посещаем **С**.

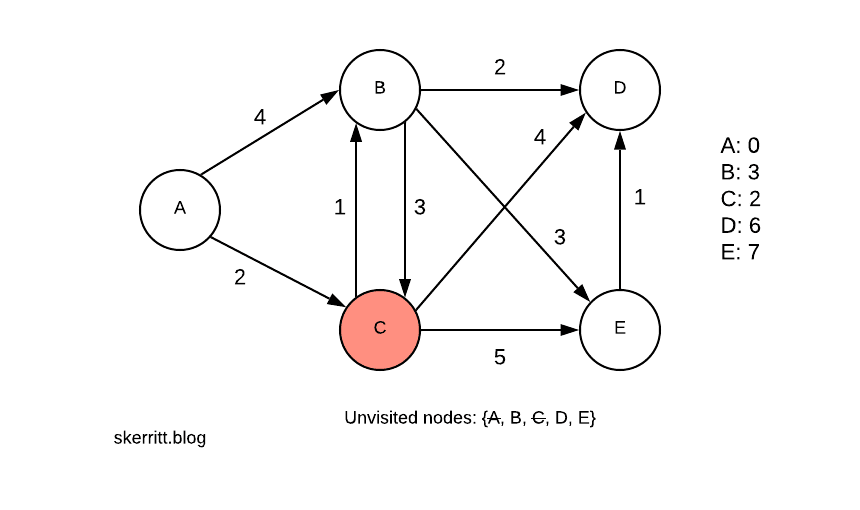
Обратите внимание, как мы выбираем наименьшее расстояние от нашего текущего узла до узла, который мы еще не посетили. Мы жадные. В этом случае жадный метод является глобальным оптимальным решением.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--4aWBozH6--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-36-.png)

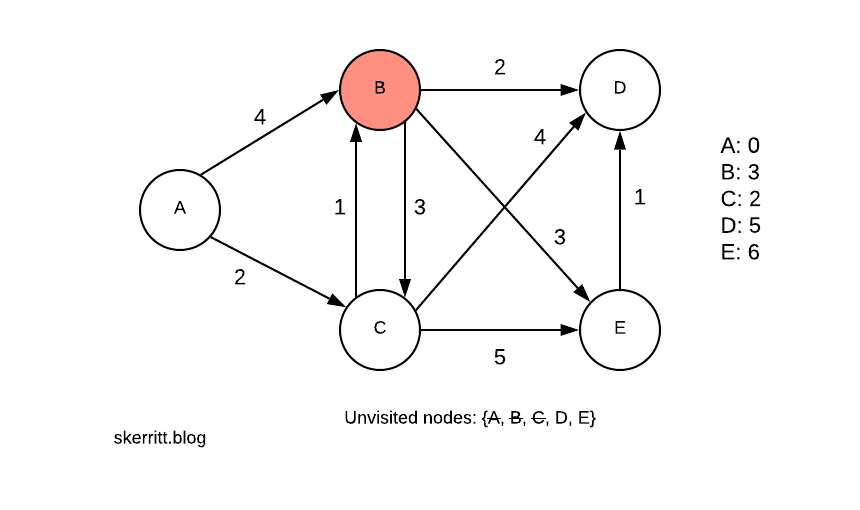
Мы можем добраться до **B** из **C.** Теперь нам нужно выбрать минимум. **min(4, 2 + 1) = 3**.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--F69uLCim--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-37-.png)

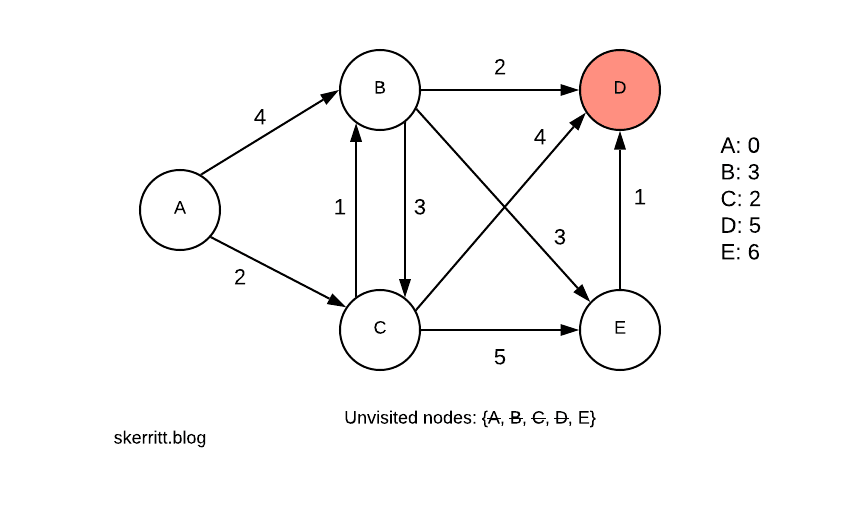
Поскольку **A->C->B** меньше, чем **A->B**, мы обновляем **B** этой информацией. Затем мы добавляем расстояния от других узлов, которые теперь можем достичь.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--Q5phll_---/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-40-.png)

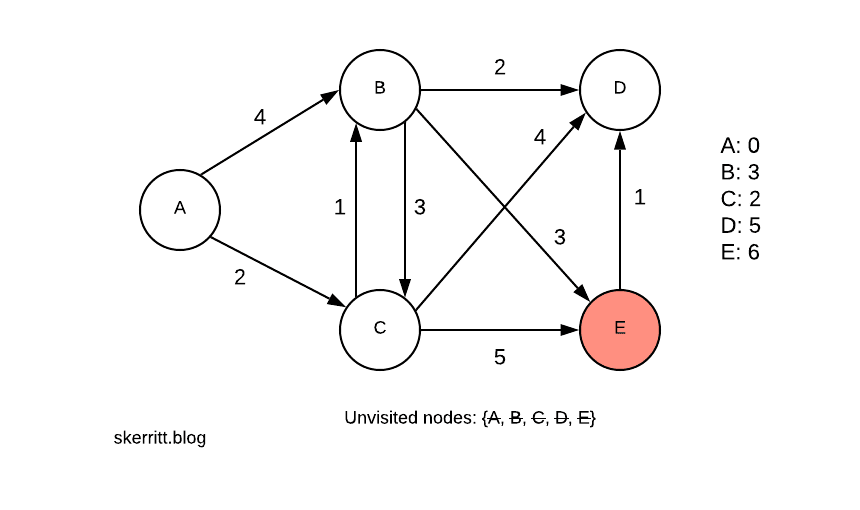
Наша следующая наименьшая вершина с узлом, который мы еще не посетили, это **B**, с 3. Мы посещаем **B**.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--T1THiw3E--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-41-.png)

Мы делаем то же самое для **B**. Затем мы выбираем самую маленькую вершину, которую мы еще не посетили, **D**.

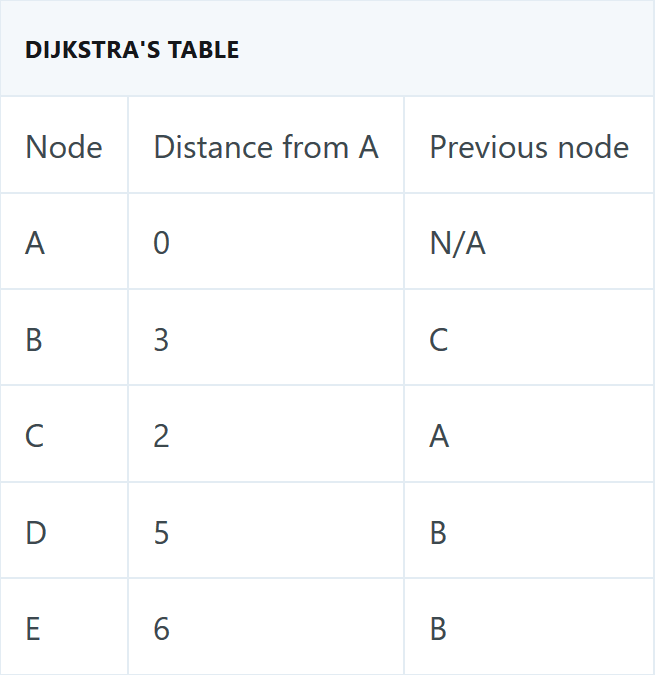
[](https://res.cloudinary.com/practicaldev/image/fetch/s--JbHSvQ1B--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-42-.png)

На этот раз мы не обновляем расстояния. Наш последний узел тогда **E**.

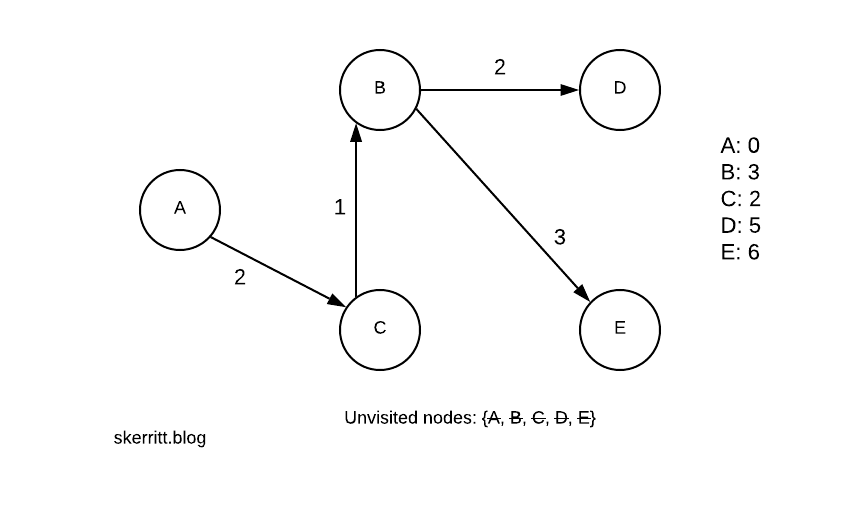
[](https://res.cloudinary.com/practicaldev/image/fetch/s---sid7lGU--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-43-.png)

Там нет обновлений снова. Чтобы найти кратчайший путь от **A** до других узлов, мы пройдемся по нашему графику.

Сначала мы выбираем **A**, затем **C**, а затем **B**. Если вам нужно создать кратчайший путь от **A** до каждого другого узла в виде графика, вы можете запустить этот алгоритм, используя таблицу с правой стороны.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--SIprxT36--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Screenshot_2019-06-23-Greedy-Algorithms.png)

Используя эту таблицу легко нарисовать кратчайшее расстояние от **A** до каждого другого узла на графике:

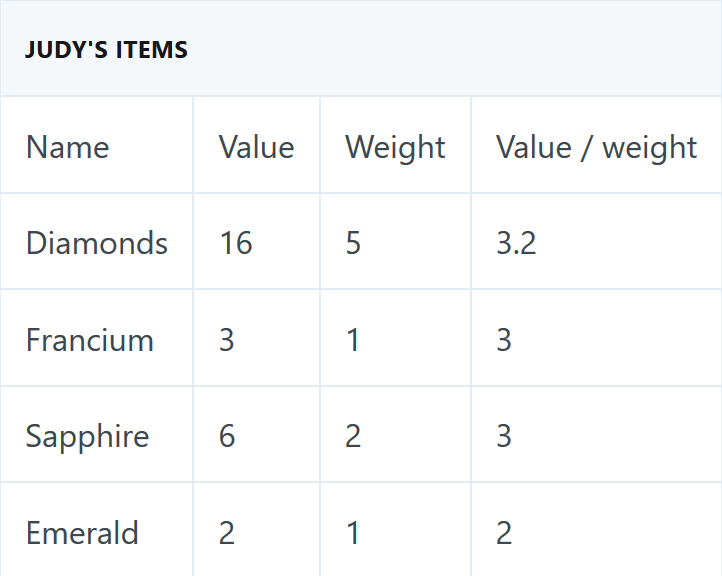
[](https://res.cloudinary.com/practicaldev/image/fetch/s--f4LQulX6--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Blank-Diagram-44-.png)

## Дробная задача о ранце с использованием жадного алгоритма

Представь, что ты вор. Ты врываешься в дом Джуди Холлидей – обладателя Оскара 1951 года за лучшую женскую роль. Джуди кладезь драгоценных камней. Дом Джуди выложен до краев драгоценными камнями.

Ты принес с собой сумку – рюкзак. Максимальный вместимость этой сумки – 7 пунктов веса. У тебя был список всех предметов Джуди из какой-то страховой бумаги. Пункты читаются как:

Первым шагом к решению проблемы дробного ранца является расчет **стоимости/веса** для каждого предмета.

[](https://res.cloudinary.com/practicaldev/image/fetch/s--2GqnZ95v--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https:/skerritt.blog/content/images/2019/06/Screenshot_2019-06-23-Greedy-Algorithms-2-.png)

И теперь мы жадно отбираем самые крупные. Для этого мы можем отсортировать их по значению/весу в порядке убывания. К счастью для нас, они уже отсортированы. Самый большой – 3,2.

knapsack value = 16

knapsack total weight = 5 (out **of** 7)

Затем мы выбираем **Francium** (я знаю, что это не драгоценный камень, но Джуди немного странная )

knapsack value = 19

knapsack weight = 6

Теперь мы добавляем **Сапфир**. Но если мы добавим Сапфир, наш общий вес достигнет 8 пунктов.

В задаче дробного ранца мы можем разрезать предметы, чтобы взять их дроби. У нас в сумке осталось 1 пункт веса. Наш сапфир имеет вес 2 пункта. Мы рассчитываем соотношение:

Свободное место в рюкзаке / вес товара

А затем умножьте это соотношение на стоимость предмета, чтобы узнать, сколько стоимости этого предмета мы можем взять.

**1/2 \* 6 = 3**

knapsack value = 21

knapsack weight = 7

Жадный алгоритм может оптимально решить проблему дробного ранца, но он не может оптимально решить проблему ранцев **{0, 1}**. В этой задаче вместо того, чтобы брать часть элемента, вы либо берете его **{1}**, либо не берете **{0}**. Чтобы решить эту проблему, вам нужно использовать [динамическое программирование](https://skerritt.blog/dynamic-programming/).

Сложность алгоритма для него **O(n log n)**. Расчетное **значение/вес** O (1). Наш основной шаг – сортировка по наибольшему **значению/весу**, что занимает **O(n log n)** времени.

# 2. Реализация алгоритма

def Gadniy\_algoritm(maze, start, end):  
 #Создаем список соседних клеток  
 neighbors=[(0,1),(0,-1),(1,0),(-1,0)]  
 #Создаем список посещенных клеток  
 visited=set()  
 #создадим очередь для хранения пути  
 queue = [[start]]  
 #пока очередь не пуста  
 while queue:  
 path = queue.pop(0) #достаем 1 путь из очереди  
 current = path[-1] #берем последнюю точку пути  
 if current not in visited:  
 visited.add(current)  
 for neighbor in neighbors: #для каждого соседа текущей точки  
 #находим координаты точки  
 x = current[0] + neighbor[0]  
 y = current[1] + neighbor[1]  
 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":#Если сосед находится в пределах лабиринта и не стена  
 new\_path = path+[(x,y)] #создаем новый путь до соседа  
 # если сосед равен конечной точку то возвращаем путь  
 if(x, y) == end:  
 return new\_path  
 else: # иначе добавляем новый путь в очередь  
 queue.append(new\_path)  
 return[]# Если не удалось найти путь до конечной точки, возвращаем пустой список

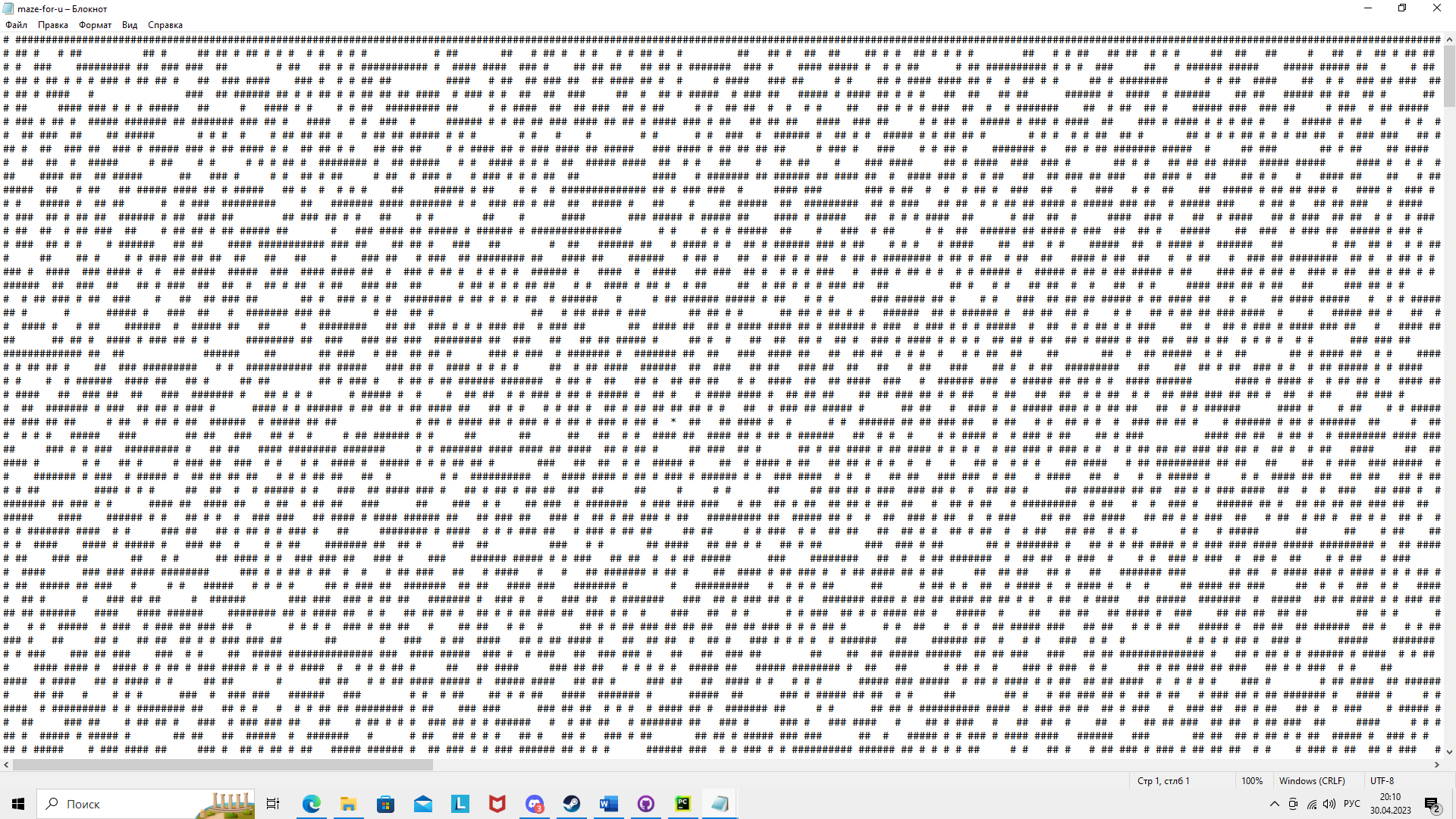
Этот алгоритм, я использовал дважды.

1. Для нахождения кротчайшего пути к ключу.
2. Для нахождения от ключа до выхода.

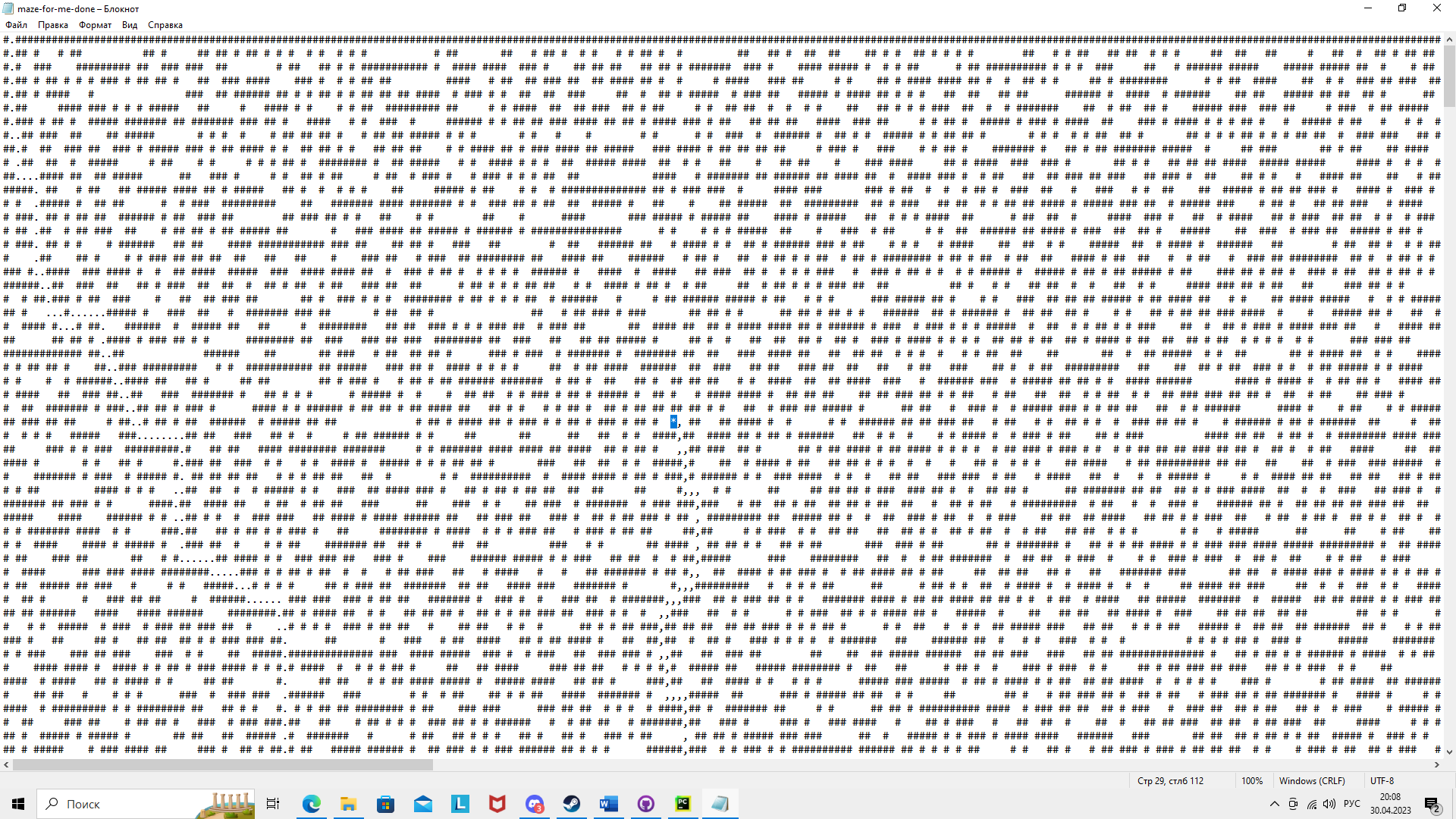
# 

# Пример работы

1. Используем программу «gen\_lab\_origin» для создания файла «maze-for-u» и получаем лабиринт:



1. Используем программу под названием «laba 3» после ее запуска появляется текстовый файл «maze-for-me-done»
2. Открываем текстовый файл и получаем результат:



1. Путь до ключа отображается «.», путь от ключа к выходу отображает «,».

# Заключение

В ходе проделанной работы была создана программа для нахождения ключа в лабиринте и нахождения выхода. Благодаря ей мы можем увидеть самый эффективный путь в лабиринте.

В результате было разработан метод, который позволяет брать элементы из текстового документа и заполнять их в массив.

# Список литературы

1. Скиена С. Стивен Алгоритмы. Руководство по разработке. 3-е изд / Скиена С. Стивен. – СПб.: Springer, 2022. – 848 с. – Текст:  
   непосредственный.
2. Станковец А.В. АЛГОРИТМЫ НА ГРАФАХ / Станковец А.В. – Текст: электронный // M[ODERN SCIENCE](https://www.elibrary.ru/contents.asp?id=44150018). — 2020. — №10-2. — С.532-536. — EDN: [pdmxpx](https://www.elibrary.ru/pdmxpx)
3. Книга «Парадигмы алгоритмического проектирования(жадные алгоритмы, разделяй и властвуй и динамическое программирование)- [Бесплатная электронная книга о жадных алгоритмах, «разделяй и властвуй» и динамическом программировании - DEV Community](https://dev.to/brandonskerritt/a-free-ebook-on-greedy-algorithms-divide-conquer-and-dynamic-programming-1712)

# Приложение 1

## Листинг программы

#функция для преобразования лабиринта  
def readmaze(file\_name):  
 with open(file\_name) as f:  
 maze = [list(line.strip()) for line in f]#создаем список который содержит строки текстового файла  
 return maze  
  
#поиск точек начало и конца  
def tochky\_start\_end(maze):  
 start = (0, maze[0].index(" ")) #Ищем вход в первой линии  
 end = (len(maze)-1, maze[-1].index(" ")) #Ищем выход в последней  
 return start, end  
  
#поиск ключа  
def key\_tochka(maze):  
 for i, g in enumerate(maze): #мы ищем координаты ключа(i,g.index("\*"))  
 if "\*" in g:  
 print(i,g.index("\*"))  
 return(i, g.index("\*"))  
  
#Жадный алгоритм для поиска пути в лабиринте  
def Gadniy\_algoritm(maze, start, end):  
 #Создаем список соседних клеток  
 neighbors=[(0,1),(0,-1),(1,0),(-1,0)]  
 #Создаем список посещенных клеток  
 visited=set()  
 #создадим очередь для хранения пути  
 queue = [[start]]  
 #пока очередь не пуста  
 while queue:  
 path = queue.pop(0) #достаем 1 путь из очереди  
 current = path[-1] #берем последнюю точку пути  
 if current not in visited:  
 visited.add(current)  
 for neighbor in neighbors: #для каждого соседа текущей точки  
 #находим координаты точки  
 x = current[0] + neighbor[0]  
 y = current[1] + neighbor[1]  
 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":#Если сосед находится в пределах лабиринта и не стена  
 new\_path = path+[(x,y)] #создаем новый путь до соседа  
 # если сосед равен конечной точку то возвращаем путь  
 if(x, y) == end:  
 return new\_path  
 else: # иначе добавляем новый путь в очередь  
 queue.append(new\_path)  
 return[]# Если не удалось найти путь до конечной точки, возвращаем пустой список  
def new\_maze(maze, path, mark): #обьявляем лабиринт и помечаем путь  
 for cord in path:  
 x, y = cord  
 maze[x][y] = mark  
 return maze  
def new\_file(maze, file\_name): #запись лабиринта в отдельный файл  
 with open(file\_name, "w") as f:  
 for g in maze:  
 f.write("".join(g)+"\n")  
  
maze = readmaze("maze-for-u.txt")  
start, end = tochky\_start\_end(maze)  
key = key\_tochka(maze)  
path1 = Gadniy\_algoritm(maze, start, key)  
path2 = Gadniy\_algoritm(maze, key, end)  
maze = new\_maze(maze, path1, ".")  
maze = new\_maze(maze, path2, ",")  
x, y = key  
maze[x][y] = "\*"  
new\_file(maze, "maze-for-me-done.txt")