Виды сортировок

1) Сортировка вставкой

Что мы делаем? А вот что. Мы ставит инлекс i на массив, начиная с первого элемента, а слево от него j. Key -это элемент массива i-ый по счету. Нам его надо запомнить, чтобы в дальнейшем перенести в нужное место. Если key становится меньше левого элемента мы начинаем его переносить, по очередно сравненивая элементы. Важный момент в момент заключается в том, что когда мы сравниваем, мы в правый элемент записываем значение левого, то есть пошагово сдвигаем весь ряд правее. Далее либо j становится равным -1, либо сравнения перестают работать, и мы записываем key в нужно место.

```
from random import randint
def insertion_sort(X):
    for i in range(1, len(X)):
        key = X[i]
        j = i-1
        while j >=0 and key < X[j] :
            X[j+1] = X[j]
            j -= 1
        X[j+1] = key
N = 10
X = [randint(1, 99) for item in range(N)]
print(*X)
insertion_sort(X)
print(*X)
```

2) Сортировка слиянием

Общий принцип состоит в том, что мы действуем по принципу «разделяй и властвуй». Алгоритм разбивает список на 2 части, каждую разбивает ещё на 2 и так далее, пока не останутся единичные элементы. В результате соседние элементы становятся сортированными парами. Потом эти пары объединяют и сортируют с другими парами. Процесс продолжается, пока не отсортируются все элементы.

```
from random import randint
def merge_sort(L):
    if len(L) < 2:
        return L
    else:
        middle = int(len(L) / 2)
        left = merge_sort(L[:middle])
        right = merge_sort(L[middle:])
        return merge(left, right)
```

```
def merge(left, right):
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    while i < len(left):
        result.append(left[i])
        i += 1
    while j < len(right):
        result.append(right[j])
        j += 1
    return result
```

```
N = 10
X = [randint(1, 99) for item in range(N)]
print(*X)
ANS = merge_sort(X)
print(*ANS)
```

3) Сортировка выбором

Разберемся в том, что происходит. Что мы делаем? Сначала мы находим наименьший элемент в массива, ставим его на место первого элемента, а первый на место наименьшего. Далее рассматриваем массив длины на 1 меньше. Делаем со вторым элементом аналогичную перестановку, что и с первом. Так действуем, пока все не отсортируем.

```
from random import randint
N = 10


def sel_sort(row):
    n = len(row)
    for i in range(n-1):
        m = i
        for j in range(i+1, n):
            if row[j] < row[m]:
                m = j
        row[i], row[m] = row[m], row[i]


X = [randint(1, 99) for item in range(N)]
print(*X)

sel_sort(X)
print(*X)
```

4) Сортировка пузырьком

Это самый интуитивно понятный способ сортировки. Мы берем первую пару чисел, сравниваем их, если нужно поменять их местами, меняем. Так действуем со следующей по порядку парой в массиве и так далее. Тем самым мы перемещаем самый большой элемент в конец. Далее действуем аналогично.

```
N = 10
X = [randint(1, 99) for item in range(N)]
print(*X)

for i in range(N-1):
    for j in range(N-i-1):
        if X[j] > X[j+1]:
            X[j], X[j+1] = X[j+1], X[j]

print(*X)
```

5) Быстрая сортировка quicksort

Алгоритм похож на сортировку слиянием, но он предполагает деление массива на две части, в одной из которых находятся элементы меньше определённого значения, в другой – больше или равные.

```
def QuickSort(A):
    if len(A) <= 1:
        return A
    else:
        q = int(len(A) / 2)
        L = []
        M = []
        R = []
        for elem in range (len(A)):
            if A[elem] < A[q]:
                L.append(A[elem])
            elif A[elem] > A[q]:
                R.append(A[elem])
            else:
                M.append(A[elem])

        return QuickSort(L) + M + QuickSort(R)
```

```
[ ] N = 5
    X = [1,6,5,4,3,7]
    print(*X)
    ANS = QuickSort(X)
    print(*ANS)
```

6) Пирамидальная сортировка

Пирамидальная сортировка – это алгоритм сортировки, основанный на структуре данных «куча». Вначале массив преобразуется в бинарную кучу, затем происходит постепенное удаление максимального элемента и перестроение кучи до тех пор, пока все элементы не будут отсортированы.

```
def heapify(sort_nums, heap_size, root):
    l = root
    left = (2 * root) + 1
    right = (2 * root) + 2
    if left < heap_size and sort_nums[left] > sort_nums[l]:
        l = left
    if right < heap_size and sort_nums[right] > sort_nums[l]:
        l = right
    if l != root:
        sort_nums[root], sort_nums[l] = sort_nums[l], sort_nums[root]
        heapify(sort_nums, heap_size, l)

def heap(sort_nums):
    size = len(sort_nums)
    for i in range(size, -1, -1):
        heapify(sort_nums, size, i)
    for i in range(size - 1, 0, -1):
        sort_nums[i], sort_nums[0] = sort_nums[0], sort_nums[i]
        heapify(sort_nums, i, 0)
nums = [3,2,54,0,6]
heap(nums)
print(nums)
```

Сложности алгоритмов:

1) Сложность алгоритма: O(n^2)
2) Сложность алгоритма: O(nlogn)
3) Сложность алгоритма: O(n^2)
4) Сложность алгоритма: O(n^2)
5) Сложность алгоритма: O(nlogn)
6) Сложность алгоритма: O(nlogn)