ПРИМЕЧАНИЕ

Бинарный поиск работает только в том случае, если список отсортирован. Например, имена в телефонной книге хранятся в алфавитном порядке, и вы можете воспользоваться бинарным поиском. А что произойдет, если имена не будут отсортированы?

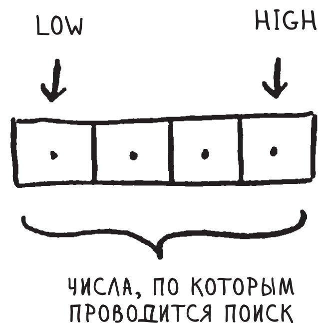
Посмотрим, как написать реализацию бинарного поиска на Python. В следующем примере кода используется массив. Если вы не знаете, как работают массивы, не беспокойтесь: эта тема рассматривается в следующей главе. Пока достаточно знать, что серию элементов можно сохранить в непрерывной последовательности ячеек, которая называется массивом. Нумерация ячеек начинается с 0: первая ячейка находится в позиции с номером 0, вторая — в позиции с номером 1 и т. д.

ПРИМЕЧАНИЕ

Поскольку в Python массивы называются списками, я буду использовать термины "список" и "массив" как взаимозаменяемые.

Функция `binary_search` получает отсортированный массив и значение. Если значение присутствует в массиве, то функция возвращает его позицию. При этом мы должны следить за тем, в какой части массива проводится поиск. Вначале это весь массив:

```
low = 0
high = len(arr) - 1
```



Каждый раз алгоритм проверяет средний элемент:

```
mid = (low + high) // 2
guess = arr[mid]
```

Если значение (low+high) нечетно, то Python автоматически округляет значение mid в меньшую сторону

Если названное число было слишком мало, то переменная `low` обновляется соответственно:

```
if guess < item:
  low = mid + 1
```

А если догадка была слишком велика, то обновляется переменная `high`. Полный код выглядит так:

```
def binary_search(arr, item):
  low = 0
  high = len(arr) - 1

  while low <= high:
    mid = (low + high) // 2
    guess = arr[mid]
    if guess == item:
      return mid
    elif guess > item:
      high = mid - 1
    else:
      low = mid + 1
  return None

my_list = [1, 3, 5, 7, 9]

print(binary_search(my_list, 3)) # => 1
print(binary_search(my_list, -1)) # => None
```

В переменных low и high хранятся границы той части списка, в которой выполняется поиск

Пока эта часть не сократится до одного элемента...

...проверяем средний элемент

Значение найдено

Много

Мало

Значения не существует

А теперь протестируем функцию!

Вспомните: нумерация элементов начинается с 0. Второй ячейке соответствует индекс 1

"None" в Python означает "ничто". Это признак того, что элемент не найден

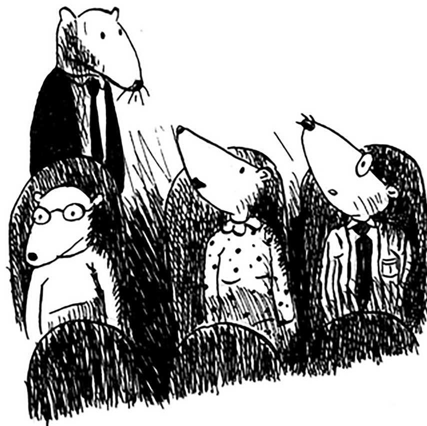
УПРАЖНЕНИЯ

1.1 Имеется отсортированный список из 128 имен, и вы ищете в нем значение методом бинарного поиска. Какое максимальное количество проверок для этого может потребоваться?

1.2 Предположим, размер списка увеличился вдвое. Как изменится максимальное количество проверок?

Теперь на основе этой функции можно написать функцию сортировки выбором:

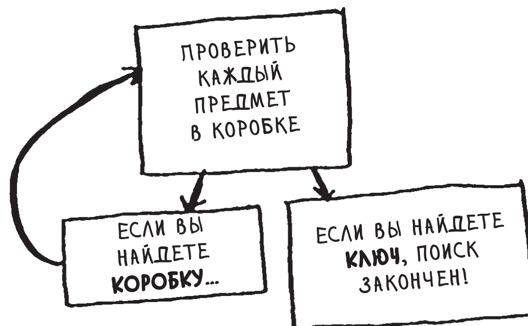
```
def selectionSort(arr):              ◄·········· Сортирует массив
  newArr = []
  copiedArr = list(arr) // копирует массив перед изменением
  for i in range(len(copiedArr)):
      smallest = findSmallest(copiedArr)  ◄········ Находит наименьший элемент
      newArr.append(copiedArr.pop(smallest))         в массиве и добавляет его
  return newArr                                      в новый массив

print(selectionSort([5, 3, 6, 2, 10]))
```



Шпаргалка

- Память компьютера напоминает огромный шкаф с ящиками.
- Если вам потребуется сохранить набор элементов, воспользуйтесь массивом или списком.
- В массиве все элементы хранятся в памяти рядом друг с другом.
- В связанном списке элементы распределяются в произвольных местах памяти, при этом в каждом элементе хранится адрес следующего элемента.
- Массивы обеспечивают быстрое чтение.
- Списки обеспечивают быструю вставку и удаление.

Есть и альтернативное решение.



1. Просмотреть содержимое коробки.
2. Если вы найдете коробку, вернуться к шагу 1.
3. Если вы найдете ключ, поиск закончен!

Какое решение кажется вам более простым? Первое решение можно построить на цикле `while`. Пока куча коробок не пуста, взять очередную коробку и проверить ее содержимое:

```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print("found the key!")
```

Второй способ основан на рекурсии. *Рекурсией* называется вызов функцией самой себя. Второе решение на псевдокоде может выглядеть так:

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(item)      <- Рекурсия!
    elif item.is_a_key():
      print("found the key!")
```

```
> 3...2...1...0...-1...-2...
```

Чтобы прервать выполнение сценария, нажмите Ctrl+C.

Когда вы пишете рекурсивную функцию, в ней необходимо указать, в какой момент следует прервать рекурсию. Вот почему *каждая рекурсивная функция состоит из двух частей: базового случая и рекурсивного случая*. В рекурсивном случае функция вызывает сама себя. В базовом случае функция себя не вызывает... чтобы предотвратить зацикливание.

Добавим базовый случай в функцию `countdown`:

```
def countdown(i):
  print(i)
  if i <= 1:        <.........   Базовый случай
    return
  else:             <.........   Рекурсивный случай
    countdown(i-1)

countdown(3)
```

Теперь функция работает, как было задумано. Это выглядит примерно так:

Такая структура данных называется *стеком*. Стек — простая структура данных. А теперь самое неожиданное: все это время вы пользовались стеком, не подозревая об этом!

Стек вызовов

Во внутренней работе вашего компьютера используется стек, называемый *стеком вызовов*. Давайте посмотрим, как он работает. Предположим, имеется простая функция:

```
def greet(name):
    print("hello, " + name + "!")
    greet2(name)
    print("getting ready to say bye...")
    bye()
```

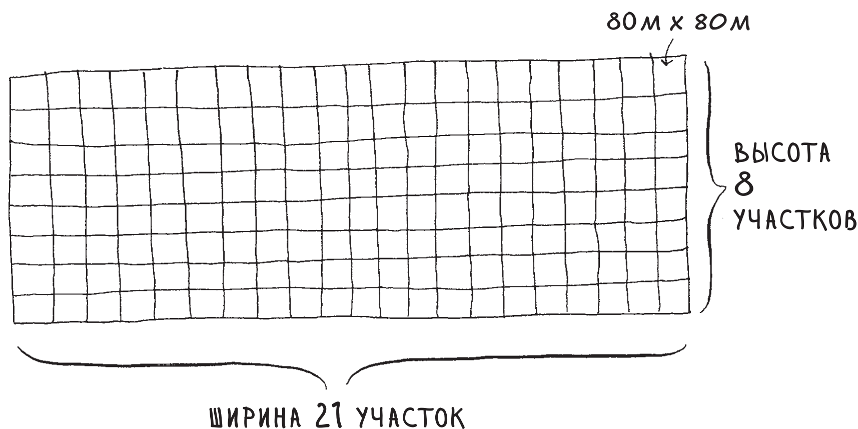
Эта функция приветствует вас, после чего вызывает две другие функции. Вот эти две функции:

```
def greet2(name):
    print("how are you, " + name + "?")

def bye():
    print("ok bye! ")
```

Разберемся, что происходит при вызове функции.

> **ПРИМЕЧАНИЕ**
>
> Для простоты я привожу только вызовы функций `greet`, `greet2` и `bye` и опускаю вызовы функции `print`.

Предположим, в программе используется вызов `greet("maggie")`. Сначала ваш компьютер выделяет блок памяти для этого вызова функции.



Вспомните, как работает стратегия «разделяй и властвуй».

1. Определите простейший случай как базовый.
2. Придумайте, как свести задачу к базовому случаю.

«Разделяй и властвуй» — не простой алгоритм, который можно применить для решения задачи. Скорее это подход к решению задачи. Рассмотрим еще один пример.

Имеется массив чисел.

Нужно просуммировать все числа и вернуть сумму. Сделать это в цикле совсем не сложно:

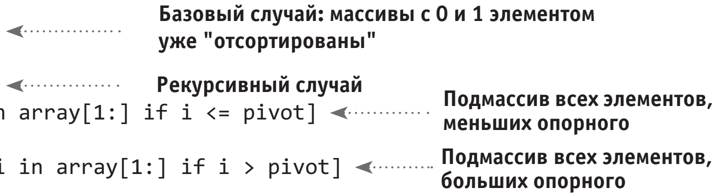
```
def sum(arr):
  total = 0
  for x in arr:
    total += x
  return total

print(sum([1, 2, 3, 4]))
```

Но как сделать то же самое с использованием рекурсивной функции?

Шаг 1: определить базовый случай. Как выглядит самый простой массив, который вы можете получить? Подумайте, как должен выглядеть про-

А вот как выглядит программный код быстрой сортировки:

```
def quicksort(array):
  if len(array) < 2:
    return array
  else:
    pivot = array[0]
    less = [i for i in array[1:] if i <= pivot]

    greater = [i for i in array[1:] if i > pivot]

    return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10, 5, 2, 3]))
```



Снова об «О-большом»

Алгоритм быстрой сортировки уникален тем, что его скорость зависит от выбора опорного элемента. Прежде чем рассматривать быструю сортировку, вспомним наиболее типичные варианты времени выполнения для «О-большое».

Оценки для медленного компьютера, выполняющего 10 операций в секунду

На графиках приведены примерные оценки времени при выполнении 10 операций в секунду. Они не претендуют на точность, а всего лишь дают представление о том, насколько различается время выполнения. Конечно,

на практике ваш компьютер способен выполнять гораздо больше 10 операций в секунду.

Для каждого времени выполнения также приведен пример алгоритма. Возьмем алгоритм сортировки выбором, о котором вы узнали в главе 2. Он обладает временем $O(n^2)$, и это довольно медленный алгоритм.

Другой алгоритм сортировки — так называемая *сортировка слиянием* — работает за время $O(n \log n)$. Намного быстрее! С быстрой сортировкой дело обстоит сложнее. В худшем случае быстрая сортировка работает за время $O(n^2)$.

Ничуть не лучше сортировки выбором! Но это худший случай, а в среднем быстрая сортировка выполняется за время $O(n \log n)$. Вероятно, вы спросите:

- что в данном случае понимается под «*худшим*» и «*средним*» случаем?
- если быстрая сортировка в среднем выполняется за время $O(n \log n)$, а сортировка слиянием выполняется за время $O(n \log n)$ всегда, то почему бы не использовать сортировку слиянием? Разве она не быстрее?

Сортировка слиянием и быстрая сортировка

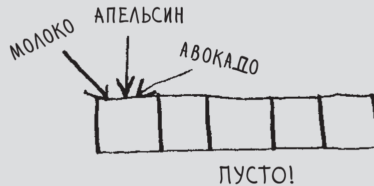
Допустим, у вас имеется простая функция для вывода каждого элемента в списке:

```
def print_items(myList):
  for item in myList:
    print(item)
```

Эта функция последовательно перебирает все элементы списка и выводит их. Так как функция перебирает весь список, она выполняется за время $O(n)$. Теперь предположим, что вы изменили эту функцию и она делает секундную паузу перед выводом:

```
from time import sleep
def print_items2(myList):
  for item in myList:
    sleep(1)
    print(item)
```

Посмотрите: все товары находятся в своих позициях. В реальности такого идеального однозначного сопоставления, скорее всего, не получится. Товарам придется соседствовать. В одной позиции будут находиться несколько товаров, а другие позиции останутся пустыми.



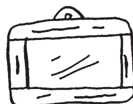
Мы обсудим эту ситуацию в разделе о коллизиях. А пока просто знайте, что, хотя хеш-таблицы очень полезны, они редко точно связывают элементы с позициями.

Кстати, такое взаимно-однозначное связывание называется инъективной функцией.

Запомните это словечко: пригодится, чтобы произвести впечатление на друзей!

Скорее всего, вам никогда не придется заниматься реализацией хеш-таблиц самостоятельно. В любом приличном языке существует реализация хеш-таблиц. В Python тоже есть хеш-таблицы; они называются *словарями*. Новая хеш-таблица задается пустыми фигурными скобками:

```
>>> book = {}
```



`book` — новая хеш-таблица. Добавим в `book` несколько цен:

```
>>> book["апельсин"] = 0.67      <-- Апельсины стоят 67 центов
>>> book["молоко"] = 1.49        <-- Молоко стоит 1 доллар 49 центов
>>> book["авокадо"] = 1.49
>>> print(book)
{'avocado': 1.49, 'apple': 0.67, 'milk': 1.49}
```

Пока все просто! А теперь запросим цену авокадо:

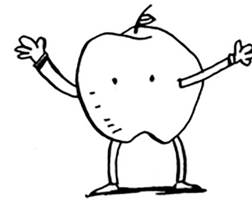
```
>>> print(book["авокадо"])
1.49
```
Цена авокадо

Хеш-таблица состоит из ключей и значений. В хеше book имена продуктов являются ключами, а цены — значениями. Хеш-таблица связывает ключи со значениями.

В следующем разделе приведены примеры, в которых хеш-таблицы приносят большую пользу.

УПРАЖНЕНИЯ

Очень важно, чтобы хеш-функции были последовательными, то есть неизменно возвращали один и тот же результат для одинаковых входных данных. Если это условие будет нарушено, вы не сможете найти свой элемент после того, как он будет помещен в хеш-таблицу!



Какие из следующих функций являются последовательными?

5.1 `f(x) = 1` ← **Возвращает "1" для любых входных значений**

5.2 `f(x) = random.random()` ← **Возвращает случайное число**

5.3 `f(x) = next_empty_slot()` ← **Возвращает индекс следующего пустого элемента в хеш-таблице**

5.4 `f(x) = len(x)` ← **Возвращает длину полученной строки**

Добавим в телефонную книгу несколько номеров:

```
>>> phone_book["Дженни"] = "8675309"
>>> phone_book["служба спасения"] = "911"
```

Вот и все! Теперь предположим, что вы хотите найти номер телефона Дженни (Jenny). Просто передайте ключ хешу:

```
>>> print(phone_book["Дженни"])
8675309    <·········  Номер Дженни
```

ХЕШ-ТАБЛИЦА КАК ТЕЛЕФОННАЯ КНИГА

А теперь представьте, что то же самое вам пришлось бы делать с массивом.

Как бы вы это сделали? Хеш-таблицы упрощают моделирование отношений между объектами.

Хеш-таблицы используются для поиска соответствий в гораздо большем масштабе. Например, представьте, что вы хотите перейти на веб-сайт — допустим, *http://adit.io*. Ваш компьютер должен преобразовать символическое имя *adit.io* в IP-адрес.

ADIT.IO → 173.255.248.55

Для любого посещаемого веб-сайта его имя преобразуется в IP-адрес:

GOOGLE.COM → 74.125.239.133
FACEBOOK.COM → 173.252.120.6
SCRIBD.COM → 23.235.47.175

Связать символическое имя с IP-адресом? Идеальная задача для хеш-таблиц! Этот процесс называется *преобразованием DNS*. Хеш-таблицы — всего лишь один из способов реализации этой функциональности. На компьютерах существует кэш DNS, в котором хранятся подобные сопоставления для недавно посещенных сайтов и который удобно создавать с помощью хеш-таблицы.

Сначала создадим хеш для хранения информации об уже проголосовавших людях:

```
>>> voted = {}
```

Когда кто-то приходит голосовать, проверьте, присутствует ли его имя в хеше:

```
>>> value = "Том" in voted
```

Функция `value` принимает значение `True`, если ключ `"Том"` присутствует в хеш-таблице. В противном случае она принимает значение `False`. С помощью этой функции можно проверить, голосовал избиратель ранее или нет!

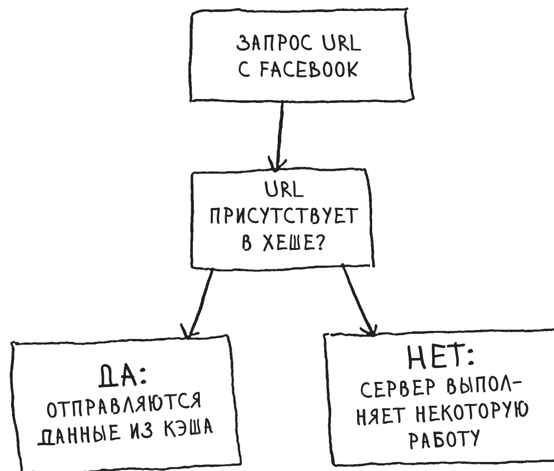
Код выглядит так:

```
voted = {}

def check_voter(name):
  if name in voted:
    print("Выгнать его!")
  else:
    voted[name] = True
    print("Допустить к голосованию!")
```

facebook.com/about → ДАННЫЕ СТРАНИЦЫ С ИНФОРМАЦИЕЙ О FACEBOOK

facebook.com → ДАННЫЕ ДОМАШНЕЙ СТРАНИЦЫ

Когда вы посещаете страницу на сайте Facebook, сайт сначала проверяет, хранится ли страница в хеше.



Вот как это выглядит в коде:

```
cache = {}

def get_page(url):
  if url in cache:
    return cache[url]          ◄······ Возвращаются кэшированные данные
  else:
    data = get_data_from_server(url)
    cache[url] = data          ◄······ Данные сначала сохраняются в кэше
    return data
```

Здесь сервер выполняет работу только в том случае, если URL не хранится в кэше. Однако перед тем, как возвращать данные, вы сохраняете их в кэше. Когда пользователь в следующий раз запросит тот же URL-адрес, данные

Напомню, что выражение `graph["вы"]` вернет список всех ваших соседей, например `["Алиса", "Боб", "Клэр"]`. Все они добавляются в очередь поиска.

А теперь рассмотрим остальное:

```
while search_queue:                        ← Пока очередь непуста...
    person = search_queue.popleft()        ← из очереди извлекается первый человек
    if person_is_seller(person):           ← Проверяем, является ли этот человек
                                             продавцом манго
        print(person + " продавец манго!") ← Да, это продавец манго
        return True
    else:
        search_queue += graph[person]      ← Нет, не является. Все друзья этого че-
                                             ловека добавляются в очередь поиска
return False   ← Если выполнение дошло
                 до этой строки, значит,
                 в очереди нет продавца манго
```

И последнее: нужно определить функцию `person_is_seller`, которая сообщает, является ли человек продавцом манго. Например, функция может выглядеть так:

```
def person_is_seller(name):
    return name[-1] == 'm'
```

Эта функция проверяет, заканчивается ли имя на букву «m», и если заканчивается, этот человек считается продавцом манго. Проверка довольно глупая, но для нашего примера сойдет. А теперь посмотрим, как работает поиск в ширину.

А вот окончательная версия кода поиска в ширину, в которой учтено это обстоятельство:

```
def search(name):
    search_queue = deque()
    search_queue += graph[name]
    searched = set()          # Этот массив используется для отслеживания уже проверенных людей
    while search_queue:
        person = search_queue.popleft()
        if not person in searched:     # Человек проверяется только в том случае, если он не проверялся ранее
            if person_is_seller(person):
                print(person + " продавец манго!")
                return True
            else:
                search_queue += graph[person]
                searched.add(person)     # Человек помечается как уже проверенный
    return False

search("вы")
```

Попробуйте выполнить этот код самостоятельно. Замените функцию `person_is_seller` чем-то более содержательным и посмотрите, выведет ли она то, что вы ожидали.

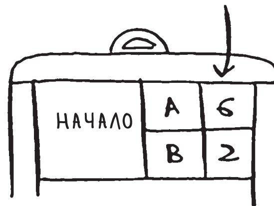
Время выполнения

Если поиск продавца манго был выполнен по всей сети, значит, вы прошли по каждому ребру (напомню: ребром называется соединительная линия или линия со стрелкой, ведущая от одного человека к другому). Таким образом, время выполнения составляет как минимум *O*(*количество ребер*).

Также в программе должна храниться очередь поиска. Добавление одного человека в очередь выполняется за постоянное время: $O(1)$. Выполнение операции для каждого человека потребует суммарного времени *O*(*количество людей*). Поиск в ширину выполняется за время *O*(*количество людей + количество ребер*), что обычно записывается в форме $O(V+E)$ (V — количество вершин, E — количество ребер).

Как представить веса этих ребер? Почему бы не воспользоваться другой хеш-таблицей?

```
graph["начало"] = {}
graph["начало"]["a"] = 6
graph["начало"]["b"] = 2
```



Итак, `graph["начало"]` является хеш-таблицей. Для получения всех соседей начального узла можно воспользоваться следующим выражением:

```
>>> print(list(graph["начало"].keys()))
["a", "b"]
```

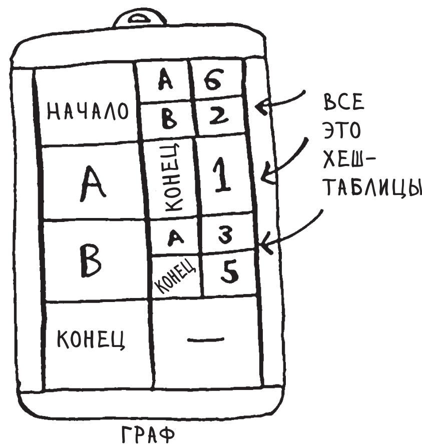
Одно ребро ведет из начального узла в A, а другое — из начального узла в B. А если вы захотите узнать веса этих ребер?

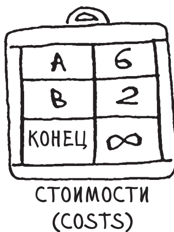
```
>>> print(graph["начало"]["a"])
6
>>> print(graph["начало"]["b"])
2
```

Включим в граф остальные узлы и их соседей:

```
graph["a"] = {}
graph["a"]["fin"] = 1

graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["конец"] = 5

graph["конец"] = {}    <-------- У конечного узла нет соседей
```

Полная хеш-таблица графа выглядит так:





Также понадобится хеш-таблица для хранения стоимостей всех узлов.

Стоимость узла определяет, сколько времени потребуется для перехода к этому узлу от начального узла. Вы знаете, что переход от начального узла к узлу B занимает 2 минуты. Вы знаете, что для перехода к узлу A требуется 6 минут (хотя, возможно, вы найдете более быстрый путь). Вы не знаете, сколько времени потребуется для достижения конечного узла. Если стоимость еще неизвестна, она считается бесконечной. Можно ли представить *бесконечность* (infinity) в Python? Оказывается, можно:

```
infinity = math.inf
```

Код создания таблицы стоимостей `costs`:

```
infinity = math.inf
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity
```

Для родителей также создается отдельная таблица:

A	НАЧАЛО
B	НАЧАЛО
КОНЕЦ	—

РОДИТЕЛИ
(PARENTS)

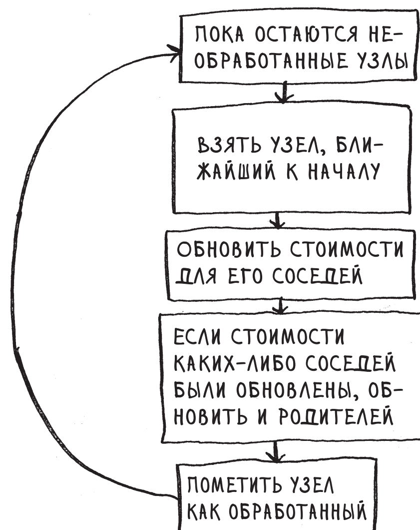
Код создания хеш-таблицы родителей:

```
parents = {}
parents["a"] = "начало"
parents["b"] = "начало"
parents["fin"] = None
```

Наконец, вам нужен массив для отслеживания всех уже обработанных узлов, так как один узел не должен обрабатываться многократно:

```
processed = set()
```

На этом подготовка завершается. Теперь обратимся к алгоритму.



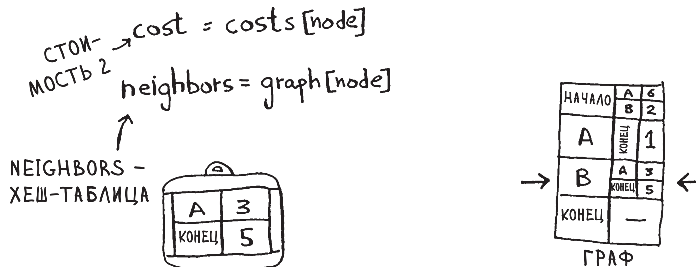
Сначала я приведу код, а потом мы разберем его более подробно.

```
node = find_lowest_cost_node(costs)    # Найти узел с наименьшей стоимостью среди необработанных
while node is not None:    # Если обработаны все узлы, цикл while завершен
    cost = costs[node]
    neighbors = graph[node]
    for n in neighbors.keys():    # Перебрать всех соседей (neighbors) текущего узла
        new_cost = cost + neighbors[n]
        if costs[n] > new_cost:    # Если к соседу можно быстрее добраться через текущий узел...
            costs[n] = new_cost    # ...обновить стоимость для этого узла
            parents[n] = node    # Этот узел становится новым родителем для соседа
    processed.add(node)    # Узел помечается как обработанный
    node = find_lowest_cost_node(costs)    # Найти следующий узел для обработки и повторить цикл
```

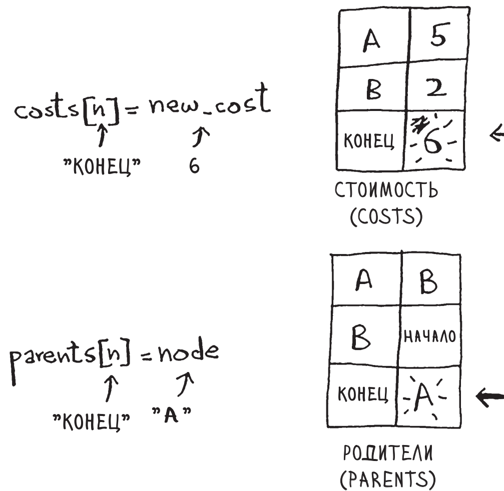
Так выглядит алгоритм Дейкстры на языке Python! Код функции будет приведен далее, а пока рассмотрим пример использования алгоритма в действии.

Найти узел с наименьшей стоимостью.

Получить стоимость и соседей этого узла.



Через узел A можно добраться быстрее! Обновим стоимость и родителя.



После того как все узлы будут обработаны, алгоритм завершается. Надеюсь, этот пошаговый разбор помог вам чуть лучше понять алгоритм. С функцией `find_lowest_cost_node` узел с наименьшей стоимостью находится проще простого. Код выглядит так:

```
def find_lowest_cost_node(costs):
    lowest_cost = math.inf
    lowest_cost_node = None
    for node in costs:  ◄········ Перебрать все узлы
        cost = costs[node]
        if cost < lowest_cost and node not in processed: ◄······· Если это узел с наименьшей стоимостью из уже виденных и он еще не был обработан...
            lowest_cost = cost  ◄········ ...он назначается новым узлом с наименьшей стоимостью
            lowest_cost_node = node
    return lowest_cost_node
```

Чтобы найти узел с наименьшей стоимостью, мы каждый раз перебираем все узлы. Существует более эффективный вариант этого алгоритма. Он использует структуру данных, называемую очередью с приоритетом. Эта очередь строится на основе другой структуры данных — кучи. Если вам интересны очереди с приоритетом и кучи, ознакомьтесь с разделом о кучах в последней главе книги.

Если условие выполняется, то станция сохраняется в `best_station`. Наконец, после завершения цикла `best_station` добавляется в итоговый список станций:

```
final_stations.add(best_station)
```

Также необходимо обновить содержимое `states_needed`. Те штаты, которые входят в зону покрытия станции, больше не нужны:

```
states_needed -= states_covered
```

Цикл продолжается, пока множество `states_needed` не станет пустым. Полный код цикла `for` выглядит так:

```
while states_needed:
  best_station = None
  states_covered = set()
  for station, states in stations.items():
    covered = states_needed & states
    if len(covered) > len(states_covered):
      best_station = station
      states_covered = covered

states_needed -= states_covered
final_stations.add(best_station)
```

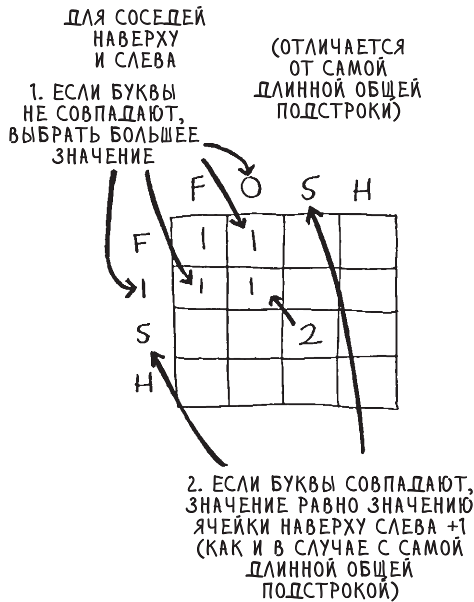
Остается вывести содержимое `final_stations`:

```
>>> print(final_stations)
set(['ktwo', 'kthree', 'kone', 'kfive'])
```

Этот результат совпадает с вашими ожиданиями? Вместо станций 1, 2, 3 и 5 можно было выбрать станции 2, 3, 4 и 5. Сравним время выполнения жадного алгоритма со временем точного алгоритма.

КОЛИЧЕСТВО СТАНЦИЙ	$O(2^n)$ ТОЧНЫЙ АЛГОРИТМ	$O(n^2)$ ЖАДНЫЙ АЛГОРИТМ
5	3.2 с	2.5 с
10	102.4 с	10 с
32	13.6 года	102.4 с
100	4×10^{21} года	16.67 мин

А теперь моя формула для заполнения каждой ячейки:



На псевдокоде эта формула реализуется так:

```
if word_a[i] == word_b[j]:        ◄········ Буквы совпадают
  cell[i][j] = cell[i-1][j-1] + 1
else:   ◄································ Буквы не совпадают
  cell[i][j] = max(cell[i-1][j], cell[i][j-1])
```

Поздравляю — вы справились! Безусловно, это была одна из самых сложных глав в книге. Находит ли динамическое программирование практическое применение? Да, находит.

- Биологи используют самую длинную общую подпоследовательность для выявления сходства в цепях ДНК. По этой метрике можно судить о сходстве двух видов животных, двух заболеваний и т. д. Самая длинная общая подпоследовательность используется для поиска лекарства от рассеянного склероза.