Сортировка

Сортировка — это упорядочивание элементов. Элементы могут являться элементами массива, контейнера, файлами на диске, записями в базе данных и т. д.

При проведении сортировки используется понятие ключа.

Ключ (в контексте сортировки) — это некое значение, имеющееся у каждого элемента, участвующего в сортировке, которое можно сравнить с таким же значением другого элемента, используя операции «Больше» или «Меньше».

При сортировке файлов, ключом может являться имя файла, его размер или дата модификации, в зависимости от запроса пользователя.

В тривиальном случае — массив чисел — каждое число также является и ключом сортировки.

Свойства сортировки

В зависимости от направления, различают:

- Сортировка по **неубыванию**. Для любой пары элементов с номерами i и i+1, ключи находятся в соотношении $A_i \leq A_{i+1}$.
- Сортировка по **невозрастанию**. Для любой пары элементов с номерами i и i+1, ключи находятся в соотношении $A_i \geq A_{i+1}$.

В зависимости от свойств алгоритма, различают:

- **Устойчивая** или **стабильная** сортировка. После проведения сортировки элементы с одинаковым ключом сохраняют относительное положение.
- **Неустойчивая** или **нестабильная** сортировка. После проведения сортировки элементы с одинаковым ключом могут поменять свой порядок следования.

Вычислительная сложность

Вычислительная сложность — это функция зависимости объема работы, которая выполняется некоторым алгоритмом, от размера входных данных.

Объем работы измеряется абстрактными понятиями: время и пространство.

Время — это количество элементарных операций, необходимых для выполнения алгоритма.

Пространство — это объем памяти, который требуется алгоритму.

Асимптотическая сложность

Конкретные значения времени и затрат памяти зависят от платформы. Поэтому на практике интересна зависимость времени от объема решаемой задачи.

Асимптотическая вычислительная сложность показывает, какой характер будет иметь вычислительная сложность алгоритма при устремлении объема решаемой задачи в бесконечность. Для ее обозначения используют O-нотацию.

Запись вида **O(f(n))** означает, что время работы алгоритма (количество операций) или расход памяти ограничены сверху функцией f(n), где n – объем решаемой задачи.

Запись вида **Ω(f(n))** означает, что время работы алгоритма ограничено снизу функцией f(n).

Таким образом, Ω(f(n)) и O(f(n)) представляют собой наилучший и наихудший случаи соответственно

Асимптотическая сложность

На практике, оценка времени работы как **O(f(n))** означает, что время работы алгоритма будет равно **T = C · f(n)**, где C — константа, включающая в себя все особенности платформы, которая может быть определена экспериментально.

Пример. Алгоритм имеет сложность $O(n^2)$. Замер показал, что обработка 10^3 элементов занимает около 100 мс. Обработка $2 \cdot 10^3$ элементов займет в 4 раза больше времени, то есть 400 мс.

Применив для той же задачи алгоритм со сложностью $O(n \cdot \log n)$, получим время работы 150 мс для 10^3 элементов, что означает, что константа C для первого алгоритма меньше. Однако уже для $2 \cdot 10^3$ элементов время работы второго алгоритма составит 330 мс. Дальнейшее увеличение объема задачи будет увеличивать отрыв второго алгоритма.

Сложность алгоритмов сортировки

Алгоритмам сортировки доступно всего 2 операции: сравнение и перестановка. Объем решаемой задачи n — это количество сортируемых элементов.

При проведении k сравнений, возможно 2^k вариантов результатов сравнений.

Количество перестановок n элементов равно n!

Чтобы каждой перестановке соответствовало хотя бы одно сравнение, необходимо сделать не менее $\log_2 n!$ сравнений.

Можно показать, что функция $\log_2 n!$ ограничена снизу функцией $n \cdot \log n$, что дает нижнюю оценку сложности алгоритма сортировки $\Omega(n \cdot \log n)$.

Если верхняя оценка сложности алгоритма сортировки равна $O(n \cdot \log n)$, то такой алгоритм можно считать оптимальным.

Идеальный алгоритм сортировки

- Является стабильным.
- Работает на месте, используя O(1) дополнительного места.
- Имеет сложность O(n·log n), совершая не более O(n·log n) сравнений и не более O(n) перестановок.
- Адаптивный: сложность приближается к O(n) при работе с почти отсортированными данными, либо когда данные содержат много одинаковых ключей.

Ни один алгоритм сортировки не соответствует всем критериям, следует выбирать алгоритм исходя из решаемой задачи.

Сортировка выбором

Один из наиболее простых в написании алгоритмов.

Имеет сложность $O(n^2)$ и затраты памяти $O(1)$.

Может быть как устойчивым, так и неустойчивым.

1) Объявляется переменная-индекс и устанавливается в первый по счету элемент. Все элементы слева от индекса отсортированы.

2) При помощи последовательного перебора выбирается наименьший элемент справа от индекса и меняется местами с элементом по индексу.

3) Индекс увеличивается на 1.

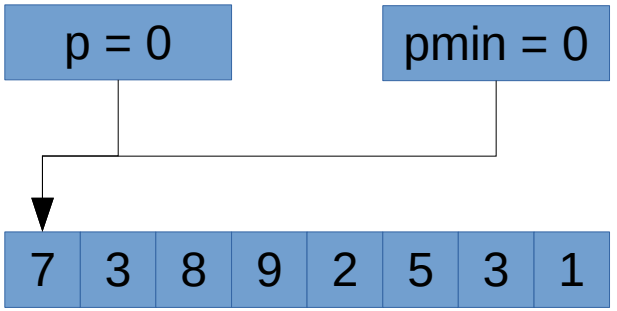
4) Пункты 2-3 повторяются пока индекс не дойдет до последнего элемента.

Сортировка выбором

```
def sort_select(ar):




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```

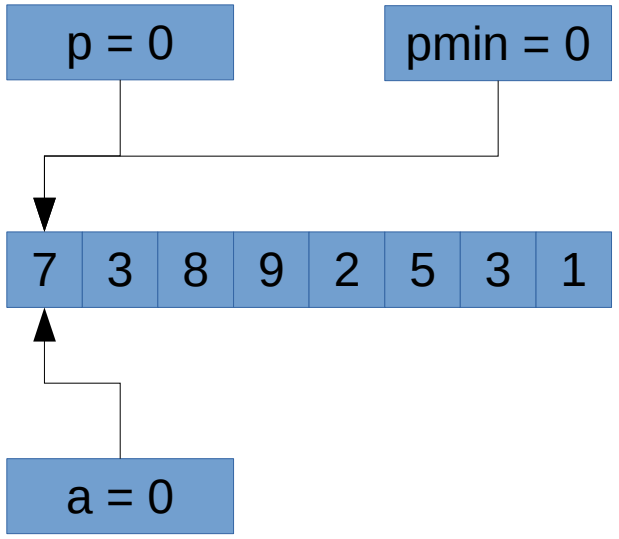
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



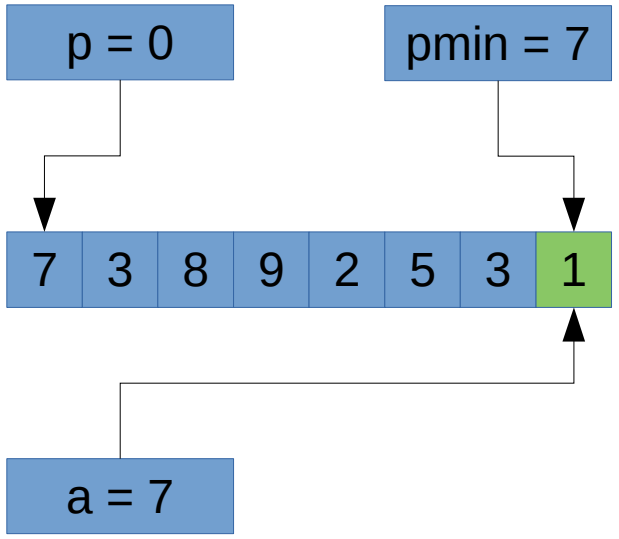
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



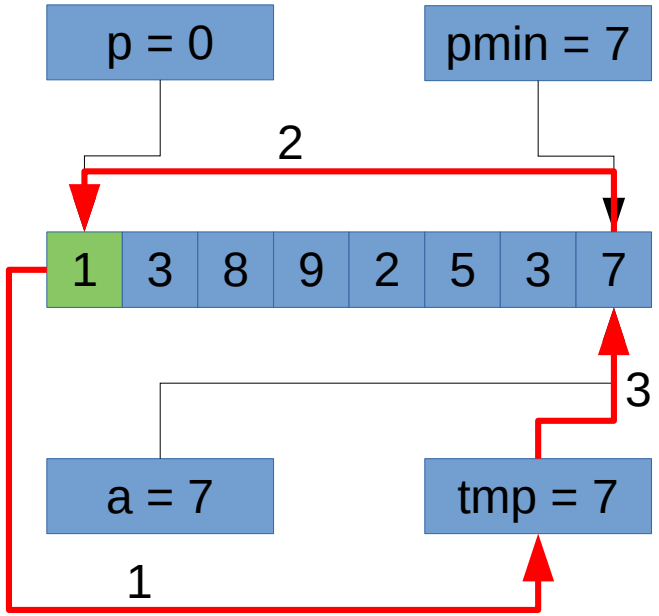
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



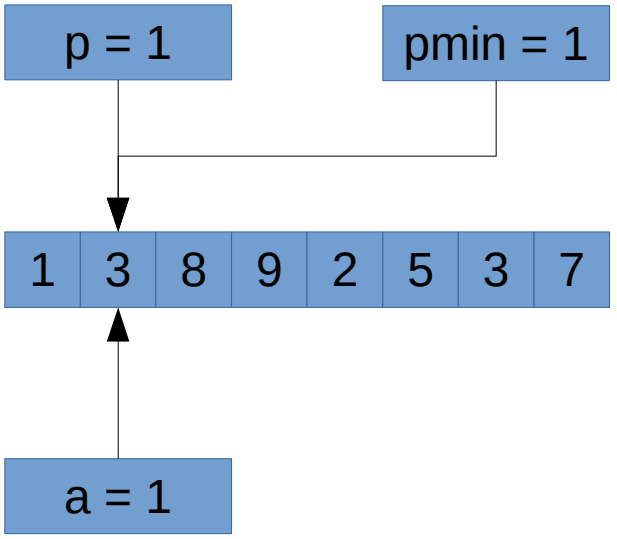
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



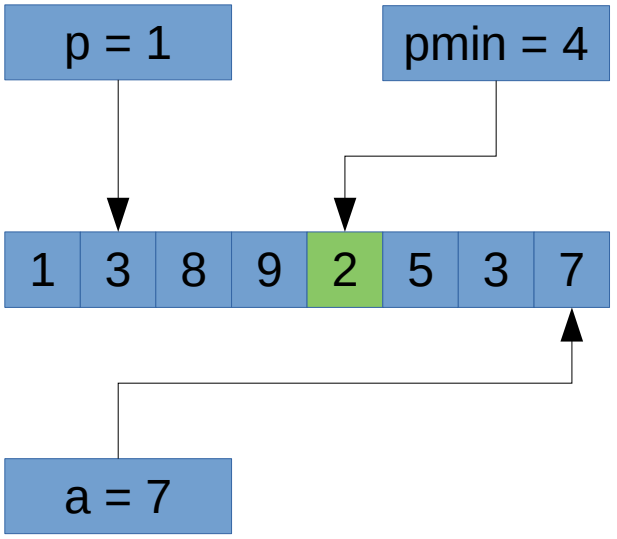
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



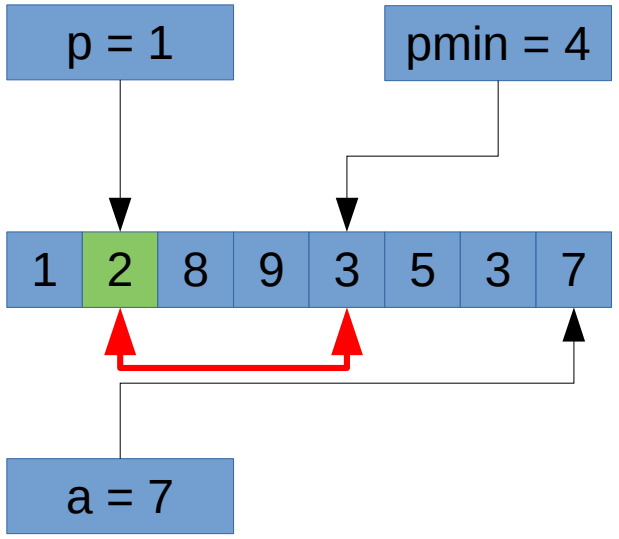
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



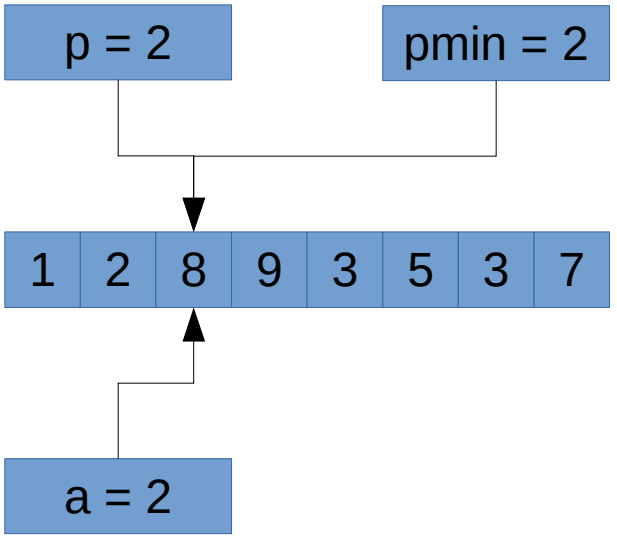
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



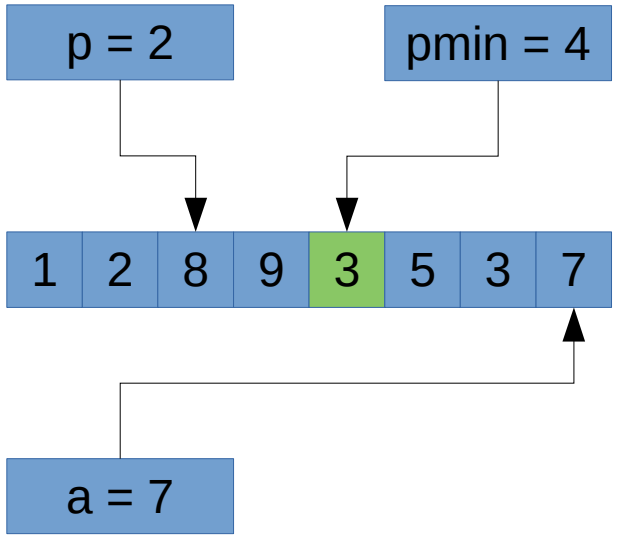
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



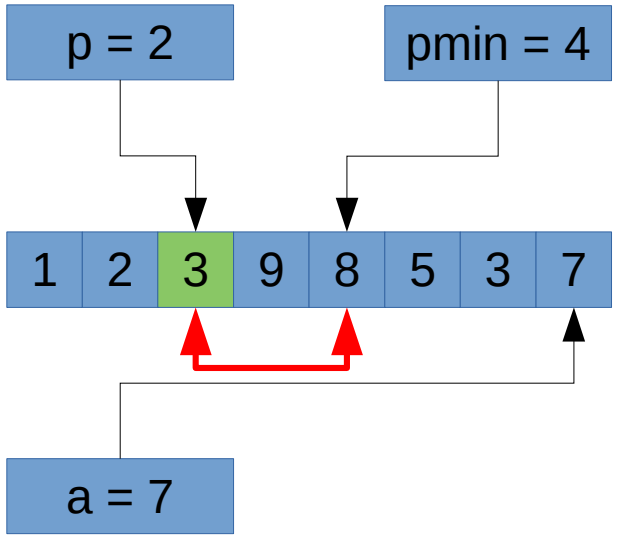
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



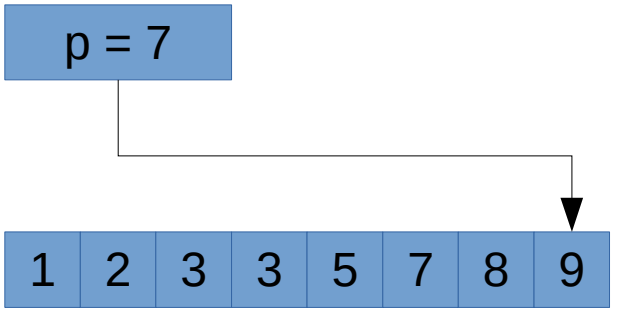
Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



Сортировка выбором

```
def sort_select(ar):
  for p in range(len(ar)):
    pmin = p
    for a in range(p, len(ar)):
      if ar[a] < ar[pmin]:
        pmin = a
    tmp       = ar[p]
    ar[p]     = ar[pmin]
    ar[pmin] = tmp
    # ar[p], ar[pmin] = ar[pmin], ar[p]



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_select(A)
```



Сортировка вставками

Алгоритм близок по сути к сортировке выбором.

Имеет сложность $O(n^2)$ и затраты памяти $O(1)$.

Является устойчивым.

1) Объявляется переменная-индекс и устанавливается в элемент с индексом 1. Все элементы слева от индекса отсортированы.

2) Выбирается элемент, расположенный по данному индексу, и меняется местами с предыдущим, пока не встанет на свое место в отсортированных данных.

3) Индекс увеличивается на 1.

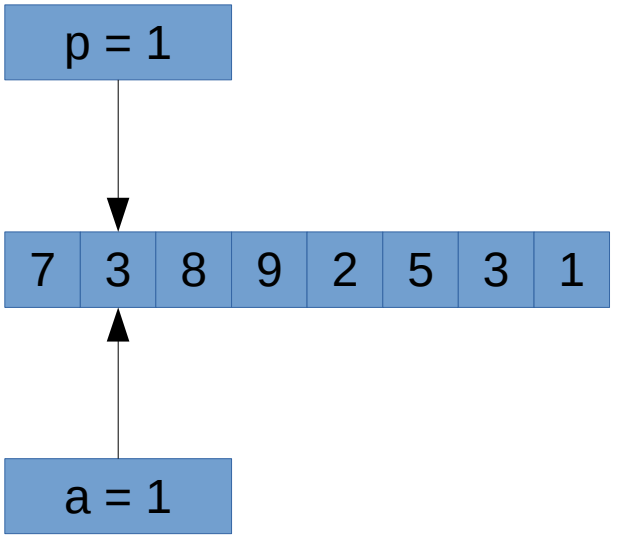
4) Пункты 2-3 повторяются пока индекс не дойдет до последнего элемента.

Сортировка вставками

```
def sort_insert(ar):




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```

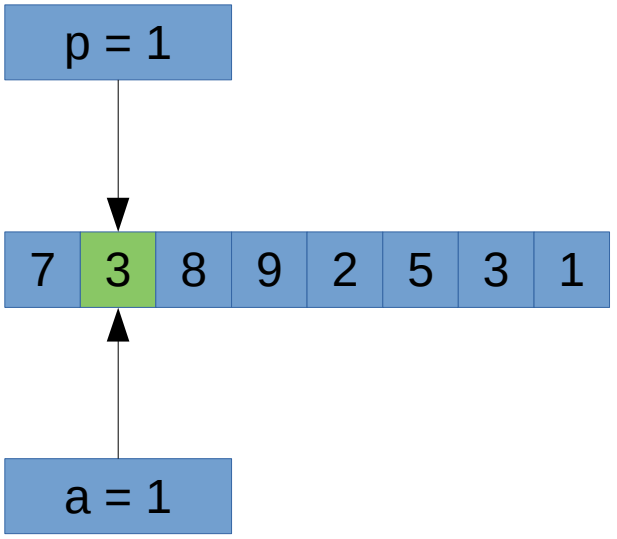
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



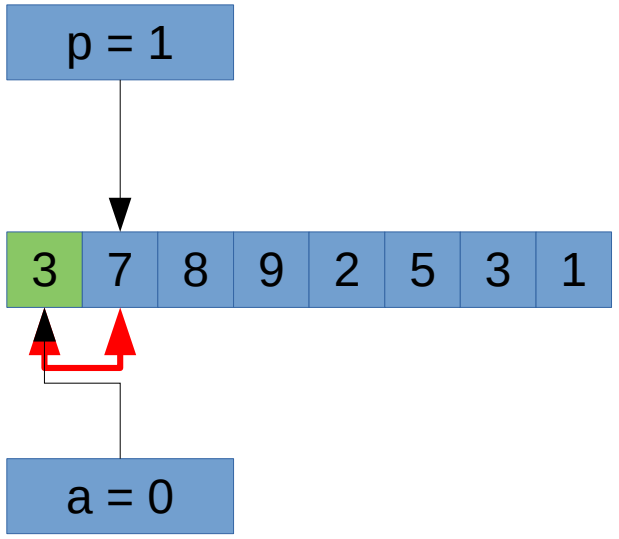
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



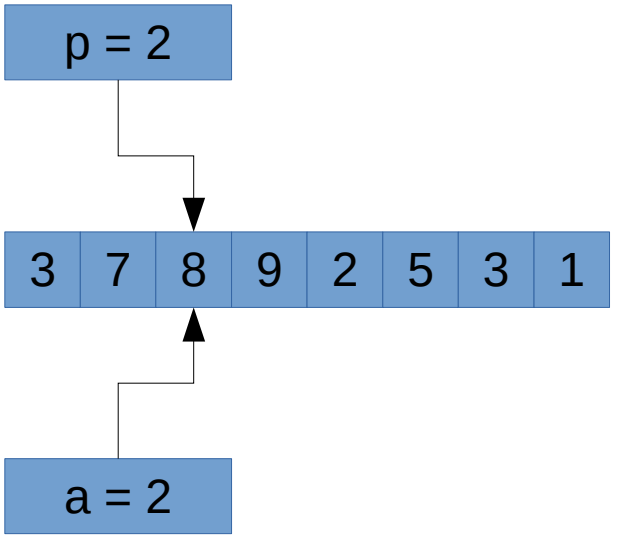
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



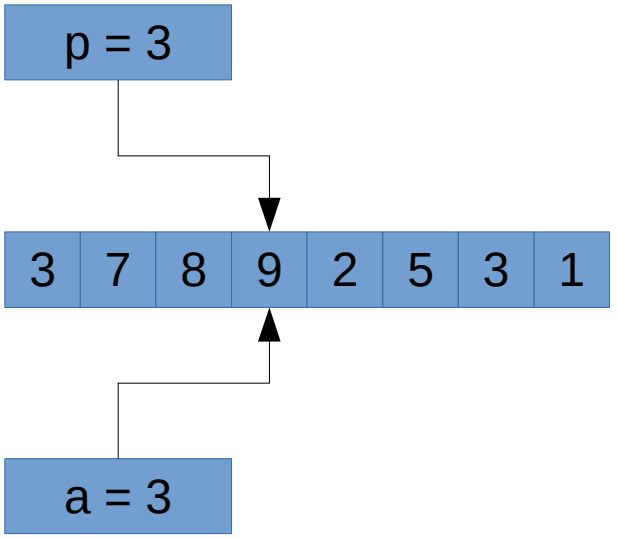
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1

A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



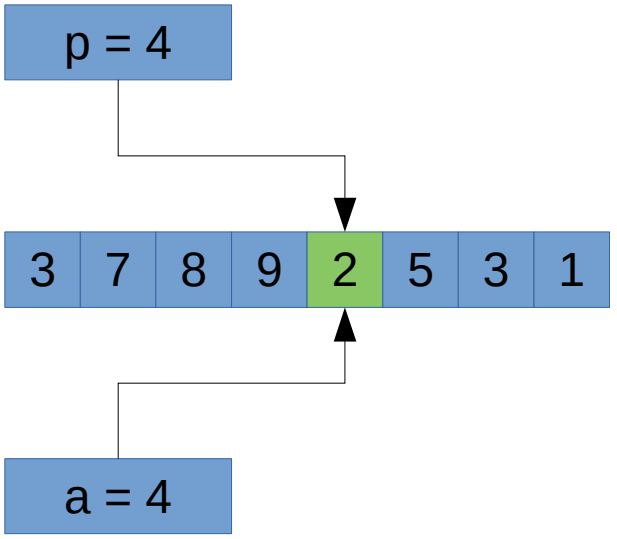
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



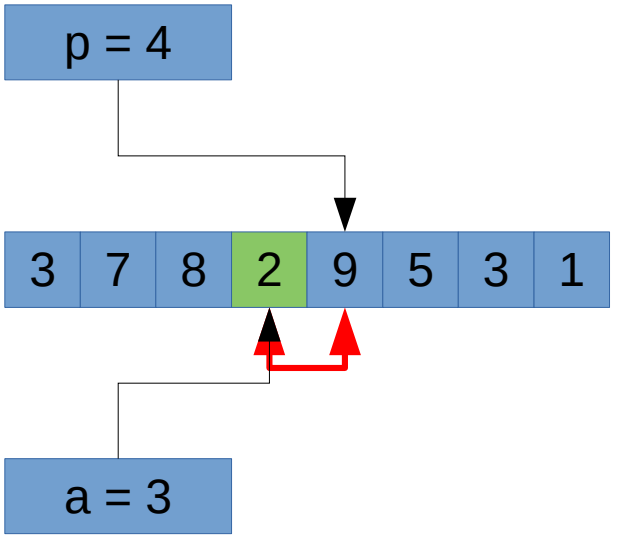
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



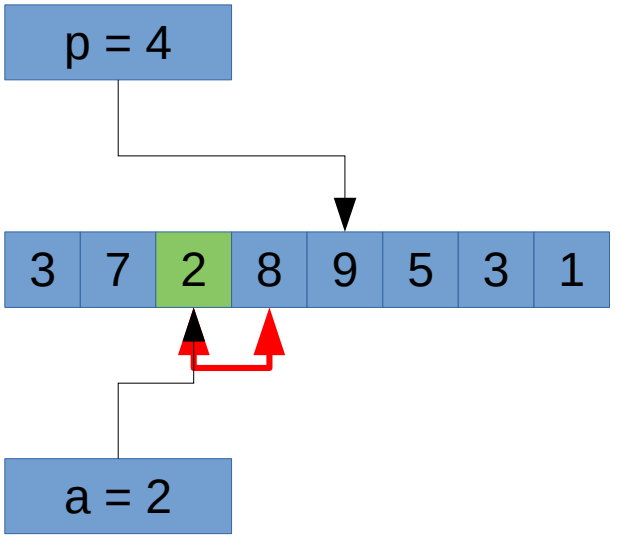
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



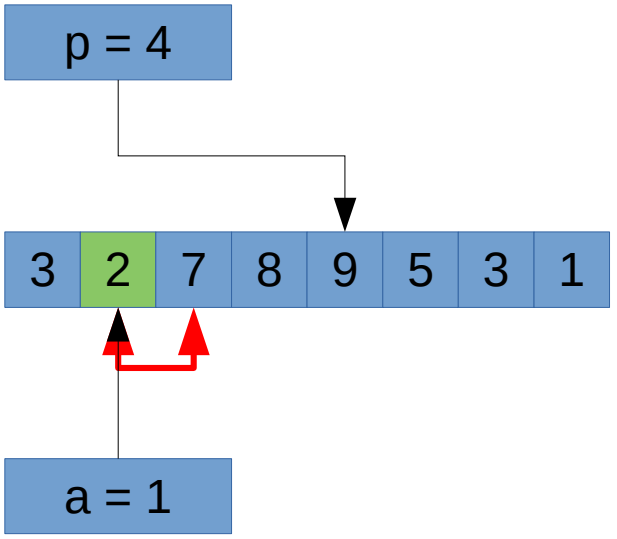
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



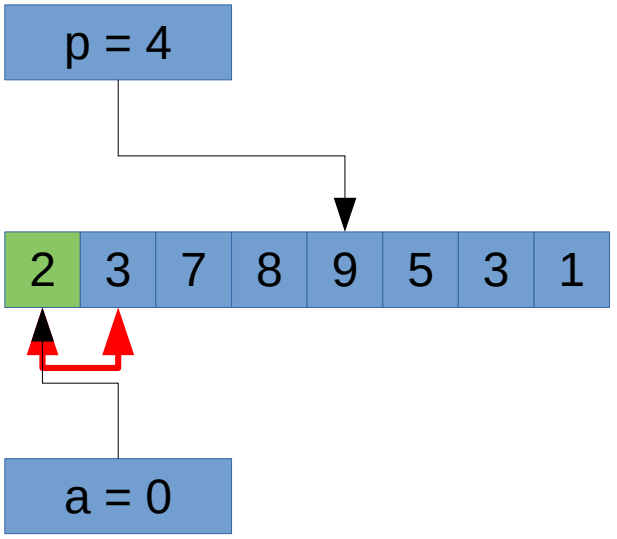
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



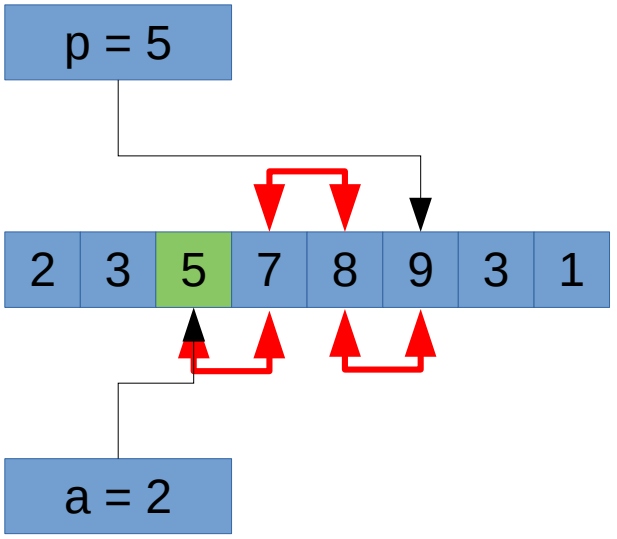
Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



Сортировка вставками

```
def sort_insert(ar):
  for p in range(1, len(ar)):
    a = p
    while ar[a] < ar[a-1] and a > 0:
      ar[a], ar[a-1] = ar[a-1], ar[a]
      a -= 1

A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_insert(A)
```



Пузырьковая сортировка

Алгоритм основан на последовательной замене соседних элементов.

Имеет сложность $O(n^2)$ и затраты памяти $O(1)$.

Является устойчивым.

1) Объявляется переменная-индекс, определяющая границу движения «пузырька».

2) Запускается цикл от начала массива до значения индекса.

3) В цикле текущий элемент меняется местами со следующим, если пара элементов не удовлетворяет условию сортировки.

4) Индекс уменьшается на 1.

5) Пункты 2-4 повторяются пока индекс не дойдет до начала массива.

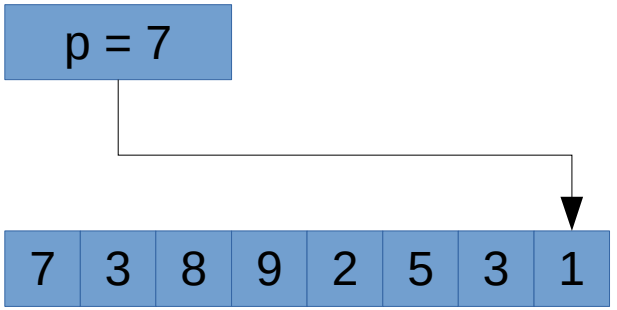
Пузырьковая сортировка

```
def sort_bubble(ar):









A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```

7	3	8	9	2	5	3	1

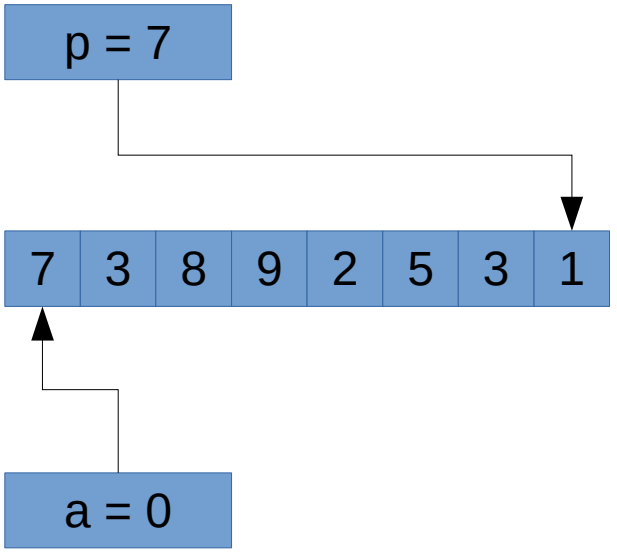
Пузырьковая сортировка

```
def sort_bubble(ar):
  p = len(ar) – 1









A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



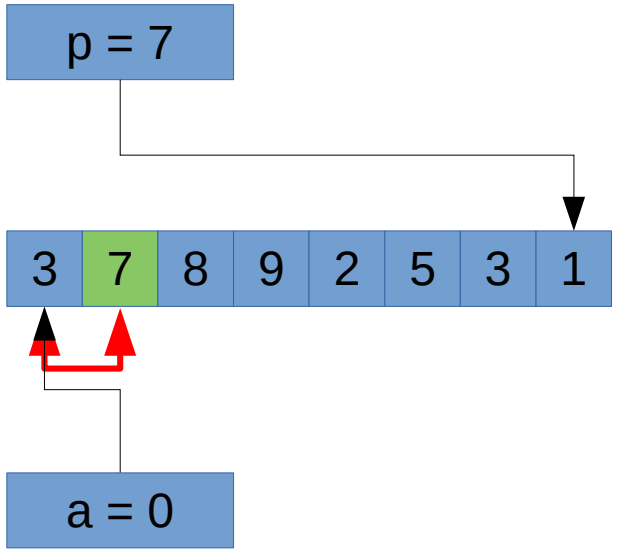
Пузырьковая сортировка

```
def sort_bubble(ar):
  p = len(ar) – 1
  while p >= 0:
    for a in range(p):









A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



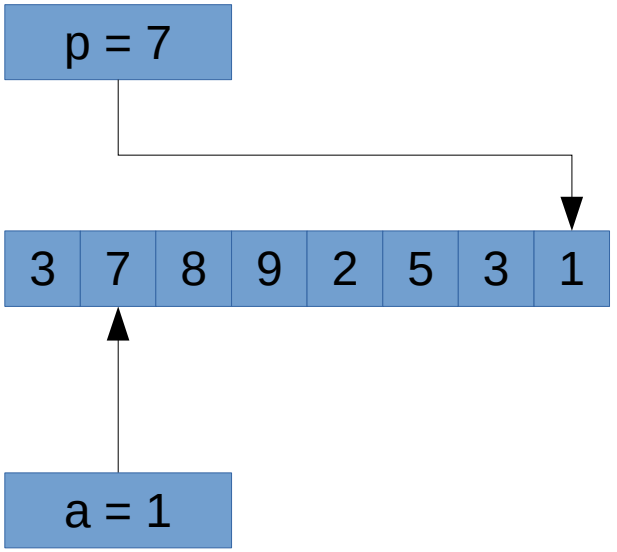
Пузырьковая сортировка

```
def sort_bubble(ar):
  p = len(ar) – 1
  while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



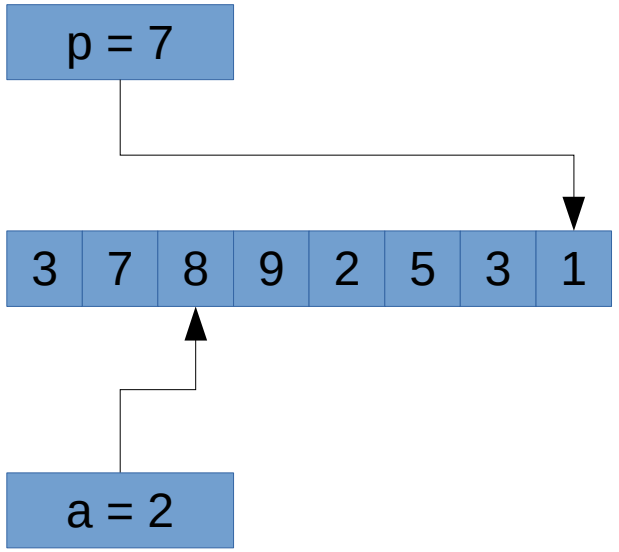
Пузырьковая сортировка

```
def sort_bubble(ar):
  p = len(ar) – 1
  while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]
    p -= 1



A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



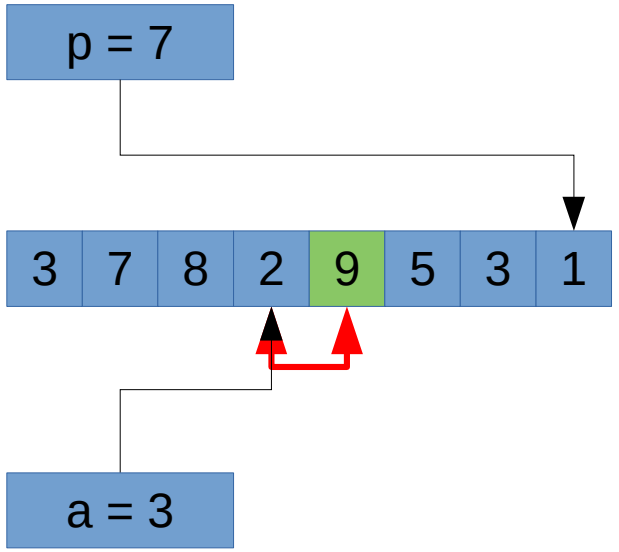
Пузырьковая сортировка

```
def sort_bubble(ar):
  for p in range(len(ar)-1, -1, -1):
  # p = len(ar) – 1
  # while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]
    # p -= 1




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



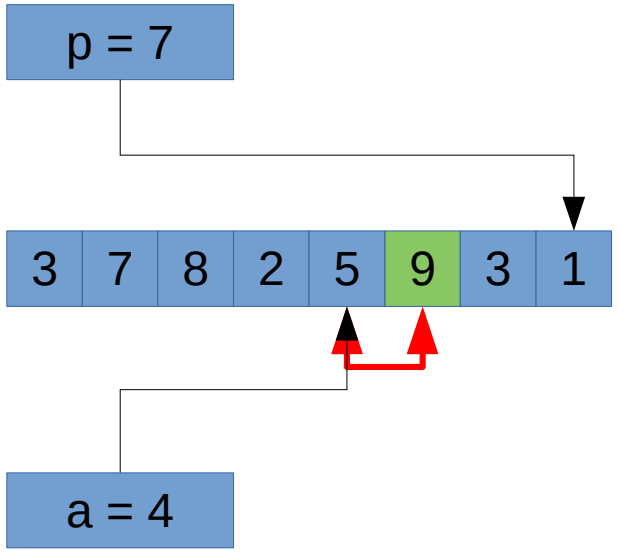
Пузырьковая сортировка

```
def sort_bubble(ar):
  for p in range(len(ar)-1, -1, -1):
  # p = len(ar) – 1
  # while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]
    # p -= 1




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



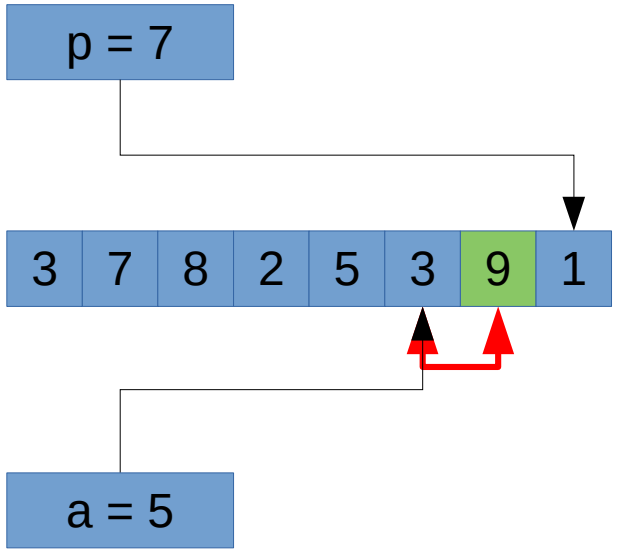
Пузырьковая сортировка

```
def sort_bubble(ar):
  for p in range(len(ar)-1, -1, -1):
  # p = len(ar) – 1
  # while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]
    # p -= 1




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



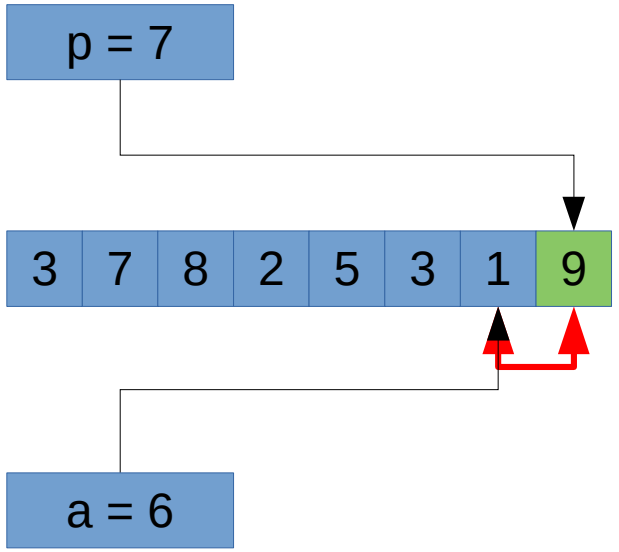
Пузырьковая сортировка

```
def sort_bubble(ar):
  for p in range(len(ar)-1, -1, -1):
  # p = len(ar) – 1
  # while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]
    # p -= 1




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



Пузырьковая сортировка

```
def sort_bubble(ar):
  for p in range(len(ar)-1, -1, -1):
  # p = len(ar) – 1
  # while p >= 0:
    for a in range(p):
      if ar[a] > ar[a+1]:
        ar[a], ar[a+1] = ar[a+1], ar[a]
    # p -= 1




A = [7, 3, 8, 9, 2, 5, 3, 1]
sort_bubble(A)
```



Быстрая сортировка

Алгоритм появился как попытка модернизации алгоритма прямого обмена (один из его вариантов — алгоритм пузырьковой сортировки). Был предложен Чарльзом Хоаром в 1960 году.

Имеет среднюю сложность $O(n \cdot \log n)$. Однако в плохих случаях может деградировтаь до $O(n^2)$. Затраты памяти порядка $O(\log n)$.

Является неустойчивым.

В основу модернизации положено два принципа:

1) Перестановки осуществляются на максимально возможном расстоянии. Цель — разделить массив на две части, в одной из которых все элементы меньше, а во второй — больше некоторого.

2) После проведения ряда перестановок, части массива сортируются по отдельности.

Быстрая сортировка

1) Определяются 2 индекса: L, смотрящий на нулевой элемент, и R, смотрящий на последний.

2) Определяется переменная M, хранящая значение медианного элемента (например, элемента из середины массива).

3) Значение L увеличивается до тех пор, пока элемент с индексом L меньше медианного. Аналогично, значение R уменьшается до тех пор, пока элемент с индексом R больше медианного.

4) Элементы под индексами R и L меняются местами, после чего R и L делают еще один шаг.

5) Пункты 3 и 4 повторяются до тех пор, пока L не больше R.

6) Полученные два подмассива — от начала до R и от L до конца — содержат элементы меньше и больше M соответственно. Они сортируются рекурсивно.

Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):









A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```

start = 0

size = -1

7	3	8	9	6	5	3	1

Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return









A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```

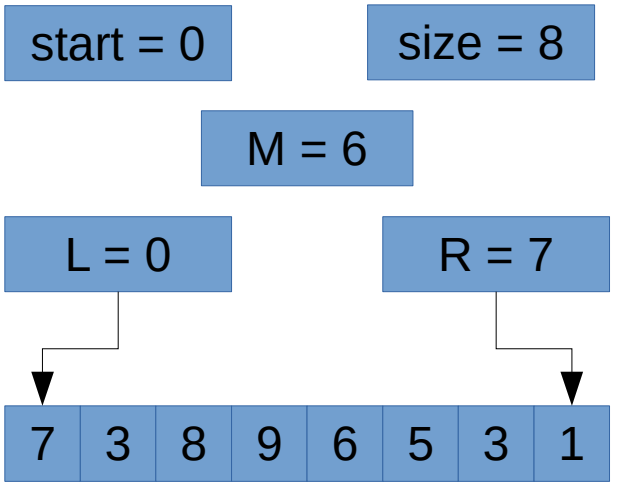
start = 0

size = 8

7	3	8	9	6	5	3	1

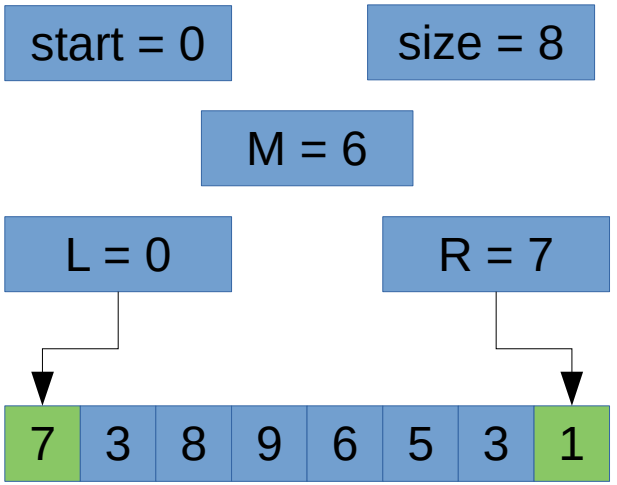
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]









A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



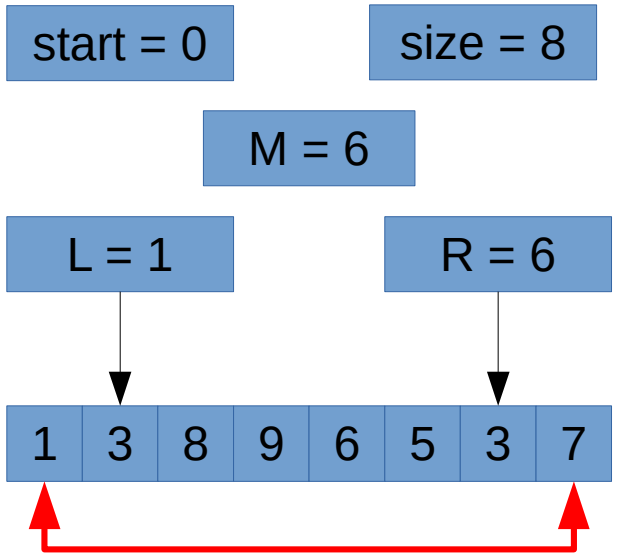
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1




A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



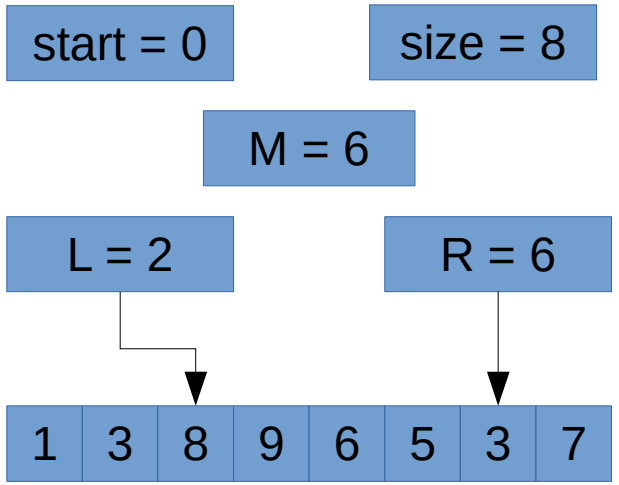
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1



A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



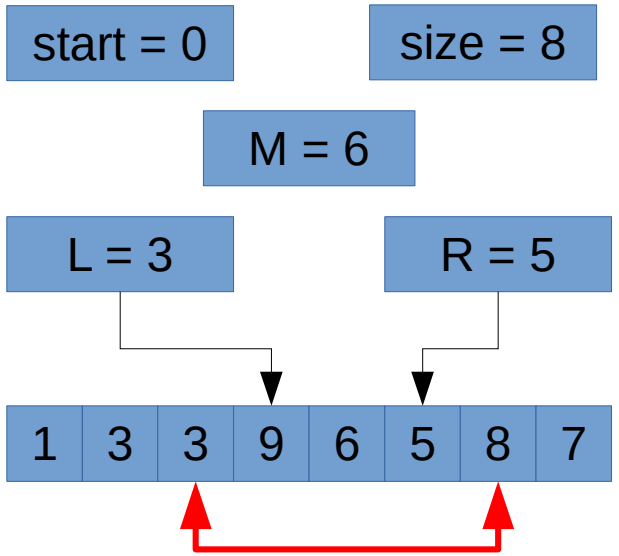
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1



A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



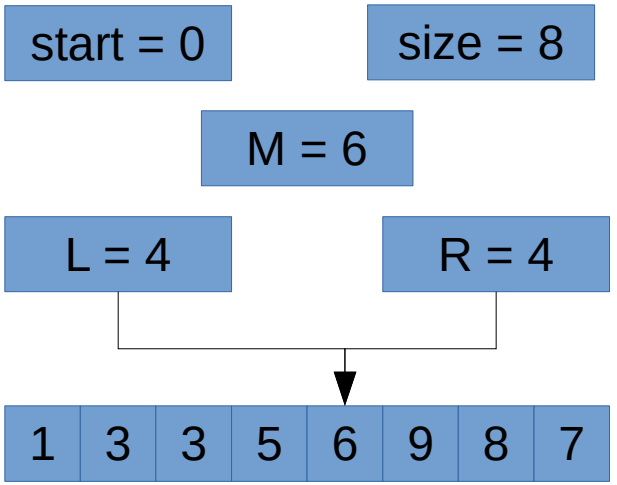
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1



A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



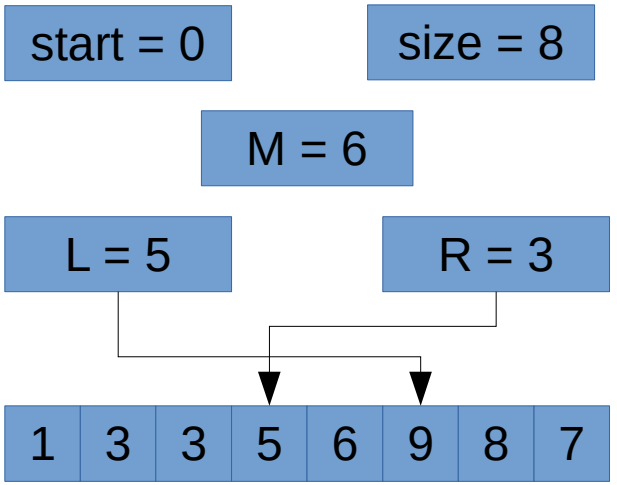
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1



A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



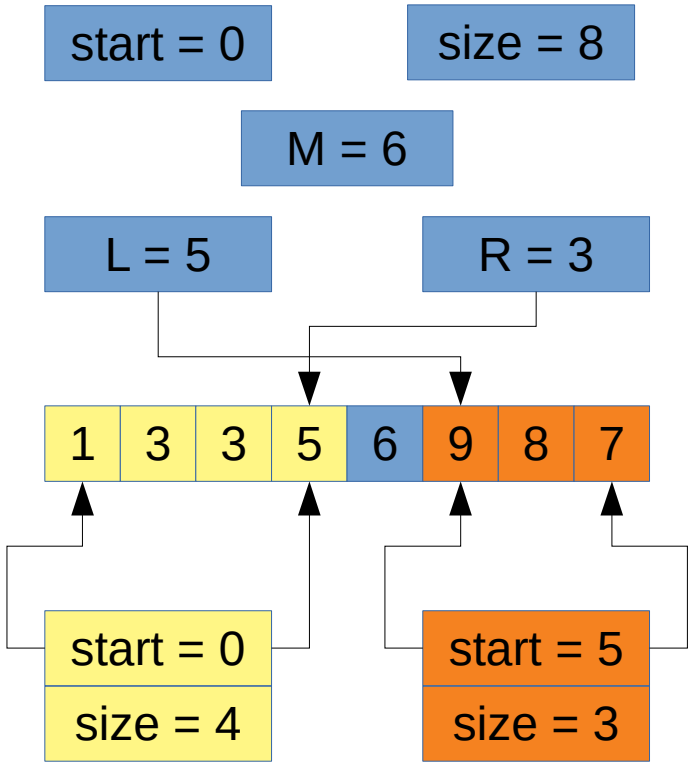
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1



A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



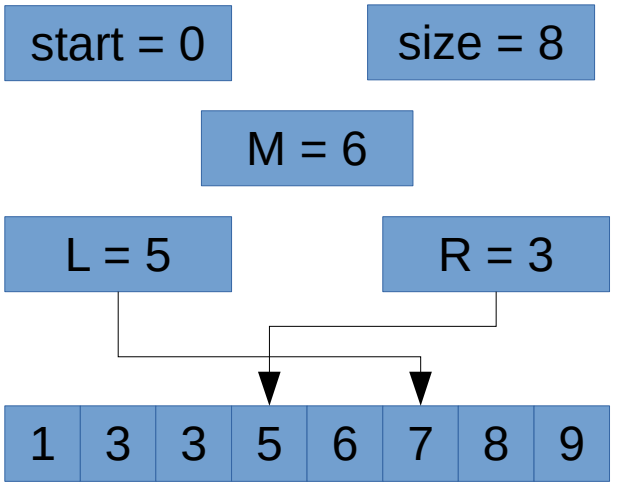
Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R -= 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1
  sort_quick(ar, start, R + 1 - start)
  sort_quick(ar, L, start + size – L)


A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



Быстрая сортировка

```
def sort_quick(ar, start = 0, size = -1):
  if size < 0: size = len(ar)
  if size < 2: return
  L = start
  R = start + size – 1
  M = ar[start + size // 2]
  while L <= R:
    while ar[L] < M: L += 1
    while ar[R] > M: R += 1
    if L <= R:
      ar[L], ar[R] = ar[R], ar[L]
      L += 1
      R -= 1
  sort_quick(ar, start, R + 1 - start)
  sort_quick(ar, L, start + size – L)


A = [7, 3, 8, 9, 6, 5, 3, 1]
sort_quick(A)
```



Быстрая сортировка

Недостатки алгоритма:

- Неустойчивость;
- Зависимость от способа выбора медианного значения, склонность к деградации вплоть до $O(n^2)$;
- Высокая константа.

Иногда применяют сортировку подмассивов другими методами, если их размер мал.

Сортировка слиянием

Алгоритм сортировки слиянием основан на двух фактах:

1) Массив из одного элемента отсортирован по определению;

2) Объединение двух отсортированных массивов в один — сравнительно простая операция.

Имеет среднюю сложность $O(n \cdot \log n)$. Затраты памяти порядка $O(n)$.

Является устойчивым.

Сортировка слиянием

1) Если исходный массив единичного размера, алгоритм ничего с ним не делает.

2) Исходный массив разбивается на 2 части, каждая из которых сортируется тем же алгоритмом.

3) Объявляются два индекса p1 и p2, устанавливаемых на нулевые элементы каждого из подмассивов.

4) Определяется индекс p, устанавливаемый на нулевой элемент конечного массива.

5) Если значение под индексом p1 меньше, то оно помещается в конечный массив под индексом p, после чего p1 и p увеличиваются на единицу. Иначе то же самое проделывается с p2 и p. Процесс повторяется до тех пор, пока p1 или p2 не дойдут до конца своего массива.

6) Оставшиеся элементы переписываются в конечный массив.

** Чтобы избежать выделения памяти на каждом этапе, следует заранее выделить массив для хранения результатов.*

Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
```

start = 0

size = -1

Ai | 7 | 3 | 8 | 9 | 6 | 5 | 3 | 1 |

Ao | 7 | 3 | 8 | 9 | 6 | 5 | 3 | 1 |

```
ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```

ar | 7 | 3 | 8 | 9 | 6 | 5 | 3 | 1 |

Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return




ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```

start = 0

size = 8

Ai	7	3	8	9	6	5	3	1
Ao	7	3	8	9	6	5	3	1
ar	7	3	8	9	6	5	3	1

Сортировка слиянием

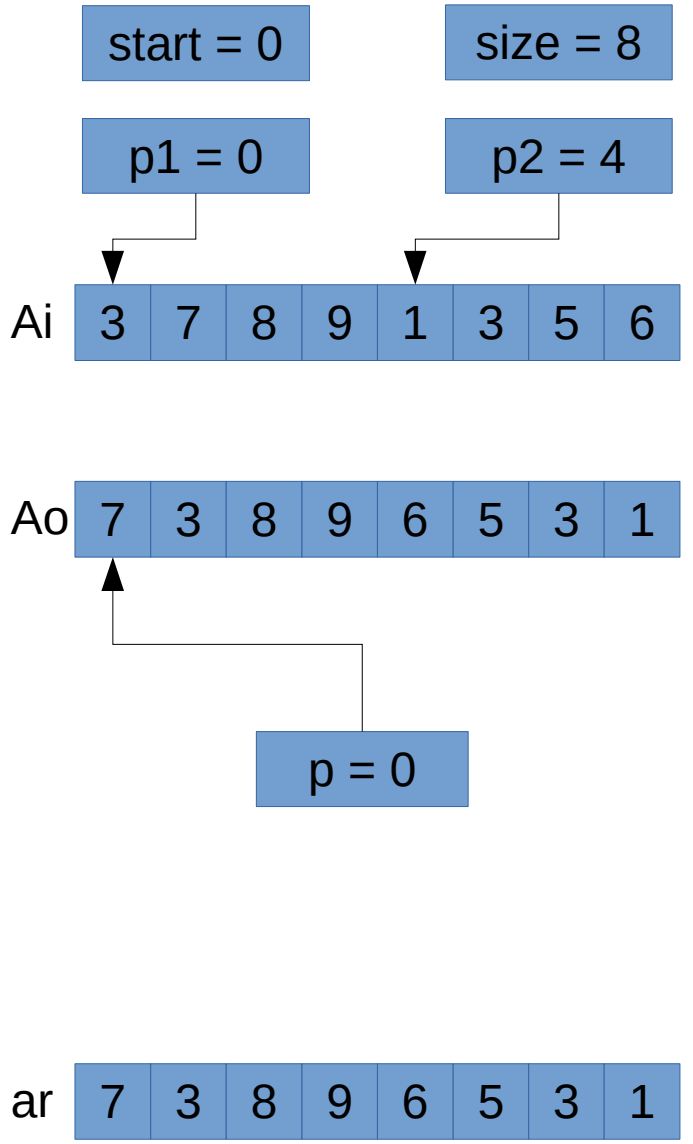
```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```

start = 0

size = 8

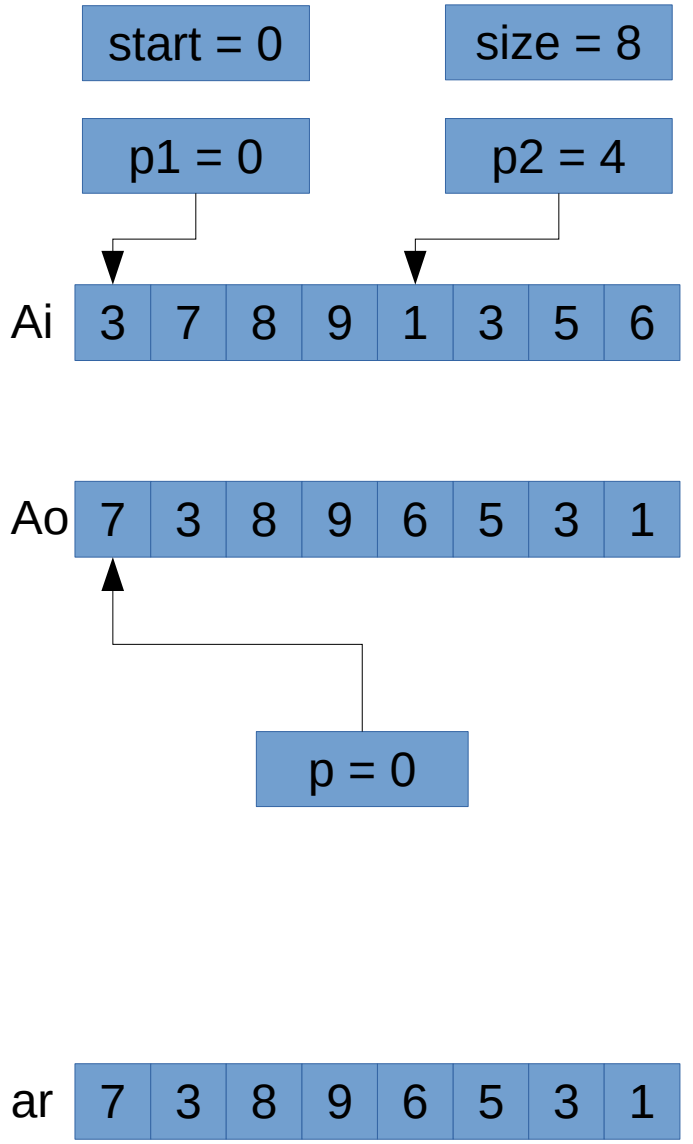
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start









ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



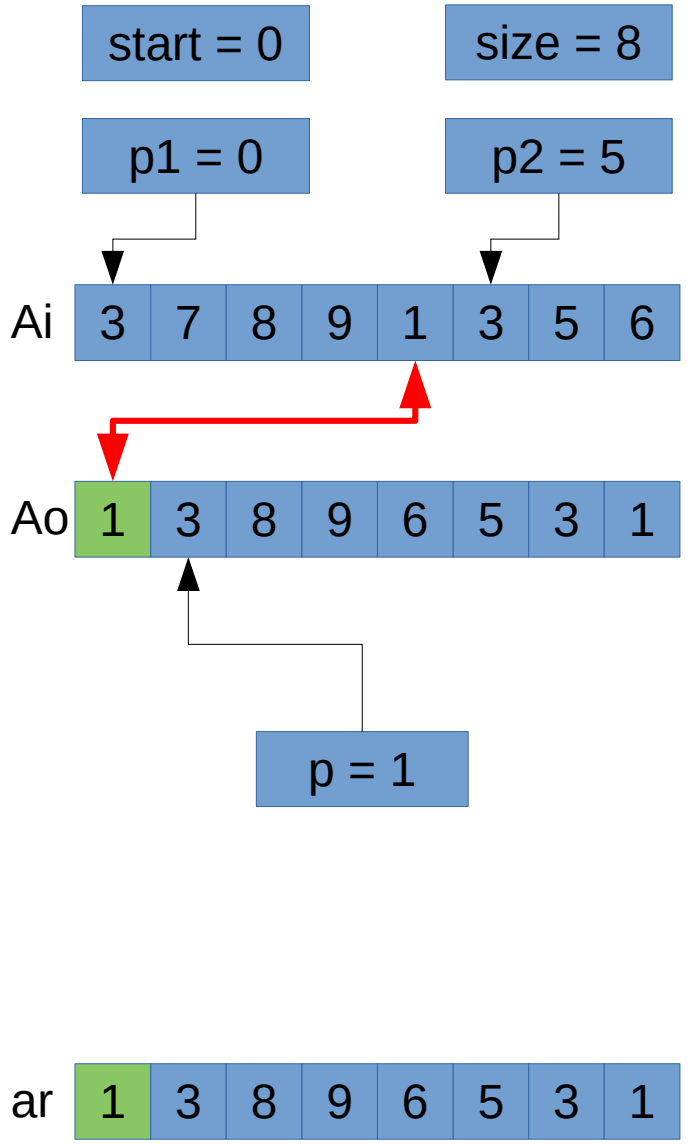
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):




ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



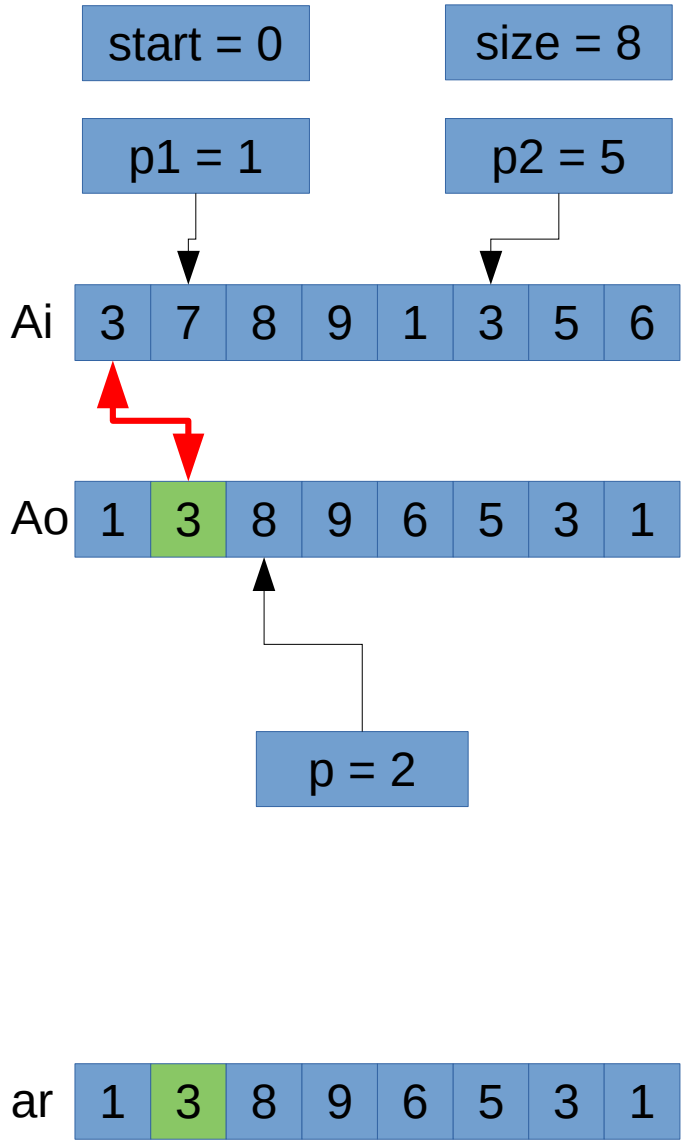
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):
      Ao[p] = Ai[p1]
      p1 += 1
    else:
      Ao[p] = Ai[p2]
      p2 += 1

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



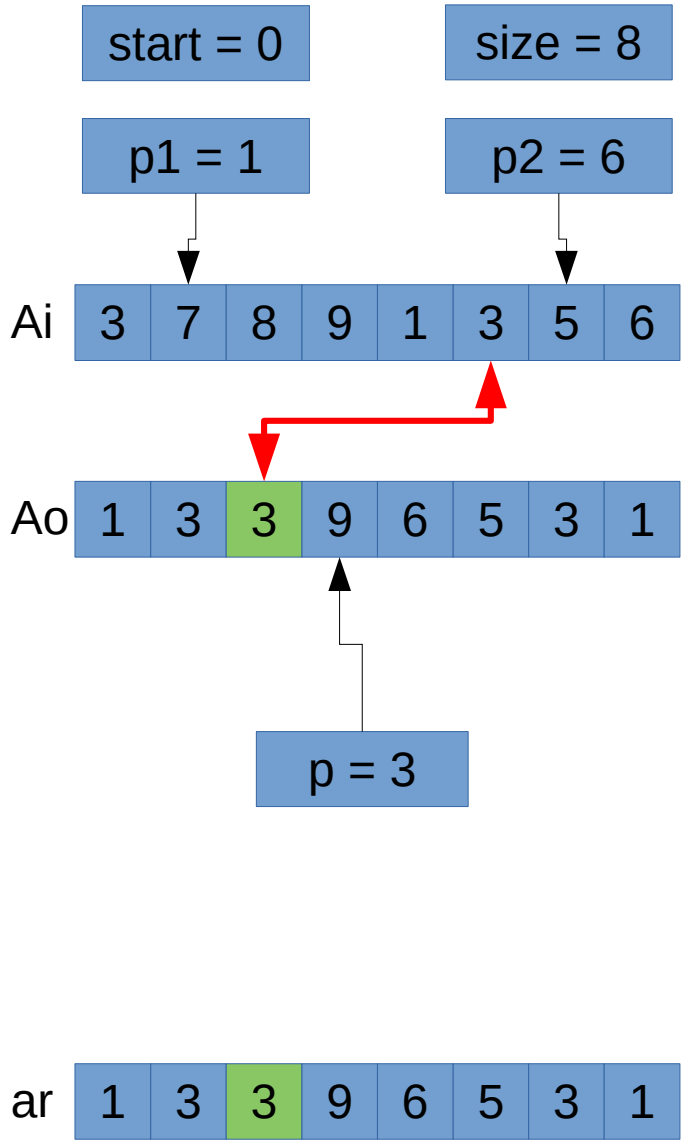
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):
      Ao[p] = Ai[p1]
      p1 += 1
    else:
      Ao[p] = Ai[p2]
      p2 += 1

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



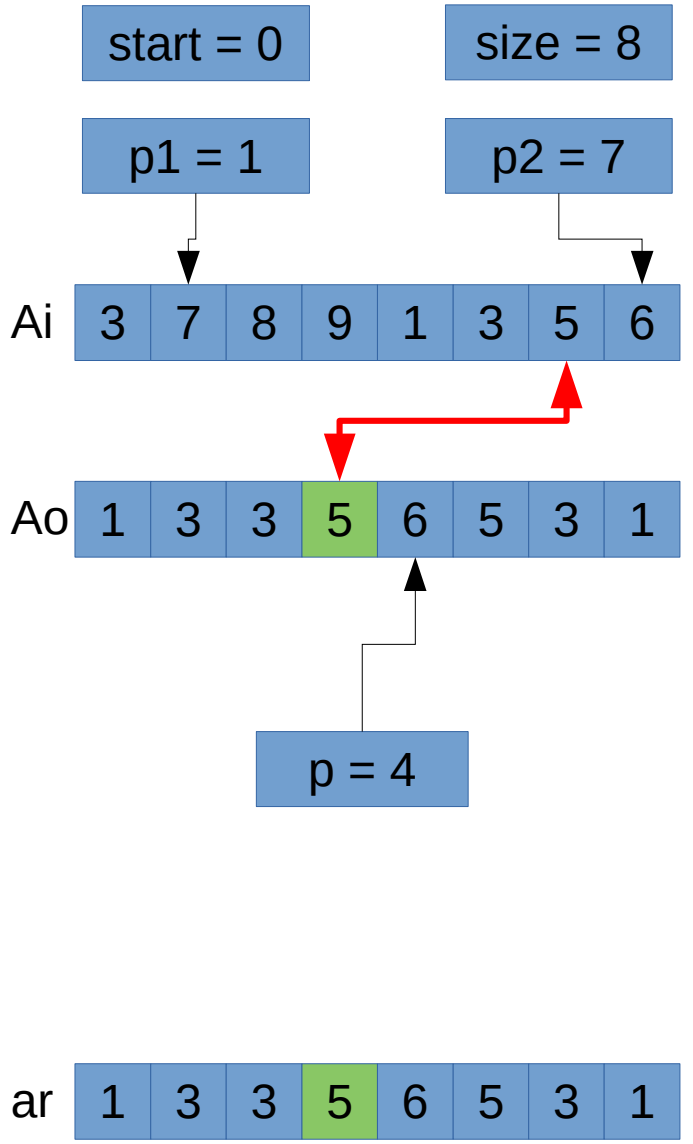
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):
      Ao[p] = Ai[p1]
      p1 += 1
    else:
      Ao[p] = Ai[p2]
      p2 += 1

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



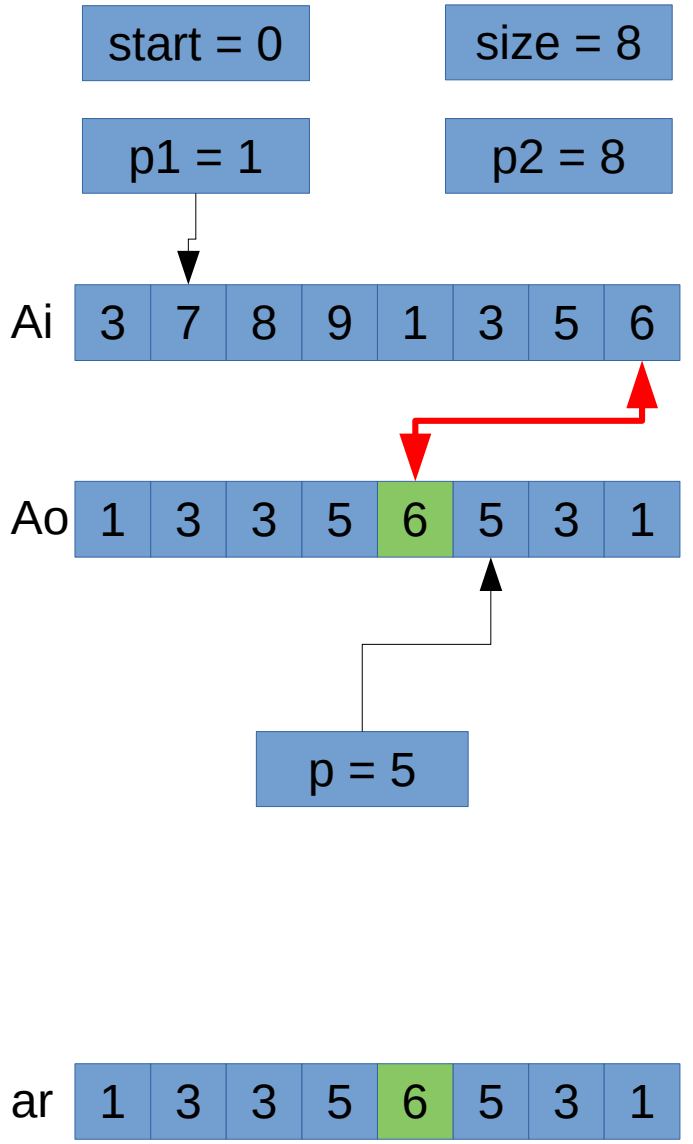
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):
      Ao[p] = Ai[p1]
      p1 += 1
    else:
      Ao[p] = Ai[p2]
      p2 += 1

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



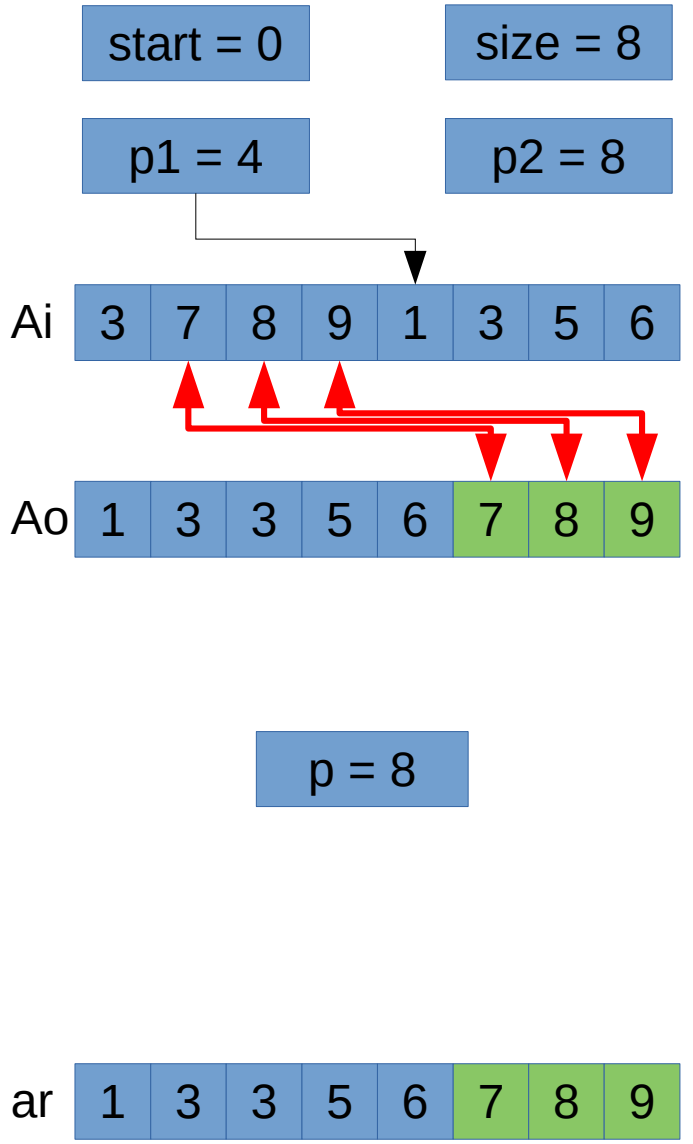
Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):
      Ao[p] = Ai[p1]
      p1 += 1
    else:
      Ao[p] = Ai[p2]
      p2 += 1

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



Сортировка слиянием

```
def sort_merge(Ai, Ao, start=0, size=-1):
  if size < 0: size = len(Ai)
  if size < 2: return
  sort_merge(Ao, Ai, start, size//2)
  sort_merge(Ao, Ai, start+size//2, size-size//2)

  p1 = start
  p2 = start + size//2
  p = start

  while p < start + size:
    if p2 == start + size \
    or (Ai[p1] <= Ai[p2] and p1 < start+size//2):
      Ao[p] = Ai[p1]
      p1 += 1
    else:
      Ao[p] = Ai[p2]
      p2 += 1

ar = [7, 3, 8, 9, 6, 5, 3, 1]
sort_merge(ar[:], ar)
```



Выбор алгоритма сортировки

- В общем случае следует использовать быструю сортировку.
- Для малых массивов, порядка десятков элементов, хорошо подходит сортировка вставками, так как она обладает очень низкими накладными расходами, является стабильной и адаптивной.
- Если требуется минимальное количество перестановок, даже в ущерб скорости, следует использовать сортировку выбором.
- Если требуется устойчивость и скорость, следует использовать сортировку слиянием.