11.8 Сортировка списков

Шаг 1

Тема урока: сортировка списков

1. Задача сортировки
2. Сортировка пузырьком
3. Сортировка выбором
4. Сортировка простыми вставками

Аннотация. Задачи и способы (алгоритмы) сортировки списков.

Задача сортировки

Задача сортировки списка заключается в перестановке его элементов так, чтобы они были упорядочены по возрастанию или убыванию. Это одна из основных задач программирования. Мы сталкиваемся с ней очень часто: при записи фамилий учеников в классном журнале, при подведении итогов соревнований и т.д.

Алгоритмы сортировки

Алгоритм сортировки — это алгоритм упорядочивания элементов в списке. Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

- время — основной параметр, характеризующий быстродействие алгоритма;
- память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных.

Алгоритмы сортировки, не потребляющие дополнительной памяти, относят к **сортировкам на месте**.

Основные алгоритмы сортировки

Медленные:

1. Пузырьковая сортировка (Bubble sort);
2. Сортировка выбором (Selection sort);
3. Сортировка простыми вставками (Insertion sort).

Быстрые:

1. Сортировка Шелла (Shell sort);
2. Быстрая сортировка (Quick sort);
3. Сортировка слиянием (Merge sort);
4. Пирамидальная сортировка (Heap sort);
5. Сортировка TimSort (используется в Java и Python).

Большинство алгоритмов сортировки, в частности, указанные выше, основаны на сравнении двух элементов списка. Существуют однако алгоритмы не основанные на сравнениях. Такие алгоритмы, как правило, используют наперед заданные условия относительно элементов списка. Например, элементами списка являются натуральные или целые числа в некотором диапазоне, элементами являются строки и т.д.

К алгоритмам, не основанным на сравнениях, можно отнести следующие:

1. Сортировка подсчетом (Counting sort)

2. Блочная сортировка (Bucket sort)
3. Поразрядная сортировка (Radix sort)

В рамках курса мы рассмотрим несложные алгоритмы пузырьковой сортировки, сортировки выбором и сортировки простыми вставками.

Примечания

Примечание 1. Подробнее об алгоритмах сортировки можно почитать по ссылке (https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D1%81%D0%BE%D1%80%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%BA%D0%B8).

Примечание 2. Мы называем некоторые алгоритмы сортировки **медленными**, поскольку они тратят много времени на сортировку больших списков. Например, если список содержит порядка миллиона элементов, то такие алгоритмы тратят часы, а то и дни на выполнение сортировки, в то время как быстрые алгоритмы справляются с задачей за секунды.

Примечание 3. Наглядную работу алгоритмов сортировки на разных входных данных можно посмотреть по ссылке (https://www.toptal.com/developers/sorting-algorithms).

Happy Pythoning!

Шаг 2

Сортировка пузырьком

Алгоритм сортировки пузырьком состоит из повторяющихся проходов по сортируемому списку. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по списку повторяются $n - 1$ раз, где n – длина списка. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент списка ставится на свое место в конце списка рядом с предыдущим «наибольшим элементом».

Наибольший элемент каждый раз «всплывает» до нужной позиции, как пузырёк в воде — отсюда и название алгоритма.

Алгоритм пузырьковой сортировки считается учебным и практически не применяется вне учебной литературы, а на практике применяются более эффективные.

Рассмотрим работу алгоритма на примере сортировки списка `a = [5, 1, 4, 2, 8]` по возрастанию.

Первый проход:

1. `[5, 1, 4, 2, 8]` → `[1, 5, 4, 2, 8]`: меняем местами первый и второй элементы, так как $5 > 1$;
2. `[1, 5, 4, 2, 8]` → `[1, 4, 5, 2, 8]`: меняем местами второй и третий элементы, так как $5 > 4$;
3. `[1, 4, 5, 2, 8]` → `[1, 4, 2, 5, 8]`: меняем местами третий и четвертый элементы, так как $5 > 2$;
4. `[1, 4, 2, 5, 8]` → `[1, 4, 2, 5, 8]`: не меняем четвертый и пятый элементы местами, так как $5 < 8$;
5. Самый большой элемент встал («всплыл») на свое место.

Второй проход:

1. `[1, 4, 2, 5, 8]` → `[1, 4, 2, 5, 8]`: не меняем первый и второй элементы местами, так как $1 < 4$;
2. `[1, 4, 2, 5, 8]` → `[1, 2, 4, 5, 8]`: меняем местами второй и третий элементы, так как $4 > 2$;
3. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`: не меняем местами третий и четвертый элементы, так как $4 < 5$;
4. Второй по величине элемент встал («всплыл») на свое место.

Теперь список полностью отсортирован, но алгоритму это неизвестно и он работает дальше.

Третий проход:

1. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`: не меняем первый и второй элементы местами, так как $1 < 2$;

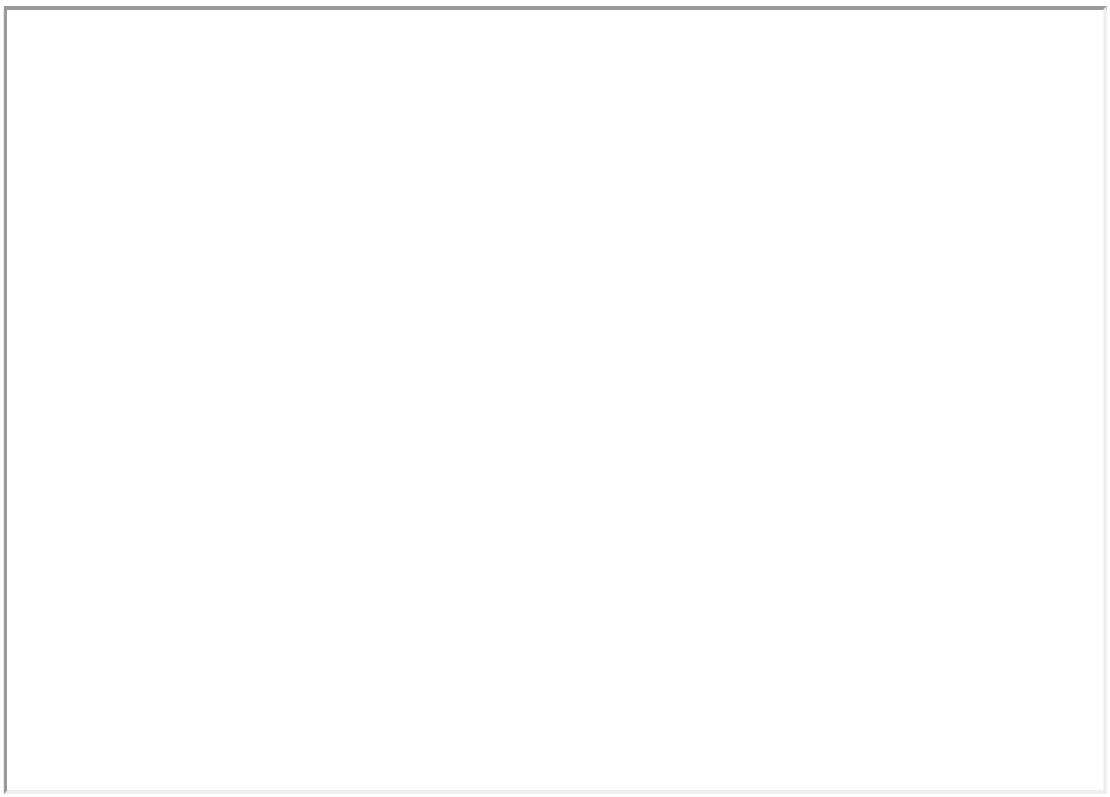
2. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`: не меняем второй и третий элементы местами, так как $2 < 4$;
3. Третий по величине элемент встал («всплыл») на свое место. (на котором и был)

Четвертый проход:

1. `[1, 2, 4, 5, 8]` → `[1, 2, 4, 5, 8]`:
2. Четвертый по величине элемент встал («всплыл») на свое место.

Теперь список отсортирован и алгоритм может быть завершен.

Визуализация алгоритма



Реализация алгоритма

Пусть требуется отсортировать по возрастанию список чисел: `a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]`.

Следующий программный код реализует алгоритм пузырьковой сортировки:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]
n = len(a)

for i in range(n - 1):
    for j in range(n - 1 - i):
        if a[j] > a[j + 1]:                  # если порядок элементов пары неправильный
            a[j], a[j + 1] = a[j + 1], a[j]  # меняем элементы пары местами

print('Отсортированный список:', a)
```

Результатом выполнения такого кода будет:

```
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]
```

Оптимизация алгоритма

Алгоритм пузырьковой сортировки можно немного ускорить. Если на одном из очередных проходов окажется, что обмены больше не нужны, то это означает, что все элементы списка находятся на своих местах, то есть список отсортирован. Для реализации такого ускорения нужно воспользоваться сигнальной меткой, то есть флажком и оператором прерывания `break`.

❤️ Happy Pythoning!

Шаг 3

Чтобы просмотреть это видео откройте https://stepik.org/lesson/310445/step/3

Шаг 4

Оптимизируйте приведённый ниже код, реализующий алгоритм пузырьковой сортировки.

Чтобы решить это задание откройте https://stepik.org/lesson/310445/step/4

Сортировка выбором

Сортировка выбором улучшает пузырьковую сортировку, совершая всего **один обмен за каждый проход по списку**. Для этого алгоритм ищет максимальный элемент и помещает его на соответствующую позицию. Как и для пузырьковой сортировки, после первого прохода самый большой элемент находится на правильном месте. После второго прохода на своё место становится следующий максимальный элемент. Проходы по списку повторяются $n - 1$ раз, где n – длина списка, поскольку последний из них автоматически оказывается на своем месте.

Алгоритм сортировки выбором также считается учебным и практически не применяется вне учебной литературы. На практике используют более эффективные алгоритмы.

Рассмотрим работу алгоритма на примере сортировки списка `a = [5, 1, 8, 2, 4]` по возрастанию.

Первый проход:

Находим максимальный элемент `8` в неотсортированной части списка и меняем его с последним элементом списка:

`[5, 1, 4, 2, 8]`.

Второй проход:

Находим максимальный элемент `5` в неотсортированной части списка и меняем его с предпоследним элементом списка:

`[2, 1, 4, 5, 8]`.

Третий проход:

Находим максимальный элемент `4` в неотсортированной части списка и меняем его с пред-предпоследним элементом списка:

`[2, 1, 4, 5, 8]`.

Четвертый проход:

Находим максимальный элемент `2` в неотсортированной части списка и меняем его с вторым элементом списка:

`[1, 2, 4, 5, 8]`.

Теперь список отсортирован и алгоритм может быть завершен.

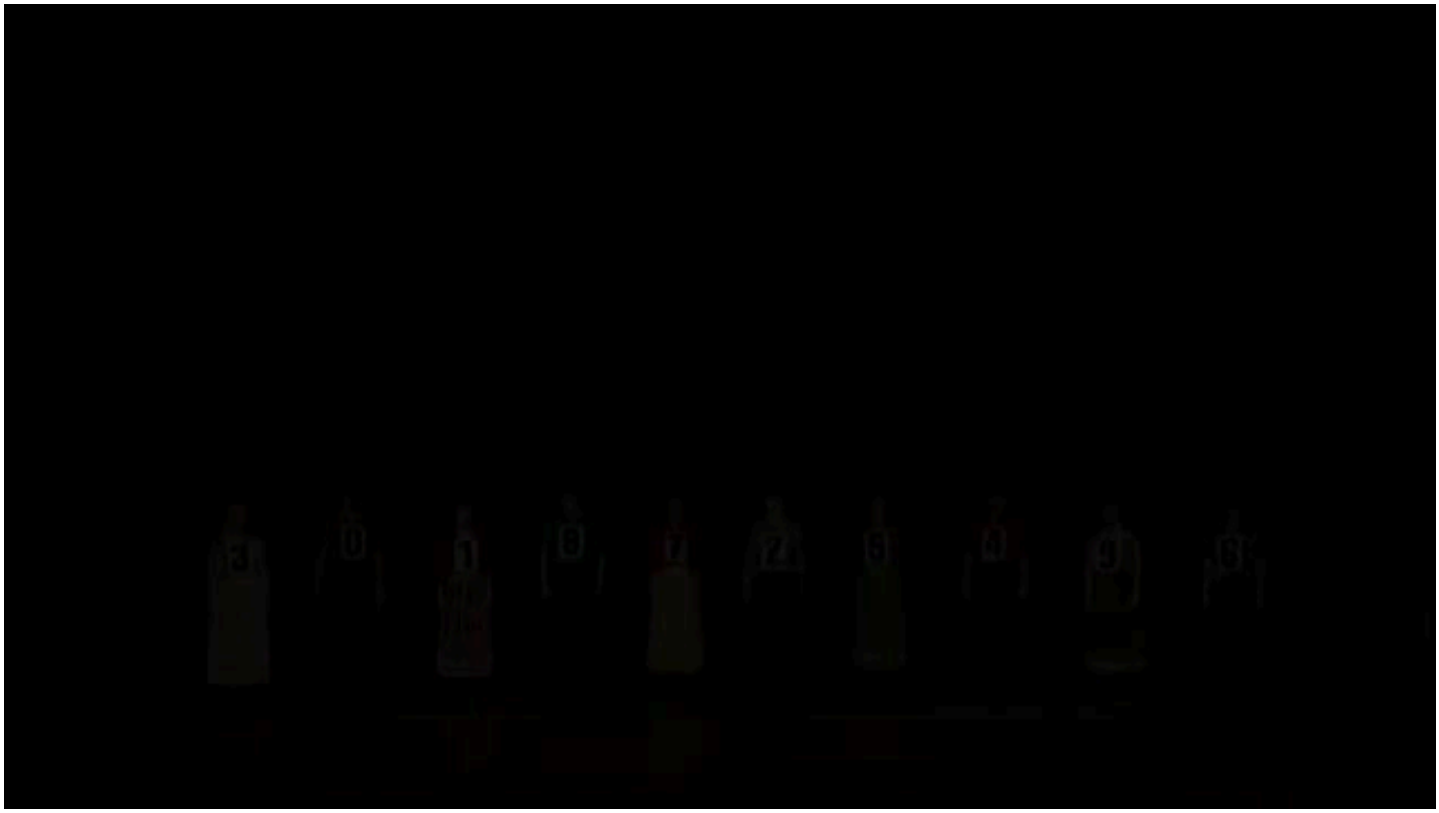
Вместо максимального элемента можно искать минимальный.

Визуализация алгоритма

Выберите в выпадающем списке алгоритм: **Selection sort**.

❤️ **Happy Pythoning!**

Шаг 6



Чтобы просмотреть это видео откройте

https://stepik.org/lesson/310445/step/6

Шаг 7

Отсортируйте список по возрастанию, реализовав алгоритм сортировки выбором.

Чтобы решить это задание откройте https://stepik.org/lesson/310445/step/7

Шаг 8

Сортировка простыми вставками

Алгоритм сортировки простыми вставками делит список на 2 части — отсортированную и неотсортированную. Из неотсортированной части извлекается очередной элемент и вставляется на нужную позицию в отсортированной части, в результате чего отсортированная часть списка увеличивается, а неотсортированная уменьшается. Так происходит, пока не исчерпан набор входных данных и не отсортированы все элементы.

Сортировка простыми вставками наиболее эффективна, когда список уже частично отсортирован и элементов массива немного. Если элементов в списке меньше 10, то этот алгоритм — один из самых быстрых.

Рассмотрим его работу на примере сортировки списка `a = [5, 1, 8, 2, 4]` по возрастанию.

Первый проход:

Делим список на две части: отсортированную `[5]` и неотсортированную `[1, 8, 2, 4]`.

Извлекаем первый элемент `1` из неотсортированной части списка и находим ему место в отсортированной части:

`[1, 5, 8, 2, 4]`.

Второй проход:

Делим список на две части: отсортированную `[1, 5]` и неотсортированную `[8, 2, 4]`.

Извлекаем первый элемент `8` из неотсортированной части списка и находим ему место в отсортированной части:

`[1, 5, 8, 2, 4]`.

Третий проход:

Делим список на две части: отсортированную `[1, 5, 8]` и неотсортированную `[2, 4]`.

Извлекаем первый элемент `2` из неотсортированной части списка и находим ему место в отсортированной части:

`[1, 2, 5, 8, 4]`.

Четвертый проход:

Делим список на две части: отсортированную `[1, 2, 5, 8]` и неотсортированную `[4]`.

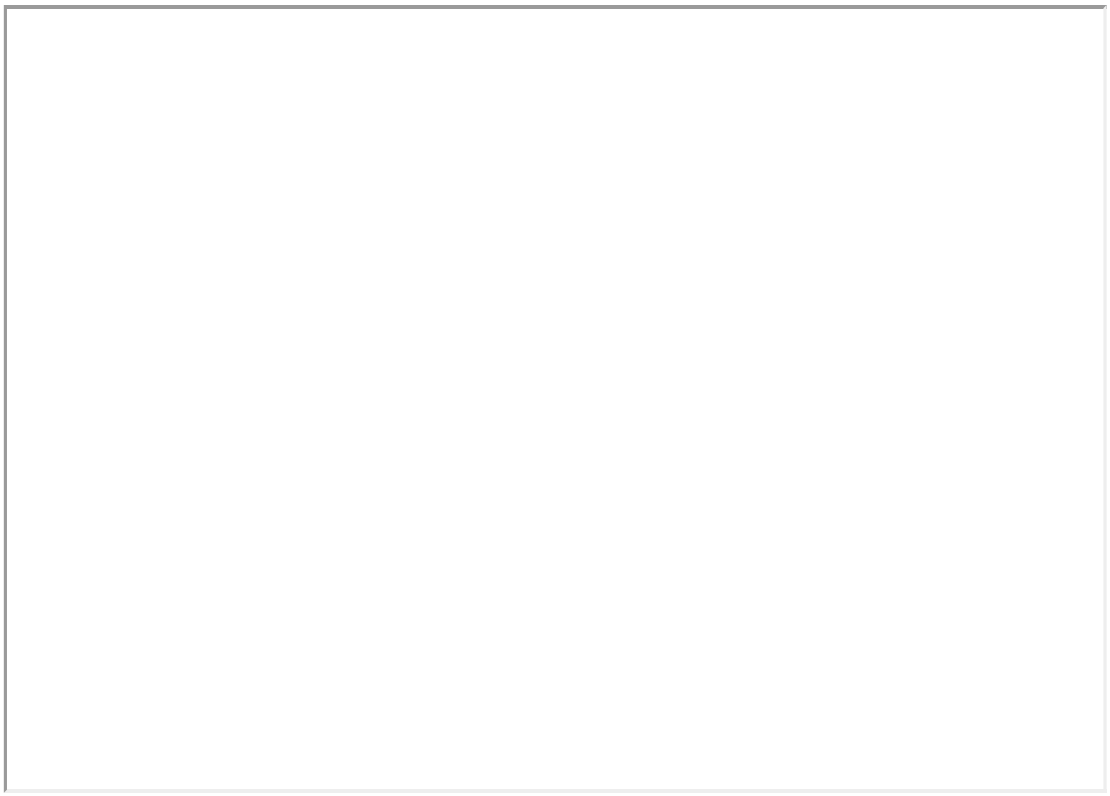
Извлекаем первый элемент `4` из неотсортированной части списка и находим ему место в отсортированной части:

`[1, 2, 4, 5, 8]`.

Теперь список отсортирован, и алгоритм может быть завершен.

Визуализация алгоритма

Выберите в выпадающем списке алгоритм: **Insert sort**.



Реализация алгоритма

Пусть требуется отсортировать по возрастанию список чисел: `a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]`.

Следующий программный код реализует алгоритм сортировки простыми вставками:

```
a = [1, 7, -3, 9, 0, -67, 34, 12, 45, 1000, 6,  8, -2, 99]
n = len(a)

for i in range(1, n):
    elem = a[i]  # берем первый элемент из неотсортированной части списка
    j = i

    # пока элемент слева существует и больше нашего текущего элемента
    while j >= 1 and a[j - 1] > elem:
        # смещаем j-й элемент отсортированной части вправо
        a[j] = a[j - 1]
        # сами идём влево, дальше ищем место для нашего текущего элемента
        j -= 1

    # нашли место для нащего текущего элемента из неотсортированной части
    # и вставляем его на индекс j в отсортированной части
    a[j] = elem


print('Отсортированный список:', a)
```

Результатом выполнения такого кода будет:

```
Отсортированный список: [-67, -3, -2, 0, 1, 6, 7, 8, 9, 12, 34, 45, 99, 1000]
```

Оптимизация алгоритма

Алгоритм сортировки простыми вставками можно значительно ускорить, если осуществлять поиск нужной позиции для вставки очередного элемента из неотсортированной части списка с помощью бинарного поиска (https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA).

Шаг 9

Чтобы просмотреть это видео откройте

https://stepik.org/lesson/310445/step/9