Funkcje pomocnicze (testujące poprawność algorytmów)

Funkcja generująca losowe dane testowe

In [1]:
```
import random
import math


def non_uniform_randrange(range_, count, k=.2, ints_only=True, round_=2):
    result = []
    limit = k / (math.log(count) or 1 if count else 1)

    while count:
        range_size = range_[1] - range_[0]
        while random.random() > limit and count:
            if ints_only:
                result.append(random.randint(*range_))
            else:
                random_num = round(random.random() * range_size + range_[0], round_)
                result.append(random_num)
            count -= 1

        avg = sum(range_) / 2
        delta = (avg - range_[0]) * k
        range_ = (int(range_[0] + delta - .5), int(range_[1] - delta + .5))

    return result


def generate_test_data(val_counts=(0, 50),        # A minimum and maximum number of values to sort that will be
                       range_=(-100, 100),        # A range which will be used to create a random list of values
                       ints_only=True,            # Set this flag to False in order to generate random non-integ
                       round_=2,                  # A number of floating point digits (works only with ints_only
                       uniform_distribution=True # If set to false, non-uniform values will be generated using
                       ):
    if uniform_distribution:
        if ints_only:
            random_lst = [random.randint(*range_) for _ in range(random.randint(*val_counts))]
        else:
            random_lst = []
            range_size = range_[1] - range_[0]
            for _ in range(random.randint(*val_counts)):
                # Shift a random value in order to be in range_ (not lower than the
                # lower bound and not greater than the greater bound)
                random_num = round(random.random() * range_size + range_[0], round_)
                random_lst.append(random_num)
    else:
        random_lst = non_uniform_randrange(range_, random.randint(*val_counts), random.random() / 3, ints_only,

    return random_lst
```

Funkcja testująca poprawność algorytmu

In [2]:
```
def test_sort(sorting_fn, *,
              samples=20,                 # A number of tests that will be performed
              val_counts=(0, 50),         # A minimum and maximum number of values to sort that will be generate
              range_=(-100, 100),         # A range which will be used to create a random list of values from th
              uniform_distribution=True, # If set to false, non-uniform values will be generated using a functi
              modifies_arr=True,          # Information whether an algorithm modifies the initial array or retur
              failed_only=False,          # Show only failed tests. Works only if no_results is set to False
              print_out_fn=None,          # An user-defined function to print additional information. Works only
              no_results=False,           # When set to True, no results will be printed (useful only for benchm
              ints_only=True,             # Set this flag to False in order to generate random non-integer numbe
              round_=2                    # A number of floating point digits (works only with ints_only set to
             ):
    passed = 0
    for i in range(samples):
        random_lst = generate_test_data(val_counts, range_, ints_only, round_, uniform_distribution)
        random_lst_before = random_lst[:] if modifies_arr else random_lst

        # Test if is correct
        expected = sorted(random_lst)
        if not modifies_arr:
            result = sorting_fn(random_lst)
        else:
            sorting_fn(random_lst)
            result = random_lst
        is_correct = result == expected
        passed += is_correct
```

```
        if not no_results:
            if not failed_only or (failed_only and not is_correct):
                print(f'TEST #{i+1}:')
                print(f'Before sorting: {random_lst_before}')
                print(f'After sorting: {result}')
                print(f'Expected result: {expected}')
                print(f'Test {"PASSED" if is_correct else "FAILED"}')
                print(f'Current passed-to-tested ratio: {passed}/{i+1}')
                if print_out_fn:
                    print(f'========== Additional results after sorting  ==========')
                    print_out_fn(random_lst)
                print()

    if not no_results:
        print(f'Sorting algorithm is {"correct" if passed == samples else "wrong"}')
        print(f'Passed tests in total: {passed}/{samples}')
```

Funkcja badająca wydajność algorytmów

In [3]:

```
import time
from decimal import Decimal
from colorama import Fore, Style

# ATTENTION: This function doesn't check if an algorithm, works properly. In order to test
# your algorithm's correctness, please use a function above
def compare_performance(sorting_functions, *,       # sorting_functions[<fn name>] = (<fn variable>, <globals di
                        samples=100,                # A number of tests that will be performed
                        val_counts=(0, 10_000),     # A minimum and maximum number of values to sort that will b
                        range_=(-10_000, 10_000),   # A range which will be used to create a random list of valu
                        progress_interval=10,       # A number of tests that will be performed between showing
                                                    # current sorting times (temporary results). When set to 0 r
                        uniform_distribution=True,  # If set to false, non-uniform values will be generated usir
                        ints_only=True              # Set this flag to False in order to generate random non-int
                       ):
    times = dict.fromkeys([fn[0] for fn in sorting_functions], Decimal(0))
    max_fn_name_len = max(len(entry[0]) for entry in sorting_functions)
    inf = float('inf')

    for i in range(1, samples + 1):
        random_lst = generate_test_data(val_counts, range_, ints_only, 10, uniform_distribution)

        for name, (fn, globals_) in sorting_functions:
            if times[name] == inf: continue
            # Overwrite current global values to make an algorithm able to work (use a proper funciton)
            for obj_name, obj in globals_.items():
                globals()[obj_name] = obj
            # Perform a test
            input_lst = random_lst[:]
            try:
                start_time = time.time()
                fn(input_lst)
                end_time = time.time()
            except RecursionError:
                times[name] = inf
            else:
                times[name] += Decimal(end_time - start_time)

        if progress_interval > 0 and not i % progress_interval:
            print(f"===== Results after {i} tests: =====")
            names = sorted(times, key=lambda name: times[name])
            fastest_time = times[names[0]]
            for name in names:
                total_time = times[name]
                if total_time < inf:
                    print(f"{name.ljust(max_fn_name_len)}  Total (in seconds): {total_time:<8.4f}  Average: {to
                    if fastest_time and name != names[0]:
                        ratio = total_time / fastest_time
                        print(f"\t({ratio:.2f}x slower)")
                    else:
                        print()
                else:
                    print(f"{name.ljust(max_fn_name_len)}   {Fore.RED}RECURSION ERROR{Style.RESET_ALL}")
            print()

    names = sorted(times, key=lambda name: times[name])
    fastest_time = times[names[0]]
    print("===== Final results: =====")
    print(f"Tests performed: {samples}")
    print("\nTotal times (in seconds):")
    for name in names:
        total_time = times[name]
        if total_time < inf:
            print(f"\t{name.ljust(max_fn_name_len)}:   {total_time:.7f}")
        else:
```

```
            print(f"\t{name.ljust(max_fn_name_len)}:   {Fore.RED}RECURSION ERROR{Style.RESET_ALL}")
    print("\nAverage times (in seconds):")
    for name in names:
        total_time = times[name]
        if total_time < inf:
            print(f"\t{name.ljust(max_fn_name_len)}:   {total_time/samples:<8.7f}", end='')
            if fastest_time and name != names[0]:
                ratio = total_time / fastest_time
                print(f"\t({ratio:.2f}x slower)")
            else:
                print()
        else:
            print(f"\t{name.ljust(max_fn_name_len)}:   {Fore.RED}RECURSION ERROR{Style.RESET_ALL}")
```

Słownik służący do zapisania funkcji, w celu porównania wydajności

In [4]:
```
sorting_functions = {}  # Please store sorting functions using function's name as a string
                        # key and a (function variable, your specified global variables
                        # such as helper functions that re used by a sorting function)
                        # pairs as a value
```

Algorytmy o złożoności $O(n^2)$

Zazwyczaj wolne, ale łatwe w implementacji i wytłumaczeniu algorytmy. Pozwalają na sortowanie dowolnych danych o dowolnym rozkładzie z dowolnego zakresu.

Co ciekawe, algorytmy te są często szybsze od algorytmów szybkiego sortowania, jeżeli do posortowania mamy mały zbiór danych (kilkanaście elementów).

Bubble Sort

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n^2)$

Najlepszy przypadek

$O(n^2)$

Złożoność pamięciowa

Najgorszy przypadek

$O(1)$

Najlepszy przypadek

$O(1)$

Stabilność

Stabilny

Ponieważ wykonujemy zamiany sąsiednych wartości tylko wtedy, gdy jedna jest większa od drugiej. Kolejność żadnych dwóch elementów o tych samych kluczach, według których sortujemy daną sekwencję, nie zostanie zmieniona, ponieważ zamieniamy tylko te elementy, których klucze są różne, a klucz tego, który znajduje się wcześniej, jest większy od klucza tego, który znajduje się później (przy sortowaniu w kolejności malejącej, postępujemy odwrotnie)

Czy sortowanie odbywa się w miejscu?*

*** Powiemy, że sortowanie odbywa się w miejscu, jeżeli podczas procesu sortowania jest wykorzystywana stała ilość pamięci, niezależna od wielkości sortowanej struktury (tzn. elementy nie są tymczasowo przepisywane do struktur pomocniczych). Zazwyczaj** sortowanie w miejscu można odróżnić po tym, że po procesie sortowania zmodyfikowana zostaje sortowana struktura, a w przypadku sortowania, które nie sortuje w miejscu, zwrócona jest nowa struktura (tej samej wielkości, co wejściowa), zawierająca posortowane dane, a wejściowa struktura pozostaje nienaruszona.**

**** Zazwyczaj, ponieważ możemy napisać funkcję, która "przepisze" posortowane dane z używanej tymczasowo struktury do sortowanej struktury, więc wejściowa struktura zostanie zmodyfikowana (patrz implementacja Merge Sort #2).**

Tak

Implementacja algorytmu

In [5]:
```
def bubble_sort(arr):
    for i in range(len(arr)-1):
        for j in range(len(arr)-1-i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [6]:
```
sorting_functions['Bubble Sort'] = (bubble_sort, {})
```

Kilka testów

In [7]:
```
test_sort(bubble_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Selection Sort

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n^2)$

Najlepszy przypadek

$O(n^2)$

Złożoność pamięciowa

Najgorszy przypadek

$O(1)$

Najlepszy przypadek

$O(1)$

Stabilność

Niestabilny

Wynika to stąd, że w każdym przebiegu wewnętrznej pętli, poszukujemy elementu, którgo klucz jest w danym momencie najmniejszy (największy - przy sortowaniu w kolejności nierosnącej) spośród kluczy elementów, znajdujących się w sprawdzanej w drugiej pętli elementów. Ponieważ sprawdzamy tylko te elementy sekwencji, które znajdują się dalej od elementu, na którym w danym momencie "zatrzymała" się zewnętrzna pętla oraz poszukujemy elementu najmniejszego spośród sprawdzanych, istnieje możliwość, że wskutek zamiany elementu najmniejszego z elementem z zewnętrznej pętli, element z zewnętrznej pętli znajdzie się dalej niż inny element, który posiada ten sam klucz. Wynika stąd, że wzajemna odległość elementów, które posiadają te same klucze, może zostać (i zazwyczaj jest) zmieniona.

Przykład:

Selection sort

```
def selection_sort(arr):
    for i in range(len(arr)-1):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]: min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

| 7 | 2 | 5 | 1 | 2 | 4 | 8 | 1 | 2 |

Czy sortowanie odbywa się w miejscu?

Tak

Implementacja algorytmu

In [8]:
```
def selection_sort(arr):
    for i in range(len(arr)-1):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]: min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [9]:
```
sorting_functions['Selection Sort'] = (selection_sort, {})
```

Kilka testów

In [10]:
```
test_sort(selection_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Insertion Sort

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n^2)$

Najlepszy przypadek

$O(n)$

Złożoność pamięciowa

Najgorszy przypadek

$O(1)$

Najlepszy przypadek

$O(1)$

Stabilność

Stabilny

Wynika to stąd, że dany element, który jest mniejszy od poprzedniego (przy sortowaniu w kolejności malejącej - większy od poprzedniego) przesuwamy w lewo, dopóki jest on mniejszy (większy) od elementu, który go w danym momencie poprzedza. Dlatego, jeżeli zdarzy się kilka elementów, które mają ten sam klucz, według którego przeprowadzamy sortowanie, przesuwany element wstawiony zostanie za ostatnim elementem, którego klucz jest taki sam jak klucz elementu, który przemieszczamy, więc nie istnieje możliwość wzajemnej zmiany kolejności elementów, które posiadają ten sam klucz.

Czy sortowanie odbywa się w miejscu?

Tak

Implementacja algorytmu

In [11]:
```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i-1
        temp = arr[i]

        while j >= 0 and temp < arr[j]:
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [12]:
```
sorting_functions['Insertion Sort'] = (insertion_sort, {})
```

Kilka testów

In [13]:
```
test_sort(insertion_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Algorytmy o złożoności $O(n \cdot log(n))$

Najszybsze uniwersalne (do sortowania danych dowolnego typu o dowolnym rozkładzie z dowolnego zakresu).

Merge Sort

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n \cdot log(n))$

Najlepszy przypadek

$O(n \cdot log(n))$

Złożoność pamięciowa

Najgorszy przypadek

$O(n)$

Najlepszy przypadek

$O(n)$

Stabilność

Stabilny

Ponieważ zawsze iterujemy przez elementy tablicy (lub jej fragmentów) w kolejności od lewej strony do prawej, a zamiana kolejności elementów występuje wyłącznie podczas łączenia dwóch fragmentów tablic w jedną posortowaną (funkcja $merge$) i polega na tym, że najpierw do tablicy wynikowej przepisujemy ten element z jednej z tablic, który jest nie większy niż ten z drugiej z tablic, nie istnieje możliwość wzajemnej zamiany kolejności elementów o tych samych kluczach (jeżeli oba porównywane elementy z obu tablic mają taki sam klucz, w funkcji złączającej dwie tablice zawsze podjęta zostanie ta sama decyzja, dotycząca wyboru pierwszego przepisanego elementu, np. pierwszy będzie ten z pierwszej z tablic, a ponieważ zawsze będzie obowiązywała taka sama zasada, nie ma możliwości zamiany kolejności tych elementów względem początkowej).

Czy sortowanie odbywa się w miejscu?

Nie

Implementacja algorytmu #1

(zwraca nową sekwencję) (NAJGORSZA implementacja - tworzy kopie fragmentów tablicy)

In [14]:
```
def merge_sort(arr):
    if len(arr) <= 1: return arr

    mid_idx = len(arr) // 2
    # Sort recursively a left and a right part of an array
    left = merge_sort(arr[:mid_idx])
    right = merge_sort(arr[mid_idx:])

    # Return a merged array
    return _merged(left, right)


def _merged(left, right):
    result = []
    left_idx = right_idx = 0

    # Rewrite values in a right (non-descending) order
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] < right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    # Rewrite remaining values if there are some in one of the parts
    for i in range(left_idx, len(left)):
        result.append(left[i])

    for i in range(right_idx, len(right)):
        result.append(right[i])

    return result
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [15]:
```
sorting_functions['Merge Sort (#1)'] = (merge_sort, {
    '_merged': _merged
})
```

Kilka testów

In [16]:
```
test_sort(merge_sort, samples=100, modifies_arr=False, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #2

(modyfikuje sortowaną sekwencję) (o dziwo nawet wolniejsza implementacja niż powyższa - przekazuje indeksy a nie tablice)

In [17]:
```
def merge_sort(arr):
    # _sort takes left_idx inclusive and right_idx exclusive
    # (same as Python's range function does)
    def _sort(left_idx, right_idx):
        # If there are less than 2 elements in a part, do nothing
        if right_idx - left_idx < 2: return

        mid_idx = (left_idx + right_idx) // 2
        _sort(left_idx, mid_idx)
        _sort(mid_idx, right_idx)
        merge(left_idx, mid_idx, right_idx)

    def merge(i, j, right_end):
        left_begin = i
        left_end = j
        merged = []

        while i < left_end and j < right_end:
            if arr[i] < arr[j]:
                merged.append(arr[i])
                i += 1
            else:
                merged.append(arr[j])
                j += 1

        for i in range(i, left_end):
            merged.append(arr[i])

        for j in range(j, right_end):
            merged.append(arr[j])

        idx = left_begin
        for val in merged:
            arr[idx] = val
            idx += 1

    _sort(0, len(arr))
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [18]:
```
sorting_functions['Merge Sort (#2)'] = (merge_sort, {})
```

Kilka testów

In [19]:
```
test_sort(merge_sort, samples=100, modifies_arr=True, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #3

(zwraca nową sekwencję) (najlepsza implementacja - nie trzeba alokować co chwilę nowych tablic oraz niepotrzebnie kilka razy przepisywać wartości) (wersja bez rekurencji)

In [20]:
```
def merge_sort(arr):
    temp_arr = [None] * len(arr)
    step = 1

    while step < len(arr):
        for left_idx in range(0, len(arr)-step, 2 * step):
            mid_idx = left_idx + step
            right_idx = mid_idx + step

            if right_idx > len(arr): right_idx = len(arr)

            # Perform merging
            i = left_idx
            j = mid_idx
            k = left_idx

            while i < mid_idx and j < right_idx:
                if arr[i] <= arr[j]:
                    temp_arr[k] = arr[i]
                    i += 1
                else:
                    temp_arr[k] = arr[j]
                    j += 1
                k += 1
```

```
            # Rewrite the remaining values
            for i in range(i, mid_idx):
                temp_arr[k] = arr[i]
                k += 1

            for j in range(j, right_idx):
                temp_arr[k] = arr[j]
                k += 1

        # If there are still some values remaining, rewrite them
        for k in range(k, len(arr)):
            temp_arr[k] = arr[k]

        # Swap arrays
        arr, temp_arr = temp_arr, arr
        step *= 2

    return arr
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [21]:
```
sorting_functions['Merge Sort (#3)'] = (merge_sort, {})
```

Kilka testów

In [22]:
```
test_sort(merge_sort, samples=100, modifies_arr=False, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Heap Sort

Heap Sort polega na zbudowaniu KOMPLETNEGO drzewa binarnego z kolejnych wartości, jakie znajdują się w sortowanej sekwencji (tablicy), a dokładniej struktury, która nazywa się Max Heap, a następnie odczycie kolejnych największych wartości z pozostałej części struktury i przenoszeniu wartości już posortowanych na koniec sekwencji (tablicy). Budowa struktury jest szybka, a jej złożoność czasowa wynosi $O(n)$, natomiast sam odczyt wartości z "Maksymalnego Kopca" (Max Heap) wymaga już wykonania $O(n \cdot log(n))$ operacji. Wynika to stąd, że za każdym razem ściągamy wartość z korzenia kopca (kompletnego drzewa binarnego), tym samym "psując" kopiec. W miejsce usuniętej wartości wstawiamy dowolną wartość (najlepiej ostatni z liści), a następnie naprawiamy drzewo w czasie $O(log(n))$.

Dokładniejsze wyjaśnienie sposobu działania algorytmu oraz funkcjonowania samej struktury, jaką jest Heap, znajduje się poniżej:

Binary Heap: https://www.youtube.com/watch?v=g9YK6sftDi0

Heap Sort: https://www.youtube.com/watch?v=k72DtCnY4MU

Warto jeszcze nadmienić, że Binary Heap nie wymaga tworzenia drzewa binarnego, ani korzystania z jakichkolwiek pomocniczych struktur. Wszystkie operacje można przeprowadzić bezpośrednio na tablicy, jedynie korzystając z tablicy jako odpowiedniej reprezentacji wartości zapisanych w "kopcu". Mimo to, Heap Sort zaliczany jest do najwolniejszych algorytmów sortowania (jest najwolniejszy z omawianych algorytmów o złożoności $O(n \cdot log(n))$ - patrz testy na końcu pliku)

Grafika z Cormena:

Chapter 6 Heapsort

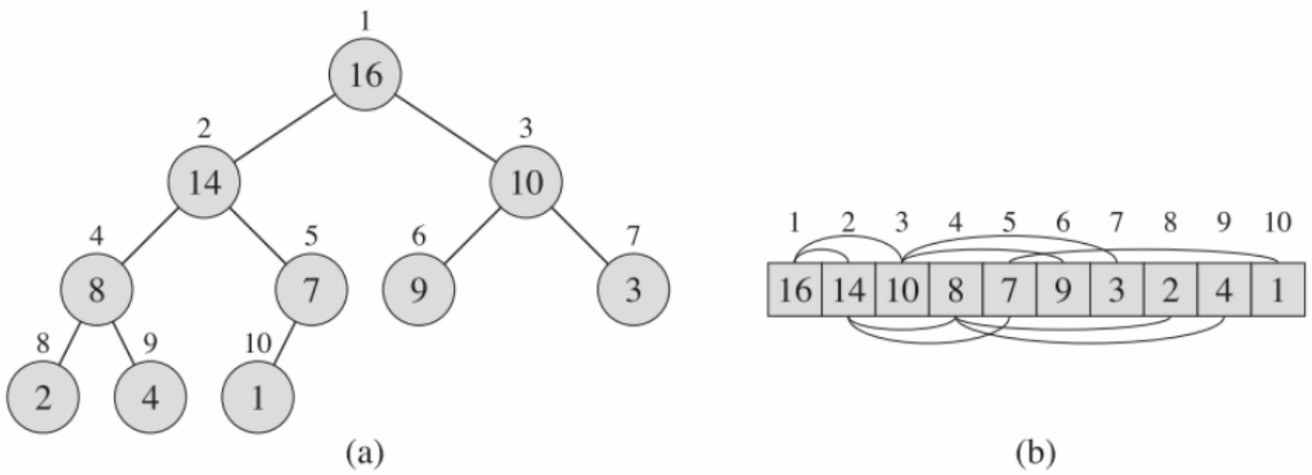

Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents

are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n \cdot log(n))$

Najlepszy przypadek

$O(n \cdot log(n))$

Złożoność pamięciowa

Najgorszy przypadek

$O(1)$

Najlepszy przypadek

$O(1)$

Stabilność

Niestabilny

Czy sortowanie odbywa się w miejscu?

Tak

Implementacja algorytmu #1

(z użyciem klasy do reprezentacji struktury Max Heap)

In [23]:

```
class MaxHeap:
    def __init__(self, values=None):
        self.heap = values  # We don't make a copy of these values
        self.build_heap()

    @property
    def heap_size(self):
        return len(self.heap)

    @staticmethod
    def left_child_idx(curr_idx):
        return curr_idx * 2 + 1

    @staticmethod
    def right_child_idx(curr_idx):
        return curr_idx * 2 + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

    def max_heapify(self, curr_idx, end_idx):  # O(log n)
        # Loop till the current node has a child larger than itself
        # We assume that when we enter a node which both children are
        # smaller than this node, a subtree which a current node is a
        # root of must fulfill a max-heap property.
        while True:
            l = self.left_child_idx(curr_idx)
            r = self.right_child_idx(curr_idx)
            largest_idx = curr_idx

            if l < end_idx:
                if self.heap[l] > self.heap[curr_idx]:
                    largest_idx = l
                if r < end_idx and self.heap[r] > self.heap[largest_idx]:
                    largest_idx = r

            if largest_idx != curr_idx:
                self.swap(curr_idx, largest_idx)
                curr_idx = largest_idx
            else:
                break

    def build_heap(self):  # O (n)
        for i in range((self.heap_size - 1) // 2, -1, -1):
            self.max_heapify(i, self.heap_size)
```

```python
def heap_sort(arr):
    max_heap = MaxHeap(arr)
    # Swap the currently greatest value (a heap's root) with a value
    # stored at index i in an array representing a heap and fix
    # a heap after doing such a change.
    for i in range(len(arr)-1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        # Fix a heap
        max_heap.max_heapify(0, i)
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [24]:
```python
sorting_functions['Heap Sort (#1)'] = (heap_sort, {'MaxHeap': MaxHeap})
```

Funkcja do wizualizacji drzew (tu: struktury Max Heap)

In [25]:
```python
def print_heap(values):
    if values:
        just = 0
        data = []

        limit = 1
        values_row = []
        branches_row = []
        prev_nodes = 0

        for i in range(1, len(values) + 1):
            curr_nodes = i - prev_nodes
            val_str = str(values[i-1])
            just = max(just, len(val_str))
            values_row.append(val_str)
            right_child_idx = 2 * i
            left_child_idx = right_child_idx - 1
            if left_child_idx < len(values):
                branches_row.append('/')
            if right_child_idx < len(values):
                branches_row.append('\\')

            if curr_nodes == limit:
                prev_nodes = i
                limit *= 2
                data.append([values_row, branches_row])
                values_row = []
                branches_row = []

        if values_row:
            data.append([values_row, branches_row])

        begin_sep = sep = 3 if just % 2 else 2
        data_iter = iter(data[::-1])
        result = [''] * (len(data) * 2 - 1)
        result[-1] = (' ' * sep).join(val.center(just) for val in next(data_iter)[0])

        # Format the tree string
        for i, (values, branches) in enumerate(data_iter):
            mul = 2 * i + 1
            # Values
            indent = (2 ** (i + 1) - 1) * (just + begin_sep) // 2
            sep = 2 * sep + just
            result[-(mul + 2)] = f"{' ' * indent}{(' ' * sep).join(val.center(just) for val in values)}"
            # Branches
            branch_indent = (3 * indent + just) // 4
            branches_row = []
            d_indent = indent - branch_indent
            branches_sep = ' ' * (2 * (d_indent - 1) + just)
            for i in range(0, len(branches), 2):
                branches_row.append(f"{branches[i]}{branches_sep}{branches[i + 1] if i + 1 < len(branches) else
            result[-(mul + 1)] = f"{' ' * branch_indent}{(' ' * (sep - 2 * d_indent)).join(branches_row)}"

        print('\n'.join(result))
    else:
        print('')
```

Kilka testów

In [26]:
```python
test_sort(heap_sort, samples=3, val_counts=(0, 31), print_out_fn=print_heap, ints_only=False, round_=1)
```

```
TEST #1:
Before sorting: [-80.6, -72.5, -32.8, 53.5]
After sorting: [-80.6, -72.5, -32.8, 53.5]
Expected result: [-80.6, -72.5, -32.8, 53.5]
Test PASSED
Current passed-to-tested ratio: 1/1
========== Additional results after sorting  ==========
            -80.6
          /       \
    -72.5           -32.8
    /
 53.5

TEST #2:
Before sorting: [-62.0, 97.4, 0.8, 70.1, -63.3, -94.7, 98.6, -22.1, 90.8]
After sorting: [-94.7, -63.3, -62.0, -22.1, 0.8, 70.1, 90.8, 97.4, 98.6]
Expected result: [-94.7, -63.3, -62.0, -22.1, 0.8, 70.1, 90.8, 97.4, 98.6]
Test PASSED
Current passed-to-tested ratio: 2/2
========== Additional results after sorting  ==========
                              -94.7
                     /               \
            -63.3                           -62.0
          /       \                       /       \
    -22.1            0.8             70.1            90.8
```

```
    /   \
 97.4    98.6

TEST #3:
Before sorting: [53.0, 10.9, 38.5, 23.4, 85.8, 94.3, -14.1, -55.0, 34.7, -29.5, -41.3, 11.9]
After sorting: [-55.0, -41.3, -29.5, -14.1, 10.9, 11.9, 23.4, 34.7, 38.5, 53.0, 85.8, 94.3]
Expected result: [-55.0, -41.3, -29.5, -14.1, 10.9, 11.9, 23.4, 34.7, 38.5, 53.0, 85.8, 94.3]
Test PASSED
Current passed-to-tested ratio: 3/3
========== Additional results after sorting  ==========
                            -55.0
                      /               \
            -41.3                           -29.5
          /       \                       /       \
    -14.1            10.9            11.9            23.4
    /   \           /   \           /
 34.7    38.5    53.0    85.8    94.3

Sorting algorithm is correct
Passed tests in total: 3/3
```

Implementacja algorytmu #2

(jedynie z użyciem funkcji)

In [27]:

```python
_left = lambda i: 2 * i + 1
_right = lambda i: 2 * i + 2


def max_heapify(arr, curr_idx, end_idx):
    # Loop till the current node has a child larger than itself
    # We assume that when we enter a node which both children are
    # smaller than this node, a subtree which a current node is a
    # root of must fulfill a max-heap property.
    while True:
        i = _left(curr_idx)
        j = _right(curr_idx)
        k = curr_idx

        if i < end_idx:
            if arr[i] > arr[k]:
                k = i
            if j < end_idx and arr[j] > arr[k]:
                k = j

        if k == curr_idx: return
        # Swap the current with the largest child
        arr[curr_idx], arr[k] = arr[k], arr[curr_idx]
        curr_idx = k


def build_heap(arr):
    for i in range((len(arr) - 1) // 2, -1, -1):
        max_heapify(arr, i, len(arr))


def heap_sort(arr):
    build_heap(arr)
    # Swap the currently greatest value (a heap's root) with a value
    # stored on index i in an array representing a heap and fix
    # a heap after doing such a change.
    for i in range(len(arr)-1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        # Fix a heap
        max_heapify(arr, 0, i)
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [28]:

```python
sorting_functions['Heap Sort (#2)'] = (heap_sort, {
    'max_heapify': max_heapify,
    'build_heap': build_heap,
    '_left': _left,
    '_right': _right
})
```

Kilka testów

In [111…

```python
test_sort(heap_sort, samples=3, val_counts=[0, 20], print_out_fn=print_heap, ints_only=False, round_=2)
```

```
TEST #1:
Before sorting: [42.89, -30.58, 28.58, -9.08, -70.52, 78.16, -11.86, -84.55, -80.27, -71.1, -10.1, 24.44, -84.5
8]
After sorting: [-84.58, -84.55, -80.27, -71.1, -70.52, -30.58, -11.86, -10.1, -9.08, 24.44, 28.58, 42.89, 78.1
6]
Expected result: [-84.58, -84.55, -80.27, -71.1, -70.52, -30.58, -11.86, -10.1, -9.08, 24.44, 28.58, 42.89, 78.
16]
```

```
Test PASSED
Current passed-to-tested ratio: 1/1
========== Additional results after sorting  ==========
                            -84.58
                      /                \
            -84.55                          -80.27
          /        \                      /        \
    -71.1           -70.52          -30.58          -11.86
    /    \          /    \          /    \
-10.1   -9.08   24.44   28.58   42.89   78.16

TEST #2:
Before sorting: [-27.82, 61.6, 25.91, 92.01, -32.27, 13.23, 49.75, 37.86]
After sorting: [-32.27, -27.82, 13.23, 25.91, 37.86, 49.75, 61.6, 92.01]
Expected result: [-32.27, -27.82, 13.23, 25.91, 37.86, 49.75, 61.6, 92.01]
Test PASSED
Current passed-to-tested ratio: 2/2
========== Additional results after sorting  ==========
                            -32.27
                      /                \
            -27.82                          13.23
          /        \                      /        \
    25.91           37.86           49.75            61.6
    /
92.01

TEST #3:
Before sorting: [83.24, 82.39, 25.51, 39.09, -50.35, 29.96, 8.44, -20.73, -47.51, -98.6, -72.39, 50.7, -46.27]
After sorting: [-98.6, -72.39, -50.35, -47.51, -46.27, -20.73, 8.44, 25.51, 29.96, 39.09, 50.7, 82.39, 83.24]
Expected result: [-98.6, -72.39, -50.35, -47.51, -46.27, -20.73, 8.44, 25.51, 29.96, 39.09, 50.7, 82.39, 83.24]
Test PASSED
Current passed-to-tested ratio: 3/3
========== Additional results after sorting  ==========
                            -98.6
                      /                \
            -72.39                          -50.35
          /        \                      /        \
    -47.51          -46.27          -20.73           8.44
    /    \          /    \          /    \
25.51   29.96   39.09    50.7   82.39   83.24

Sorting algorithm is correct
Passed tests in total: 3/3
```

Quick Sort

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n^2)$

lub: $O(n \cdot log(n))$ przy skorzystaniu z wyboru pivota przy pomocy algorytmu Mediana Median (zwanego dla $k = 5$, gdzie k - liczebność grup, na które dzielimy tablicę, "Magiczne Piątki")

UWAGA: W tym przypadku można zauważyć, że fakt iż algorytm z użyciem Magicznych Piątek ma niższą najgorszą złożoność czasową i taką zamą złożoność oczekiwaną, nie świadczy wcale o jego szybszym działaniu. W większości przypadków, gdy otrzymujemy dane, które nie są posortowane, zauważymy spadek wydajności, ponieważ zawsze konieczne jest wyznaczenie dodatkowo mediany median, która w danym przypadku nic nie poprawia.

Najlepszy przypadek

$O(n \cdot log(n))$

Złożoność pamięciowa

Najgorszy przypadek

$O(n)$

Jedynie dla niezoptymalizowanych pod kątem pamięci implementacji (są to 3 pierwsze implementacje niżej, kolejne już tylko wymagają $O(log(n))$ pamięci)

Najlepszy przypadek

O(log(n))

Zazwyczaj ten algorytm sortowania ma taką złożoność pamięciową. Można go zaimplementować w taki sposób, że $O(lon(n))$ jest jednocześnie jego średnią oraz najgorszą złożonością pamięciową (patrz implementacja #4 i kolejne)

Stabilność

Niestabilny

Czy sortowanie odbywa się w miejscu?

Tak (przy założeniu, że pamięć, jaką zajmuje stos rekurencyjny, wliczamy do złożoności)

Głosy są podzielone. Niby nie rezerwujemy pamięci w sposób jawny w celu tymczasowego przechowywania częściowo posortowanych danych, ale jednak złożoność pamięciowa algorytmu nie wynosi $O(1)$, ponieważ zmienne przechowywane na stosie rekurencyjnym również zajmują pamięć, więc stwierdzenie, czy sortowanie odbywa się w miejscu, zależy od tego, jak na to spojrzymy.

Implementacja algorytmu #1

(Wersja rekurencyjna ze sztywno ustalonym pivotem - my wybieramy ostatni element tablicy - z wykorzystaniem algorytmu podziału zaproponowanego przez Lomuto)

In [30]:
```
def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)


def _quick_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        pivot_position = partition(arr, left_idx, right_idx)
        _quick_sort(arr, left_idx, pivot_position - 1)
        left_idx = pivot_position + 1  # I removed a tailing recursion

''' Bez usuwania rekursji ogonowej, algorytm wygląda następująco: '''
#     if left_idx < right_idx:
#         pivot_position = partition(arr, left_idx, right_idx)
#         _quick_sort(arr, left_idx, pivot_position - 1)
#         _quick_sort(arr, pivot_position + 1, right_idx)


def partition(arr, left_idx, right_idx):
    pivot = arr[right_idx]

    # Partition an array into 2 subarrays of elements lower than
    # pivot and of elements greater than a pivot
    i = left_idx
    for j in range(left_idx, right_idx):
        if arr[j] < pivot:
            swap(arr, i, j)
            i += 1

    # Place a pivot element on its destination index
    swap(arr, i, right_idx)

    return i  # Return a pivot position after the last swap


def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [31]:
```
sorting_functions['Quick Sort (#1 - Lomuto parition)'] = (quick_sort, {
    '_quick_sort': _quick_sort,
    'partition': partition,
    'swap': swap
})
```

Kilka testów

In [32]:
```
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #2

(Wersja rekurencyjna ze sztywno ustalonym pivotem - korzysta z algorytmu Hoare'a podziału tablicy)

UWAGA:

O ile w implementacji funkcji partition zaproponowanej przez Lomuto mamy pewność, że po podziale, pivot znajduje się na swojej końcowej pozycji, tj. wyznaczyliśmy jego położenie końcowe w posortowanej tablicy, więc w kolejnych iteracjach sortujemy pozostałe części tablicy z pominięciem pivota, tak w implementacji funkcji podziału Hoare'a, wiemy jedynie, że pivot znajduje się pod koniec na zwróconej przez funkcję partition pozycji i należy do części tablicy, składającej się z elementów od niego mniejszych i mu równych. Nie oznacza to, że pivot znalazł się na swojej końcowej pozycji, więc musimy posortować lewą część tablicy wraz z pivotem oraz prawą (bez pivota).

In [33]:
```
def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)
```

```python
def _quick_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        pivot_position = partition(arr, left_idx, right_idx)
        _quick_sort(arr, left_idx, pivot_position),  '<- zamiast pivot_position + 1 jest samo pivot_position'
        left_idx = pivot_position + 1  # I removed a tailing recursion

''' Funkcja partition jest zmieniona: '''
def partition(arr, left_idx, right_idx):
    pivot = arr[left_idx]

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            swap(arr, i, j)
        else:
            return j  # Return a pivot position after the last swap
''' Koniec funkcji partition '''

def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [34]:
```python
sorting_functions['Quick Sort (#2 - Hoare\'s partition)'] = (quick_sort, {
    '_quick_sort': _quick_sort,
    'partition': partition,
    'swap': swap
})
```

Kilka testów

In [35]:
```python
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #3

(Wersja rekurencyjna z losowo wybieranym pivotem - korzystamy z funkcji partition Hoare'a, ponieważ jest nieco szybsza od wersji Lomuto (nie wykonuje niepotrzebnych zamian elementów))

In [36]:
```python
import random


def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)


def _quick_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        pivot_position = partition(arr, left_idx, right_idx)
        _quick_sort(arr, left_idx, pivot_position)
        left_idx = pivot_position + 1  # I removed a tailing recursion


def partition(arr, left_idx, right_idx):
    pivot = arr[random.randint(left_idx, right_idx)]

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            swap(arr, i, j)
        else:
```

```
            return j  # Return a pivot position after the last swap


def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [37]:
```
sorting_functions['Quick Sort (#3 - random pivot)'] = (quick_sort, {
    '_quick_sort': _quick_sort,
    'partition': partition,
    'swap': swap
})
```

In [38]:
```
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #4

(Wersja rekurencyjna z losowo wybieranym pivotem, która zajmuje tylko $O(logn)$ dodatkowej pamięci - na wywołania rekurencyjne)

In [39]:
```
import random


def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)


def _quick_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        pivot_position = _partition(arr, left_idx, right_idx)

        ''' Zmiany poniżej: '''
        if pivot_position - left_idx < right_idx - pivot_position:
            _quick_sort(arr, left_idx, pivot_position)
            left_idx = pivot_position + 1  # I removed a tailing recursion
        else:
            _quick_sort(arr, pivot_position + 1, right_idx)
            right_idx = pivot_position  # I removed a tailing recursion
        ''' Koniec zmian '''


def _partition(arr, left_idx, right_idx):
    pivot = arr[random.randint(left_idx, right_idx)]
    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            _swap(arr, i, j)
        else:
            return j  # Return a pivot position after the last swap


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [40]:
```
sorting_functions['Quick Sort (#4 - O(log n) memory)'] = (quick_sort, {
    '_quick_sort': _quick_sort,
    '_partition': _partition,
    '_swap': _swap
})
```

Kilka testów

In [41]:
```
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #5

(Wersja iteracyjna z użyciem stosu o stałym rozmiarze (jako stos używamy tabicę o długości $log_2(n)$). Jest to zoptymalizowana pamięciowo i wydajnościowo wersja algorytmu).

In [42]:
```
import math
import random


def quick_sort(arr):
    if len(arr) < 2: return
    stack = [None] * math.ceil(math.log(len(arr), 2))
    stack_idx = 0  # An index of the last element in a stack
    stack[0] = (0, len(arr) - 1)

    while stack_idx >= 0:
#         print(stack)
        # Take the last indices pair out of stack
        left_idx, right_idx = stack[stack_idx]
        stack[stack_idx] = None
        stack_idx -= 1

        pivot_position = _partition(arr, left_idx, right_idx)

        if pivot_position - left_idx < right_idx - pivot_position:
            # Store indices of the longer part at first if has at least 2 elements
            if pivot_position + 1 < right_idx:
                stack_idx += 1
                stack[stack_idx] = (pivot_position + 1, right_idx)
            # Then store indices of a shorter part (as it will produce less parts after sorting) (if has at lea
            if left_idx < pivot_position:
                stack_idx += 1
                stack[stack_idx] = (left_idx, pivot_position)
        else:
            # Store indices of the longer part at first if has at least 2 elements
            if left_idx < pivot_position:
                stack_idx += 1
                stack[stack_idx] = (left_idx, pivot_position)
            # Then store indices of a shorter part (as it will produce less parts after sorting) (if has at lea
            if pivot_position + 1 < right_idx:
                stack_idx += 1
                stack[stack_idx] = (pivot_position + 1, right_idx)


def _partition(arr, left_idx, right_idx):
    pivot = arr[random.randint(left_idx, right_idx)]

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            _swap(arr, i, j)
        else:
            return j  # Return a pivot position after the last swap


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [43]:
```
sorting_functions['Quick Sort (#5 - static stack)'] = (quick_sort, {
    '_partition': _partition,
    '_swap': _swap
})
```

Kilka testów

In [44]:
```
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #6

(Wersja iteracyjna z użyciem stosu o dynamicznym rozmiarze (działa jak wyżej, ale korzysta z metod .append() i .pop() Pythonowych list). Jest to zoptymalizowana pamięciowo i wydajnościowo wersja algorytmu).

In [45]:
```
import random


def quick_sort(arr):
    if len(arr) < 2: return
    stack = [(0, len(arr)-1)]

    while stack:
#         print(stack)
        # Take the last indices pair out of the stack
        left_idx, right_idx = stack.pop()

        pivot_position = _partition(arr, left_idx, right_idx)

        if pivot_position - left_idx < right_idx - pivot_position:
            # Store indices of the longer part at first if has at least 2 elements
            if pivot_position + 1 < right_idx:
                stack.append((pivot_position + 1, right_idx))
                # Then store indices of a shorter part (as it will produce less parts after sorting)
                # (if has at least 2 elements and we are sure the longer part has also at least 2 elements)
                if left_idx < pivot_position:
                    stack.append((left_idx, pivot_position))
        else:
            # Store indices of the longer part at first if has at least 2 elements
            if left_idx < pivot_position:
                stack.append((left_idx, pivot_position))
                # Then store indices of a shorter part (as it will produce less parts after sorting)
                # (if has at least 2 elements and we are sure the longer part has also at least 2 elements)
                if pivot_position + 1 < right_idx:
                    stack.append((pivot_position + 1, right_idx))


def _partition(arr, left_idx, right_idx):
    pivot = arr[random.randint(left_idx, right_idx)]

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            _swap(arr, i, j)
        else:
            return j  # Return a pivot position after the last swap


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [46]:
```
sorting_functions['Quick Sort (#6 - dynamic stack)'] = (quick_sort, {
    '_partition': _partition,
    '_swap': _swap
})
```

Kilka testów

In [47]:
```
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
# test_sort(quick_sort, range_=(-100_000, 100_000), val_counts=(0, 10_000), samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #7

(Wersja rekurencyjna z pivotem wybieranym przy pomocy algorytmu Magiczne Piątki, która zajmuje tylko $O(log(n))$ dodatkowej pamięci - na wywołania rekurencyjne)

In [48]:
```
import random
```

```python
def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)


def _quick_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        pivot_position = _partition(arr, left_idx, right_idx)

        if pivot_position - left_idx < right_idx - pivot_position:
            _quick_sort(arr, left_idx, pivot_position)
            left_idx = pivot_position + 1  # I removed a tailing recursion
        else:
            _quick_sort(arr, pivot_position + 1, right_idx)
            right_idx = pivot_position  # I removed a tailing recursion


def _partition(arr, left_idx, right_idx):
    # Use a median of medians algorithm to select a pivot value
    pivot = median_of_medians(arr, left_idx, right_idx)

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            _swap(arr, i, j)
        else:
            return j  # Return a pivot position after the last swap


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]


def median_of_medians(arr, left_idx, right_idx, k=5):
    # Store the position on which the next median will be stored
    # (we will store each median of current k-element subarrays one
    # after another at the beginning of the subarray which begins
    # on the left_index and ends on the right_idx (inclusive)
    next_swap_idx = left_idx

    # Loop till the current subarray has more than k elements
    while right_idx - left_idx >= k:
        # Calculate and store a median of each full k-element subarray
        for end_idx in range(left_idx + k-1, right_idx + 1, k):
            # Store a median on the next index just after the last median stored
            # (swap a median with a value placed after previously calculated medians)
            _swap(arr, next_swap_idx, _select_median(arr, end_idx - k + 1, end_idx))
            next_swap_idx += 1

        # Calculate and store a median of the remaining subarray
        # (which has less than k elements)
        if end_idx < right_idx - 1:
            _swap(arr, next_swap_idx, _select_median(arr, end_idx, right_idx))
            next_swap_idx += 1

        # Prepare variables for the next loop (we will calculate a median of
        # the subarray of medians calculated above, so the right_idx will now
        # be equal to the index of the last median previously determined)
        right_idx = next_swap_idx - 1
        next_swap_idx = left_idx

    # Return a value of a median
    return arr[_select_median(arr, left_idx, right_idx)]


def _select_median(arr, left_idx, right_idx):
    # Using the Selection Sort concept, sort only elements of the
    # subarray which are placed up to the middle index (including
    # the middle element)
```

```
    mid_idx = (right_idx + left_idx) // 2
    for i in range(left_idx, mid_idx + 1):
        min_idx = i
        for j in range(i + 1, right_idx + 1):
            if arr[j] < arr[min_idx]:
                min_idx = j
        _swap(arr, min_idx, i)
    # Return the middle index which is a position of the median
    # after sorting a part of the subarray
    return mid_idx
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [49]:
```
sorting_functions['Quick Sort (#7 - median of medians pivot)'] = (quick_sort, {
    '_quick_sort': _quick_sort,
    '_partition': _partition,
    '_swap': _swap,
    'median_of_medians': median_of_medians,
    '_select_median': _select_median
})
```

Kilka testów

In [50]:
```
# test_sort(quick_sort, samples=10000, val_counts=(0, 1000), failed_only=True)
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #8

(Wersja rekurencyjna ze sztywno ustalonym pivotem, która zajmuje tylko $O(log(n))$ dodatkowej pamięci - na wywołania rekurencyjne, pozwalająca na sortowanie dowolnych danych według zdefiniowanych przez własną funkcję wartości)

Domyślnie funkcja, według której odbywa się sortowanie, sortuje elementy po nich samych (tzn. odbywa się tradycyjne sortowanie)

In [51]:
```
def quick_sort(arr, *, fn=lambda x: x):
    _quick_sort(arr, 0, len(arr) - 1, fn)


def _quick_sort(arr, left_idx, right_idx, fn):
    while left_idx < right_idx:
        pivot_position = _partition(arr, left_idx, right_idx, fn)

        if pivot_position - left_idx < right_idx - pivot_position:
            _quick_sort(arr, left_idx, pivot_position - 1, fn)
            left_idx = pivot_position + 1
        else:
            _quick_sort(arr, pivot_position + 1, right_idx, fn)
            right_idx = pivot_position - 1


def _partition(arr, left_idx, right_idx, fn):
    pivot = fn(arr[right_idx])

    # Partition an array into 2 subarrays of elements lower than
    # pivot and of elements greater than a pivot
    i = left_idx
    for j in range(left_idx, right_idx):
        if fn(arr[j]) < pivot:
            _swap(arr, i, j)
            i += 1

    # Place a pivot element on its destination index
    _swap(arr, i, right_idx)

    return i  # Return a pivot position after the last swap


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [52]:
```
sorting_functions['Quick Sort (#8 - own sorting function)'] = (quick_sort, {
    '_quick_sort': _quick_sort,
    '_partition': _partition,
    '_swap': _swap
})
```

Kilka testów

In [53]:
```
# test_sort(quick_sort, samples=10000, val_counts=(0, 1000), range_=(-10_000, 10_000),
#           ints_only=False, failed_only=True)

# The test below should be failed when there are nagative numbers in a sorted array
print('Sorting by absolute value:')
print('(should be failed because testing function checks if an array is sorted in a non-decreasing order)\n')
test_sort(lambda arr: quick_sort(arr, fn=lambda x: abs(x)), samples=3, ints_only=False, failed_only=True)
print('\n\nThese test will be passed:\n')
test_sort(quick_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting by absolute value:
(should be failed because testing function checks if an array is sorted in a non-decreasing order)

TEST #1:
Before sorting: [47.16, 87.5, -95.3, 67.67, 67.69, -44.56, 39.37, -75.41, 6.72, 9.33, 15.02, 92.55, -7.13, 81.6
8, 67.07, -21.35, -53.68, 48.47, 13.18, -85.66, 19.47, 62.13, -80.81, -41.48, -98.61, -59.09, -1.47, 27.98, -0.
53, 49.65, 59.81, 60.45, 39.0, -92.24]
After sorting: [-0.53, -1.47, 6.72, -7.13, 9.33, 13.18, 15.02, 19.47, -21.35, 27.98, 39.0, 39.37, -41.48, -44.5
6, 47.16, 48.47, 49.65, -53.68, -59.09, 59.81, 60.45, 62.13, 67.07, 67.67, 67.69, -75.41, -80.81, 81.68, -85.6
6, 87.5, -92.24, 92.55, -95.3, -98.61]
Expected result: [-98.61, -95.3, -92.24, -85.66, -80.81, -75.41, -59.09, -53.68, -44.56, -41.48, -21.35, -7.13,
-1.47, -0.53, 6.72, 9.33, 13.18, 15.02, 19.47, 27.98, 39.0, 39.37, 47.16, 48.47, 49.65, 59.81, 60.45, 62.13, 6
7.07, 67.67, 67.69, 81.68, 87.5, 92.55]
Test FAILED
Current passed-to-tested ratio: 0/1

TEST #2:
Before sorting: [-19.96, -47.83, 85.34, 2.99, 90.33, -35.07, 74.19, 48.99, 22.95, 48.32, -88.36]
After sorting: [2.99, -19.96, 22.95, -35.07, -47.83, 48.32, 48.99, 74.19, 85.34, -88.36, 90.33]
Expected result: [-88.36, -47.83, -35.07, -19.96, 2.99, 22.95, 48.32, 48.99, 74.19, 85.34, 90.33]
Test FAILED
Current passed-to-tested ratio: 0/2

TEST #3:
Before sorting: [-23.62, 42.76, -38.18, 26.5, -50.77, -90.07, -0.11, -67.99, -51.78, 1.94, 67.49, -91.16, -48.6
7, -37.07, 29.15, 7.45, 96.73, 46.13, -82.52, -29.2, -84.06, -26.06, 38.4, -48.7, 98.86, -56.44, 35.47, -21.15,
-59.9, 82.32, -83.94, -4.54, 40.16, 17.32, -22.38, 86.81, -36.09, 84.8, -74.29, 18.28, -78.78, 23.06, -53.34, -
93.13, -81.15, 74.15]
After sorting: [-0.11, 1.94, -4.54, 7.45, 17.32, 18.28, -21.15, -22.38, 23.06, -23.62, -26.06, 26.5, 29.15, -2
9.2, 35.47, -36.09, -37.07, -38.18, 38.4, 40.16, 42.76, 46.13, -48.67, -48.7, -50.77, -51.78, -53.34, -56.44, -
59.9, 67.49, -67.99, 74.15, -74.29, -78.78, -81.15, 82.32, -82.52, -83.94, -84.06, 84.8, 86.81, -90.07, -91.16,
-93.13, 96.73, 98.86]
Expected result: [-93.13, -91.16, -90.07, -84.06, -83.94, -82.52, -81.15, -78.78, -74.29, -67.99, -59.9, -56.4
4, -53.34, -51.78, -50.77, -48.7, -48.67, -38.18, -37.07, -36.09, -29.2, -26.06, -23.62, -22.38, -21.15, -4.54,
-0.11, 1.94, 7.45, 17.32, 18.28, 23.06, 26.5, 29.15, 35.47, 38.4, 40.16, 42.76, 46.13, 67.49, 74.15, 82.32, 84.
8, 86.81, 96.73, 98.86]
Test FAILED
Current passed-to-tested ratio: 0/3

Sorting algorithm is wrong
Passed tests in total: 0/3


These test will be passed:

Sorting algorithm is correct
Passed tests in total: 100/100
```

Quicker Sort

Jest to zmodyfikowana wersja algorytmu Quick Sort, dzięki czemu algorytm ten jest jeszcze szybszy, ale jedynie w przypadku, gdy w tablicy występują powtórki danej wartości. Tzn. wartość wybierana za pivota musi wystąpić przynajmniej kilka razy, abyśmy zauważyli wzrost wydajności. Jedyną różnicą względem zwykłego Quick Sorta jest idea podziału tablicy (dzielimy na 3 części, z których pierwsza zawiera elementy mniejsze od pivota, druga mu równe, a trzecia większe od niego).

Złożoność sortowania

Złożoność czasowa

Najgorszy przypadek

$O(n^2)$

lub: $O(n \cdot log(n))$ przy skorzystaniu z pivota, wybieranego przy pomocy algorytmu Mediana Median (zwanego dla $k = 5$, gdzie k - liczebność grup, na które dzielimy tablicę, "Magiczne Piątki")

UWAGA: W tym przypadku można zauważyć, że fakt iż algorytm z użyciem Magicznych Piątek ma niższą najgorszą złożoność czasową i taką samą złożoność oczekiwaną, nie świadczy wcale o jego szybszym działaniu. W większości przypadków, gdy otrzymujemy dane, które nie są posortowane, zauważymy spadek wydajności, ponieważ zawsze konieczne jest wyznaczenie dodatkowo mediany median, która w danym przypadku nic nie poprawia.

Najlepszy przypadek

$O(n \cdot log(k(n)))$

Taką złożoność osiągniemy, jeżeli w tablicy będzie bardzo dużo powtarzających się wartości. (W powyższym oszacowaniu złożoności przyjmujemy, że w n- elementowej tablicy znajduje się tylko $k(n) = log(n)$ różnych wartości).

Złożoność pamięciowa

Podobnie jak Quick Sort

Stabilność

Niestabilny

Czy sortowanie odbywa się w miejscu?

Ta sama zasada, co dla Quick Sorta

Implementacja algorytmu #1

(Wersja rekurencyjna z usuniętą rekursją ogonową i zoptymalizowanym wywoływaniem rekurencyjnym w taki sposób, by zużywać maksymalnie $O(log(n))$ miejsca na stosie rekurencyjnym).

In [54]:
```
def quicker_sort(arr):
    _quicker_sort(arr, 0, len(arr) - 1)


def _quicker_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        lt_pivot_last, gt_pivot_first = _partition(arr, left_idx, right_idx)

        # If a number of elements lower than a pivot is greater than a number
        # of elements which are greater than a pivot, sort recursively the shorter part
        # of elements which are greater than a pivot
        if lt_pivot_last - left_idx > right_idx - gt_pivot_first:
            _quicker_sort(arr, gt_pivot_first, right_idx)
            right_idx = lt_pivot_last  # I removed a tailing recursion
        # Otherwise, sort a subarray of elementslower than a pivot first
        # as it is shorter than a subarray of elements greater than a pivot
        else:
            _quicker_sort(arr, left_idx, lt_pivot_last)
            left_idx = gt_pivot_first  # I removed a tailing recursion


def _partition(arr, left_idx, right_idx):
    pivot = arr[left_idx]

    # Partition an array into 3 subarrays (lower than, equal to and
    # greater than a pivot value)
    i = left_idx       # A pointer of the first pivot
    j = left_idx + 1   # A pointer of elements which are after the last pivot
    k = j              # A pointer of the currently checked element
    while k <= right_idx:
        if arr[k] < pivot:
            _two_swaps(arr, i, j, k)
            i += 1
            j += 1
        elif arr[k] == pivot:
            _one_swap(arr, j, k)
            j += 1
        k += 1

    # Return the last index of the subarray of elements lower than a pivot
    # and the first index of the subarray of elements greater than a pivot
    return i - 1, j


def _one_swap(arr, i, j):
    # Swap two elements in an array
    arr[i], arr[j] = arr[j], arr[i]


def _two_swaps(arr, i, j, k):
    # Rotate right the elements of the indices specified
    arr[k], arr[i], arr[j] = arr[j], arr[k], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [55]:
```
sorting_functions['Quicker Sort (#1 - recursive)'] = (quicker_sort, {
    '_quicker_sort': _quicker_sort,
    '_partition': _partition,
    '_one_swap': _one_swap,
    '_two_swaps': _two_swaps
})
```

Kilka testów

In [56]:
```
# test_sort(quicker_sort, range_=(0, 100), val_counts=(0, 10_000), samples=1000, failed_only=True)
# test_sort(quicker_sort, range_=(-20, 20), val_counts=(100, 200), ints_only=True, uniform_distribu
test_sort(quicker_sort, samples=100, ints_only=False, uniform_distribution=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #2

(Wersja rekurencyjna ze zoptymalizowanym wybieraniem pivota jako mediany median (patrz plik z algorytmami wyszukiwania i wyboru) oraz z usuniętą rekursją ogonową i zoptymalizowanym wywoływaniem rekurencyjnym w taki sposób, by zużywać maksymalnie $O(log(n))$ miejsca na stosie rekurencyjnym).

Poniższa implementacja rozwiązuje problem wyboru pivota, dzięki czemu zminimalizowane jest ryzyko wystąpienia najgorszego przypadku, w którym np. na wejściu dostaniemy posortowaną tablicę. Dzięki temu niemożliwe jest ukwadratowienie się poniższego algorytmu sortowania.

In [57]:
```
def quicker_sort(arr):
    _quicker_sort(arr, 0, len(arr) - 1)


def _quicker_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        lt_pivot_last, gt_pivot_first = _partition(arr, left_idx, right_idx)

        # If a number of elements lower than a pivot is greater than a number
        # of elements which are greater than a pivot, sort recursively the shorter part
        # of elements which are greater than a pivot
        if lt_pivot_last - left_idx > right_idx - gt_pivot_first:
            _quicker_sort(arr, gt_pivot_first, right_idx)
            right_idx = lt_pivot_last  # I removed a tailing recursion
        # Otherwise, sort a subarray of elementslower than a pivot first
        # as it is shorter than a subarray of elements greater than a pivot
        else:
            _quicker_sort(arr, left_idx, lt_pivot_last)
            left_idx = gt_pivot_first  # I removed a tailing recursion


def _partition(arr, left_idx, right_idx):
    pivot_idx = median_of_medians(arr, left_idx, right_idx)
    pivot = arr[pivot_idx]
    # Place a pivot value in the left_idx position of an array
    _one_swap(arr, left_idx, pivot_idx)

    # Partition an array into 3 subarrays (lower than, equal to and
    # greater than a pivot value)
    i = left_idx       # A pointer of the first pivot
    j = left_idx + 1   # A pointer of elements which are after the last pivot
    k = j              # A pointer of the currently checked element
    while k <= right_idx:
        if arr[k] < pivot:
            _two_swaps(arr, i, j, k)
            i += 1
            j += 1
        elif arr[k] == pivot:
            _one_swap(arr, j, k)
            j += 1
        k += 1

    # Return the last index of the subarray of elements lower than a pivot
    # and the first index of the subarray of elements greater than a pivot
    return i - 1, j


def _one_swap(arr, i, j):
    # Swap two elements in an array
    arr[i], arr[j] = arr[j], arr[i]


def _two_swaps(arr, i, j, k):
    # Rotate right the elements of the indices specified
    arr[k], arr[i], arr[j] = arr[j], arr[k], arr[i]


def median_of_medians(arr, left_idx, right_idx, k=5):
    # Store the position on which the next median will be stored
    # (we will store each median of current k-element subarrays one
    # after another at the beginning of the subarray which begins
    # on the left_idx index and ends on the right_idx (inclusive)
    next_swap_idx = left_idx

    # Loop till the current subarray has more than k elements
    while right_idx - left_idx >= k:
        # Calculate and store a median of each full k-element subarray
        for end_idx in range(left_idx + k-1, right_idx + 1, k):
            # Store a median on the next index just after the last median stored
            # (swap a median with a value placed after previously calculated medians)
            _one_swap(arr, next_swap_idx, _select_median(arr, end_idx - k + 1, end_idx))
            next_swap_idx += 1

        # Calculate and store a median of the remaining subarray
        # (which has less than k elements)
        if end_idx < right_idx - 1:
            _one_swap(arr, next_swap_idx, _select_median(arr, end_idx, right_idx))
            next_swap_idx += 1

        right_idx = next_swap_idx - 1
        next_swap_idx = left_idx

    # Return a value of a median
    return _select_median(arr, left_idx, right_idx)


def _select_median(arr, left_idx, right_idx):
    # Using the Selection Sort concept, sort only elements of the
    # subarray which are placed up to the middle index (including
    # the middle element)
    mid_idx = (right_idx + left_idx) // 2
    for i in range(left_idx, mid_idx + 1):
        min_idx = i
        for j in range(i + 1, right_idx + 1):
            if arr[j] < arr[min_idx]:
                min_idx = j
        _one_swap(arr, min_idx, i)
    # Return the middle index which is a position of the median
    # after sorting a part of the subarray
    return mid_idx
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [58]:
```
sorting_functions['Quicker Sort (#2 - median of medians pivot)'] = (quicker_sort, {
    '_quicker_sort': _quicker_sort,
    '_partition': _partition,
    '_one_swap': _one_swap,
    '_two_swaps': _two_swaps,
    'median_of_medians': median_of_medians,
    '_select_median': _select_median
})
```

Kilka testów

In [59]:
```
test_sort(quicker_sort, samples=100, uniform_distribution=False, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Algorytmy o złożoności $O(n)$

Poniższe algorytmy są bardzo szybkie (szybsze od wszystkich powyższych), ale jedynie wtedy, gdy spełnione są pewne założenia. W zależności od algorytmu, przyjmujemy, że będzie on sortował dane konkretnego rodzaju, o zakładanym rozkładzie lub z odpowiedniego przedziału.

Counting Sort

Założenia, aby korzystanie z algorytmu było opłacalne

- Z reguły zakładamy, że sortowane elementy to liczby całkowite z przedziału od 0 do pewnej wartości k, będącej największą wartością w sortowanej tablicy (możliwe jest oczywiście sortowanie również liczb ujemnych, przy pomocy tego algorytmu, bądź także liter (ciągów tekstowych więcej niż 1-literowych lepiej nie sortować tym sposobem), po wprowadzeniu odpowiednich modyfikacji algorytmu,
- Counting Sort działa najlepiej dla danych, w których wiele wartości się powtarza, choć nie wpływa to znacząco na jego złożoność, która ZAWSZE zależy w największym stopniu od zakresu wartości, czyli jaka jest wartość najmniejsza i jaka największa (patrz niżej).

Złożoność sortowania

Złożoność czasowa

Każdy przypadek

$O(n+k)$

n - liczba elementów w sortowanej tablicy (wszystkich), k - zakres unikatowych wartości (zazwyczaj liczba liczb całkowitych, jakie znajdują się w przedziale $[0, k]$, gdzie k jest największą liczbą z sortowanej tablicy) (rzeczywiście na przedziale domkniętym jest $k+1$ unikatowych wartości, ale różnica o stałą równą 1 nie jest uwzględniana w złożoności)

Złożoność pamięciowa

Każdy przypadek

$O(n+k)$

k - tworzymy pomocniczą tablicę, w której będziemy przechowywać liczby wystąpień poszczególnych wartości pod indeksami, które odpowiadają tym wartościom (dlatego najlepiej działa ten algorytm na liczbach z małych przedziałów). Później modyfikujemy tę tablicę tak, by dla danego indeksu (odpowiadającego danej liczbie), w komórce tablicy pomocniczej o tym indeksie, występowała liczba wartości, które są nie większe niż indeks tej komórki. n - pomocnicza tablica, do której przepisujemy wartości w odpowiedniej kolejności, tak, aby je ponownie później przepisać do tablicy wyjściowej.

Stabilność

Stabilny

W poniższej implementacji, gdzie przepisywanie wartości do pomocniczej tablicy rozpoczynamy od końca sortowanej tablicy. Wiele implementacji w Internecie zawiera niestabilną wersję tego algorytmu.

Czy sortowanie odbywa się w miejscu?

Nie

Zawsze musimy użyć dodatkowej pamięci, która jest zależna od danych wejściowych (tu od liczby elementów do posortowania oraz liczby unikatowych wartości z tego przedziału, do którego należą sortowane wartości).

Implementacja algorytmu #1 (modyfikuje tablicę)

(Wersja JEDYNIE dla liczb całkowitych nieujemnych od 0 do wskazanej wartości k)

In [60]:
```
def counting_sort(arr, k: 'the upper bound of the values range'):
    # Allocate memory for required temporary arrays
    counts = [0] * (k + 1)
    temp = [None] * len(arr)
    # Count values repetitions
    for val in arr:
        counts[val] += 1
    # Modify the counts array to indicate how many values are not greater than the current one
    for i in range(1, len(counts)):
        counts[i] += counts[i-1]
    # Rewrite values to the temp sorted array
    for i in range(len(arr)-1, -1, -1):
        counts[arr[i]] -= 1
        temp[counts[arr[i]]] = arr[i]
    # Rewrite sorted values to the initial array
    for i in range(len(temp)):
        arr[i] = temp[i]
```

Kilka testów

In [61]:
```
import random

max_val = 100
val_counts = (0, 100)
arr = [random.randint(0, max_val) for _ in range(random.randint(*val_counts))]
print('Input:', arr)
expected = sorted(arr)
k = expected[-1]
counting_sort(arr, k)
is_correct = expected == arr
print('Expected:', expected)
print('Result:', arr)
print('Correct?:', is_correct)
```

```
Input: [56, 78, 66, 81, 38, 11, 6, 77, 94, 82, 69, 25, 52, 31, 48, 19, 19, 56, 28, 57, 39, 47, 45, 29, 22, 17,
97, 41, 44, 73, 82, 35, 29, 40, 9, 95, 74, 57, 85, 92, 59, 57, 50, 44, 88, 13, 75, 51, 23, 97, 81, 89, 87, 87,
84]
Expected: [6, 9, 11, 13, 17, 19, 19, 22, 23, 25, 28, 29, 29, 31, 35, 38, 39, 40, 41, 44, 44, 45, 47, 48, 50, 5
1, 52, 56, 56, 57, 57, 57, 59, 66, 69, 73, 74, 75, 77, 78, 81, 81, 82, 82, 84, 85, 87, 87, 88, 89, 92, 94, 95,
97, 97]
Result: [6, 9, 11, 13, 17, 19, 19, 22, 23, 25, 28, 29, 29, 31, 35, 38, 39, 40, 41, 44, 44, 45, 47, 48, 50, 51,
52, 56, 56, 57, 57, 57, 59, 66, 69, 73, 74, 75, 77, 78, 81, 81, 82, 82, 84, 85, 87, 87, 88, 89, 92, 94, 95, 97,
97]
Correct?: True
```

Implementacja algorytmu #2 (nie modyfikuje tablicy)

(Wersja JEDYNIE dla liczb całkowitych nieujemnych od 0 do wskazanej wartości k)

In [62]:
```
def counting_sort(arr, k: 'the upper bound of the values range'):
    # Allocate memory for required temporary arrays
    counts = [0] * (k + 1)
    new = [None] * len(arr)
    # Count values repetitions
    for val in arr:
        counts[val] += 1
    # Modify the counts array to indicate how many values are not greater than the current one
    for i in range(1, len(counts)):
        counts[i] += counts[i-1]
    # Rewrite values to the temp sorted array
    for i in range(len(arr)-1, -1, -1):
        counts[arr[i]] -= 1
        new[counts[arr[i]]] = arr[i]
    return new
```

Kilka testów

In [63]:
```
import random

max_val = 100
val_counts = (1, 100)
arr = [random.randint(0, max_val) for _ in range(random.randint(*val_counts))]
expected = sorted(arr)
k = expected[-1]
res = counting_sort(arr, k)
is_correct = expected == res
print('Input:', arr)
print('Expected:', expected)
print('Result:', res)
print('Correct?:', is_correct)
```

```
Input: [42, 30, 44, 15, 47, 2, 74, 20, 96, 57, 43, 25, 53, 25, 27, 9, 15, 89, 15, 26, 41, 43]
Expected: [2, 9, 15, 15, 15, 20, 25, 25, 26, 27, 30, 41, 42, 43, 43, 44, 47, 53, 57, 74, 89, 96]
Result: [2, 9, 15, 15, 15, 20, 25, 25, 26, 27, 30, 41, 42, 43, 43, 44, 47, 53, 57, 74, 89, 96]
Correct?: True
```

Implementacja algorytmu #3 (modyfikuje tablicę - można to łatwo zmienić)

(Wersja, JEDYNIE dla liczb całkowitych nieujemnych od 0 do największej wartości - której nie wskazujemy jawnie (po prostu szukamy jej liniowo, bo tablica jest nieposortowana))

In [64]:
```
def counting_sort(arr):
    if arr: _counting_sort(arr, _max_val(arr))


def _counting_sort(arr, k: 'the upper bound of the values range'):
    # Allocate memory for required temporary arrays
    counts = [0] * (k + 1)
    temp = [None] * len(arr)
    # Count values repetitions
    for val in arr:
        counts[val] += 1
    # Modify the counts array to indicate how many values are not greater than the current one
    for i in range(1, len(counts)):
        counts[i] += counts[i-1]
    # Rewrite values to the temp sorted array
    for i in range(len(arr)-1, -1, -1):
        counts[arr[i]] -= 1
        temp[counts[arr[i]]] = arr[i]
    # Rewrite sorted values to the initial array
    for i in range(len(temp)):
        arr[i] = temp[i]


def _max_val(arr):
    if not arr: return None
    max_ = arr[0]
    for i in range(1, len(arr)):
        if arr[i] > max_:
            max_ = arr[i]
    return max_
```

Kilka testów

In [65]:
```
test_sort(counting_sort, range_=(0, 100), samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #4 (modyfikuje tablicę)

(Wersja, dla liczb całkowitych (MOGĄ BYĆ UJEMNE) bez konieczności wskazywania wartości najmniejszej i największej (po prostu szukamy ich liniowo w jednej pętli, bo tablica jest nieposortowana))

In [66]:
```
def counting_sort(arr):
    if arr:
        min_, max_ = minmax(arr)
        _counting_sort(arr, min_, max_)


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max


def _counting_sort(arr, k: 'the lower bound', m: 'the upper bound'):
    # Allocate memory for required temporary arrays
    counts = [0] * (m - k + 1)
    temp = [None] * len(arr)
    # Count values repetitions
    for val in arr:
        counts[val - k] += 1
    # Modify the counts array to indicate how many values are not greater than the current one
    for i in range(1, len(counts)):
        counts[i] += counts[i-1]
    # Rewrite values to the temp sorted array
    for i in range(len(arr)-1, -1, -1):
        counts[arr[i] - k] -= 1
        temp[counts[arr[i] - k]] = arr[i]
    # Rewrite sorted values to the initial array
    for i in range(len(temp)):
        arr[i] = temp[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [67]:
```
sorting_functions['Counting Sort (#4 - negative & positive)'] = (counting_sort, {
    '_counting_sort': _counting_sort,
    'minmax': minmax
})
```

Kilka testów

In [68]:
```
test_sort(counting_sort, samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Radix Sort

Założenia, aby korzystanie z algorytmu było opłacalne

- Dane wejściowe musi się dać podzielić na poszczególne fragmenty, odpowiadające pojedynczej cyfrze (lub pojedynczemu znakowi w przypadku sortowania tekstu),
- Algorytm pomocniczy, przy pomocy którego będziemy sortować wartości według znaków (poszczególnych cyfr), musi mieć niską złożoność i, przede wszystkim, być stabilny (tylko wtedy Radix Sort działa prawidłowo).

Złożoność sortowania

Złożoność czasowa

Każdy przypadek

$O(d \cdot (n+b))$

$d = log_b(k)$, gdzie k - maksymalna wartość w sortowanej tablicy,
n - liczba elementów w sortowanej tablicy,
b - podstawa systemu liczbowego, w jakim odbywa się sortowanie (bo tyle elementów będzie miała tablica, służąca do zliczania powrótek wartości (Radix Sort zazwyczaj korzysta z Counting Sorta) (W przypadku sortowania tekstu, jako podstawę przyjmujemy różnicę między największym kodem ASCII liter a najmniejszym (plus 1), zwykle będzie to liczba 26, ponieważ tyle jest małych liter alfabetu łacińskiego. Samo sortowanie wygląda jednak nieco inaczej niż w przypadku sortowania liczb, co zostało zaprezentowane w jednej z poniższych implementacji)

UWAGA

Najlepszą złożoność obliczeniową uzyskamy wtedy, gdy $b \approx n$, ponieważ wówczas $d \approx log_n(k)$ ma niewielką wartość i równocześnie suma $n + b \approx 2 \cdot n$, a więc 2-krotne zwiększenie czynnika $(n+b)$ spowodowało nam znacznie zmniejszenie czynnika d. Nie możemy jednak wziąć dowolnie dużej wartości b, ponieważ wpłynie to negatywnie na złożoność pamięciową, która wynosi $O(n+b)$, a także obliczeniową, która wtedy, mimo iż $d \to 0$, będzie wysoka, ponieważ będziemy musieli zaalokować $O(n+b)$ pamięci, a alokacja odbywa się w czasie proporcjonalnym do alokowanego miejsca, czyli w tym przypadku będzie to $O(n+b)$.
Przy założeniu, że $b = \theta(n)$, otrzymujemy złożoność:
$O(d \cdot (n+n)) = O(d \cdot n)$
Jeżeli również mamy stałe górne ograniczenie na wielkość liczb (tzn. k nie przekracza pewnej ustalonej wartości), z dobrym przybliżeniem można przyjąć, że $O(d) = O(log_b(k)) = |b = \theta(n)| = O(log_n(k)) \approx O(1)$ (w notacji dużego O i tak zawsze $k = O(1)$, jeżeli k - ustalone, ale tu dodatkowo bierzemy logarytm o podstawie n z liczby k, więc tym bardziej wartość $d = log_n(k)$ jest bliska jedynce).
Otrzymujemy wówczas:
$O(d \cdot n) = O(1 \cdot n) = O(n)$

Złożoność pamięciowa

Każdy przypadek

$O(n+b)$

n - liczba sortowanych elementów (tyle elementów będzie miała tablica temp),
b - podstawa systemu liczbowego (tyle elementów będzie miała tablica count). Podstawa ta może być dowolna (nie musimy koniecznie sortować liczb dziesiętnych, biorąc za podstawę systemu liczbowego wartość $b = 10$).

UWAGA

Dla $b = \theta(n)$, otrzymujemy:
$O(n+n) = O(n)$

Stabilność

Stabilny

Jest to jedna z cech tego algorytmu, że żeby działał on prawidłowo, do sortowania wybieramy jako algorytm pomocniczy algorytm stabilny, np. Counting Sort.

Czy sortowanie odbywa się w miejscu?

Nie

Zawsze musimy użyć dodatkowej pamięci, która jest zależna od danych wejściowych (tu od liczby elementów do posortowania oraz liczby unikatowych wartości z tego przedziału, do którego należą sortowane wartości).

Implementacja algorytmu #1 (z pomocą Counting Sorta)

(Wersja JEDYNIE dla liczb całkowitych nieujemnych)

In [69]:
```
def radix_sort(arr, base=None):
    n = len(arr)
    if n > 1:
        if base is None: base = n
        # Store the greatest value
        max_val = _max(arr)

        digit_place = 1
        while max_val >= digit_place:
            _counting_sort(arr, base, digit_place)
            digit_place *= base


def _counting_sort(arr, base, digit_place):
    # Allocate memory for required temporary arrays
    counts = [0] * base
    temp = [None] * len(arr)
    # Count digits repetitions
    for val in arr:
        digit = (val // digit_place) % base
        counts[digit] += 1
    # Modify the counts array to indicate how many digits are not greater than the current one
    for i in range(1, base):
        counts[i] += counts[i-1]
    # Rewrite values to the temp sorted array
    for i in range(len(arr)-1, -1, -1):
        digit = (arr[i] // digit_place) % base
        counts[digit] -= 1
        temp[counts[digit]] = arr[i]
    # Rewrite sorted values to the initial array
    for i in range(len(temp)):
        arr[i] = temp[i]


def _max(arr):
    if arr:
        max_val = arr[0]
        for i in range(1, len(arr)):
            if arr[i] > max_val:
                max_val = arr[i]
        return max_val
    else:
        return None
```

Kilka testów

In [70]:
```
test_sort(radix_sort, range_=(0, 100), samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #2 (z pomocą Counting Sorta)

(Wersja dla WSZYSTKICH liczb całkowitych)

In [71]:
```
def radix_sort(arr, base=None):
    n = len(arr)
    if n > 1:
        if base is None: base = n
        first_non_negative_idx, min_negative, max_positive = _partition(arr)
        # If not all the numbers are positive (there are some negative integers)
        if first_non_negative_idx > 0:
            _radix_sort(arr, 0, first_non_negative_idx - 1, min_negative, base)
        # If not all the numbers are negative (there are some positive integers)
        if first_non_negative_idx < len(arr):
            _radix_sort(arr, first_non_negative_idx, len(arr) - 1, max_positive, base)


def _radix_sort(arr, begin_idx, end_idx, limit, base):
    digit_place = 1
    limit = abs(limit)
    while limit >= digit_place:
        _counting_sort(arr, begin_idx, end_idx, digit_place, base)
        digit_place *= base

# This function moves negative elements to the left and positive ones to the right side
# of an array. Time complexity is only O(n) as it uses approach similar to the Quick Sort
# Lomuto's partition with the pivot set to -1 as it partitions an array into a part of values
# lower than or equal to a pivot and values greater than a pivot
def _partition(arr):
    pivot = -1
    max_val = arr[0]
    # If there is no negative number, we will still return -1 value (this value is used only
    # to store the lowest number (if there are negative numbers))
    min_val = -1

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater than a pivot
    i = 0
    for j in range(len(arr)):
        if arr[j] <= pivot:
            if arr[j] < min_val:
                min_val = arr[j]
            _swap(arr, i, j)
            i += 1
        elif arr[j] > max_val:
            max_val = arr[j]

    return i, min_val, max_val


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]


def _counting_sort(arr, begin_idx, end_idx, digit_place, base):
    # Allocate memory for required temporary arrays
    counts = [0] * base
    result = [None] * (end_idx - begin_idx + 1)
    # Count digits repetitions
    for i in range(begin_idx, end_idx + 1):
        digit = (arr[i] // digit_place) % base
        counts[digit] += 1
    # Modify the counts array to indicate how many digits are not greater than the current one
    for i in range(1, base):
        counts[i] += counts[i - 1]
    # Rewrite values to the result sorted array
    for i in range(end_idx, begin_idx - 1, -1):
        digit = (arr[i] // digit_place) % base
        counts[digit] -= 1
        result[counts[digit]] = arr[i]
    # Rewrite sorted values to the initial array
    for i in range(len(result)):
        arr[begin_idx + i] = result[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [72]:
```
sorting_functions['Radix Sort (#2 - negative & positive)'] = (radix_sort, {
    '_counting_sort': _counting_sort,
    '_radix_sort': _radix_sort,
    '_partition': _partition,
    '_swap': _swap
})
```

Kilka testów

In [73]:
```
test_sort(radix_sort, samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Bucket Sort

Założenia, aby korzystanie z algorytmu było opłacalne

- MAMY PEWNOŚĆ, że otrzymujemy dane wejściowe o ROZKŁADZIE JEDNOSTAJNYM. Oznacza to, że każde z wiaderek, na które podzielimy dane wejściowe (w czasie liniowym), będzie zawierało zbliżoną liczbę elementów do pozostałych wiaderek,
- Tak naprawdę algorytm sortowania Bucket Sort sam w sobie nie jest algorytmem sortowania, a jedynie algorytmem, który dzieli dane wejściowe na mniejsze, szybsze i łatwiejsze w sortowaniu zbiory danych,
- Podzielone na wiaderka dane musimy posortować jakimś wydajnym (w zależności od liczby elementów w wiaderku) algorytmem. Jeżeli liczba elementów jest niewielka, najlepiej się sprawdzi uniwersalny algorytm sortowania, jakim jest Insertion Sort (dla małych danych algorytm ten jest szybki, poza tym, jest to stabilny algorytm sortowania, więc możemy otrzymać stabilną wersję Bucket Sorta i jeszcze jedno, dla prawie posortowanych danych, złożoność Insertion Sorta jest liniowa, a więc istnieje największa szansa, że użycie tego prostego algorytmu o średniej złożoności obliczeniowej $O(n^2)$, spowoduje posortowanie niektórych wiaderek w czasie liniowym), natomiast dla większych wiaderek, lepiej wykorzystać jakiś algorytm o złożoności $O(n \cdot log(n))$,
- Warto dodać, że jeżeli pojemniki (wiaderka) zawierają liczby całkowite lub ciągi tekstowe, można wykorzystać np. Radix Sorta do posortowania każdego z pojemników lub Counting Sorta, jeżeli zakres liczb jest niewielki, a liczb o powtarzającej się wartości jest dużo,
- Ponieważ nie mamy pewności, ile dokładnie elementów trafi do danego pojemnika, nie opłaca się alokować pamięci "na sztywno" dla każdego z pojemników. Lepiej wykorzystać listę odsyłaczową (patrz plik z sortowaniem list odsyłaczowych) lub dynamiczną tablicę (np. Pythonową listę - my ją wykorzystamy w poniższych przykładach).

Złożoność sortowania

Złożoność czasowa

Najlepszy przypadek

$O(n+k)$

$O(n)$ - czas utworzenia kubełków (jest on równy $O(n)$, ponieważ zawsze liczba kubełków powinna zależeć od wielkości danych wejściowych oraz konieczne jest liniowe przejście po tablicy i umieszczenie wartości w odpowiednich wiaderkach)

$O(k)$ - czas sortowania wszystkich kubełków (zależny od liczby kubełków) (przyjmujemy, że czas sortowania pojedynczego kubełka jest w przybliżeniu stały, ponieważ zawiera on niewielką liczbę elementów; dokładniej, to skoro k n, to iloraz $\frac{n}{k}$, który wynosi tyle, ile przybliżona liczba elementów, które przypadają na jeden kubełek, jest stały (zwykle ustalamy tę liczbę odgórnie, zamiast liczby kubełków))

UWAGA:

Aby osiągnąć złożoność liniową, musimy ustalić tak wartość k, aby zależała ona od n.

W powyższych oszacowaniach przyjąłem następujące oznaczenia:
k - liczba kubełków,
n - liczba elementów w sortowanej tablicy,

W implementacjach algorytmu, jakie zamieściłem niżej, k oznacza stosunek liczby elementów w tablicy (n) do liczby kubełków, jakie mają zostać utworzone, czyli średnią liczbę elementów, które przypadają na pojedynczy kubełek.

Najgorszy przypadek

$O(n^2)$

W praktyce, przy dobrym doborze wiaderek, istnieje niewielka szansa na osiągnięcie złożoności $O(n^2)$. Złożoność będzie tym bliższa złożoności pesymistycznej ($O(n^2)$), im więcej wartości trafi do pojedynczego wiaderka (w szczególności, gdy wszystkie wartości znajdą się w jednym wiaderku, a wiaderka sortujemy algorytmem o złożoności $O(n^2)$)

Złożoność pamięciowa

Każdy przypadek

$O(n+k)$

Zależy od liczby elementów, jakie otrzymujemy w tablicy do posortowania oraz od liczby kubełków, jakie tworzymy. Ponieważ powinniśmy dobrać tak liczbę kubełków k, aby $k = \theta(n)$ (asymptotycznie k musi być liniowe względem rozmiaru tablicy do posortowania), otrzymamy złożoność pamięciową $O(n+n) = O(n)$.

Stabilność

Zależy

Stabilność jest zależna od algorytmu użytego do sortowania wiaderek. Sam Bucket Sort nie przeprowadza sortowania, więc jego stabilność jest uzależniona od stabilności algorytmu, który odpowiada za posortowanie każdego z wiaderek. Jeżeli korzystamy z Insertion Sorta, algorytm Bucket Sort jest stabilny.

Czy sortowanie odbywa się w miejscu?

Nie

Zawsze musimy użyć dodatkowej pamięci, która jest zależna od danych wejściowych (tu od liczby elementów do posortowania oraz liczby wiaderek, która również zależy od wielkości danych wejściowych).

Implementacja algorytmu #1 (z pomocą Insertion Sorta)

(Wersja dla dowolnych liczb rzeczywistych)

Zdecydowałem się na wybór Insertion Sorta do posortowania wiaderek, ponieważ jest on stabilnym algorytmem sortowania oraz dla małych danych (do ok. 24 elementów - patrz porównanie wydajności dla takich danych na końcu pliku) jest bardzo szybki. W poniższej implementacji wykorzystuję liczbę wiaderek, która jest równa $\frac{1}{16}$ wielkości danych ($\frac{2}{3} \cdot 24 = 16$) wejściowych tak, aby każde z wiaderek miało odpowiednio niewielką liczbę elementów, dzięki czemu da się je szybko posortować. Oczywiście można użyć innej wartości tego współczynnika, ale, uwzględniając możliwe odchylenia w obie strony od liczby 16, chcę, aby w jak największej liczbie wiaderek, liczba elementów nie przekroczyła zbyt mocno liczby 24.

In [74]:
```
def bucket_sort(arr, k: 'threshold' = 24):
    if len(arr) <= k:
        insertion_sort(arr)
    else:
        # Create buckets
        m = int(2/3 * k)
        buckets_count = len(arr) // m + 1
        buckets = [[] for _ in range(buckets_count)]
        min_val, max_val = minmax(arr)
        val_interval = (max_val - min_val) / buckets_count
        # Distribute values to the proper buckets
        for val in arr:
            # Calculate the bucket's index depending on how much the
            # current value is greater than the lowest one
            bucket_idx = int((val - min_val) / val_interval - .5)  # Round down in order not to overflow an arr
            buckets[bucket_idx].append(val)
        # Sort each bucket separately
        for bucket in buckets:
            insertion_sort(bucket)
        # Rewrite sorted values from buckets to the inintial array
        i = 0
        for bucket in buckets:
            for val in bucket:
                arr[i] = val
                i += 1


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max


def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i-1
        temp = arr[i]

        while j >= 0 and temp < arr[j]:
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [75]:
```
sorting_functions['Bucket Sort (#1 - with Insertion Sort)'] = (bucket_sort, {
    'insertion_sort': insertion_sort,
    'minmax': minmax
})
```

Kilka testów

In [76]:
```
# test_sort(bucket_sort, range_=(-1000, 10000), val_counts=(0, 10_000), samples=1000, failed_only=True)
# test_sort(bucket_sort, range_=(-1000, 1000), val_counts=(0, 10_000), samples=1000, ints_only=False, failed_o
test_sort(bucket_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #2 (z pomocą Counting Sorta)

(Wersja dla liczb CAŁKOWITYCH NIEUJEMNYCH)

Ta implementacja tak w rzeczywistości jest alternatywą dla zwykłego Counting Sorta, ponieważ zamiast komórek, które wskazują na liczbę wartości niewiększych niż wartość równa indeksowi danej kolumny, stosujemy wiaderka, w których od razu umieszczamy liczby (nie inkrementujemy licznika). Ta implementacja używa list Pythonowych. Implementacja na listach odsyłaczowych znajduje się w pliku z algorytmami sortowania listy odsyłaczowych. (Oczywiście ten, jak i poniższy algorytm sortowania, jest niewydajny pamięciowo, jeżeli sortowana jest tablica bardzo różniących się między sobą wartości).

In [77]:
```python
def bucket_counting_sort(arr):
    if len(arr) > 1:
        max_val = _max(arr)
        # Allocate memory for values (our buckets)
        buckets = [[] for _ in range(max_val + 1)]
        # Store values in the proper buckets
        for val in arr:
            buckets[val].append(val)
        # Rewrite all values from buckets to the initial array
        i = 0
        for bucket in buckets:
            for val in bucket:
                arr[i] = val
                i += 1


def _max(arr):
    if not arr: return None
    max_val = arr[0]
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]
    return max_val
```

Kilka testów

In [78]:
```python
test_sort(bucket_counting_sort, range_=(0, 100), samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #3 (z pomocą Counting Sorta)

(Wersja dla WSZYSTKICH liczb CAŁKOWITYCH)

Ta implementacja tak w rzeczywistości jest alternatywą dla zwykłego Counting Sorta w wersji dla wszystkich liczb całkowitych. Działa analogicznie jak wyżej. Tak naprawdę ten algorytm nie wymaga żadnego sortowania ani zliczania wartości, a jedynie ich włożenie do odpowiednich wiaderek, a następnie ich połączenie.

In [79]:
```python
def bucket_counting_sort(arr):
    if len(arr) > 1:
        min_val, max_val = minmax(arr)
        # Allocate memory for values (our buckets)
        buckets = [[] for _ in range(max_val - min_val + 1)]
        # Store values in the proper buckets
        for val in arr:
            buckets[val - min_val].append(val)
        # Rewrite all values from buckets to the initial array
        i = 0
        for bucket in buckets:
            for val in bucket:
                arr[i] = val
                i += 1


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [80]:
```python
sorting_functions['Bucket Sort (#3 - Counting Sort variation)'] = (bucket_counting_sort, {
    'minmax': minmax
})
```

Kilka testów

In [81]:
```python
test_sort(bucket_counting_sort, samples=100, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #4 (z pomocą Bucket Sorta oraz Insertion Sorta)

(Wersja dla dowolnych liczb rzeczywistych)

Jest to ulepszona wersja Bucket Sorta, która sortuje wiaderka, przy pomocy Bucket Sorta, jeżeli w danym wiaderku znajduje się duża liczba elementów. W poniższej implementacji uznałem, że jeżeli zawartość wiaderka przekracza liczbę równą 24, należy dane wiaderko sortować również Bucket Sortem. W rzeczywistości przy danych wejściowych o rozkładzie jednostajnym nie zauważymy żadnego wzrostu wydajności. Pewne przyspieszenie działania algorytmu zaobserwujemy w przypadku danych, które nie przypominają rozkładu normalnego.

In [82]:
```python
# Use k as a threshold which indicates when to start using Insertion Sort
def bucket_sort(arr, *, k: 'threshold' = 24):
    # If a bucket is small enough, use an Insertion Sort algorithm to
    # sort this bucket
    if len(arr) <= k:
        insertion_sort(arr)
    else:
        _bucket_sort(arr, k)


def _bucket_sort(arr, k):
    # Store the maximum and the minimum value of a bucket
    min_val, max_val = minmax(arr)
    # Sort a bucket if only there is more than one unique value
    if min_val != max_val:
        # Make a threshold k a bit smaller as a number of elements in each
        # bucket can slightly vary and we don't want to make unnecessary
        # recursive calls.
        m = int(2/3 * k)
        # Create buckets
        buckets_count = len(arr) // m + 1
        buckets = [[] for _ in range(buckets_count)]
        val_interval = (max_val - min_val) / buckets_count
        # Distribute values to the proper buckets
        for val in arr:
            # Calculate the bucket's index depending on how much the
            # current value is greater than the lowest one
            bucket_idx = int((val - min_val) / val_interval - .5)
            buckets[bucket_idx].append(val)
        # Sort each bucket separately
        for bucket in buckets:
            # Bucket sort all of the buckets again
            bucket_sort(bucket, k=k)
        # Rewrite sorted values from buckets to the initial array
        i = 0
        for bucket in buckets:
            for val in bucket:
                arr[i] = val
                i += 1


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max


def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i-1
        temp = arr[i]

        while j >= 0 and temp < arr[j]:
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [83]:
```python
sorting_functions['Bucket Sort (#4 - recursive with Bucket Sort)'] = (bucket_sort, {
    'bucket_sort': bucket_sort,
    '_bucket_sort': _bucket_sort,
    'insertion_sort': insertion_sort,
    'minmax': minmax
})
```

Kilka testów

In [84]:
```python
# test_sort(bucket_sort, samples=100, ints_only=False, uniform_distribution=False)
test_sort(bucket_sort, samples=100, range_=(-1000, 1000),
          val_counts=(0, 1000), ints_only=False, uniform_distribution=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #5 (z pomocą Bucket Sorta oraz Insertion Sorta)

(Wersja iteracyjna z użyciem stosu dla dowolnych liczb rzeczywistych)

Jak wyżej, ale nie występuje problem z możliwym przepełnieniem stosu przy zastosowaniu stosu rekurencyjnego dla bardzo dużych danych wejściowych (W rzeczywistości sytuacja taka jest prawie niemożliwa w rekurencyjnej wersji Bucket Sorta przedstawionej wyżej, ponieważ zawsze dzielimy wiaderka na odpowiednią liczbę mniejszych wiaderek, która jest wyznaczana odpowiednio dla zakresu wartości w wiaderku).

In [85]:
```python
# Use k as a threshold which indicates when to start using Insertion Sort
def bucket_sort(arr, *, k: 'threshold' = 24):
    buckets_stack = [arr]
    idx = len(arr) - 1
    while buckets_stack:
        # Get the last bucket from a stack
        bucket = buckets_stack.pop()
        # If it is small enough, sort it and rewrite values to the initial array
        if len(bucket) < k:
            insertion_sort(bucket)
            # Rewrite sorted values from buckets to the initial array
            idx = _rewrite_bucket(arr, idx, bucket)
        # Split a bucket into smaller buckets otherwise
        else:
            min_val, max_val = minmax(bucket)
            # If a bucket consists of all the same values
            if max_val == min_val:
                # Rewrite sorted values from buckets to the initial array
                idx = _rewrite_bucket(arr, idx, bucket)
            else:
                # Add buckets to the stack in order to sort them later
                for bucket in _divide_into_buckets(bucket, min_val, max_val, k):
                    buckets_stack.append(bucket)


def _rewrite_bucket(arr, idx, bucket):
    for i in range(len(bucket)-1, -1, -1):
        arr[idx] = bucket[i]
        idx -= 1
    return idx


def _divide_into_buckets(bucket, min_val, max_val, k):
    m = int(2/3 * k)
    buckets_count = len(bucket) // m + 1
    buckets = [[] for _ in range(buckets_count)]
    val_interval = (max_val - min_val) / buckets_count
    # Distribute values to the proper buckets
    for val in bucket:
        # Calculate the bucket's index depending on how much the
        # current value is greater than the lowest one
        bucket_idx = int((val - min_val) / val_interval - .5)
        buckets[bucket_idx].append(val)
    return buckets


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max


def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i-1
        temp = arr[i]

        while j >= 0 and temp < arr[j]:
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [86]:
```python
sorting_functions['Bucket Sort (#5 - iterative with Bucket Sort)'] = (bucket_sort, {
    'insertion_sort': insertion_sort,
    'minmax': minmax,
    '_rewrite_bucket': _rewrite_bucket,
    '_divide_into_buckets': _divide_into_buckets
})
```

Kilka testów

In [87]:
```python
test_sort(bucket_sort, samples=100_000, ints_only=False, uniform_distribution=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100000/100000
```

Implementacja algorytmu #6 (z pomocą Quick Sorta oraz Insertion Sorta)

(Algorytm combo, który najpierw dzieli liczby na wiaderka, a następnie duże wiaderka sortuje Quick Sortem, a małe Insertion Sortem)

(Takie podejście pozwala nam zniwelować ryzyko utworzenia ogromnej liczby zbędnych wiaderek, jak to było możliwe w przypadku Bucket Sorta, który korzystał z Bucket Sorta dla dużych wiaderek i jednocześnie zachować najlepszą złożoność czasową równą $O(n)$ - w przypadku rozkładu jednostajnego oraz $O(n \cdot log(n))$ - w najgorszym przypadku (zazwyczaj i tak część wiaderek zostanie posortowana Insertion Sortem w czasie stałym, bo mają one określoną, niezależną od n wielkość)).

In [88]:
```python
def combo_sort(arr, *, k: 'threshold' = 24):
    if len(arr) <= k:
        insertion_sort(arr)
    else:
        # Create buckets
        m = int(2/3 * k)
        buckets_count = len(arr) // m + 1
        buckets = [[] for _ in range(buckets_count)]
        min_val, max_val = minmax(arr)
        val_interval = (max_val - min_val) / buckets_count
        # Distribute values to the proper buckets
        for val in arr:
            # Calculate the bucket's index depending on how much the
            # current value is greater than the lowest one
            bucket_idx = int((val - min_val) / val_interval - .5)  # Round down in order not to overflow an array
            buckets[bucket_idx].append(val)
        # Sort each bucket separately
        for bucket in buckets:
            if len(bucket) < k:
                insertion_sort(bucket)
            else:
                quick_sort(bucket)
        # Rewrite sorted values from buckets to the initial array
        i = 0
        for bucket in buckets:
            for val in bucket:
                arr[i] = val
                i += 1


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max


def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i-1
        temp = arr[i]

        while j >= 0 and temp < arr[j]:
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp


def quick_sort(arr):
    _quick_sort(arr, 0, len(arr) - 1)


def _quick_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        pivot_position = _partition(arr, left_idx, right_idx)
        if pivot_position - left_idx < right_idx - pivot_position:
            _quick_sort(arr, left_idx, pivot_position)
            left_idx = pivot_position + 1  # I removed a tailing recursion
        else:
            _quick_sort(arr, pivot_position + 1, right_idx)
            right_idx = pivot_position  # I removed a tailing recursion


def _partition(arr, left_idx, right_idx):
    pivot = arr[left_idx]

    # Partition an array into 2 subarrays of elements lower than or
    # equal to a pivot and of elements greater or equal to a pivot
    # (in this partition algorithm pivot isn't placed on a fixed position
    # but can be also swapped like all the remaining values)
    i = left_idx - 1
    j = right_idx + 1
    while True:
        i += 1
        while arr[i] < pivot: i += 1

        j -= 1
        while arr[j] > pivot: j -= 1

        if i < j:
            _swap(arr, i, j)
        else:
            return j  # Return a pivot position after the last swap


def _swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [89]:
```python
sorting_functions['Bucket Sort (#6 - Quick Sort & Insertion Sort)'] = (combo_sort, {
    'insertion_sort': insertion_sort,
    'quick_sort': quick_sort,
    '_quick_sort': _quick_sort,
    '_partition': _partition,
    '_swap': _swap,
    'minmax': minmax
})
```

Kilka testów

In [90]:
```python
# test_sort(combo_sort, samples=10000, range_=(-10_000, 10_000), val_counts=(0, 10_000),
#           ints_only=False, failed_only=True, uniform_distribution=False)
test_sort(combo_sort, samples=100, ints_only=False, uniform_distribution=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Implementacja algorytmu #7 (z pomocą Quicker Sorta oraz Insertion Sorta)

(Algorytm combo, który najpierw dzieli liczby na wiaderka, a następnie duże wiaderka sortuje Quicker Sortem, a małe Insertion Sortem)

(Podobnie jak wyżej, ale powinien działać szybciej, jeżeli dużo identycznych liczb trafi do tego samego wiaderka (Zazwyczaj wzrost wydajności widać od razu dla liczb całkowitych. W przypadku liczb rzeczywistych jest mniejsze prawdopodobieństwo wystąpienia kilku takich samych wartości)).

In [91]:
```python
def combo_sort(arr, *, k: 'threshold' = 24):
    if len(arr) <= k:
        insertion_sort(arr)
    else:
        # Create buckets
        m = int(2/3 * k)
        buckets_count = len(arr) // m + 1
        buckets = [[] for _ in range(buckets_count)]
        min_val, max_val = minmax(arr)
        val_interval = (max_val - min_val) / buckets_count
        # Distribute values to the proper buckets
        for val in arr:
            # Calculate the bucket's index depending on how much the
            # current value is greater than the lowest one
            bucket_idx = int((val - min_val) / val_interval - .5)  # Round down in order not to overflow an array
            buckets[bucket_idx].append(val)
        # Sort each bucket separately
        for bucket in buckets:
            if len(bucket) < k:
                insertion_sort(bucket)
            else:
                quicker_sort(bucket)
        # Rewrite sorted values from buckets to the initial array
        i = 0
        for bucket in buckets:
            for val in bucket:
                arr[i] = val
                i += 1


def minmax(arr):
    global_min = global_max = arr[-1]

    for i in range(0, len(arr)-1, 2):
        if arr[i] > arr[i+1]:
            if arr[i] > global_max:   global_max = arr[i]
            if arr[i+1] < global_min: global_min = arr[i+1]
        else:
            if arr[i+1] > global_max: global_max = arr[i+1]
            if arr[i] < global_min:   global_min = arr[i]
    return global_min, global_max


def insertion_sort(arr):
    for i in range(1, len(arr)):
        j = i-1
        temp = arr[i]

        while j >= 0 and temp < arr[j]:
            arr[j+1] = arr[j]
            j -= 1

        arr[j+1] = temp


def quicker_sort(arr):
    _quicker_sort(arr, 0, len(arr)-1)


def _quicker_sort(arr, left_idx, right_idx):
    while left_idx < right_idx:
        lt_pivot_last, gt_pivot_first = _partition(arr, left_idx, right_idx)
        # If a number of elements lower than a pivot is greater than a number
        # of elements greater than a pivot, sort recursively the shorter part
        # of elements which are greater than a pivot
        if lt_pivot_last - left_idx > right_idx - gt_pivot_first:
            _quicker_sort(arr, gt_pivot_first, right_idx)
            right_idx = lt_pivot_last  # I removed a tailing recursion
        # Otherwise, sort a subarray of elements lower than a pivot first
        # as it is shorter than a subarray of elements greater than a pivot
        else:
            _quicker_sort(arr, left_idx, lt_pivot_last)
            left_idx = gt_pivot_first  # I removed a tailing recursion


def _partition(arr, left_idx, right_idx):
    pivot = arr[left_idx]

    # Partition an array into 3 subarrays (lower than, equal to and
    # greater than a pivot value)
    i = left_idx      # A pointer of the first pivot
    j = left_idx + 1  # A pointer of the element after the last pivot
    k = left_idx + 1  # A pointer of the currently checked element
    while k <= right_idx:
        if arr[k] < pivot:
            _two_swaps(arr, i, j, k)
            i += 1
            j += 1
        elif arr[k] == pivot:
            _one_swap(arr, j, k)
            j += 1
        k += 1

    # Return the last index of the subarray of elements lower than a pivot
    # and the first index of the subarray of elements greater than a pivot
    return i - 1, j


def _one_swap(arr, i, j):
    # Swap two elements in an array
    arr[i], arr[j] = arr[j], arr[i]


def _two_swaps(arr, i, j, k):
    # Rotate right the elements of the indices specified (i <- j <- k)
    arr[i], arr[j], arr[k] = arr[k], arr[i], arr[j]
```

Zapisujemy funkcję do późniejszego benchmarku (nie jest to częścią algorytmu)

In [92]:
```python
sorting_functions['Bucket Sort (#7 - Quicker Sort & Insertion Sort)'] = (combo_sort, {
    'insertion_sort': insertion_sort,
    'quicker_sort': quicker_sort,
    '_quicker_sort': _quicker_sort,
    '_partition': _partition,
    '_one_swap': _one_swap,
    '_two_swaps': _two_swaps,
    'minmax': minmax
})
```

Kilka testów

In [93]:
```python
# test_sort(combo_sort, samples=10000, range_=(-10_000, 10_000), val_counts=(0, 10_000),
#           ints_only=False, failed_only=True, uniform_distribution=False)
test_sort(combo_sort, samples=100, ints_only=False, failed_only=True)
```

```
Sorting algorithm is correct
Passed tests in total: 100/100
```

Porównanie szybkości

Dodamy jeszcze dla porównania wbudowaną w Pythona funkcję sortującą

(ponieważ nie możemy dodać metody .sort(), bo funkcja porównująca szybkość sortowań akceptuje tylko funkcje jako argument, użyjemy funkcji sorted, która zwraca posortowaną tablicę, zamiast sortować ją w miejscu)

Jak łatwo będzie można zauważyć, wbudowana w Pythona funkcja sortująca bije na głowę każdy inny algorytm, jaki napisaliśmy. Nie wynika to z faktu, że nasze algorytmy są złe, ale z tego, że Python jest napisany w języku kompilowanym C (przynajmniej jego najbardziej popularna odmiana, czyli tzw. CPython, z którego korzystam), a więc wszystkie wbudowane w niego elementy są dużo szybsze od tych, które możemy sami napisać. Jeżeli piszemy algorytm sortujący, Python, jako, że jest językiem interpretowanym, musi interpretować na bieżąco nasz kod, na co schodzi dużo czasu. Jeżeli użyjemy gotowych rozwiązań, które są dostarczane wraz z językiem, Python nie musi interpretować kodu, sprawdzać składni funkcji sortującej, badać wyjątków, bo jest ona przetestowana i skompilowana, a więc przetłumaczona na kod maszynowy, który nie musi być na bieżąco interpretowany i tłumaczony, jak ma to miejsce w przypadku naszych programów.

In [94]:
```python
sorting_functions['Sorted (Python\'s built-in)'] = (sorted, {})
```

In [95]:
```python
for i, (name, value) in enumerate(sorting_functions.items()):
    print(f'[{i}] {name}')
```

```
[0] Bubble Sort
[1] Selection Sort
[2] Insertion Sort
[3] Merge Sort (#1)
[4] Merge Sort (#2)
[5] Merge Sort (#3)
[6] Heap Sort (#1)
[7] Heap Sort (#2)
[8] Quick Sort (#1 - Lomuto partition)
[9] Quick Sort (#2 - Hoare's partition)
[10] Quick Sort (#3 - random pivot)
[11] Quick Sort (#4 - O(log n) memory)
[12] Quick Sort (#5 - static stack)
[13] Quick Sort (#6 - dynamic stack)
[14] Quick Sort (#7 - median of medians pivot)
[15] Quick Sort (#8 - own sorting function)
[16] Quicker Sort (#1 - recursive)
[17] Quicker Sort (#2 - median of medians pivot)
[18] Counting Sort (#4 - negative & positive)
[19] Radix Sort (#2 - negative & positive)
[20] Bucket Sort (#1 - with Insertion Sort)
[21] Bucket Sort (#3 - Counting Sort variation)
[22] Bucket Sort (#4 - recursive with Bucket Sort)
[23] Bucket Sort (#5 - iterative with Bucket Sort)
[24] Bucket Sort (#6 - Quick Sort & Insertion Sort)
[25] Bucket Sort (#7 - Quicker Sort & Insertion Sort)
[26] Sorted (Python's built-in)
```

In [96]:
```python
def choose(*numbers):  # If no number is specified, all functions will be compared
    result = []
    numbers = set(numbers)
    for i, item in enumerate(sorting_functions.items()):
        if i in numbers or not numbers:
            result.append(item)

    return result
```

Tylko dla liczb całkowitych

Rozkład jednostajny - dużo liczb z małego zakresu - porównanie wszystkich algorytmów

Można zauważyć, że w przypadku wielu liczb $n = O(10000)$, gdzie maksymalnie mamy 2000 różnych liczb, algorytmy sortowania liniowego działają bardzo szybko i zbliżają się do wbudowanej funkcji sortującej w Pythona. (W kolejnym teście algorytmy sortowania liniowego vs wbudowana funkcja w Pythona).

In [97]:
```python
compare_performance(choose(),
                    samples=100,
                    val_counts=(0, 10_000),
                    range_=(-1_000, 1_000),
                    progress_interval=25,
                    uniform_distribution=True,
                    ints_only=True
                    )
```

```
===== Results after 25 tests: =====
Sorted (Python's built-in)                           Total (in seconds): 0.0150    Average: 0.0006
Bucket Sort (#3 - Counting Sort variation)           Total (in seconds): 0.0470    Average: 0.0019    (3.13x slower)
Counting Sort (#4 - negative & positive)             Total (in seconds): 0.0720    Average: 0.0029    (4.80x slower)
Radix Sort (#2 - negative & positive)                Total (in seconds): 0.1220    Average: 0.0049    (8.14x slower)
Bucket Sort (#1 - with Insertion Sort)               Total (in seconds): 0.1300    Average: 0.0052    (8.67x slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)     Total (in seconds): 0.1321    Average: 0.0053    (8.81x slower)
Bucket Sort (#4 - recursive with Bucket Sort)        Total (in seconds): 0.1350    Average: 0.0054    (9.00x slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)       Total (in seconds): 0.1360    Average: 0.0054    (9.07x slower)
Bucket Sort (#5 - iterative with Bucket Sort)        Total (in seconds): 0.1420    Average: 0.0057    (9.47x slower)
Quick Sort (#2 - Hoare's partition)                  Total (in seconds): 0.3270    Average: 0.0131    (21.81x slower)
Quick Sort (#1 - Lomuto partition)                   Total (in seconds): 0.3420    Average: 0.0137    (22.81x slower)
Merge Sort (#3)                                      Total (in seconds): 0.4379    Average: 0.0175    (29.21x slower)
Quick Sort (#3 - random pivot)                       Total (in seconds): 0.4430    Average: 0.0177    (29.55x slower)
Merge Sort (#1)                                      Total (in seconds): 0.4450    Average: 0.0178    (29.68x slower)
Quick Sort (#8 - own sorting function)               Total (in seconds): 0.4560    Average: 0.0182    (30.41x slower)
Quick Sort (#4 - O(log n) memory)                    Total (in seconds): 0.4590    Average: 0.0184    (30.61x slower)
Quicker Sort (#1 - recursive)                        Total (in seconds): 0.4760    Average: 0.0190    (31.75x slower)
Quick Sort (#5 - static stack)                       Total (in seconds): 0.4800    Average: 0.0192    (32.01x slower)
Quick Sort (#6 - dynamic stack)                      Total (in seconds): 0.4800    Average: 0.0192    (32.01x slower)
Merge Sort (#2)                                      Total (in seconds): 0.5460    Average: 0.0218    (36.42x slower)
Heap Sort (#2)                                       Total (in seconds): 0.8200    Average: 0.0328    (54.69x slower)
Quicker Sort (#2 - median of medians pivot)          Total (in seconds): 1.2260    Average: 0.0490    (81.77x slower)
Heap Sort (#1)                                       Total (in seconds): 1.2771    Average: 0.0511    (85.17x slower)
Quick Sort (#7 - median of medians pivot)            Total (in seconds): 1.4220    Average: 0.0569    (94.84x slower)
Selection Sort                                       Total (in seconds): 28.8290   Average: 1.1532    (1922.65x slower)
Insertion Sort                                       Total (in seconds): 35.8170   Average: 1.4327    (2388.69x slower)
Bubble Sort                                          Total (in seconds): 75.7979   Average: 3.0319    (5055.01x slower)

===== Results after 50 tests: =====
Sorted (Python's built-in)                           Total (in seconds): 0.0340    Average: 0.0007
Bucket Sort (#3 - Counting Sort variation)           Total (in seconds): 0.1110    Average: 0.0022    (3.26x slower)
Counting Sort (#4 - negative & positive)             Total (in seconds): 0.1440    Average: 0.0029    (4.23x slower)
Radix Sort (#2 - negative & positive)                Total (in seconds): 0.2540    Average: 0.0051    (7.47x slower)
Bucket Sort (#1 - with Insertion Sort)               Total (in seconds): 0.2680    Average: 0.0054    (7.88x slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)     Total (in seconds): 0.2681    Average: 0.0054    (7.88x slower)
Bucket Sort (#4 - recursive with Bucket Sort)        Total (in seconds): 0.2790    Average: 0.0056    (8.20x slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)       Total (in seconds): 0.2850    Average: 0.0057    (8.38x slower)
Bucket Sort (#5 - iterative with Bucket Sort)        Total (in seconds): 0.2880    Average: 0.0058    (8.47x slower)
Quick Sort (#2 - Hoare's partition)                  Total (in seconds): 0.6740    Average: 0.0135    (19.81x slower)
Quick Sort (#1 - Lomuto partition)                   Total (in seconds): 0.7051    Average: 0.0141    (20.72x slower)
Merge Sort (#3)                                      Total (in seconds): 0.9048    Average: 0.0181    (26.60x slower)
Quick Sort (#3 - random pivot)                       Total (in seconds): 0.9110    Average: 0.0182    (26.78x slower)
Merge Sort (#1)                                      Total (in seconds): 0.9160    Average: 0.0183    (26.92x slower)
Quick Sort (#8 - own sorting function)               Total (in seconds): 0.9400    Average: 0.0188    (27.63x slower)
Quick Sort (#4 - O(log n) memory)                    Total (in seconds): 0.9480    Average: 0.0190    (27.86x slower)
Quicker Sort (#1 - recursive)                        Total (in seconds): 0.9710    Average: 0.0194    (28.54x slower)
Quick Sort (#5 - static stack)                       Total (in seconds): 0.9850    Average: 0.0197    (28.95x slower)
Quick Sort (#6 - dynamic stack)                      Total (in seconds): 0.9890    Average: 0.0198    (29.07x slower)
Merge Sort (#2)                                      Total (in seconds): 1.1281    Average: 0.0226    (33.16x slower)
Heap Sort (#2)                                       Total (in seconds): 1.6930    Average: 0.0339    (49.76x slower)
Quicker Sort (#2 - median of medians pivot)          Total (in seconds): 2.5280    Average: 0.0506    (74.30x slower)
Heap Sort (#1)                                       Total (in seconds): 2.6241    Average: 0.0525    (77.13x slower)
Quick Sort (#7 - median of medians pivot)            Total (in seconds): 2.9320    Average: 0.0586    (86.18x slower)
Selection Sort                                       Total (in seconds): 59.1110   Average: 1.1822    (1737.41x slower)
Insertion Sort                                       Total (in seconds): 73.5390   Average: 1.4708    (2161.48x slower)
Bubble Sort                                          Total (in seconds): 155.4959  Average: 3.1099    (4570.37x slower)

===== Results after 75 tests: =====
Sorted (Python's built-in)                           Total (in seconds): 0.0470    Average: 0.0006
Bucket Sort (#3 - Counting Sort variation)           Total (in seconds): 0.1580    Average: 0.0021    (3.36x slower)
Counting Sort (#4 - negative & positive)             Total (in seconds): 0.2220    Average: 0.0030    (4.72x slower)
Radix Sort (#2 - negative & positive)                Total (in seconds): 0.3810    Average: 0.0051    (8.10x slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)     Total (in seconds): 0.4050    Average: 0.0054    (8.61x slower)
Bucket Sort (#1 - with Insertion Sort)               Total (in seconds): 0.4080    Average: 0.0054    (8.68x slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)       Total (in seconds): 0.4220    Average: 0.0056    (8.97x slower)
Bucket Sort (#5 - iterative with Bucket Sort)        Total (in seconds): 0.4330    Average: 0.0058    (9.21x slower)
Bucket Sort (#4 - recursive with Bucket Sort)        Total (in seconds): 0.4351    Average: 0.0058    (9.25x slower)
Quick Sort (#2 - Hoare's partition)                  Total (in seconds): 1.0189    Average: 0.0136    (21.67x slower)
Quick Sort (#1 - Lomuto partition)                   Total (in seconds): 1.0721    Average: 0.0143    (22.80x slower)
Merge Sort (#3)                                      Total (in seconds): 1.3678    Average: 0.0182    (29.09x slower)
Quick Sort (#3 - random pivot)                       Total (in seconds): 1.3840    Average: 0.0185    (29.43x slower)
Merge Sort (#1)                                      Total (in seconds): 1.3890    Average: 0.0185    (29.54x slower)
Quick Sort (#8 - own sorting function)               Total (in seconds): 1.4300    Average: 0.0191    (30.41x slower)
Quick Sort (#4 - O(log n) memory)                    Total (in seconds): 1.4341    Average: 0.0191    (30.50x slower)
Quicker Sort (#1 - recursive)                        Total (in seconds): 1.4661    Average: 0.0195    (31.18x slower)
Quick Sort (#5 - static stack)                       Total (in seconds): 1.4910    Average: 0.0199    (31.71x slower)
Quick Sort (#6 - dynamic stack)                      Total (in seconds): 1.4970    Average: 0.0200    (31.83x slower)
Merge Sort (#2)                                      Total (in seconds): 1.7131    Average: 0.0228    (36.43x slower)
Heap Sort (#2)                                       Total (in seconds): 2.5659    Average: 0.0342    (54.57x slower)
Quicker Sort (#2 - median of medians pivot)          Total (in seconds): 3.8220    Average: 0.0510    (81.28x slower)
Heap Sort (#1)                                       Total (in seconds): 3.9731    Average: 0.0530    (84.49x slower)
Quick Sort (#7 - median of medians pivot)            Total (in seconds): 4.4440    Average: 0.0593    (94.50x slower)
Selection Sort                                       Total (in seconds): 90.3001   Average: 1.2040    (1920.29x slower)
Insertion Sort                                       Total (in seconds): 112.1810  Average: 1.4957    (2385.60x slower)
Bubble Sort                                          Total (in seconds): 237.8589  Average: 3.1715    (5058.22x slower)

===== Results after 100 tests: =====
Sorted (Python's built-in)                           Total (in seconds): 0.0580    Average: 0.0006
Bucket Sort (#3 - Counting Sort variation)           Total (in seconds): 0.1939    Average: 0.0019    (3.34x slower)
Counting Sort (#4 - negative & positive)             Total (in seconds): 0.2820    Average: 0.0028    (4.86x slower)
Radix Sort (#2 - negative & positive)                Total (in seconds): 0.4860    Average: 0.0049    (8.37x slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)     Total (in seconds): 0.5211    Average: 0.0052    (8.98x slower)
Bucket Sort (#1 - with Insertion Sort)               Total (in seconds): 0.5230    Average: 0.0052    (9.01x slower)
Bucket Sort (#4 - recursive with Bucket Sort)        Total (in seconds): 0.5380    Average: 0.0054    (9.27x slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)       Total (in seconds): 0.5500    Average: 0.0055    (9.48x slower)
Bucket Sort (#5 - iterative with Bucket Sort)        Total (in seconds): 0.5551    Average: 0.0056    (9.56x slower)
Quick Sort (#2 - Hoare's partition)                  Total (in seconds): 1.3039    Average: 0.0130    (22.47x slower)
Quick Sort (#1 - Lomuto partition)                   Total (in seconds): 1.3492    Average: 0.0135    (23.25x slower)
Merge Sort (#3)                                      Total (in seconds): 1.7307    Average: 0.0173    (29.82x slower)
Merge Sort (#1)                                      Total (in seconds): 1.7560    Average: 0.0176    (30.26x slower)
Quick Sort (#3 - random pivot)                       Total (in seconds): 1.7560    Average: 0.0176    (30.26x slower)
Quick Sort (#8 - own sorting function)               Total (in seconds): 1.8009    Average: 0.0180    (31.03x slower)
Quick Sort (#4 - O(log n) memory)                    Total (in seconds): 1.8101    Average: 0.0181    (31.19x slower)
Quicker Sort (#1 - recursive)                        Total (in seconds): 1.8640    Average: 0.0186    (32.12x slower)
Quick Sort (#5 - static stack)                       Total (in seconds): 1.8940    Average: 0.0189    (32.63x slower)
Quick Sort (#6 - dynamic stack)                      Total (in seconds): 1.8970    Average: 0.0190    (32.69x slower)
Merge Sort (#2)                                      Total (in seconds): 2.1751    Average: 0.0218    (37.48x slower)
Heap Sort (#2)                                       Total (in seconds): 3.2559    Average: 0.0326    (56.10x slower)
Quicker Sort (#2 - median of medians pivot)          Total (in seconds): 4.8461    Average: 0.0485    (83.50x slower)
Heap Sort (#1)                                       Total (in seconds): 5.0292    Average: 0.0503    (86.66x slower)
Quick Sort (#7 - median of medians pivot)            Total (in seconds): 5.6250    Average: 0.0563    (96.92x slower)
Selection Sort                                       Total (in seconds): 112.2361  Average: 1.1224    (1933.90x slower)
Insertion Sort                                       Total (in seconds): 139.7330  Average: 1.3973    (2407.69x slower)
Bubble Sort                                          Total (in seconds): 295.5109  Average: 2.9551    (5091.85x slower)

===== Final results: =====
Tests performed: 100

Total times (in seconds):
        Sorted (Python's built-in)                         :   0.0580361
        Bucket Sort (#3 - Counting Sort variation)         :   0.1939428
        Counting Sort (#4 - negative & positive)           :   0.2819991
        Radix Sort (#2 - negative & positive)              :   0.4860096
        Bucket Sort (#7 - Quicker Sort & Insertion Sort)   :   0.5210695
        Bucket Sort (#1 - with Insertion Sort)             :   0.5229938
        Bucket Sort (#4 - recursive with Bucket Sort)      :   0.5379734
        Bucket Sort (#6 - Quick Sort & Insertion Sort)     :   0.5499506
        Bucket Sort (#5 - iterative with Bucket Sort)      :   0.5550594
        Quick Sort (#2 - Hoare's partition)                :   1.3038993
        Quick Sort (#1 - Lomuto partition)                 :   1.3491526
        Merge Sort (#3)                                    :   1.7307475
        Merge Sort (#1)                                    :   1.7559979
        Quick Sort (#3 - random pivot)                     :   1.7560017
        Quick Sort (#8 - own sorting function)             :   1.8009441
        Quick Sort (#4 - O(log n) memory)                  :   1.8100715
        Quicker Sort (#1 - recursive)                      :   1.8640399
        Quick Sort (#5 - static stack)                     :   1.8939662
        Quick Sort (#6 - dynamic stack)                    :   1.8969815
        Merge Sort (#2)                                    :   2.1750751
        Heap Sort (#2)                                     :   3.2559373
        Quicker Sort (#2 - median of medians pivot)        :   4.8460846
        Heap Sort (#1)                                     :   5.0292068
        Quick Sort (#7 - median of medians pivot)          :   5.6250215
        Selection Sort                                     : 112.2361290
        Insertion Sort                                     : 139.7329614
        Bubble Sort                                        : 295.5108597

Average times (in seconds):
        Sorted (Python's built-in)                         :   0.0005804
        Bucket Sort (#3 - Counting Sort variation)         :   0.0019394   (3.34x slower)
        Counting Sort (#4 - negative & positive)           :   0.0028200   (4.86x slower)
        Radix Sort (#2 - negative & positive)              :   0.0048601   (8.37x slower)
        Bucket Sort (#7 - Quicker Sort & Insertion Sort)   :   0.0052107   (8.98x slower)
        Bucket Sort (#1 - with Insertion Sort)             :   0.0052299   (9.01x slower)
        Bucket Sort (#4 - recursive with Bucket Sort)      :   0.0053797   (9.27x slower)
        Bucket Sort (#6 - Quick Sort & Insertion Sort)     :   0.0054995   (9.48x slower)
        Bucket Sort (#5 - iterative with Bucket Sort)      :   0.0055506   (9.56x slower)
        Quick Sort (#2 - Hoare's partition)                :   0.0130390   (22.47x slower)
        Quick Sort (#1 - Lomuto partition)                 :   0.0134915   (23.25x slower)
        Merge Sort (#3)                                    :   0.0173075   (29.82x slower)
        Merge Sort (#1)                                    :   0.0175600   (30.26x slower)
        Quick Sort (#3 - random pivot)                     :   0.0175600   (30.26x slower)
        Quick Sort (#8 - own sorting function)             :   0.0180094   (31.03x slower)
        Quick Sort (#4 - O(log n) memory)                  :   0.0181007   (31.19x slower)
        Quicker Sort (#1 - recursive)                      :   0.0186404   (32.12x slower)
        Quick Sort (#5 - static stack)                     :   0.0189397   (32.63x slower)
        Quick Sort (#6 - dynamic stack)                    :   0.0189698   (32.69x slower)
        Merge Sort (#2)                                    :   0.0217508   (37.48x slower)
        Heap Sort (#2)                                     :   0.0325594   (56.10x slower)
        Quicker Sort (#2 - median of medians pivot)        :   0.0484608   (83.50x slower)
        Heap Sort (#1)                                     :   0.0502921   (86.66x slower)
        Quick Sort (#7 - median of medians pivot)          :   0.0562502   (96.92x slower)
        Selection Sort                                     :   1.1223613   (1933.90x slower)
        Insertion Sort                                     :   1.3973296   (2407.69x slower)
        Bubble Sort                                        :   2.9551086   (5091.85x slower)
```

Rozkład jednostajny - bardzo dużo liczb z małego zakresu - porównanie sortowań liniowych i wbudowanego

In [97]:

```
compare_performance(choose(*range(18, 27)),
                    samples=40,
                    val_counts=(0, 1_000_000),
                    range_=(-1_000, 1_000),
                    progress_interval=10,
                    uniform_distribution=True,
                    ints_only=True
                   )
```

```
===== Results after 10 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.8499    Average: 0.0850
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 1.8820    Average: 0.1882        (2.21x
slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 3.0080    Average: 0.3008        (3.54x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 3.2549    Average: 0.3255        (3.83x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 3.6081    Average: 0.3608        (4.25x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 4.1410    Average: 0.4141        (4.87x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 4.8810    Average: 0.4881        (5.74x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 5.3080    Average: 0.5308        (6.25x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 12.4200   Average: 1.2420        (14.61x
slower)

===== Results after 20 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 1.7069    Average: 0.0853
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 3.8450    Average: 0.1922        (2.25x
slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 6.0661    Average: 0.3033        (3.55x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 6.6069    Average: 0.3303        (3.87x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 7.2780    Average: 0.3639        (4.26x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 8.3590    Average: 0.4180        (4.90x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 9.8580    Average: 0.4929        (5.78x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 10.7190   Average: 0.5360        (6.28x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 24.9500   Average: 1.2475        (14.62x
slower)

===== Results after 30 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 2.4059    Average: 0.0802
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 5.4540    Average: 0.1818        (2.27x
slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 8.5920    Average: 0.2864        (3.57x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 9.3809    Average: 0.3127        (3.90x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 10.3300   Average: 0.3443        (4.29x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 11.8290   Average: 0.3943        (4.92x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 13.9779   Average: 0.4659        (5.81x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 15.2330   Average: 0.5078        (6.33x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 35.1681   Average: 1.1723        (14.62x
slower)

===== Results after 40 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 3.2549    Average: 0.0814
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 7.3019    Average: 0.1825        (2.24x
slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 11.5870   Average: 0.2897        (3.56x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 12.6279   Average: 0.3157        (3.88x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 13.8040   Average: 0.3451        (4.24x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 15.8401   Average: 0.3960        (4.87x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 18.7649   Average: 0.4691        (5.77x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 20.3790   Average: 0.5095        (6.26x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 47.1220   Average: 1.1781        (14.48x
slower)

===== Final results: =====
Tests performed: 40

Total times (in seconds):
        Sorted (Python's built-in)                        :   3.2548690
        Bucket Sort (#3 - Counting Sort variation)        :   7.3019497
        Counting Sort (#4 - negative & positive)          :   11.5870423
        Bucket Sort (#5 - iterative with Bucket Sort)     :   12.6278894
        Bucket Sort (#4 - recursive with Bucket Sort)     :   13.8040257
        Bucket Sort (#1 - with Insertion Sort)            :   15.8400776
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   18.7649372
        Radix Sort (#2 - negative & positive)             :   20.3789907
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   47.1220288

Average times (in seconds):
        Sorted (Python's built-in)                        :   0.0813717
        Bucket Sort (#3 - Counting Sort variation)        :   0.1825487   (2.24x slower)
        Counting Sort (#4 - negative & positive)          :   0.2896761   (3.56x slower)
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.3156972   (3.88x slower)
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.3451006   (4.24x slower)
        Bucket Sort (#1 - with Insertion Sort)            :   0.3960019   (4.87x slower)
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   0.4691234   (5.77x slower)
        Radix Sort (#2 - negative & positive)             :   0.5094748   (6.26x slower)
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   1.1780507   (14.48x slower)
```

Rozkład niejednostajny - duży zakres liczb - porównanie wszystkich sortowań

In [98]:

```
compare_performance(choose(),
                    samples=100,
                    val_counts=(0, 500),
                    range_=(-500_000, 500_000),
                    progress_interval=25,
                    uniform_distribution=False,
                    ints_only=True
                   )
```

```
===== Results after 25 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0000    Average: 0.0000
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.0070    Average: 0.0003
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.0090    Average: 0.0004
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 0.0110    Average: 0.0004
Merge Sort (#3)                                   Total (in seconds): 0.0110    Average: 0.0004
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.0120    Average: 0.0005
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 0.0130    Average: 0.0005
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.0140    Average: 0.0006
Radix Sort (#2 - negative & positive)             Total (in seconds): 0.0150    Average: 0.0006
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 0.0160    Average: 0.0006
Quicker Sort (#1 - recursive)                     Total (in seconds): 0.0170    Average: 0.0007
Merge Sort (#1)                                   Total (in seconds): 0.0180    Average: 0.0007
Quick Sort (#3 - random pivot)                    Total (in seconds): 0.0200    Average: 0.0008
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 0.0200    Average: 0.0008
Quick Sort (#5 - static stack)                    Total (in seconds): 0.0200    Average: 0.0008
Quick Sort (#8 - own sorting function)            Total (in seconds): 0.0210    Average: 0.0008
Merge Sort (#2)                                   Total (in seconds): 0.0220    Average: 0.0009
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 0.0230    Average: 0.0009
Heap Sort (#2)                                    Total (in seconds): 0.0300    Average: 0.0012
Heap Sort (#1)                                    Total (in seconds): 0.0430    Average: 0.0017
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 0.0440    Average: 0.0018
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 0.0480    Average: 0.0019
Selection Sort                                    Total (in seconds): 0.0780    Average: 0.0031
Insertion Sort                                    Total (in seconds): 0.0790    Average: 0.0032
Bubble Sort                                       Total (in seconds): 0.1790    Average: 0.0072
Counting Sort (#4 - negative & positive)          Total (in seconds): 2.7649    Average: 0.1106
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 8.6281    Average: 0.3451

===== Results after 50 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0010    Average: 0.0000
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.0170    Average: 0.0003        (16.98x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.0170    Average: 0.0003        (17.00x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.0210    Average: 0.0004        (20.98x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 0.0230    Average: 0.0005        (22.99x
slower)
Merge Sort (#3)                                   Total (in seconds): 0.0240    Average: 0.0005        (23.98x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 0.0270    Average: 0.0005        (26.99x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.0270    Average: 0.0005        (27.02x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 0.0310    Average: 0.0006        (30.99x
slower)
Merge Sort (#1)                                   Total (in seconds): 0.0320    Average: 0.0006        (32.01x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 0.0370    Average: 0.0007        (36.97x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 0.0370    Average: 0.0007        (36.98x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 0.0370    Average: 0.0007        (36.99x
slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 0.0380    Average: 0.0008        (38.00x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 0.0390    Average: 0.0008        (39.01x
slower)
Merge Sort (#2)                                   Total (in seconds): 0.0390    Average: 0.0008        (39.04x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 0.0400    Average: 0.0008        (40.00x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 0.0430    Average: 0.0009        (43.04x
slower)
Heap Sort (#2)                                    Total (in seconds): 0.0520    Average: 0.0010        (51.98x
slower)
Heap Sort (#1)                                    Total (in seconds): 0.0820    Average: 0.0016        (82.03x
slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 0.0821    Average: 0.0016        (82.07x
slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 0.0870    Average: 0.0017        (87.00x
slower)
Selection Sort                                    Total (in seconds): 0.1390    Average: 0.0028        (139.00
x slower)
Insertion Sort                                    Total (in seconds): 0.1540    Average: 0.0031        (153.98
x slower)
Bubble Sort                                       Total (in seconds): 0.3310    Average: 0.0066        (331.00
x slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 5.4229    Average: 0.1085        (5423.2
9x slower)
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 16.8302   Average: 0.3366        (16831.
38x slower)

===== Results after 75 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0020    Average: 0.0000
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.0250    Average: 0.0003        (12.48x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.0260    Average: 0.0003        (12.99x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.0260    Average: 0.0003        (13.03x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.0330    Average: 0.0004        (16.52x
slower)
Merge Sort (#3)                                   Total (in seconds): 0.0350    Average: 0.0005        (17.50x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 0.0360    Average: 0.0005        (17.99x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 0.0360    Average: 0.0005        (18.00x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 0.0430    Average: 0.0006        (21.50x
slower)
Merge Sort (#1)                                   Total (in seconds): 0.0460    Average: 0.0006        (23.01x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 0.0520    Average: 0.0007        (26.01x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 0.0530    Average: 0.0007        (26.49x
slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 0.0530    Average: 0.0007        (26.51x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 0.0540    Average: 0.0007        (27.02x
slower)
Merge Sort (#2)                                   Total (in seconds): 0.0540    Average: 0.0007        (27.03x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 0.0550    Average: 0.0007        (27.50x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 0.0580    Average: 0.0008        (29.03x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 0.0590    Average: 0.0008        (29.51x
slower)
Heap Sort (#2)                                    Total (in seconds): 0.0730    Average: 0.0010        (36.51x
slower)
Heap Sort (#1)                                    Total (in seconds): 0.1160    Average: 0.0015        (58.03x
slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 0.1171    Average: 0.0016        (58.55x
slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 0.1250    Average: 0.0017        (62.52x
slower)
Selection Sort                                    Total (in seconds): 0.1940    Average: 0.0026        (97.04x
slower)
Insertion Sort                                    Total (in seconds): 0.2180    Average: 0.0029        (109.03
x slower)
Bubble Sort                                       Total (in seconds): 0.4669    Average: 0.0062        (233.57
x slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 7.9840    Average: 0.1065        (3993.6
9x slower)
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 25.0392   Average: 0.3339        (12524.
97x slower)

===== Results after 100 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0030    Average: 0.0000
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.0310    Average: 0.0003        (10.32x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.0320    Average: 0.0003        (10.68x
slower)
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.0330    Average: 0.0003        (10.99x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.0431    Average: 0.0004        (14.35x
slower)
Merge Sort (#3)                                   Total (in seconds): 0.0440    Average: 0.0004        (14.66x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 0.0440    Average: 0.0004        (14.67x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 0.0470    Average: 0.0005        (15.66x
slower)
Radix Sort (#2 - negative & positive)             Total (in seconds): 0.0540    Average: 0.0005        (18.00x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 0.0580    Average: 0.0006        (19.32x
slower)
Merge Sort (#1)                                   Total (in seconds): 0.0590    Average: 0.0006        (19.67x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 0.0640    Average: 0.0006        (21.35x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 0.0660    Average: 0.0007        (22.00x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 0.0690    Average: 0.0007        (23.00x
slower)
Merge Sort (#2)                                   Total (in seconds): 0.0690    Average: 0.0007        (23.01x
slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 0.0700    Average: 0.0007        (23.34x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 0.0730    Average: 0.0007        (24.35x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 0.0770    Average: 0.0008        (25.67x
slower)
Heap Sort (#2)                                    Total (in seconds): 0.0920    Average: 0.0009        (30.67x
slower)
Heap Sort (#1)                                    Total (in seconds): 0.1470    Average: 0.0015        (49.00x
slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 0.1490    Average: 0.0015        (49.69x
slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 0.1570    Average: 0.0016        (52.33x
slower)
Selection Sort                                    Total (in seconds): 0.2420    Average: 0.0024        (80.68x
slower)
Insertion Sort                                    Total (in seconds): 0.2650    Average: 0.0026        (88.34x
slower)
Bubble Sort                                       Total (in seconds): 0.5730    Average: 0.0057        (191.03
x slower)
Counting Sort (#4 - negative & positive)          Total (in seconds): 10.6540   Average: 0.1065        (3552.1
5x slower)
Bucket Sort (#3 - Counting Sort variation)        Total (in seconds): 33.5162   Average: 0.3352        (11174.
66x slower)

===== Final results: =====
Tests performed: 100

Total times (in seconds):
        Sorted (Python's built-in)                        :   0.0029993
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   0.0309546
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.0320361
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.0329733
        Quick Sort (#2 - Hoare's partition)               :   0.0430512
        Merge Sort (#3)                                   :   0.0439816
        Quick Sort (#1 - Lomuto parition)                 :   0.0439880
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   0.0469677
        Radix Sort (#2 - negative & positive)             :   0.0539873
        Bucket Sort (#1 - with Insertion Sort)            :   0.0579596
        Merge Sort (#1)                                   :   0.0590055
        Quick Sort (#8 - own sorting function)            :   0.0640242
        Quick Sort (#4 - O(log n) memory)                 :   0.0659893
        Quick Sort (#3 - random pivot)                    :   0.0689771
        Merge Sort (#2)                                   :   0.0690229
        Quicker Sort (#1 - recursive)                     :   0.0700030
        Quick Sort (#6 - dynamic stack)                   :   0.0730195
        Quick Sort (#5 - static stack)                    :   0.0770018
        Heap Sort (#2)                                    :   0.0919795
        Heap Sort (#1)                                    :   0.1469779
        Quicker Sort (#2 - median of medians pivot)       :   0.1490431
        Quick Sort (#7 - median of medians pivot)         :   0.1569571
        Selection Sort                                    :   0.2419922
        Insertion Sort                                    :   0.2649670
        Bubble Sort                                       :   0.5729666
        Counting Sort (#4 - negative & positive)          :   10.6539798
        Bucket Sort (#3 - Counting Sort variation)        :   33.5162218

Average times (in seconds):
        Sorted (Python's built-in)                        :   0.0000300
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   0.0003095   (10.32x slower)
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.0003204   (10.68x slower)
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.0003297   (10.99x slower)
        Quick Sort (#2 - Hoare's partition)               :   0.0004305   (14.35x slower)
        Merge Sort (#3)                                   :   0.0004398   (14.66x slower)
        Quick Sort (#1 - Lomuto parition)                 :   0.0004399   (14.67x slower)
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   0.0004697   (15.66x slower)
        Radix Sort (#2 - negative & positive)             :   0.0005399   (18.00x slower)
        Bucket Sort (#1 - with Insertion Sort)            :   0.0005796   (19.32x slower)
        Merge Sort (#1)                                   :   0.0005901   (19.67x slower)
        Quick Sort (#8 - own sorting function)            :   0.0006402   (21.35x slower)
        Quick Sort (#4 - O(log n) memory)                 :   0.0006599   (22.00x slower)
        Quick Sort (#3 - random pivot)                    :   0.0006898   (23.00x slower)
        Merge Sort (#2)                                   :   0.0006902   (23.01x slower)
        Quicker Sort (#1 - recursive)                     :   0.0007000   (23.34x slower)
        Quick Sort (#6 - dynamic stack)                   :   0.0007302   (24.35x slower)
        Quick Sort (#5 - static stack)                    :   0.0007700   (25.67x slower)
        Heap Sort (#2)                                    :   0.0009198   (30.67x slower)
        Heap Sort (#1)                                    :   0.0014698   (49.00x slower)
        Quicker Sort (#2 - median of medians pivot)       :   0.0014904   (49.69x slower)
        Quick Sort (#7 - median of medians pivot)         :   0.0015696   (52.33x slower)
        Selection Sort                                    :   0.0024199   (80.68x slower)
        Insertion Sort                                    :   0.0026497   (88.34x slower)
        Bubble Sort                                       :   0.0057297   (191.03x slower)
        Counting Sort (#4 - negative & positive)          :   0.1065398   (3552.15x slower)
        Bucket Sort (#3 - Counting Sort variation)        :   0.3351622   (11174.66x slower)
```

Dla dowolnych liczb wymiernych (zmiennoprzecinkowych)

Odrzucamy algorytm Counting Sort, który się nie nadaje do niczego w przypadku liczb zmiennoprzecinkowych, a także wersję Bucket Sorta, która na nim bazuje. Odrzucamy również Radix Sorta, choć można go zmodyfikować tak, by część dziesiętną traktował jak osobną liczbę i sobie wówczas poradzi z sortowaniem liczb zmiennoprzecinkowych. Poza tymi algorytmami, porównujemy wszystkie pozostałe.

Rozkład niejednostajny - duży zakres liczb - wszystkie bez odrzuconych

In [101…

```
compare_performance(choose(*(set(range(0, 27)) - {18, 19, 21})),
                    samples=100,
                    val_counts=(0, 10_000),
                    range_=(-1_000_000_000, 1_000_000_000),
                    progress_interval=25,
                    uniform_distribution=False,
                    ints_only=False
                   )
```

```
===== Results after 25 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0100    Average: 0.0004
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.2020    Average: 0.0081        (20.26x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.2160    Average: 0.0086        (21.67x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.2840    Average: 0.0114        (28.48x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.2969    Average: 0.0119        (29.79x
slower)
Merge Sort (#1)                                   Total (in seconds): 0.3820    Average: 0.0153        (38.32x
slower)
Merge Sort (#3)                                   Total (in seconds): 0.3860    Average: 0.0154        (38.72x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 0.4200    Average: 0.0168        (42.13x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 0.4280    Average: 0.0171        (42.94x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 0.4520    Average: 0.0181        (45.35x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 0.4580    Average: 0.0183        (45.94x
slower)
Merge Sort (#2)                                   Total (in seconds): 0.4800    Average: 0.0192        (48.15x
slower)
Heap Sort (#2)                                    Total (in seconds): 0.7360    Average: 0.0294        (73.83x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 0.7380    Average: 0.0295        (74.03x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 0.7850    Average: 0.0314        (78.74x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 0.9740    Average: 0.0390        (97.70x
slower)
Heap Sort (#1)                                    Total (in seconds): 1.1280    Average: 0.0451        (113.16
x slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 1.1710    Average: 0.0468        (117.47
x slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 1.2510    Average: 0.0500        (125.49
x slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 1.2820    Average: 0.0513        (128.60
x slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 17.6636   Average: 0.7065        (1771.9
0x slower)
Selection Sort                                    Total (in seconds): 25.5451   Average: 1.0218        (2562.5
1x slower)
Insertion Sort                                    Total (in seconds): 32.9886   Average: 1.3195        (3309.2
0x slower)
Bubble Sort                                       Total (in seconds): 67.8160   Average: 2.7126        (6802.8
5x slower)

===== Results after 50 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0180    Average: 0.0004
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.3280    Average: 0.0066        (18.27x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.3520    Average: 0.0070        (19.60x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.4490    Average: 0.0090        (25.00x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.5109    Average: 0.0102        (28.46x
slower)
Merge Sort (#1)                                   Total (in seconds): 0.6590    Average: 0.0132        (36.70x
slower)
Merge Sort (#3)                                   Total (in seconds): 0.6610    Average: 0.0132        (36.82x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 0.7300    Average: 0.0146        (40.66x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 0.7450    Average: 0.0149        (41.49x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 0.7800    Average: 0.0156        (43.44x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 0.7970    Average: 0.0159        (44.39x
slower)
Merge Sort (#2)                                   Total (in seconds): 0.8309    Average: 0.0166        (46.28x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 1.0729    Average: 0.0215        (59.76x
slower)
Heap Sort (#2)                                    Total (in seconds): 1.2700    Average: 0.0254        (70.73x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 1.2919    Average: 0.0258        (71.95x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 1.6050    Average: 0.0321        (89.39x
slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 1.7870    Average: 0.0357        (99.53x
slower)
Heap Sort (#1)                                    Total (in seconds): 1.9560    Average: 0.0391        (108.94
x slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 2.1550    Average: 0.0431        (120.02
x slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 2.2060    Average: 0.0441        (122.86
x slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 23.0015   Average: 0.4600        (1281.0
4x slower)
Selection Sort                                    Total (in seconds): 41.3501   Average: 0.8270        (2302.9
5x slower)
Insertion Sort                                    Total (in seconds): 53.2736   Average: 1.0655        (2967.0
1x slower)
Bubble Sort                                       Total (in seconds): 109.7300  Average: 2.1946        (6111.2
9x slower)

===== Results after 75 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0280    Average: 0.0004
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.4890    Average: 0.0065        (17.49x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.5200    Average: 0.0069        (18.60x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.6470    Average: 0.0086        (23.14x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 0.7580    Average: 0.0101        (27.11x
slower)
Merge Sort (#1)                                   Total (in seconds): 0.9731    Average: 0.0130        (34.81x
slower)
Merge Sort (#3)                                   Total (in seconds): 0.9770    Average: 0.0130        (34.95x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 1.0820    Average: 0.0144        (38.70x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 1.1101    Average: 0.0148        (39.71x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 1.1650    Average: 0.0155        (41.67x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 1.1820    Average: 0.0158        (42.28x
slower)
Merge Sort (#2)                                   Total (in seconds): 1.2320    Average: 0.0164        (44.07x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 1.5630    Average: 0.0208        (55.91x
slower)
Heap Sort (#2)                                    Total (in seconds): 1.8770    Average: 0.0250        (67.14x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 1.8789    Average: 0.0251        (67.21x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 2.2980    Average: 0.0306        (82.20x
slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 2.5970    Average: 0.0346        (92.90x
slower)
Heap Sort (#1)                                    Total (in seconds): 2.8770    Average: 0.0384        (102.91
x slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 3.2040    Average: 0.0427        (114.61
x slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 3.2750    Average: 0.0437        (117.15
x slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 31.7915   Average: 0.4239        (1137.2
[illegible]x slower)
Selection Sort                                    Total (in seconds): 58.3672   Average: 0.7782        (2087.8
[illegible]x slower)
Insertion Sort                                    Total (in seconds): 75.3986   Average: 1.0053        (2697.0
[illegible]x slower)
Bubble Sort                                       Total (in seconds): 156.7970  Average: 2.0840        (5537.2
[illegible]x slower)

===== Results after 100 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.0400    Average: 0.0004
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 0.7020    Average: 0.0070        (17.56x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 0.7460    Average: 0.0075        (18.66x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 0.9420    Average: 0.0094        (23.57x
slower)
Quick Sort (#2 - Hoare's partition)               Total (in seconds): 1.0699    Average: 0.0107        (26.77x
slower)
Quick Sort (#3 - random pivot)                    Total (in seconds): 1.3731    Average: 0.0137        (34.35x
slower)
Merge Sort (#1)                                   Total (in seconds): 1.3740    Average: 0.0137        (34.37x
slower)
Merge Sort (#3)                                   Total (in seconds): 1.5270    Average: 0.0153        (38.20x
slower)
Quick Sort (#4 - O(log n) memory)                 Total (in seconds): 1.5600    Average: 0.0156        (39.04x
slower)
Quick Sort (#5 - static stack)                    Total (in seconds): 1.6420    Average: 0.0164        (41.08x
slower)
Quick Sort (#6 - dynamic stack)                   Total (in seconds): 1.6600    Average: 0.0166        (41.53x
slower)
Merge Sort (#2)                                   Total (in seconds): 1.7390    Average: 0.0174        (43.50x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 2.2550    Average: 0.0225        (56.41x
slower)
Heap Sort (#2)                                    Total (in seconds): 2.6600    Average: 0.0266        (66.55x
slower)
Quick Sort (#1 - Lomuto parition)                 Total (in seconds): 2.8319    Average: 0.0283        (70.85x
slower)
Quick Sort (#8 - own sorting function)            Total (in seconds): 3.4671    Average: 0.0347        (86.74x
slower)
Quicker Sort (#1 - recursive)                     Total (in seconds): 3.7390    Average: 0.0374        (93.54x
slower)
Heap Sort (#1)                                    Total (in seconds): 4.0570    Average: 0.0406        (101.49
x slower)
Quicker Sort (#2 - median of medians pivot)       Total (in seconds): 4.5250    Average: 0.0452        (113.20
x slower)
Quick Sort (#7 - median of medians pivot)         Total (in seconds): 4.6210    Average: 0.0462        (115.60
x slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 50.3596   Average: 0.5036        (1259.8
5x slower)
Selection Sort                                    Total (in seconds): 85.0911   Average: 0.8509        (2128.7
3x slower)
Insertion Sort                                    Total (in seconds): 109.3706  Average: 1.0937        (2736.1
3x slower)
Bubble Sort                                       Total (in seconds): 225.3290  Average: 2.2533        (5637.0
6x slower)

===== Final results: =====
Tests performed: 100

Total times (in seconds):
        Sorted (Python's built-in)                        :   0.0399728
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.7020175
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.7460265
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   0.9420044
        Quick Sort (#2 - Hoare's partition)               :   1.0699444
        Quick Sort (#3 - random pivot)                    :   1.3730850
        Merge Sort (#1)                                   :   1.3739896
        Merge Sort (#3)                                   :   1.5269797
        Quick Sort (#4 - O(log n) memory)                 :   1.5600491
        Quick Sort (#5 - static stack)                    :   1.6419985
        Quick Sort (#6 - dynamic stack)                   :   1.6599944
        Merge Sort (#2)                                   :   1.7390027
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   2.2549646
        Heap Sort (#2)                                    :   2.6599936
        Quick Sort (#1 - Lomuto parition)                 :   2.8318846
        Quick Sort (#8 - own sorting function)            :   3.4670665
        Quicker Sort (#1 - recursive)                     :   3.7389727
        Heap Sort (#1)                                    :   4.0570300
        Quicker Sort (#2 - median of medians pivot)       :   4.5249920
        Quick Sort (#7 - median of medians pivot)         :   4.6210113
        Bucket Sort (#1 - with Insertion Sort)            :   50.3595541
        Selection Sort                                    :   85.0911145
        Insertion Sort                                    :   109.3706236
        Bubble Sort                                       :   225.3289876

Average times (in seconds):
        Sorted (Python's built-in)                        :   0.0003997
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.0070202   (17.56x slower)
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.0074603   (18.66x slower)
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   0.0094200   (23.57x slower)
        Quick Sort (#2 - Hoare's partition)               :   0.0106994   (26.77x slower)
        Quick Sort (#3 - random pivot)                    :   0.0137309   (34.35x slower)
        Merge Sort (#1)                                   :   0.0137399   (34.37x slower)
        Merge Sort (#3)                                   :   0.0152698   (38.20x slower)
        Quick Sort (#4 - O(log n) memory)                 :   0.0156005   (39.03x slower)
        Quick Sort (#5 - static stack)                    :   0.0164200   (41.08x slower)
        Quick Sort (#6 - dynamic stack)                   :   0.0165999   (41.53x slower)
        Merge Sort (#2)                                   :   0.0173900   (43.50x slower)
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   0.0225496   (56.41x slower)
        Heap Sort (#2)                                    :   0.0265999   (66.55x slower)
        Quick Sort (#1 - Lomuto parition)                 :   0.0283188   (70.85x slower)
        Quick Sort (#8 - own sorting function)            :   0.0346707   (86.74x slower)
        Quicker Sort (#1 - recursive)                     :   0.0373897   (93.54x slower)
        Heap Sort (#1)                                    :   0.0405703   (101.49x slower)
        Quicker Sort (#2 - median of medians pivot)       :   0.0452499   (113.20x slower)
        Quick Sort (#7 - median of medians pivot)         :   0.0462101   (115.60x slower)
        Bucket Sort (#1 - with Insertion Sort)            :   0.5035955   (1259.85x slower)
        Selection Sort                                    :   0.8509111   (2128.73x slower)
        Insertion Sort                                    :   1.0937062   (2736.13x slower)
        Bubble Sort                                       :   2.2532899   (5637.06x slower)
```

Rozkład jednostajny - duży zakres liczb - tylko liniowe (bez odrzuconych) i wbudowana

In [102…

```
compare_performance(choose(*(set(range(20, 27)) - {21})),
                    samples=100,
                    val_counts=(0, 100_000),
                    range_=(-100_000_000_000_000, 100_000_000_000_000),
                    progress_interval=25,
                    uniform_distribution=True,
                    ints_only=False
                   )
```

```
===== Results after 25 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.1500    Average: 0.0060
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 1.3410    Average: 0.0536        (8.94x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 1.3640    Average: 0.0546        (9.10x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 1.3765    Average: 0.0551        (9.18x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 1.3781    Average: 0.0551        (9.19x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 1.4209    Average: 0.0568        (9.47x
slower)

===== Results after 50 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.2910    Average: 0.0058
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 2.5639    Average: 0.0513        (8.81x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 2.6125    Average: 0.0523        (8.98x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 2.6170    Average: 0.0523        (8.99x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 2.6192    Average: 0.0524        (9.00x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 2.7048    Average: 0.0541        (9.30x
slower)

===== Results after 75 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.4350    Average: 0.0058
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 3.8489    Average: 0.0513        (8.85x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 3.8875    Average: 0.0518        (8.94x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 3.9349    Average: 0.0525        (9.05x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 3.9803    Average: 0.0531        (9.15x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 4.0318    Average: 0.0538        (9.27x
slower)

===== Results after 100 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 0.6279    Average: 0.0063
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 5.5420    Average: 0.0554        (8.83x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 5.5574    Average: 0.0556        (8.85x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 5.6609    Average: 0.0566        (9.02x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 5.6932    Average: 0.0569        (9.07x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 5.7458    Average: 0.0575        (9.15x
slower)

===== Final results: =====
Tests performed: 100

Total times (in seconds):
        Sorted (Python's built-in)                        :   0.6279166
        Bucket Sort (#5 - iterative with Bucket Sort)     :   5.5419843
        Bucket Sort (#4 - recursive with Bucket Sort)     :   5.5573978
        Bucket Sort (#1 - with Insertion Sort)            :   5.6608813
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   5.6932232
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   5.7457874

Average times (in seconds):
        Sorted (Python's built-in)                        :   0.0062792
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.0554198   (8.83x slower)
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.0555740   (8.85x slower)
        Bucket Sort (#1 - with Insertion Sort)            :   0.0566088   (9.02x slower)
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   0.0569322   (9.07x slower)
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   0.0574579   (9.15x slower)
```

Rozkład jednostajny - bardzo mały zakres liczb - tylko liniowe (bez odrzuconych) i wbudowana

In [103…

```
compare_performance(choose(*(set(range(20, 27)) - {21})),
                    samples=100,
                    val_counts=(0, 1_000_000),
                    range_=(-100, 100),
                    progress_interval=25,
                    uniform_distribution=True,
                    ints_only=False
                   )
```

```
===== Results after 25 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 2.8860    Average: 0.1154
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 17.5510   Average: 0.7020        (6.08x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 18.5520   Average: 0.7421        (6.43x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 18.6769   Average: 0.7471        (6.47x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 18.8218   Average: 0.7529        (6.52x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 18.8944   Average: 0.7558        (6.55x
slower)

===== Results after 50 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 4.9240    Average: 0.0985
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 30.5130   Average: 0.6103        (6.20x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 32.0650   Average: 0.6413        (6.51x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 32.5458   Average: 0.6509        (6.61x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 32.6594   Average: 0.6532        (6.63x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 32.6799   Average: 0.6536        (6.64x
slower)

===== Results after 75 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 7.2900    Average: 0.0972
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 45.4990   Average: 0.6067        (6.24x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 48.0960   Average: 0.6413        (6.60x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 48.5657   Average: 0.6475        (6.66x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 48.7228   Average: 0.6496        (6.68x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 48.7375   Average: 0.6498        (6.69x
slower)

===== Results after 100 tests: =====
Sorted (Python's built-in)                        Total (in seconds): 10.0810   Average: 0.1008
Bucket Sort (#5 - iterative with Bucket Sort)     Total (in seconds): 62.3699   Average: 0.6237        (6.19x
slower)
Bucket Sort (#4 - recursive with Bucket Sort)     Total (in seconds): 65.6050   Average: 0.6561        (6.51x
slower)
Bucket Sort (#7 - Quicker Sort & Insertion Sort)  Total (in seconds): 66.0768   Average: 0.6608        (6.55x
slower)
Bucket Sort (#6 - Quick Sort & Insertion Sort)    Total (in seconds): 66.1306   Average: 0.6613        (6.56x
slower)
Bucket Sort (#1 - with Insertion Sort)            Total (in seconds): 66.1668   Average: 0.6617        (6.56x
slower)

===== Final results: =====
Tests performed: 100

Total times (in seconds):
        Sorted (Python's built-in)                        :   10.0810065
        Bucket Sort (#5 - iterative with Bucket Sort)     :   62.3699045
        Bucket Sort (#4 - recursive with Bucket Sort)     :   65.6050382
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   66.0767705
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   66.1305652
        Bucket Sort (#1 - with Insertion Sort)            :   66.1667883

Average times (in seconds):
        Sorted (Python's built-in)                        :   0.1008101
        Bucket Sort (#5 - iterative with Bucket Sort)     :   0.6236990   (6.19x slower)
        Bucket Sort (#4 - recursive with Bucket Sort)     :   0.6560504   (6.51x slower)
        Bucket Sort (#7 - Quicker Sort & Insertion Sort):   0.6607677   (6.55x slower)
        Bucket Sort (#6 - Quick Sort & Insertion Sort)    :   0.6613057   (6.56x slower)
        Bucket Sort (#1 - with Insertion Sort)            :   0.6616679   (6.56x slower)
```