Оглавление

Сортировка

Сортировка пузырьком

Сортировка пузырьком — один из самых известных алгоритмов сортировки. Здесь нужно последовательно сравнивать значения соседних элементов и менять числа местами, если предыдущее оказывается больше последующего. Таким образом элементы с большими значениями оказываются в конце списка, а с меньшими остаются в начале.

Этот алгоритм считается учебным и почти не применяется на практике из-за низкой эффективности: он медленно работает на тестах, в которых маленькие элементы (их называют «черепахами») стоят в конце массива. Однако на нём основаны многие другие методы, например, шейкерная сортировка и сортировка расчёской.

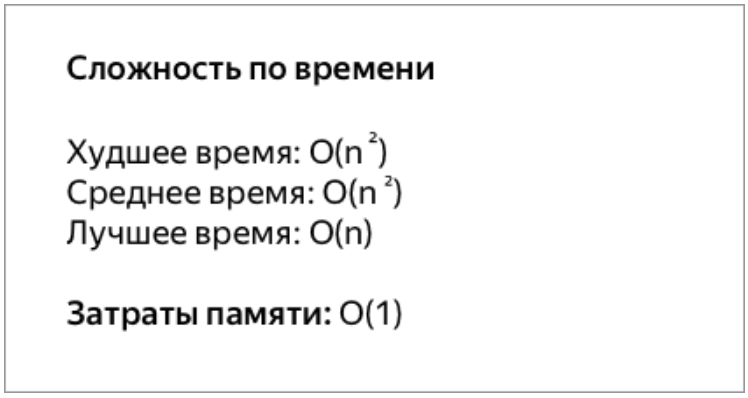


Рис. 1. Сложность во времени: сортировка пузырьком

Алгоритм

На каждом шаге мы находим наибольший элемент из двух соседних и ставим этот элемент в конец пары. Получается, что при каждом прогоне цикла большие элементы будут всплывать к концу массива, как пузырьки воздуха — отсюда и название.

Алгоритм выглядит так:

1. Берём самый первый элемент массива и сравниваем его со вторым. Если первый больше второго — меняем их местами с первым, если нет — ничего не делаем.
2. Затем берём второй элемент массива и сравниваем его со следующим — третьим. Если второй больше третьего — меняем их местами, если нет — ничего не делаем.
3. Проходим так до предпоследнего элемента, сравниваем его с последним и ставим наибольший из них в конец массива. Всё, мы нашли самое большое число в массиве и поставили его на своё место.

4. Возвращаемся в начало алгоритма и делаем всё снова точно так же, начиная с первого и второго элемента. Только теперь даём себе задание не проверять последний элемент — мы знаем, что теперь в конце массива самый большой элемент.
5. Когда закончим очередной проход — уменьшаем значение финальной позиции, до которой проверяем, и снова начинаем сначала.
6. Так делаем до тех пор, пока у нас не останется один элемент.

Рис. 2. Пример сортировки пузырьком

Эффективность работы

Этот алгоритм считается учебным и в чистом виде на практике почти не применяется. Дело в том, что среднее количество проверок и перестановок в массиве — это количество элементов в квадрате. Например, для массива из 10 элементов потребуется 100 проверок, а для массива из 100 элементов — уже в сто раз больше, 10 000 проверок.

Получается, что если у нас в массиве будет 10 тысяч элементов (а это не слишком большой массив с точки зрения реальных ИТ-задач), то для его сортировки понадобится 100

миллионов проверок — на это уйдёт какое-то время. А что, если сортировать нужно несколько раз в секунду?

В программировании эффективность работы алгоритма в зависимости от количества элементов n обозначают так: O (n). В нашем случае эффективность работы пузырьковой сортировки равна O (n^2). По сравнению с другими алгоритмами это очень большой показатель (чем он больше — тем больше времени нужно на сортировку).

Сортировка перемешиванием (шейкерная сортировка)

Шейкерная сортировка отличается от пузырьковой тем, что она двунаправленная: алгоритм перемещается не строго слева направо, а сначала слева направо, затем справа налево.

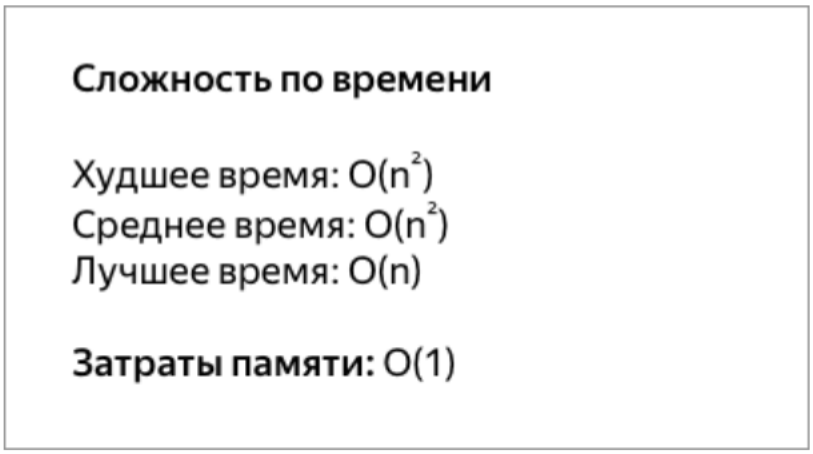


Рис. 3. Сложность по времени: шейкерная сортировка

Алгоритм

Шейкерная сортировка, также известная как коктейльная сортировка, представляет собой разновидность алгоритма пузырьковой сортировки. Он сортирует массив, сравнивая и меняя соседние элементы, но также перемещается слева направо, а затем справа налево на каждом проходе, позволяя более крупным элементам «всплывать» вправо, а меньшим элементам «опускаться» влево. Вот пошаговые инструкции для алгоритма сортировки шейкером:

1. **Инициализация**:
    - Начните с начала списка, который нужно отсортировать.
    - Установите два указателя **left** и **right** сначала на первый и последний элементы списка.

2. **Переход слева направо**:
    - Сравнивайте и меняйте местами соседние элементы, двигаясь слева направо.
    - Продолжайте этот процесс, пока не дойдете до **right** указателя.
3. **Перемещение указателя вправо**:
    - Переместите **right** указатель на одну позицию влево (уменьшите **right**).
4. **Переход справа налево**:
    - Сравнивайте и меняйте местами соседние элементы, двигаясь справа налево.
    - Продолжайте этот процесс, пока не дойдете до **left** указателя.
5. **Перемещение указателя влево**:
    - Переместите **left** указатель на одну позицию вправо (приращение **left**).
6. **Повторите**:
    - Повторяйте шаги со 2 по 5, пока в проходе больше не будут нужны замены.
7. **Прекращение**:
    - Алгоритм завершает работу, когда проход по списку не приводит к каким-либо заменам, что указывает на то, что список отсортирован.

Вот пример, иллюстрирующий этапы алгоритма шейкерной сортировки:

Несортированный список: [64, 34, 25, 12, 22, 11, 90]

1. Инициализация:
    - Начните с **left** первого элемента и **right** с последнего элемента.
    - Список: [64, 34, 25, 12, 22, 11, 90]
2. Переход слева направо:
    - Сравнивайте и меняйте местами соседние элементы, двигаясь слева направо.
        - Поменяйте местами 64 и 34: [34, 64, 25, 12, 22, 11, 90]
        - Поменяйте местами 64 и 25: [34, 25, 64, 12, 22, 11, 90]
        - Поменяйте местами 64 и 12: [34, 25, 12, 64, 22, 11, 90]
        - Поменяйте местами 64 и 22: [34, 25, 12, 22, 64, 11, 90]
        - Поменяйте местами 64 и 11: [34, 25, 12, 22, 11, 64, 90]
    - **right** указатель перемещается влево (уменьшается): **right** теперь он находится на индексе 5.
3. Перемещение указателя вправо:
    - **right** перемещается на одну позицию влево: **right** теперь находится под индексом 4.
4. Переход справа налево:
    - Сравнивайте и меняйте местами соседние элементы, двигаясь справа налево.
        - Поменяйте местами 11 и 64: [34, 25, 12, 22, 11, 64, 90]

- **left** указатель перемещается вправо (приращение): **left** теперь он находится на индексе 4.

5. Перемещение левого указателя:
    - **left** перемещается на одну позицию вправо: **left** теперь находится под индексом 5.
6. Повторить:
    - Повторите шаги со 2 по 5.
7. Прекращение:
    - После нескольких проходов никаких свопов в проходе не требуется. Алгоритм завершает работу.

Отсортированный список: [11, 12, 22, 25, 34, 64, 90]

Сортировка шейкером эффективно сортирует список, а также устраняет некоторые проблемы обычной пузырьковой сортировки за счет перемещения в обоих направлениях. Однако временная сложность в наихудшем случае по-прежнему равна O(n^2), что делает его менее эффективным, чем некоторые другие алгоритмы сортировки для больших наборов данных.

Сортировка расческой

Главный недостаток пузырьковой сортировки — её скорость и полный перебор всех элементов массива. Скорость работы алгоритма зависит не от количества сравнений (они выполняются быстро), а от количества перестановок (на них как раз и тратится много процессорного времени).

Получается, что если у нас в массиве в начале будет много больших элементов, которые нужно отправить в конец массива, то каждый раз нам придётся менять их местами с соседями, по одной перестановке за раз. Можно представить, что мы несем куда-то не слишком тяжёлый ящик, но после каждого шага ставим этот ящик на пол, потом поднимаем, делаем шаг, снова ставим, снова поднимаем. Процедуры простые, но из-за устройства алгоритма мы делаем эти процедуры слишком много раз.

Раз у нас большие элементы могут тормозить весь процесс, то можно их перекидывать не на соседнее место, а подальше. Так мы уменьшим количество перестановок, а с ними сэкономим и процессорное время, нужное на их обработку.

Но отправлять каждый большой элемент сразу в конец массива будет недальновидно — мы же не знаем, насколько этот элемент большой по сравнению с остальными элементами. Поэтому в сортировке расчёской сравниваются элементы, которые отстоят друг от друга на каком-то расстоянии. Оно не слишком большое, чтобы сильно не откидывать элементы и возвращать потом большинство назад, но и не слишком маленькое, чтобы можно было отправлять не на соседнее место, а чуть дальше.

Опытным путём программисты установили оптимальное расстояние между элементами — это длина массива, поделённая на 1,247 (понятное дело, расстояние нужно округлить до целого числа). С этим числом алгоритм работает быстрее всего.

Алгоритм

Гребенчатая сортировка — это относительно простой алгоритм сортировки, который совершенствует пузырьковую сортировку за счет исключения небольших значений, находящихся далеко от их правильного положения. Он работает путем сравнения и замены соседних элементов, аналогично сортировке пузырьком, но с различным размером промежутка. Вот пошаговые инструкции для алгоритма гребенчатой сортировки:

1. **Инициализация**:
    - Начните со всего списка, который нужно отсортировать.
    - Установите размер пробела, который изначально равен длине списка.

- Инициализируйте переменную, чтобы отслеживать, были ли сделаны какие-либо замены во время прохода. Изначально установлено значение **true**, обеспечивающее выполнение первого прохода.

2. **Пройдите по списку**:
   - Сравните и поменяйте местами соседние элементы, используя текущий размер зазора.
   - Продолжайте этот процесс для всех элементов списка, пока не будет достигнут конец.
3. **Сократить разрыв**:
   - Уменьшите размер зазора. Наиболее распространенный коэффициент усадки — 1,3, но можно использовать и другие коэффициенты.
   - Рассчитайте новый размер зазора, разделив текущий размер зазора на коэффициент сжатия и округлив его до ближайшего целого числа. Если размер зазора становится меньше 1, установите его равным 1.
4. **Повторите**:
   - Повторяйте шаги 2 и 3 до тех пор, пока размер промежутка не станет равным 1 и во время прохода не будет произведено никаких свопов (что указывает на то, что список отсортирован).

Вот пример, иллюстрирующий этапы алгоритма гребенчатой сортировки:

Несортированный список: [64, 34, 25, 12, 22, 11, 90]

1. Инициализация:
   - Начните со всего списка.
   - Размер пробела изначально равен длине списка (7).
   - Переменная **swapped** установлена в **true**.
2. Пройти по списку (пробел = 7):
   - Сравните и поменяйте местами соседние элементы с интервалом 7.
   - Во время этого прохода никаких свопов не производится, поэтому список остается прежним.
3. Сократите разрыв (разрыв = 7/1,3 ≈ 5):
   - Размер разрыва уменьшен до 5.
4. Пройти по списку (пробел = 5):
   - Сравните и поменяйте местами соседние элементы с интервалом 5.
   - Поменяйте местами 64 и 11: [11, 34, 25, 12, 22, 64, 90]
5. Сократите разрыв (разрыв = 5/1,3 ≈ 3):
   - Размер зазора уменьшен до 3.

6. Пройти по списку (пробел = 3):
    - Сравните и поменяйте местами соседние элементы с интервалом 3.
    - Поменяйте местами 11 и 25: [11, 34, 12, 25, 22, 64, 90]
    - Поменяйте местами 25 и 12: [11, 34, 12, 22, 25, 64, 90]
7. Сократите разрыв (разрыв = 3/1,3 ≈ 2):
    - Размер зазора уменьшен до 2.
8. Пройти по списку (пробел = 2):
    - Сравните и поменяйте местами соседние элементы с интервалом 2.
    - Поменяйте местами 11 и 12: [11, 12, 34, 22, 25, 64, 90]
    - Поменяйте местами 34 и 22: [11, 12, 22, 34, 25, 64, 90]
9. Сократите разрыв (разрыв = 2/1,3 ≈ 1):
    - Размер разрыва уменьшен до 1.
10. Пройти по списку (пробел = 1):
    - Сравните и поменяйте местами соседние элементы с интервалом 1.
    - Поменяйте местами 12 и 22: [11, 12, 22, 25, 34, 64, 90]
11. Сократить разрыв (пробел = 1):
    - Размер разрыва уже равен 1, поэтому дальнейшее уменьшение невозможно.
12. На последнем проходе не было произведено никаких свопов, и список отсортирован.

    Отсортированный список: [11, 12, 22, 25, 34, 64, 90]

Гребенчатая сортировка — это улучшение по сравнению с пузырьковой сортировкой с временной сложностью O(n^2), но она не так эффективна, как некоторые другие алгоритмы сортировки для больших наборов данных. Это может быть полезным вариантом для списков среднего размера.

Лучше пузырьковой сортировки?

То, что код выглядит сложнее, ничего не значит: нам нужна не оценка сложности кода, а скорость и эффективность работы.

Расчёска лучше пузырьковой сортировки, потому что в ней намного меньше операций перестановки. Именно перестановка занимает основное время процессора. В самом худшем случае алгоритм сортировки расчёской будет работать так же, как и пузырьковая, а в среднем — алгоритм работает быстрее пузырьковой.

Сортировка выбором

Алгоритм сортировки выбором заключается в поиске на необработанном срезе массива или списка минимального значения и в дальнейшем обмене этого значения с первым элементом необработанного среза (поиск минимума и перестановка). На следующем шаге необработанный срез уменьшается на один элемент.

1. Найти наименьшее значение в списке.
2. Записать его в начало списка, а первый элемент - на место, где раньше стоял наименьший.
3. Снова найти наименьший элемент в списке. При этом в поиске не участвует первый элемент.
4. Второй минимум поместить на второе место списка. Второй элемент при этом перемещается на освободившееся место.
5. Продолжать выполнять поиск и обмен, пока не будет достигнут конец списка.

Отличие сортировок выбором от сортировок вставками

Может показаться, что сортировки выбором и сортировки вставками — это суть одно и то же, общий класс алгоритмов. Ну, или сортировки вставками — разновидность сортировок выбором. Или сортировки выбором — частный случай сортировок вставками. И там и там мы по очереди из неотсортированной части массива извлекаем элементы и перенаправляем их в отсортированную область.

Главное отличие: в сортировке вставками мы извлекаем из неотсортированной части массива любой элемент и вставляем его на своё место в отсортированной части. В сортировке выбором мы целенаправленно ищем максимальный элемент (или минимальный), которым дополняем отсортированную часть массива. Во вставках мы ищем куда вставить очередной элемент, а в выборе — мы заранее уже знаем в какое место поставим, но при этом требуется найти элемент, этому месту соответствующий.

Это делает оба класса алгоритмов совершенно отличными друг от друга по своей сути и применяемым методам.

Алгоритм

Сортировка выбором — это простой алгоритм сортировки, который работает путем многократного выбора минимального (или максимального) элемента из неотсортированной

части списка и помещения его в начало (или конец) отсортированной части. Вот пошаговые инструкции для алгоритма сортировки выбором с упором на сортировку по возрастанию:

1. Начните со всего списка, который нужно отсортировать.
2. Разделите список на две части: отсортированную и неотсортированную. Изначально отсортированная часть пуста, а несортированная содержит все элементы.
3. Найдите минимальный элемент в неотсортированной части списка.
4. Поменяйте местами минимальный элемент с первым элементом в неотсортированной части. Это эффективно перемещает минимальный элемент в конец отсортированной части и расширяет отсортированную часть.
5. Теперь отсортированная часть содержит на один элемент больше, а несортированная часть содержит на один элемент меньше.
6. Повторите шаги 3–5 для оставшейся неотсортированной части (за исключением элементов, уже помещенных в отсортированную часть).
7. Продолжайте этот процесс, пока весь список не будет отсортирован. Отсортированная часть будет постепенно разрастаться, а несортированная будет сжиматься, пока не станет пустой.

Вот пример, иллюстрирующий этапы алгоритма сортировки выбором:

Несортированный список: [64, 25, 12, 22, 11]

1. Проход 1:
    - Минимальный элемент в неотсортированной части — 11.
    - Поменяйте местами 11 с первым элементом (64).
    - Отсортированная часть: [11], Неотсортированная часть: [25, 12, 22, 64]
2. Проход 2:
    - Минимальный элемент в несортированной части — 12.
    - Поменяйте местами 12 со вторым элементом (25).
    - Отсортированная часть: [11, 12], Неотсортированная часть: [25, 22, 64]
3. Проход 3:
    - Минимальный элемент в неотсортированной части — 22.
    - Поменяйте местами 22 с третьим элементом (25).
    - Отсортированная часть: [11, 12, 22], Несортированная часть: [25, 64]
4. Проход 4:
    - Минимальный элемент в неотсортированной части — 25.
    - Поменяйте местами 25 с четвертым элементом (64).
    - Отсортированная часть: [11, 12, 22, 25], Неотсортированная часть: [64]
5. Проход 5:

- Минимальный элемент в несортированной части — 64.
- Поменяйте местами 64 с пятым элементом (64). Поскольку это один и тот же элемент, фактическая замена не выполняется.
- Отсортированная часть: [11, 12, 22, 25, 64], Несортированная часть: []

Теперь весь список отсортирован, и алгоритм сортировки выбором завершен.

Отсортированный список: [11, 12, 22, 25, 64]

Сортировка выбором имеет временную сложность O(n^2) в худшем и среднем случаях, что делает ее неэффективной для больших списков. Однако его легко реализовать и понять.

Быстрая сортировка

Когда в 1960 году Тони Хоар придумывал этот алгоритм, ему нужно было отсортировать данные на магнитной ленте за один проход, чтобы не перематывать плёнку много раз. Для этого он взял за основу классическую пузырьковую сортировку и преобразовал её так:

1. На очередном шаге выбирается опорный элемент — им может быть любой элемент массива.
2. Все остальные элементы массива сравниваются с опорным и те, которые меньше него, ставятся слева от него, а которые больше или равны — справа.
3. Для двух получившихся блоков массива (меньше опорного, и больше либо равны опорному) производится точно такая же операция — выделяется опорный элемент и всё идёт точно так же, пока в блоке не останется один элемент.

Особенности алгоритма

Так как на третьем шаге мы разбиваем массив на два и для каждой части делаем то же самое, и так снова и снова, то это значит, что в нём используется рекурсия. Рекурсия — это когда функция вызывает саму себя, и при этом ей нужно держать в памяти все предыдущие этапы. Это значит, что при использовании сразу двух рекурсий (для левой и правой частей массива), может потребоваться очень много памяти. Чтобы обойти это ограничение, используют улучшенные версии быстрой сортировки — про них поговорим в другой раз.

Но несмотря на такой возможный расход памяти, у быстрой сортировки есть много плюсов:

- это один из самых быстрых алгоритмов, когда мы заранее ничего не знаем про массивы, с которыми придётся работать;
- алгоритм настолько прост, что его легко написать на любом языке программирования;
- быструю сортировку легко распараллелить и разбить на отдельные процессы;
- алгоритм работает на данных с последовательным доступом, когда мы не можем в любой момент вернуться в начало, а должны работать с данными только в одном порядке.

Выбор опорного элемента

Правильный выбор опорного элемента может сильно повысить эффективность быстрой сортировки. В зависимости от реализации алгоритма есть разные способы выбора:

- Первый элемент — в первых версиях быстрой сортировки Хоар выбирал опорным первый элемент массива. Именно так он смог обрабатывать всю магнитную ленту за один проход.
- Средний элемент — тот, который физически стоит посередине массива.
- Медианный элемент — элемент, значение которого находится посередине между всеми значениями в массиве

Есть ещё много других техник выбора — они применяются, когда программист точно знает, с какими массивами придётся работать: немного упорядоченными или когда всё вразнобой.

Алгоритм

Быстрая сортировка — это широко используемый эффективный алгоритм сортировки, который использует стратегию «разделяй и властвуй» для сортировки массива или списка. Он работает путем выбора «поворотного» элемента из массива и разделения других элементов на два подмассива в зависимости от того, меньше они или больше опорного элемента. Затем подмассивы рекурсивно сортируются. Вот пошаговые инструкции для алгоритма быстрой сортировки:

1. **Выберите опорный элемент**: выберите опорный элемент из массива. Выбор пивота может быть произвольным, но он может существенно повлиять на эффективность алгоритма.
2. **Разделение**: переставьте элементы в массиве так, чтобы все элементы меньше опорной точки находились слева от опорной точки, а все элементы, превышающие

опорную точку, были справа. Сама точка поворота находится в окончательном отсортированном положении. Этот процесс известен как секционирование.

- Инициализируйте два указателя **i** и **j** на левом и правом концах подмассива.
- Увеличивайте **i** до тех пор, пока не будет найден элемент, больший или равный опорному элементу, или до тех пор, пока **i** он не станет больше или равен **j**.
- Уменьшайте **j** до тех пор, пока не будет найден элемент, меньший или равный опорному элементу, или до тех пор, пока **j** он не будет меньше или равен **i**.
- Если **i** меньше или равно **j**, поменяйте местами элементы в позициях **i** и **j**, а затем продолжайте увеличивать **i** и уменьшать **j**.
- Повторяйте эти шаги до тех пор, пока **i** значение не станет больше **j**. На этом этапе точка поворота находится в окончательной отсортированной позиции, и все элементы слева от нее становятся меньше, а все элементы справа — больше.

3. **Рекурсия**: рекурсивно применить алгоритм быстрой сортировки к двум подмассивам, созданным на предыдущем шаге: подмассиву слева от опорной точки (содержащему элементы меньшего размера) и подмассиву справа от опорной точки (содержащему элементы большего размера).).
4. **Базовый случай**: рекурсия останавливается, когда подмассивы содержат только один или ноль элементов, поскольку массивы с одним или нулевым элементом уже считаются отсортированными.
5. **Объединение**: по мере возврата рекурсивных вызовов подмассивы объединяются для создания окончательного отсортированного массива.

Вот пример, иллюстрирующий этапы алгоритма быстрой сортировки:

Несортированный массив: [7, 2, 1, 6, 8, 5, 3, 4]

1. Выберите точку поворота, скажем **pivot = 4**.
2. Разделение:
    - Начните с **i** начала (индекс 0) и **j** с конца (индекс 7).
    - **i** находит 7, **j** находит 3 и меняются местами. Массив становится: [3, 2, 1, 6, 8, 5, 7, 4]
    - **i** находит 2, **j** находит 5 и они меняются местами. Массив становится: [3, 2, 1, 6, 8, 5, 7, 4]
    - **i** находит 1, **j** находит 8 и они меняются местами. Массив становится: [3, 2, 1, 6, 1, 5, 7, 4]
    - **i** находит 6, **j** находит 1, и они меняются местами. Массив становится: [3, 2, 1, 1, 6, 5, 7, 4]

- **i** находит 5, **j** находит 4 и меняются местами. Массив становится: [3, 2, 1, 1, 4, 5, 7, 6]
- В этот момент **i** больше **j**, а ось 4 находится в окончательном отсортированном положении.

3. Рекурсия:
   - Примените быструю сортировку к подмассиву слева от точки поворота (содержащему [3, 2, 1, 1, 4]).
   - Примените быструю сортировку к подмассиву справа от точки поворота (содержащему [5, 7, 6]).
4. Базовый вариант:
   - Подмассив [3, 2, 1, 1, 4] сортируется: [1, 1, 2, 3, 4].
   - Подмассив [5, 7, 6] сортируется: [5, 6, 7].
5. Объедините:
   - Окончательный отсортированный массив формируется путем объединения результатов рекурсивных вызовов и опорной точки: [1, 1, 2, 3, 4, 5, 6, 7, 8].

Теперь массив отсортирован по возрастанию.

Быстрая сортировка имеет среднюю и оптимальную временную сложность O(n log n) и зачастую более эффективна, чем другие алгоритмы сортировки для больших наборов данных. Однако в худшем случае временная сложность может составлять O(n^2), если используется плохая стратегия выбора опорного элемента. Для улучшения производительности алгоритма можно использовать различные методы выбора опорной точки и оптимизации.

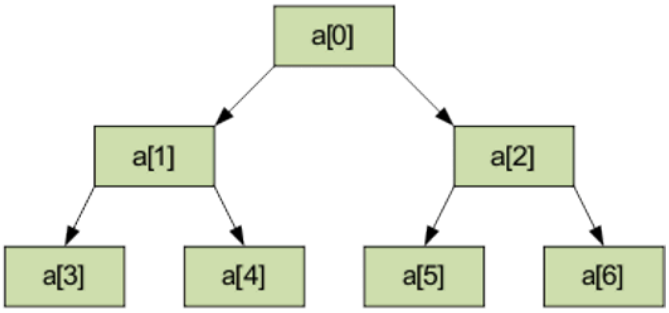
Сортировка кучей (Пирамидальная сортировка)

Метод пирамидальной сортировки, изобретенный Д. Уилльямсом, является улучшением традиционных сортировок с помощью дерева.

Пирамидой (*кучей*) называется двоичное дерево такое, что

$a[i] \le a[2i+1]$;

$a[i] \le a[2i+2]$.



a[0] — минимальный элемент пирамиды.

Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов.

Выполнение алгоритма разбивается на два этапа.

Алгоритм

Сортировка пирамидой, также известная как пирамидальная сортировка, представляет собой алгоритм сортировки на основе сравнения, который использует структуру данных двоичной кучи для сортировки элементов. Вот пошаговые инструкции для алгоритма сортировки кучей:

1. **Постройте максимальную кучу**:
    - Преобразуйте входной массив в двоичную кучу (максимальная куча). Это делается путем начала с последнего нелистового узла и «кучи» каждого узла (т. е. обеспечения того, чтобы он удовлетворял свойству максимальной кучи, где каждый родительский узел больше или равен своим дочерним узлам). Этот процесс переупорядочит элементы таким образом, чтобы самый большой элемент находился в корне кучи.
2. **Куча**:
    - Начиная с последнего нелистового узла (который можно рассчитать как **n // 2 - 1**, где **n**— количество элементов в массиве) и продвигаясь вверх, выполните следующие действия:
        - Сравните родительский узел с его дочерними узлами.
        - Если родительский элемент меньше любого из своих дочерних элементов, поменяйте родительский элемент на больший дочерний.
        - Повторяйте этот процесс до тех пор, пока свойство максимальной кучи не будет восстановлено для текущего узла.

3. **Извлечь элементы**:
    - Корень максимальной кучи теперь содержит самый большой элемент. Поменяйте местами корень с последним элементом в куче (который является наименьшим элементом в несортированной части массива).
    - Уменьшите размер кучи на единицу (исключая отсортированные элементы в конце).
4. **Снова куча**:
    - Создайте кучу в корне максимальной кучи, чтобы гарантировать, что следующий по величине элемент находится в корне.
    - Повторите этот процесс, извлекая элементы и складывая корень в кучу, пока куча не станет пустой.
5. **Объединить отсортированные элементы**:
    - Элементы, извлеченные из максимальной кучи, теперь отсортированы. Поместите их обратно в исходный массив в обратном порядке (от большего к меньшему), начиная с конца массива.

Вот пример, иллюстрирующий этапы алгоритма сортировки кучей:

Несортированный массив: [12, 11, 13, 5, 6, 7]

1. Постройте максимальную кучу:
    - Массив после построения максимальной кучи: [13, 11, 12, 5, 6, 7]
2. Куча:
    - Массив после кучи каждого узла: [13, 11, 12, 5, 6, 7]
3. Извлечь элементы:
    - Извлеките 13 (самый большой элемент) и замените его последним элементом: [7, 11, 12, 5, 6, 13]
4. Куча:
    - Массив после кучи корня: [12, 11, 7, 5, 6, 13]
5. Извлечь элементы:
    - Извлеките 12 и замените его последним неотсортированным элементом: [6, 11, 7, 5, 12, 13]
6. Куча:
    - Массив после кучи корня: [11, 6, 7, 5, 12, 13]
7. Извлечь элементы:
    - Извлеките 11 и замените его последним неотсортированным элементом: [5, 6, 7, 11, 12, 13]
8. Куча:

- Массив после кучи корня: [7, 6, 5, 11, 12, 13]

9. Извлечь элементы:
    - Извлеките 7 и замените его последним неотсортированным элементом: [5, 6, 7, 11, 12, 13]
10. Продолжайте процесс, пока куча не станет пустой.
11. Объедините отсортированные элементы обратно в исходный массив: [5, 6, 7, 11, 12, 13].

Теперь массив отсортирован по возрастанию.

Сортировка кучей имеет временную сложность O(n log n) и является эффективным алгоритмом сортировки на месте. Это особенно полезно для сортировки больших наборов данных.

Сортировка вставками

Сортировка вставками (Insertion sort) — один из простейших алгоритмов сортировки. Как всегда мы будем рассматривать сортировку по возрастанию.

Суть его заключается в том, что в цикле один за другим выбираются элементы массива и сравниваются с элементами, стоящими перед ними, до тех пор пока не будет найдет элемент, меньший текущего, или мы не дойдем до начала массива. Перед ним мы и **вставляем** текущий, для этого предварительно сдвинув все элементы, которые оказались больше текущего, в сторону увеличения на один индекс.

Этот алгоритм, в отличии от другого простейшего алгоритма — сортировки пузырьком, имеет сложность $O(n^2)$ только для худшего случая (массива, отсортированного в обратном порядке), а для лучшего случая сложность будет $O(n)$ — достаточно одного прохода, чтобы понять что массив отсортирован. При этом и затраты памяти всего $O(n)$ на сам массив и $O(1)$ на дополнительную переменную с текущим элементом.

Сортировка вставками дает несколько преимуществ:

- Это эффективно для небольших наборов данных.
- Это стабильный алгоритм сортировки, означающий, что он поддерживает относительный порядок равных элементов в отсортированном выводе.
- Это алгоритм сортировки на месте, то есть он не требует дополнительного места в памяти.

Алгоритм

1. Начните со второго элемента массива (при условии, что первый элемент уже отсортирован).
2. Сравните текущий элемент с его соседним предшественником.
3. Если элемент меньше своего предшественника, поменяйте его местами с предшественником.
4. Повторяйте шаги 2–3, пока элемент не займет правильное положение в отсортированной части массива.
5. Перейдите к следующему элементу и повторяйте шаги 2–4, пока весь массив не будет отсортирован.

Пояснение примера:

Давайте рассмотрим пример шаг за шагом:

- Исходный массив:**[12, 11, 13, 5, 6]**
- Начните со второго элемента (11).
- Сравните его с первым элементом (12) и при необходимости поменяйте местами.
    - Новый массив:**[11, 12, 13, 5, 6]**
- Перейдите к следующему элементу (13) и вставьте его в правильную позицию в отсортированной части массива.
    - Новый массив:**[11, 12, 13, 5, 6]**
- Продолжайте этот процесс, пока весь массив не будет отсортирован.

После процесса сортировки массив принимает вид: **[5, 6, 11, 12, 13]**.

Сортировка слиянием

Основной принцип сортировки слиянием такой: делим массив пополам, каждый из них сортируем слиянием и потом соединяем оба массива. Каждый разделённый массив тоже нарезаем на два подмассива до тех пор, пока в каждом не окажется по одному элементу.

Здесь тоже используется рекурсия — то есть повторение алгоритма внутри самого алгоритма. Но это только один из элементов алгоритма.

Второй элемент — соединение отсортированных элементов между собой, причём тоже хитрым способом: раз оба массива уже отсортированы, то нам достаточно сравнивать элементы друг с другом по очереди и заносить в итоговый массив данные по порядку.

Алгоритм

Сортировка слиянием — это популярный, эффективный и стабильный алгоритм сортировки, который использует подход «разделяй и властвуй» для сортировки массива или списка. Он работает путем разделения несортированного списка на более мелкие подсписки, сортировки каждого подсписка, а затем обратного слияния отсортированных подсписков вместе для создания единого отсортированного списка. Вот пошаговые инструкции для алгоритма сортировки слиянием:

1. **Разделить**:
    - Если сортируемый список содержит ноль или один элемент, он уже считается отсортированным. Верните список как есть.
    - В противном случае разделите список на два подсписка примерно одинакового размера. Вы можете выбрать средний элемент в качестве центра, чтобы разделить список на две части.
2. **Рекурсия**:
    - Рекурсивно примените алгоритм сортировки слиянием к каждому из двух подсписков, созданных на предыдущем шаге.
3. **Объединить**:
    - Объедините два отсортированных подсписка в один отсортированный список. Это делается путем многократного выбора наименьшего (или самого большого) элемента из двух подсписков и добавления его в новый объединенный список.

Вот пример, иллюстрирующий этапы алгоритма сортировки слиянием:

Несортированный список: [38, 27, 43, 3, 9, 82, 10]

1. Разделять:
    - Список разделен на два подсписка: [38, 27, 43] и [3, 9, 82, 10].
2. Рекурсия:
    - Что касается первого подсписка, [38, 27, 43], разделите его на [38] и [27, 43]. Поскольку [38] и [27] содержат только один элемент, они считаются отсортированными.
    - Для второго подсписка [3, 9, 82, 10] разделите его на [3, 9] и [82, 10].
        - Рекурсивно примените сортировку слиянием к обоим подспискам:
            - Для [3, 9] разделите его на [3] и [9], которые также считаются отсортированными.

- Для [82, 10] разделите его на [82] и [10], оба считаются отсортированными.
- Теперь объедините отсортированные подсписки: [3, 9] и [10, 82].

3. Объединить:
   - Объедините два отсортированных подсписка [3, 9] и [10, 82], чтобы получить [3, 9, 10, 82].
   - Объедините отсортированные подсписки [38] и [27, 43], чтобы получить [27, 38, 43].
   - Наконец, объедините два основных подсписка [27, 38, 43] и [3, 9, 10, 82], чтобы получить полностью отсортированный список: [3, 9, 10, 27, 38, 43, 82].

Теперь массив сортируется по возрастанию с использованием алгоритма сортировки слиянием. Сортировка слиянием имеет временную сложность O(n log n) и эффективна для больших наборов данных, что делает ее одним из предпочтительных алгоритмов сортировки для многих приложений.

Особенность алгоритма

Главное преимущество сортировки слиянием — она работает всегда с одной и той же скоростью на любых массивах. Допустим, у нас есть два массива:

1. В одном будут числа в случайном порядке.
2. В другом — уже отсортированные числа, но последний и предпоследний элементы (разные) просто поменяны местами.

В обоих случаях сортировка слиянием покажет одно и то же время: ей всё равно, какие данные обрабатывать, алгоритм всё равно сделает это за один проход и одинаковое время. Благодаря этому свойству алгоритм используют там, где нужно обработать данные за заранее известное время. Например, если для сортировки 1000 элементов серверу требуется 50 миллисекунд, то можно настроить ему отправку новой партии данных каждую 51 миллисекунду — так мы будем уверены, что к моменту отправки новых данных старые уже будут обработаны.

Сортировка подсчётом

В обычных сортировках мы разными способами сравниваем элементы массива между собой. Но есть пара сортировок, в которых нет сравнений, но массив всё равно выстраивается по порядку. Рассказываем, как работает такая магия на примере сортировки подсчётом.

Сортировка подсчётом лучше всего работает при таких условиях:

- массив очень большой — значений много;
- эти значения лежат в известном нам диапазоне (например, это диапазон работы какого-то датчика);
- диапазон намного меньше, чем размер массива, то есть единицы данных могут повторяться.

Принцип работы

Главная идея алгоритма — посчитать, сколько раз встречается каждый элемент в массиве, а потом заполнить исходный массив результатами этого подсчёта. Для этого нам нужен вспомогательный массив, где мы будем хранить результаты подсчёта. Даже если нам надо отсортировать миллион чисел, мы всё равно знаем диапазон этих чисел заранее, например, от 1 до 100. Это значит, что во вспомогательном массиве будет не миллион элементов, а сто.

В общем виде всё работает так:

1. Мы создаём вспомогательный массив и на старте заполняем его нулями.
2. Проходим по всему исходному массиву и смотрим очередное значение в ячейке.
3. Берём содержимое этой ячейки и увеличиваем на единицу значение вспомогательного массива под этим номером. Например, если мы встретили число 5, то увеличиваем на единицу пятый элемент вспомогательного массива. Если встретили 13 — тринадцатый.
4. После цикла во вспомогательном массиве у нас хранятся данные, сколько раз встречается каждый элемент.
5. Теперь мы проходим по вспомогательному массиву, и если в очередной ячейке лежит что-то больше нуля, то мы в исходный массив столько же раз отправляем номер этой ячейки. Например, в первой ячейке вспомогательного массива лежит число 7. Это значит, что в исходный массив мы отправляем единицу 7 раз подряд.

Зачем нужна эта сортировка

Допустим, у нас есть какой-то прибор, который выдаёт результаты в определённом диапазоне, например показывает сейсмоактивность этом регионе в диапазоне от 0 до 100. Он работает круглосуточно и каждые 2 секунды пишет в лог новое значение текущей активности землетрясений. За год получается 15 миллионов записей. Сортировка подсчётом

отлично подходит для такого массива, потому что мы знаем диапазон и этот диапазон гораздо меньше общего количества элементов массива.

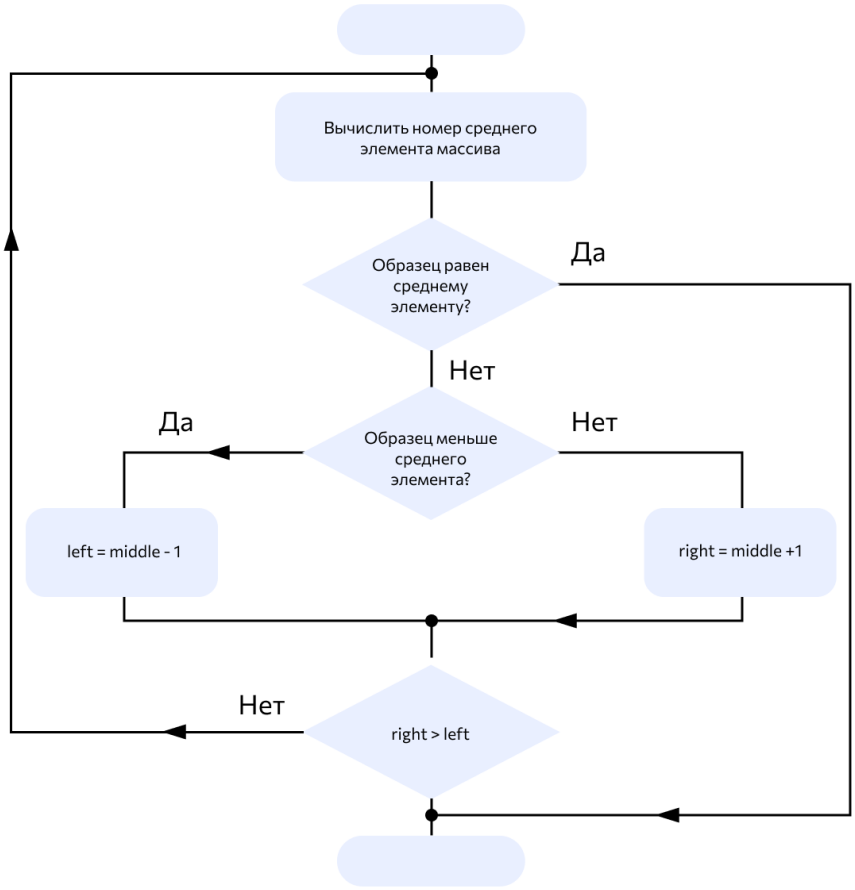
Поиск

Бинарный поиск

Бинарный поиск — тип поискового алгоритма, который последовательно делит пополам заранее отсортированный массив данных, чтобы обнаружить нужный элемент. Другие его названия — двоичный поиск, метод половинного деления, дихотомия.

Принцип работы алгоритма бинарного поиска

1. Основная последовательность действий алгоритма выглядит так:
2. Сортируем массив данных.
3. Делим его пополам и находим середину.
4. Сравниваем срединный элемент с заданным искомым элементом.
5. Если искомое число больше среднего — продолжаем поиск в правой части массива (если он отсортирован по возрастанию): делим ее пополам, повторяя пункт 3. Если же заданное число меньше — алгоритм продолжит поиск в левой части массива, снова возвращаясь к пункту 3.

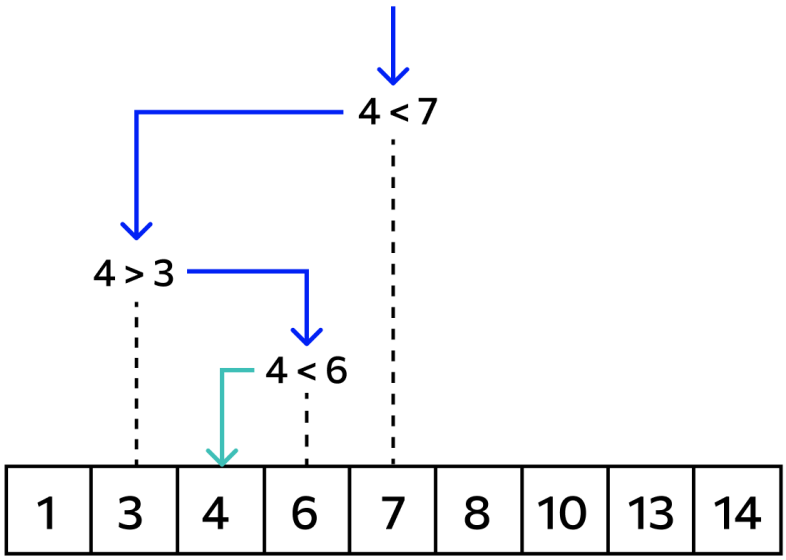


Реализация бинарного поиска

Существуют два способа реализации бинарного поиска.

1. Итерационный метод. При таком подходе используется цикл, тело которого повторяется, пока не найдется заданный элемент либо не будет установлено, что его нет в массиве. Например, в Python для этой цели удобно использовать цикл while.

2. Рекурсивный подход. В этом случае пишется функция, которая вызывает сама себя (рекурсивно), пока не будет найден искомый элемент в массиве.



В каких случаях используют бинарный поиск

Двоичный поиск подходит для нахождения позиций элемента в упорядоченном списке: в этом случае он эффективнее линейного, поскольку массив данных на каждом шаге разделяется надвое и одна половина сразу отбрасывается. Последовательная сложность двоичного метода в худшем и среднем случаях равна O(log n), в лучшем — O(1) (если обнаруживаем искомый элемент на первой итерации). Для сравнения: вычислительная сложность линейного поиска равна O(n) (обычный проход по всем элементам в поисках нужного).

У бинарного поиска есть недостаток — он требует упорядочивания данных по возрастанию. Сложность сортировки — не менее O(n log n). Поэтому, если список короткий, используется все-таки линейный поиск.

Разновидности алгоритма

На основе бинарного поиска разработано несколько дополнительных разновидностей поисковых алгоритмов:

- однородный бинарный поиск, при котором аргумент поиска A сравнивается с ключом Ki, где i — золотое сечение интервала (оно выбирается так, чтобы отношение длины большего отрезка к длине всего интервала равнялось отношению длины меньшего отрезка к длине большего отрезка);
- троичный поиск, когда интервал делится на три части вместо двух. Обычно применяется для поиска положения экстремума функции;
- интерполирующий поиск, который предсказывает позицию нужного элемента на основе разницы значений. Эффективен, если элементы распределены достаточно равномерно;
- дробный спуск — применяется для ускорения двоичного поиска в многомерных массивах данных, и другие.

Поиск в ширину (Breadth-First Search, BFS)

BFS, или Breadth First Search — алгоритм обхода графа в ширину. Граф — это структура из «вершин» и «ребер», соединяющих между собой вершины. По ребрам можно передвигаться от одной вершине к другой, и BFS делает это поуровнево: сначала проходит по всем ближайшим от начальной точки вершинам, потом спускается глубже.

Выглядит это так: алгоритм начинает в заранее выбранной вершине и сначала «посещает» и отмечает всех соседей этой вершины. Потом он переходит к соседям посещенных вершин, затем — дальше по тому же принципу. Из-за характера распространения, похожего на волну, алгоритм еще называют волновым. BFS — один из двух популярных алгоритмов обхода. Второй называется DFS и подразумевает обход в глубину: сначала алгоритм проходит по ребрам «вглубь» графа.

Кто пользуется BFS

- Дата-сайентисты, которые работают с информацией и ее распространением, часто взаимодействуют с теорией графов.
- Разработчики, имеющие дело с определенными видами задач: поиск оптимального маршрута, программирование передвижения «умных» машин, разработка интеллектуальных систем и другие.
- Математики и другие ученые, которые работают с теорией графов как с фундаментальным научным знанием или в контексте решения практических задач.
- Инженеры-электроники: конкретно алгоритм BFS используется при трассировке печатных плат.
- Технические специалисты, работающие в телекоммуникационных системах. Там тоже активно применяется теория графов и в частности BFS.
- Сетевые инженеры, так как теория графов активно используется в сетевых технологиях. BFS, например, применяют при обходе P2P-сетей, или пиринговых сетей, а на них основаны многие сетевые протоколы. В частности, пиринговую сеть реализует BitTorrent.

Для чего нужен BFS

- Для решения задач поиска оптимального пути. Классической задачей считается автоматизированный поиск выхода из лабиринта.

- Для решения задач, связанных непосредственно с теорией графов, например для поиска компонент связности. Эти задачи в свою очередь решаются в Data Science, теории сетей и электронике.
- Для задач искусственного интеллекта, связанных с поиском решения с минимальным количеством ходов. В таком случае состояния «умной машины» представляются как вершины, а переходы между ними — как ребра.
- Для оптимизации памяти при обходе графа в некоторых ситуациях, например для некоторых специфических структур.
- Для работы с информацией в определенных структурах данных, таких как деревья. Их тоже можно обходить с помощью алгоритма BFS, потому что они — подвид графов.

Особенности BFS

- Константное количество действий для каждого ребра или вершины. Это важно при расчете сложности алгоритма — при выборе оптимального метода решения той или иной задачи.
- Отсутствие проблемы «бесконечного цикла»: алгоритм не попадет в него ни при каких условиях благодаря особенностям работы.
- Высокая точность и надежная архитектура, которая позволяет полагаться на этот алгоритм в решении различных задач.
- Возможность работать и с ориентированными, и с неориентированными графами. О том, чем они различаются, можно прочитать в статье про ориентированный граф.
- Полнота алгоритма — он найдет решение, то есть кратчайший путь, и завершится на любом конечном графе. Если граф бесконечный, решение найдется только в том случае, если конечен какой-либо из его путей.
- Возможность находить кратчайший путь в графе, если все ребра одинаковой длины. Если длины ребер разные, BFS найдет путь с минимальным количеством ребер, но он не обязательно будет самым коротким. Для поиска кратчайшего пути в таком случае будет лучше алгоритм Дейкстры.

Как работает алгоритм BFS

Алгоритм простой и интуитивно понятный. Он проходит по вершинам графа, пока в том не останется непосещенных вершин, и рассчитывает самый короткий путь до целевой вершины. Чтобы показать его работу нагляднее, представим алгоритм пошагово.

Начало работы. В качестве начальной можно выбрать любую вершину. На момент начала работы алгоритма все вершины помечены как непосещенные — их называют «белыми». Первое, что делает алгоритм, — помечает начальную вершину как посещенную (также используют термины «развернутая» или «серая»). Если она и есть целевая, на этом алгоритм завершается. Но чаще всего это не так.

Поиск соседей. Алгоритм проверяет, какие соседи есть у начальной вершины. Они добавляются в «очередь действий» в том порядке, в каком алгоритм их нашел, и тоже помечаются как «серые». Это продолжается, пока у начальной вершины не останется «белых» соседей.

Переход на следующую вершину. Когда алгоритм проходит по всем соседям начальной вершины, он помечает ее как полностью обойденную. Такие вершины еще называют «черными»: алгоритм к ним не возвращается. Затем он переходит к одной из «серых» вершин — соседей начальной. Алгоритм выбирает первую вершину в очереди. Далее действия повторяются: «соседи» вершины, кроме «черной», заносятся в очередь.

Когда и эта вершина будет пройдена, переход повторится по тому же принципу — первая вершина в очереди. В этом случае ею будет второй сосед начальной вершины — мы помним, что их добавляли в очередь первыми. И только когда соседи начальной вершины в очереди закончатся, алгоритм пойдет по следующему «уровню» вершин. Так и достигается обход в ширину.

Конец алгоритма. Если очередь оказалась пустой, это значит, что «белых» и «серых» вершин больше не осталось. Алгоритм завершится. Если при этом целевая вершина не будет достигнута, это значит, что доступа к ней из начальной точки нет.

Если целевая вершина достигается раньше, чем алгоритм пройдет по всему графу, это также может означать его завершение. Алгоритм остановится, потому что задача окажется выполнена: самый короткий путь к целевой вершине будет найден.

Поиск в глубину (Depth-First Search, DFS)

DFS, или Depth First Search, — поиск в глубину, позволяющий найти маршрут от точки A до точки B. Используется в графах — особых структурах, состоящих из точек-вершин и ребер-путей. DFS ищет маршрут по графу “в глубину”: на каждом шаге “уходит” все дальше.

Классический пример использования алгоритма — поиск случайного пути в лабиринте. DFS начинает работу в заданной точке, на каждом шаге проходит по лабиринту до следующего поворота и выбирает направление. Если путь оказывается тупиковым, алгоритм возвращается к предыдущему повороту и пробует новое направление. В результате рано или поздно находится нужный путь.

Кто пользуется алгоритмом

- Математики, которые работают с теорией графов для решения фундаментальных или практических задач.
- Специалисты по анализу данных и по искусственному интеллекту, так как графы часто используются в Data Science или в машинном обучении.
- Разработчики, которым бывает необходимо решать задачи поиска маршрутов, расчета потоков и другие подобные. Такие задачи могут встретиться во многих проектах: от картографического сервиса до онлайн-игры.
- Сетевые инженеры, так как в виде графов представляют компьютерные сети, а многие сетевые протоколы основаны на алгоритмах работы с графами.
- Иногда — другие специалисты, которым бывает нужно столкнуться с теорией графов. Вариации DFS используются в том числе в жизни.

Для чего нужен алгоритм DFS

- Для поиска любого маршрута в лабиринте. В отличие от алгоритма BFS, поиск в глубину ищет не самый короткий, а случайный путь. Правило прохождения лабиринта в реальной жизни “Идти с левой рукой на стене и всегда поворачивать влево” — пример DFS вне программирования.
- Для решения задач, связанных с построением маршрута: в сети, на карте, в сервисах покупки билетов и так далее. При этом непосредственно для поиска DFS используется не так часто — он чаще нужен для исследования топологии графа.

- Как составная часть расчетов в более сложных алгоритмах, например для определения максимального транспортного потока.
- Для решения ряда задач из теории графов, которые используются в программировании и математике: поиска циклов, сортировки и так далее. Мы подробно поговорим об этом ниже.

Модификации DFS

Алгоритм можно модифицировать, чтобы решить другие задачи из теории графов:

- проверить граф на двудольность. Двудольным называется граф, вершины которого можно разбить на две группы так, чтобы концы каждого ребра принадлежали вершинам из разных групп;
- найти в ориентированном графе цикл. Ориентированный граф — такой, у ребер которого есть направление. Об этом можно подробнее прочесть в нашей статье. Цикл — это замкнутый контур из вершин и ребер внутри графа: проходя по нему, в результате придешь в начальную точку;
- отсортировать и упорядочить ориентированный граф, дать вершинам номера так, чтобы ребра шли от меньшего номера к большему;
- преобразовать “синтаксическое дерево”, подвид графа, в строку;
- найти в графе определенные точки, например, так называемые мосты или шарниры.

Все эти задачи также нужны в разработке, в математике и ряде других направлений.

При чем тут рекурсия

Классическая реализация DFS — рекурсивная. Рекурсией называется процесс, когда какая-то функция вызывает себя же, но с другими аргументами. В результате одна и та же функция одновременно остается запущенной несколько раз, пока в одной из итераций не дойдет до финального решения. Тогда она вернет результат и закроется, а по каскаду закроются и все функции, приведшие к ее вызову.

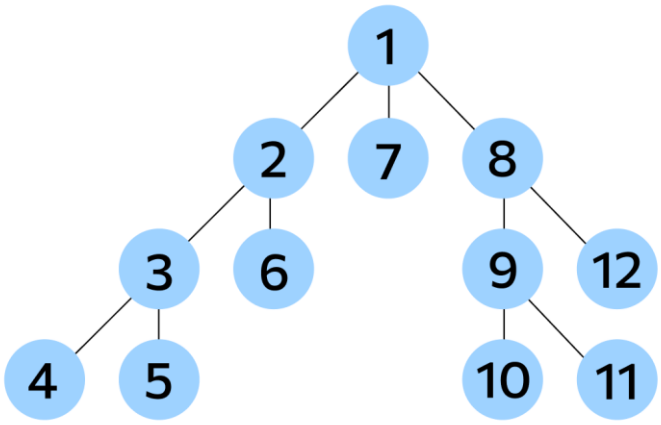
С помощью рекурсии решают задачи, связанные с “глубоким” прохождением по данным. Очевидный пример — расчет чисел Фибоначчи, когда на каждом следующем шаге функция складывает два предыдущих числа. Задача DFS тоже классически решается с помощью рекурсивного алгоритма.

Но не всегда рекурсия оптимальна. Поддержка сразу нескольких запущенных одинаковых функций может отнимать много аппаратных ресурсов, поэтому существуют

и нерекурсивные реализации. Их тоже нужно знать, чтобы избежать переполнения стека и других ситуаций, когда рекурсивная реализация может “упасть”.

Принцип обхода графа в глубину

Мы рассмотрим стандартный рекурсивный алгоритм обхода. О нерекурсивных реализациях поговорим ниже: их есть несколько, и у каждой свои особенности. Стандартный же принцип довольно простой для понимания, и его легко реализовать.



Принцип обхода графа в глубину

Первый шаг. Когда алгоритм начинает работу, все вершины считаются “белыми”, непосещенными. DFS начинает путь в заранее заданной вершине *v* и должен найти от нее путь до другой заданной вершины или же полностью составить карту графа.

Первое, что делает DFS, — красит вершину, в которой находится, в серый цвет. Это показывает, что алгоритм в ней уже был. Затем DFS проверяет соседей — вершины, которые соединены с той, где он находится.

Переход. Если какая-то из смежных вершин белая, алгоритм переходит на нее и повторяет те же действия: красит в серый, ищет соседей. Это происходит не циклически, а рекурсивно: если представить DFS как функцию, то получится, что эта функция в ходе выполнения вызывает сама себя, но для другой вершины. Поэтому сначала алгоритм работает с одним выбранным соседом и, только если упирается в тупик, возвращается обратно и пробует пройти по другому пути.

Выбор соседа происходит случайно или по заранее заданным критериям — например, это может быть самая левая или самая правая вершина. Выше мы упоминали “правило левой руки”: оно по сути является таким критерием.

Если неисследованных соседей у вершины не осталось, она красится в черный цвет как полностью посещенная.

Завершение обхода. Алгоритм завершается, если достигает нужной точки. В таком случае все вызванные “экземпляры” функции поочередно завершаются: от последнего до первого вызванного. Если задача — полностью перебрать граф, то критерий для завершения другой: все вершины должны стать черными.

Нерекурсивные реализации

Так как рекурсия отнимает много программных ресурсов, есть несколько реализаций без ее использования. Они сложнее в написании, но могут сэкономить время работы программы.

Самый простой вариант — по-особому хранить в памяти посещенные вершины. Для этого используется такая структура данных, как стек, — о ней можно прочитать в нашей статье. В стек поочередно помещаются непосещенные вершины-соседки, и тот используется как “карта” для будущих посещений. Этот способ тоже нагружает программные ресурсы, но не так сильно.

Второй вариант — хранить в стеке не сами вершины, а их номера и номера смежных с ними вершин. Это сложнее и, по сути, является имитацией так называемого стека вызовов — участка памяти, где хранятся вызванные функции. Именно его может перегрузить рекурсия.

Третий способ — реализовать сам граф на указателях: помещать в каждой вершине указатели на предыдущие и номера смежных вершин.

Это нормально, если нерекурсивные реализации кажутся вам сложными. Вы сможете разобраться с ними подробнее, когда лучше изучите программирование.

Как реализовать алгоритм DFS

Самая простая рекурсивная реализация — создать граф в виде связанного списка или другой структуры данных, а потом написать функцию для его прохождения. Как реализовать сам граф — зависит от языка программирования: обычно используются типы, позволяющие хранить множество значений. Каждый элемент такого “комплексного” типа является вершиной, а внутри вершины хранятся ссылки на другие элементы или их номера — так реализованы пути.

Сама же функция, условно названная DFS (v), довольно простая и действует по следующей логике.

1. На вход поступает белая вершина *v*.
2. Вершина *v* окрашивается в серый.
3. Ищется вершина *w,* смежная с *v* и белая.
4. Изнутри DFS (*v*) рекурсивно вызывается DFS (*w*).
5. Когда функция завершается, вершина *v* окрашивается в черный.

“Окрашивание” можно реализовать с помощью какой-либо переменной внутри вершины: например, значение 0 — белая, 1 — серая, и так далее.

Задание на лабораторную работу

1. Реализовать следующие алгоритмы сортировки (алгоритмы реализовывать в указанном порядке) и осуществить сортировку массива:
    - Сортировка пузырьком.
    - Шейкерная сортировка.
    - Сортировка расчёской.
    - Сортировка вставками.
    - Сортировка слиянием.
    - Сортировка выбором.
    - Сортировка подсчетом.
    - Быстрая сортировка.
    - Пирамидальная сортировка.
2. Реализовать алгоритмы поиска данных (алгоритмы реализовывать в указанном порядке) и осуществить поиск данных в массиве:
    - Бинарный поиск.
    - Поиск в ширину.
    - Поиск в глубину.

Источники

1. Сортировка пузырьком.
2. Шейкерная сортировка.
3. Сортирочка расческой.
4. Быстрая сортировка.
5. Сортировка вставками.
6. Сортировка слиянием.
7. Сортировка выбором.
8. Отличие сортировок выбором от сортировок вставками.
9. Сортировка подсчётом.
10. Пирамидальная сортировка.
11. Бинарный поиск.
12. Поиск в ширину.
13. Поиск в грубину.