Spis treści

Uproszczona implementacja listy jednokierunkowej #1 (obiektowa)

Implementacja jest ograniczona do funkcjonalności niezbędnych do utworzenia i posortowania listy. Bardziej rozbudowana implementacja struktury znajduje się w pliku 'Struktury danych'.

```
class Node:
    def __init__(self, val=None):
        self.val = val
        self.next = None


class LinkedList:
    def __init__(self, values: 'iterable' = None):
        self.head = self.tail = None
        self.length = 0
        values and self.extend(values)  # The same as 'if values: self.extend(values)'

    def __iter__(self):
        curr = self.head
        while curr:
            yield curr.val
            curr = curr.next

    def __str__(self):
        return ' -> '.join(map(str, self))

    def __len__(self):
        return self.length

    def append(self, val: object):
        node = Node(val)
        if not self:
            self.head = self.tail = node
        else:
            self.tail.next = node
            self.tail = node
        self.length += 1

    def extend(self, values: 'iterable'):
        if values:
            iterator = iter(values)
            if not self:
                self.head = self.tail = Node(next(iterator))
            for val in iterator:
                self.tail.next = Node(val)
                self.tail = self.tail.next
            self.length += len(values)
```

Uproszczona implementacja listy jednokierunkowej #2 (funkcyjna)

Implementacja jest ograniczona do funkcjonalności niezbędnych do utworzenia i posortowania listy. Bardziej rozbudowana implementacja struktury znajduje się w pliku 'Struktury danych'.

```
class Node:
    def __init__(self, val=None):
        self.val = val
        self.next = None


def create_linked_list(values: 'iterable' = None) -> 'linked list head (sentinel)':
    head = Node()  # A sentinel node
    if not values: return head
    head.next = curr = Node(values[0])
    for i in range(1, len(values)):
        curr.next = Node(values[i])
        curr = curr.next
    return head


def print_linked_list(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    print(ll_head.val, end=' ')
    while curr:
        print('->', curr.val, end=' ')
        curr = curr.next
    print()


def linked_list_to_list(ll_head: 'linked list head (sentinel)') -> list:
    values = []
    curr = ll_head.next
    while curr:
        values.append(curr.val)
        curr = curr.next
    return values
```

Funkcja do testowania poprawności algorytmu (dla obiektowej wersji)

```
import random

def test_sort_obj(sort_fn, *, samples=20, val_counts=(0, 50), range_=(-100, 100), modifies_list=True, failed_o
    passed = 0
    for i in range(samples):
        values = [random.randint(*range_) for _ in range(random.randint(*val_counts))]
        expected = sorted(values)
        ll = LinkedList(values)
        output = sort_fn(ll)
        if not modifies_list:
            result = list(output)
        else:
            result = list(ll)
        is_correct = result == expected
        passed += is_correct
        if not failed_only or (failed_only and not is_correct):
            print(f'#{i+1} Loop (already passed: {passed}/{i + 1}):')
            print(f'Input values:  {values}')
            print(f'Proper answer: {expected}')
            print(f'After sorting: {result}')
            print(f'Test {"PASSED" if is_correct else "FAILED"}')
            print()
    print(f'Total tests passed: {passed}/{samples}')
    print(f'A sorting algorithm is implemented {"correctly" if passed == samples else "improperly"}')
```

Funkcja do testowania poprawności algorytmu (dla funkcyjnej wersji)

```
import random

def test_sort_func(sort_fn, *, samples=20, val_counts=(0, 50), range_=(-100, 100), modifies_list=True, failed_o
    passed = 0
    for i in range(samples):
        values = [random.randint(*range_) for _ in range(random.randint(*val_counts))]
        expected = sorted(values)
        ll_head = create_linked_list(values)
        output = sort_fn(ll_head)
        if not modifies_list:
            result = output
        result = linked_list_to_list(ll_head)
        is_correct = result == expected
        passed += is_correct
        if not failed_only or (failed_only and not is_correct):
            print(f'#{i+1} Loop (already passed: {passed}/{i + 1}):')
            print(f'Input values:  {values}')
            print(f'Proper answer: {expected}')
            print('After sorting: ', end='')
            print_linked_list(ll_head)
            print(f'Test {"PASSED" if is_correct else "FAILED"}')
            print()
    print(f'Total tests passed: {passed}/{samples}')
    print(f'A sorting algorithm is implemented {"correctly" if passed == samples else "improperly"}')
```

Funkcja do porównywania wydajności algorytmów sortowania

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


def non_uniform_randrange(range_, count, k=.2):
    result = []
    limit = k / (math.log(count) or 1 if count else 1)
    while count:
        while random.random() > limit and count:
            result.append(random.randint(*range_))
            count -= 1
        avg = sum(range_) / 2
        delta = (avg - range_[0]) * k
        range_ = (int(range_[0] + delta - .5), int(range_[1] - delta + .5))
    return result

# 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>, <linked list
                        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 be
                        range_=(-10_000, 10_000), # A range which will be used to create a random list of value
                        progress_interval=10,     # A number of tests that will be performed between showing
                                                  # current sorting times (temporary results). When set to 0 no
                        uniform_distribution=True # If set to false, non-uniform values will be generated using
                        ):
    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 recursion errors

    for i in range(1, samples + 1):
        if uniform_distribution:
            random_lst = [random.randint(*range_) for _ in range(random.randint(*val_counts))]
        else:
            random_lst = non_uniform_randrange(range_, random.randint(*val_counts), random.random() / 3)

        for name, (fn, constructor, globals_) in sorting_functions:
            # continue if a sorting process lead to the recursion error
            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[:]
            # It doesn't matter if a constructor is a class or a function
            ll = constructor(input_lst)
            try:
                start_time = time.time()
                fn(ll)
                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} \tAverage: {
                    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

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

Algorytmy proste

O złożoności ($O(n^2)$)

Insertion Sort

Implementacja algorytmu #1 (dla obiektowej implementacji listy)

```
def insert_node(ll, node):  # Inserts node in a right position maintaining the ascending order
    # Prepend the node
    if node.val < ll.head.val:
        node.next = ll.head
        ll.head = node
    # Insert the node before a greater one
    else:
        curr = ll.head
        while node.val > curr.next.val:
            curr = curr.next
        node.next = curr.next
        curr.next = node


def insertion_sort(ll):
    if len(ll) < 2: return

    prev = ll.head
    while prev.next:  # We start from the second node
        # If a current node (prev.next) has a value lower than a prev node, we have to
        # shift this node to a right position before
        if prev.next.val < prev.val:
            # Remove a current node
            removed = prev.next
            prev.next = prev.next.next
            # Now we insert this node in a right position
            insert_node(ll, removed)
            # We can skip a current node otherwise.
        else:
            prev = prev.next

    # Fix the tail pointer
    ll.tail = prev
```

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

```
sorting_functions['Insertion Sort #1 (objective)'] = (insertion_sort, LinkedList, {
    'insert_node': insert_node
})
```

Kilka testów

```
test_sort_obj(insertion_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #2 (dla funkcyjnej implementacji listy)

```
def insert_node(ll_head: 'linked list head (sentinel)', node):  # Inserts node in a right position maintaining
    # Insert the node before a greater one
    curr = ll_head
    while curr.next.val < node.val:
        curr = curr.next
    node.next = curr.next
    curr.next = node


def insertion_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next or not ll_head.next.next: return

    prev = ll_head.next
    while prev.next:  # We start from the second node
        # If a current node (prev.next) has a value lower than a prev node, we have to
        # shift this node to a right position before
        if prev.next.val < prev.val:
            # Remove a current node
            removed = prev.next
            prev.next = prev.next.next
            # Now we insert this node in a right position
            insert_node(ll_head, removed)
            # We can skip a current node otherwise.
        else:
            prev = prev.next
```

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

```
sorting_functions['Insertion Sort #2 (functional)'] = (insertion_sort, create_linked_list, {
    'insert_node': insert_node
})
```

Kilka testów

```
test_sort_func(insertion_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Selection Sort

Implementacja algorytmu #1 (dla obiektowej implementacji listy)

Implementacja sortująca poprzez wybieranie dla listy odsyłaczowej tak, aby zamieniane były odpowiednie wskaźniki, jest bardzo skomplikowana. Z tego powodu ograniczymy się do implementacji podmieniającej wartości

```
def selection_sort(ll):
    if len(ll) < 2: return

    first = ll.head

    # Loop while a first pointer is not the last element
    while first.next:
        min_node = first
        second = first.next
        while second:
            # Look for a currently minimal value
            if second.val < min_node.val:
                min_node = second
            second = second.next
        # Swap values in the first node and in the minimal node
        min_node.val, first.val = first.val, min_node.val
        first = first.next
```

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

```
sorting_functions['Selection Sort #1 (objective)'] = (selection_sort, LinkedList, {})
```

Kilka testów

```
test_sort_obj(selection_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #2 (dla funkcyjnej implementacji listy)

j. w.

```
def selection_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next or not ll_head.next.next: return

    first = ll_head.next

    # Loop while a first pointer is not the last element
    while first.next:
        min_node = first
        second = first.next
        while second:
            # Look for a currently minimal value
            if second.val < min_node.val:
                min_node = second
            second = second.next
        # Swap values in the first node and in the minimal node
        min_node.val, first.val = first.val, min_node.val
        first = first.next
```

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

```
sorting_functions['Selection Sort #2 (functional)'] = (selection_sort, create_linked_list, {})
```

Kilka testów

```
test_sort_func(selection_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Bubble Sort

Implementacja algorytmu #1 (dla obiektowej implementacji listy)

```
def swap_nodes(first, second, third):
    second.next = third.next
    third.next = second
    first.next = third


def bubble_sort(ll):
    if len(ll) < 2: return
    tail_node = limit = None  # Use a limit pointer to reduce a number of loops
    # Add a sentinel node
    sentinel = Node()
    sentinel.next = ll.head
    ll.head = sentinel
    # Loop over a Linked List and modify pointers (if necessary) in order to swap nodes
    while limit is not sentinel.next.next:
        prev = ll.head
        while prev.next.next is not limit:
            if prev.next.next.val < prev.next.val:
                swap_nodes(prev, prev.next, prev.next.next)
            prev = prev.next
        limit = prev.next
        if not tail_node:
            tail_node = limit
    # Remove a sentinel node and fix the pointers
    ll.head = ll.head.next
    ll.tail = tail_node
```

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

```
sorting_functions['Bubble Sort #1 (objective)'] = (bubble_sort, LinkedList, {
    'swap_nodes': swap_nodes
})
```

Kilka testów

```
test_sort_obj(bubble_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #2 (dla funkcyjnej implementacji listy)

```
def swap_nodes(first, second, third):
    second.next = third.next
    third.next = second
    first.next = third


def bubble_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next or not ll_head.next.next: return
    limit = None  # Use a limit pointer to reduce a number of loops
    # Loop over a Linked List and modify pointers (if necessary) in order to swap nodes
    while limit is not ll_head.next.next:
        prev = ll_head
        while prev.next.next is not limit:
            if prev.next.next.val < prev.next.val:
                swap_nodes(prev, prev.next, prev.next.next)
            prev = prev.next
        limit = prev.next
```

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

```
sorting_functions['Bubble Sort #2 (functional)'] = (bubble_sort, create_linked_list, {
    'swap_nodes': swap_nodes
})
```

Kilka testów

```
test_sort_func(bubble_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Algorytmy szybkie

O złożoności ($O(nlogn)$)

Merge Sort

Implementacja algorytmu #1 (dla obiektowej implementacji listy)

Wersja rekurencyjna modyfikująca listę źródłową z rekurencyjnym łączeniem (funkcja merge) - NAJGORSZA implementacja (duże ryzyko przepełnienia stosu rekurencyjnego)

```
def merge_sort(ll):
    if len(ll) > 1:
        ll.head = _merge_sort_recur(ll.head)
        # Scan a linked list linearly to find the tail pointer
        curr = ll.head
        while curr.next:
            curr = curr.next
        ll.tail = curr


def _merge_sort_recur(begin_ptr):
    # If the 'll' part of a Linked List that is being sorted has no more than
    # 1 element, return this part
    if not begin_ptr or not begin_ptr.next: return begin_ptr

    left_ptr, right_ptr = _split(begin_ptr)
    return _merge(_merge_sort_recur(left_ptr), _merge_sort_recur(right_ptr))


def _split(begin_ptr):
    # Find a place to make a cut (split current 'll' part into to halves)
    cut_ptr = begin_ptr
    end_ptr = begin_ptr.next

    while end_ptr:
        end_ptr = end_ptr.next
        if end_ptr:
            end_ptr = end_ptr.next
            cut_ptr = cut_ptr.next

    # Perform a cutting operation (split into the left and the right part)
    left_ptr = begin_ptr
    right_ptr = cut_ptr.next
    cut_ptr.next = None  # Unlink the right part from the left part

    return left_ptr, right_ptr


def _merge(left_ptr, right_ptr):
    if not left_ptr: return right_ptr
    if not right_ptr: return left_ptr

    if left_ptr.val < right_ptr.val:
        left_ptr.next = _merge(left_ptr.next, right_ptr)
        return left_ptr
    else:
        right_ptr.next = _merge(left_ptr, right_ptr.next)
        return right_ptr
```

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

```
sorting_functions['Merge Sort #1 (objective, recursive)'] = (merge_sort, LinkedList, {
    '_merge_sort_recur': _merge_sort_recur,
    '_split': _split,
    '_merge': _merge
})
```

Kilka testów

```
test_sort_obj(merge_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #2 (dla obiektowej implementacji listy) (NAJLEPSZA)

Wersja rekurencyjna, z NIEREKURENCYJNĄ funkcją merge

```
def merge_sort(ll):
    if len(ll) > 1:
        ll.head = _merge_sort_recur(ll.head)
        # Scan a linked list linearly to find the tail pointer
        curr = ll.head
        while curr.next:
            curr = curr.next
        ll.tail = curr


def _merge_sort_recur(begin_ptr):
    # If the 'll' part of a Linked List that is being sorted has no more than
    # 1 element, return this part
    if not begin_ptr or not begin_ptr.next: return begin_ptr

    left_ptr, right_ptr = _split(begin_ptr)
    return _merge(_merge_sort_recur(left_ptr), _merge_sort_recur(right_ptr))


def _split(begin_ptr):
    # Find a place to make a cut (split current 'll' part into to halves)
    cut_ptr = begin_ptr
    end_ptr = begin_ptr.next

    while end_ptr:
        end_ptr = end_ptr.next
        if end_ptr:
            end_ptr = end_ptr.next
            cut_ptr = cut_ptr.next

    # Perform a cutting operation (split into the left and the right part)
    left_ptr = begin_ptr
    right_ptr = cut_ptr.next
    cut_ptr.next = None  # Unlink the right part from the left part

    return left_ptr, right_ptr


def _merge(ll1_curr, ll2_curr):
    res_head = res_tail = Node(None)  # Add a sentinel node to ease nodes linking

    while ll1_curr and ll2_curr:
        if ll1_curr.val < ll2_curr.val:
            res_tail.next = ll1_curr
            ll1_curr = ll1_curr.next
        else:
            res_tail.next = ll2_curr
            ll2_curr = ll2_curr.next
        res_tail = res_tail.next

    # Link the remaining nodes at the end of a result linked list
    if ll1_curr:
        res_tail.next = ll1_curr
    else:
        res_tail.next = ll2_curr

    # Remove a sentinel node
    res_head = res_head.next

    return res_head
```

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

```
sorting_functions['Merge Sort #2 (objective, recursive optimized)'] = (merge_sort, LinkedList, {
    '_merge_sort_recur': _merge_sort_recur,
    '_split': _split,
    '_merge': _merge
})
```

Kilka testów

```
test_sort_obj(merge_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #3 (dla obiektowej implementacji listy)

Wersja iteracyjna, działająca poprzez złączanie serii naturalnych (więcej informacji w zadaniach z 2. ćwiczeń)

```
def merge_sort(ll):
    if len(ll) > 1:
        ll_head = ll.head
        ll_tail = None

        while True:
            new_head = new_tail = None

            while True:
                # Cut off the first part from a linked list
                first_part = ll_head
                ll_head = cut_series(ll_head)

                if ll_head is None:
                    # If a current linked list is empty after a cut and there are
                    # no nodes in a new linked list, we have finished sorting
                    if new_head is None:
                        ll.head = first_part
                        ll.tail = ll_tail
                        return
                    # If we have cut the last part of a current linked list and
                    # there are still some nodes in a new linked list, we have to
                    # link a part cut to a new linked list.
                    else:
                        new_tail.next = first_part
                        break

                # If the inner loop hasn't been broken yet, we can cut off
                # another part from a linked list
                second_part = ll_head
                ll_head = cut_series(ll_head)

                # As we have now two parts, we have to merge them together
                # and link to a new linked list
                merged_head, merged_tail = merged(first_part, second_part)

                if not new_head:
                    new_head = merged_head
                else:
                    new_tail.next = merged_head

                new_tail = merged_tail

            # If a current linked list was exhausted, replace it with a new linked list
            if ll_head is None:
                ll_head = new_head

            ll_tail = new_tail


def merged(ll1_curr, ll2_curr):
    res_head = res_tail = Node(None)  # Add a sentinel node to ease nodes linking

    while ll1_curr and ll2_curr:
        if ll1_curr.val < ll2_curr.val:
            res_tail.next = ll1_curr
            ll1_curr = ll1_curr.next
        else:
            res_tail.next = ll2_curr
            ll2_curr = ll2_curr.next
        res_tail = res_tail.next

    # Link the remaining nodes at the end of a result linked list
    if ll1_curr: res_tail.next = ll1_curr
    else: res_tail.next = ll2_curr

    # Move a tail pointer to the last node
    while res_tail.next:
        res_tail = res_tail.next

    # Remove a sentinel node
    res_head = res_head.next

    return res_head, res_tail


def cut_series(ll_head):
    if not ll_head: return None
    curr = ll_head
    while curr.next and curr.next.val >= curr.val:
        curr = curr.next
    # Return a pointer to the beginning of the second part after split
    remaining = curr.next
    curr.next = None
    return remaining
```

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

```
sorting_functions['Merge Sort #3 (objective, iterative, natural series)'] = (merge_sort, LinkedList, {
    'merged': merged,
    'cut_series': cut_series
})
```

Kilka testów

```
test_sort_obj(merge_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #4 (dla funkcyjnej implementacji listy) (NAJGORSZA)

Wersja rekurencyjna modyfikująca listę źródłową z rekurencyjnym łączeniem list (częste przepełnienie stosu rekurencyjnego)

In [34]:
```python
def merge_sort(ll_head: 'linked list head (sentinel)'):
    ll_head.next = _merge_sort_recur(ll_head.next)


def _merge_sort_recur(begin_ptr):
    # If the 'll' part of a Linked List that is being sorted has no more than
    # 1 element, return this part
    if not begin_ptr or not begin_ptr.next: return begin_ptr

    left_ptr, right_ptr = _split(begin_ptr)
    return _merge(_merge_sort_recur(left_ptr), _merge_sort_recur(right_ptr))


def _split(begin_ptr):
    # Find a place to make a cut (split current 'll' part into halves)
    cut_ptr = begin_ptr
    end_ptr = begin_ptr.next

    while end_ptr:
        end_ptr = end_ptr.next
        if end_ptr:
            end_ptr = end_ptr.next
            cut_ptr = cut_ptr.next

    # Perform a cutting operation (split into the left and the right part)
    left_ptr = begin_ptr
    right_ptr = cut_ptr.next
    cut_ptr.next = None  # Unlink the right part from the left part

    return left_ptr, right_ptr


def _merge(left_ptr, right_ptr):
    if not left_ptr: return right_ptr
    if not right_ptr: return left_ptr

    if left_ptr.val < right_ptr.val:
        left_ptr.next = _merge(left_ptr.next, right_ptr)
        return left_ptr
    else:
        right_ptr.next = _merge(left_ptr, right_ptr.next)
        return right_ptr
```

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

In [35]:
```python
sorting_functions['Merge Sort #4 (functional, recursive)'] = (merge_sort, create_linked_list, {
    '_merge_sort_recur': _merge_sort_recur,
    '_split': _split,
    '_merge': _merge
})
```

Kilka testów

In [36]:
```python
test_sort_func(merge_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #5 (dla funkcyjnej implementacji listy) (NAJLEPSZA)

Wersja rekurencyjna, z iteracyjnym łączeniem list

In [37]:
```python
def merge_sort(ll_head: 'linked list head (sentinel)'):
    ll_head.next = _merge_sort_recur(ll_head.next)


def _merge_sort_recur(begin_ptr):
    # If the 'll' part of a Linked List that is being sorted has no more than
    # 1 element, return this part
    if not begin_ptr or not begin_ptr.next: return begin_ptr

    left_ptr, right_ptr = _split(begin_ptr)
    return _merge(_merge_sort_recur(left_ptr), _merge_sort_recur(right_ptr))


def _split(begin_ptr):
    # Find a place to make a cut (split current 'll' part into to halves)
    cut_ptr = begin_ptr
    end_ptr = begin_ptr.next

    while end_ptr:
        end_ptr = end_ptr.next
        if end_ptr:
            end_ptr = end_ptr.next
            cut_ptr = cut_ptr.next

    # Perform a cutting operation (split into the left and the right part)
    left_ptr = begin_ptr
    right_ptr = cut_ptr.next
    cut_ptr.next = None  # Unlink the right part from the left part

    return left_ptr, right_ptr


def _merge(ll1_curr, ll2_curr):
    res_head = res_tail = Node(None)  # Add a sentinel node to ease nodes linking

    while ll1_curr and ll2_curr:
        if ll1_curr.val < ll2_curr.val:
            res_tail.next = ll1_curr
            ll1_curr = ll1_curr.next
        else:
            res_tail.next = ll2_curr
            ll2_curr = ll2_curr.next
        res_tail = res_tail.next

    # Link the remaining nodes at the end of a result linked list
    if ll1_curr: res_tail.next = ll1_curr
    else: res_tail.next = ll2_curr

    # Move a tail pointer to the last node
    while res_tail.next:
        res_tail = res_tail.next

    # Remove a sentinel node
    res_head = res_head.next

    return res_head
```

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

In [38]:
```python
sorting_functions['Merge Sort #5 (functional, recursive optimized)'] = (merge_sort, create_linked_list, {
    '_merge_sort_recur': _merge_sort_recur,
    '_split': _split,
    '_merge': _merge
})
```

Kilka testów

In [39]:
```python
test_sort_func(merge_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #6 (dla funkcyjnej implementacji listy)

Wersja iteracyjna, działająca poprzez złączanie serii naturalnych (więcej informacji w zadaniach z 2. ćwiczeń)

In [40]:
```python
def merge_sort(ll_head: 'linked list head (sentinel)'):
    while True:
        new_head = new_tail = None

        while True:
            # Cut off the first part from a linked list
            first_part = ll_head.next
            ll_head.next = cut_series(ll_head)

            if not ll_head.next:
                # If a current linked list is empty after a cut and there are
                # no nodes in a new linked list, we have finished sorting
                if not new_head:
                    ll_head.next = first_part
                    return
                # If we have cut the last part of a current linked list and
                # there are still some nodes in a new linked list, we have to
                # link a part cut to a new linked list.
                else:
                    new_tail.next = first_part
                    break

            # If the inner loop hasn't been broken yet, we can cut off
            # another part from a linked list
            second_part = ll_head.next
            ll_head.next = cut_series(ll_head)

            # As we have now two parts, we have to merge them together
            # and link to a new linked list
            merged_head, merged_tail = merged(first_part, second_part)

            if not new_head:
                new_head = merged_head
            else:
                new_tail.next = merged_head

            new_tail = merged_tail

        # If a current linked list was exhausted, replace it with a new linked list
        if ll_head.next is None:
            ll_head.next = new_head


def merged(ll1_curr, ll2_curr):
    res_head = res_tail = Node()  # Add a sentinel node to ease nodes linking

    while ll1_curr and ll2_curr:
        if ll1_curr.val < ll2_curr.val:
            res_tail.next = ll1_curr
            ll1_curr = ll1_curr.next
        else:
            res_tail.next = ll2_curr
            ll2_curr = ll2_curr.next
        res_tail = res_tail.next

    # Link the remaining nodes at the end of a result linked list
    if ll1_curr: res_tail.next = ll1_curr
    else: res_tail.next = ll2_curr

    # Move a tail pointer to the last node
    while res_tail.next:
        res_tail = res_tail.next

    # Remove a sentinel node
    res_head = res_head.next

    return res_head, res_tail


def cut_series(ll_head):
    if not ll_head.next: return None
    curr = ll_head.next
    while curr.next and curr.next.val >= curr.val:
        curr = curr.next
    # Return a pointer to the beginning of the second part after split
    remaining = curr.next
    curr.next = None
    return remaining
```

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

In [41]:
```python
sorting_functions['Merge Sort #6 (functional, iterative, natural series)'] = (merge_sort, create_linked_list, {
    'merged': merged,
    'cut_series': cut_series
})
```

Kilka testów

In [42]:
```python
test_sort_func(merge_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Quick Sort

Implementacja algorytmu #1 (dla obiektowej implementacji listy)

Wersja rekurencyjna modyfikująca listę źródłową

In [43]:
```python
def quick_sort(ll: LinkedList):
    if len(ll) > 1:
        # Add a sentinel node to ease sorting
        sentinel = Node()
        sentinel.next = ll.head
        ll.head = sentinel
        # Perform sorting on a linked list
        _quick_sort(ll.head, None)
        # Remove a sentinel node which was added
        ll.head = ll.head.next
        # Scan a linked list linearly to find the tail pointer
        curr = ll.head
        while curr.next:
            curr = curr.next
        ll.tail = curr


# begin_idx will be included and end_idx excluded (as in Python's ranges)
def _quick_sort(begin_prev_node, end_node):
    # Loop till the current sublist has at least 2 elements
    # (calling a partition function for a single-element list is pointless and inefficient,
    # thus it's better to check two conditions in a while loop)
    while begin_prev_node.next is not end_node \
            and begin_prev_node.next.next is not end_node:
        first_end_node, second_begin_prev_node = _partition(begin_prev_node, end_node)
        _quick_sort(begin_prev_node, first_end_node)
        begin_prev_node = second_begin_prev_node


def _partition(begin_prev_node, end_node):
    # Store a pivot node and a current node pointers in variables
    # Take the first (leftmost) node as a pivot
    pivot_node = begin_prev_node.next
    curr_node = pivot_node.next

    # Prepare sentinel nodes for sublists which will be created
    lt_pivot_head = Node()
    eq_pivot_head = pivot_node
    gt_pivot_head = Node()

    # Prepare pointers to the sublists
    lt_pivot_curr = lt_pivot_head
    eq_pivot_curr = eq_pivot_head
    gt_pivot_curr = gt_pivot_head

    # Distribute subsequent nodes of a linked list part to appropriate sublists
    while curr_node is not end_node:
        if curr_node.val < pivot_node.val:
            lt_pivot_curr.next = curr_node
            lt_pivot_curr = lt_pivot_curr.next
        elif curr_node.val > pivot_node.val:
            gt_pivot_curr.next = curr_node
            gt_pivot_curr = gt_pivot_curr.next
        else:
            eq_pivot_curr.next = curr_node
            eq_pivot_curr = eq_pivot_curr.next
        curr_node = curr_node.next

    # Join created lists together
    # Link a list of elements lower than pivot (lt_pivot) if is not empty
    if lt_pivot_head.next:
        begin_prev_node.next = lt_pivot_head.next
        lt_pivot_curr.next = eq_pivot_head

        if gt_pivot_head.next:
            eq_pivot_curr.next = gt_pivot_head.next
            gt_pivot_curr.next = end_node
        else:
            eq_pivot_curr.next = end_node
    # Link a list of elements greater than pivot (gt_pivot) if is not empty
    elif gt_pivot_head.next:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = gt_pivot_head.next
        gt_pivot_curr.next = end_node
    # Otherwise, there will be only eq_pivot linked list (all elements are equal to a pivot)
    else:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = end_node

    return eq_pivot_head, eq_pivot_curr
```

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

In [44]:
```python
sorting_functions['Quick Sort #1 (objective, recursive)'] = (quick_sort, LinkedList, {
    '_quick_sort': _quick_sort,
    '_partition': _partition
})
```

Kilka testów

In [45]:
```python
# test_sort_obj(quick_sort, samples=10000, failed_only=True)
test_sort_obj(quick_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #2 (dla obiektowej implementacji listy)

Wersja iteracyjna z użyciem stosu, modyfikująca listę źródłową

In [46]:
```python
def quick_sort(ll: LinkedList):
    if len(ll) > 1:
        # Add a sentinel node to ease sorting
        sentinel = Node()
        sentinel.next = ll.head
        # Create stack
        stack = [(sentinel, None)]
        # Perform sorting on a linked list
        while stack:
            # Take the last indices pair out of the stack
            begin_prev_node, end_node = stack.pop()

            first_end_node, first_length, second_begin_prev_node, second_length = _partition(begin_prev_node, e

            if first_length < second_length:
                # Store indices of the longer part at first if has at least 2 elements
                if second_length > 1:
                    stack.append((second_begin_prev_node, end_node))
                    # 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 first_length > 1:
                        stack.append((begin_prev_node, first_end_node))
            else:
                # Store indices of the longer part at first if has at least 2 elements
                if first_length > 1:
                    stack.append((begin_prev_node, first_end_node))
                    # 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 second_length > 1:
                        stack.append((second_begin_prev_node, end_node))
        # Update the head node of a linked list
        ll.head = sentinel.next
        # Scan a linked list linearly to find the tail pointer
        curr = ll.head
        while curr.next:
            curr = curr.next
        ll.tail = curr


def _partition(begin_prev_node, end_node):
    # Store a pivot node and a current node pointers in variables
    # Take the first (leftmost) node as a pivot
    pivot_node = begin_prev_node.next
    curr_node = pivot_node.next

    # Prepare sentinel nodes for sublists which will be created
    lt_pivot_head = Node()
    eq_pivot_head = pivot_node
    gt_pivot_head = Node()

    # Prepare pointers to the sublists
    lt_pivot_curr = lt_pivot_head
    eq_pivot_curr = eq_pivot_head
    gt_pivot_curr = gt_pivot_head

    # Prepare variables to calculate length of the result parts
    first_part_length = second_part_length = 0

    # Distribute subsequent nodes of a linked list part to appropriate sublists
    while curr_node is not end_node:
        if curr_node.val < pivot_node.val:
            lt_pivot_curr.next = curr_node
            lt_pivot_curr = lt_pivot_curr.next
            first_part_length += 1
        elif curr_node.val > pivot_node.val:
            gt_pivot_curr.next = curr_node
            gt_pivot_curr = gt_pivot_curr.next
            second_part_length += 1
        else:
            eq_pivot_curr.next = curr_node
            eq_pivot_curr = eq_pivot_curr.next
        curr_node = curr_node.next

    # Join created lists together
    # Link a list of elements lower than pivot (lt_pivot) if is not empty
    if lt_pivot_head.next:
        begin_prev_node.next = lt_pivot_head.next
        lt_pivot_curr.next = eq_pivot_head

        if gt_pivot_head.next:
            eq_pivot_curr.next = gt_pivot_head.next
            gt_pivot_curr.next = end_node
        else:
            eq_pivot_curr.next = end_node
    # Link a list of elements greater than pivot (gt_pivot) if is not empty
    elif gt_pivot_head.next:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = gt_pivot_head.next
        gt_pivot_curr.next = end_node
    # Otherwise, there will be only eq_pivot linked list (all elements are equal to a pivot)
    else:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = end_node

    return eq_pivot_head, first_part_length, eq_pivot_curr, second_part_length
```

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

In [47]:
```python
sorting_functions['Quick Sort #2 (objective, iterative)'] = (quick_sort, LinkedList, {
    '_partition': _partition
})
```

Kilka testów

In [48]:
```python
test_sort_obj(quick_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #3 (dla funkcyjnej implementacji listy)

Wersja rekurencyjna modyfikująca listę źródłową

In [49]:
```python
def quick_sort(ll_head: 'linked list head (sentinel)'):
    # Perform sorting only if there are at least 2 elements in a linked list
    if ll_head.next and ll_head.next.next:
        # Perform sorting on a linked list
        _quick_sort(ll_head, None)


# begin_idx will be included and end_idx excluded (as in Python's ranges)
def _quick_sort(begin_prev_node, end_node):
    # Loop till the current sublist has at least 2 elements
    # (calling a partition function for a single-element list is pointless and inefficient,
    # thus it's better to check two conditions in a while loop)
    while begin_prev_node.next is not end_node\
            and begin_prev_node.next.next is not end_node:
        first_end_node, second_begin_prev_node = _partition(begin_prev_node, end_node)
        _quick_sort(begin_prev_node, first_end_node)
        begin_prev_node = second_begin_prev_node


def _partition(begin_prev_node, end_node):
    # Store a pivot node and a current node pointers in variables
    # Take the first (leftmost) node as a pivot
    pivot_node = begin_prev_node.next
    curr_node = pivot_node.next

    # Prepare sentinel nodes for sublists which will be created
    lt_pivot_head = Node()
    eq_pivot_head = pivot_node
    gt_pivot_head = Node()

    # Prepare pointers to the sublists
    lt_pivot_curr = lt_pivot_head
    eq_pivot_curr = eq_pivot_head
    gt_pivot_curr = gt_pivot_head

    # Distribute subsequent nodes of a linked list part to appropriate sublists
    while curr_node is not end_node:
        if curr_node.val < pivot_node.val:
            lt_pivot_curr.next = curr_node
            lt_pivot_curr = lt_pivot_curr.next
        elif curr_node.val > pivot_node.val:
            gt_pivot_curr.next = curr_node
            gt_pivot_curr = gt_pivot_curr.next
        else:
            eq_pivot_curr.next = curr_node
            eq_pivot_curr = eq_pivot_curr.next
        curr_node = curr_node.next

    # Join created lists together
    # Link a list of elements lower than pivot (lt_pivot) if is not empty
    if lt_pivot_head.next:
        begin_prev_node.next = lt_pivot_head.next
        lt_pivot_curr.next = eq_pivot_head

        if gt_pivot_head.next:
            eq_pivot_curr.next = gt_pivot_head.next
            gt_pivot_curr.next = end_node
        else:
            eq_pivot_curr.next = end_node
    # Link a list of elements greater than pivot (gt_pivot) if is not empty
    elif gt_pivot_head.next:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = gt_pivot_head.next
        gt_pivot_curr.next = end_node
    # Otherwise, there will be only eq_pivot linked list (all elements are equal to a pivot)
    else:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = end_node

    return eq_pivot_head, eq_pivot_curr
```

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

In [50]:
```python
sorting_functions['Quick Sort #3 (functional, recursive)'] = (quick_sort, create_linked_list, {
    '_quick_sort': _quick_sort,
    '_partition': _partition
})
```

Kilka testów

In [51]:
```python
# test_sort_func(quick_sort, samples=25_000, failed_only=True)
test_sort_func(quick_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #4 (dla funkcyjnej implementacji listy)

Wersja iteracyjna z użyciem stosu, modyfikująca listę źródłową

In [52]:
```python
def quick_sort(ll_head: 'linked list head (sentinel)'):
    # Sort if only there are at least two elements in a linked list
    if ll_head.next and ll_head.next.next:
        # Create stack
        stack = [(ll_head, None)]
        # Perform sorting on a linked list
        while stack:
            # Take the last indices pair out of the stack
            begin_prev_node, end_node = stack.pop()

            first_end_node, first_length, second_begin_prev_node, second_length = _partition(begin_prev_node, e

            if first_length < second_length:
                # Store indices of the longer part at first if has at least 2 elements
                if second_length > 1:
                    stack.append((second_begin_prev_node, end_node))
                    # 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 first_length > 1:
                        stack.append((begin_prev_node, first_end_node))
            else:
                # Store indices of the longer part at first if has at least 2 elements
                if first_length > 1:
                    stack.append((begin_prev_node, first_end_node))
                    # 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 second_length > 1:
                        stack.append((second_begin_prev_node, end_node))


def _partition(begin_prev_node, end_node):
    # Store a pivot node and a current node pointers in variables
    # Take the first (leftmost) node as a pivot
    pivot_node = begin_prev_node.next
    curr_node = pivot_node.next

    # Prepare sentinel nodes for sublists which will be created
    lt_pivot_head = Node()
    eq_pivot_head = pivot_node
    gt_pivot_head = Node()

    # Prepare pointers to the sublists
    lt_pivot_curr = lt_pivot_head
    eq_pivot_curr = eq_pivot_head
    gt_pivot_curr = gt_pivot_head

    # Prepare variables to calculate length of the result parts
    first_part_length = second_part_length = 0

    # Distribute subsequent nodes of a linked list part to appropriate sublists
    while curr_node is not end_node:
        if curr_node.val < pivot_node.val:
            lt_pivot_curr.next = curr_node
            lt_pivot_curr = lt_pivot_curr.next
            first_part_length += 1
        elif curr_node.val > pivot_node.val:
            gt_pivot_curr.next = curr_node
            gt_pivot_curr = gt_pivot_curr.next
            second_part_length += 1
        else:
            eq_pivot_curr.next = curr_node
            eq_pivot_curr = eq_pivot_curr.next
        curr_node = curr_node.next

    # Join created lists together
    # Link a list of elements lower than pivot (lt_pivot) if is not empty
    if lt_pivot_head.next:
        begin_prev_node.next = lt_pivot_head.next
        lt_pivot_curr.next = eq_pivot_head

        if gt_pivot_head.next:
            eq_pivot_curr.next = gt_pivot_head.next
            gt_pivot_curr.next = end_node
        else:
            eq_pivot_curr.next = end_node
    # Link a list of elements greater than pivot (gt_pivot) if is not empty
    elif gt_pivot_head.next:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = gt_pivot_head.next
        gt_pivot_curr.next = end_node
    # Otherwise, there will be only eq_pivot linked list (all elements are equal to a pivot)
    else:
        begin_prev_node.next = eq_pivot_head
        eq_pivot_curr.next = end_node

    return eq_pivot_head, first_part_length, eq_pivot_curr, second_part_length
```

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

In [53]:
```python
sorting_functions['Quick Sort #4 (functional, iterative)'] = (quick_sort, create_linked_list, {
    '_partition': _partition
})
```

Kilka testów

In [54]:
```python
test_sort_func(quick_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Algorytmy szybkie (liniowe)

O złożoności ($O(n)$)

Bucket Sort

Oczywiście, aby ten algorytm działał jak najlepiej, dane wejściowe powinny być o rozkładzie normalnym (patrz plik z omówieniem algorytmów sortowania dla tablic)

Implementacja algorytmu #1 (z pomocą Insertion Sorta) (dla obiektowej implementacji listy)

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. 64 elementów (w przypadku implementacji dla list odsyłaczowych) - patrz porównanie wydajności dla takich danych) jest bardzo szybki. W poniższej implementacji wykorzystuję liczbę wiaderek, która jest równa 1/48 wielkości danych 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 48, chcę, aby w jak największej liczbie wiaderek, liczba elementów nie przekroczyła zbyt mocno liczby 64.

In [55]:
```python
def bucket_sort(ll: LinkedList, k: 'threshold' = 64):
    if len(ll) < k:
        # Create a sentinel node
        sentinel = Node()
        sentinel.next = ll.head
        # Perform sorting and update the pointers
        ll.tail = insertion_sort(sentinel)
        ll.head = sentinel.next
    else:
        # Make a threshold 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)
        buckets_count = len(ll) // m + 1
        # Create buckets (a list of sentinel nodes)
        buckets = []
        for _ in range(buckets_count):
            sentinel = tail = Node()
            buckets.append([sentinel, tail])
        # Calculate an interval in order to store values in proper buckets
        min_val, max_val = minmax(ll)
        val_interval = (max_val - min_val) / buckets_count
        # Distribute values to the proper buckets
        curr = ll.head
        while curr:
            # Calculate the bucket's index depending on how much the
            # current value is greater than the lowest one
            bucket_idx = int((curr.val - min_val) / val_interval - .5)
            buckets[bucket_idx][1].next = curr
            buckets[bucket_idx][1] = curr
            curr = curr.next
        # Sort each bucket separately
        for bucket in buckets:
            # Remove tail's next node and update a tail after sorting
            bucket[1].next = None
            bucket[1] = insertion_sort(bucket[0])
        # Link buckets together to create a sorted list
        sentinel = Node()
        sorted_tail = sentinel
        for bucket in buckets:
            # If a bucket is not empty, link a list to he result
            if bucket[0].next:
                sorted_tail.next = bucket[0].next
                sorted_tail = bucket[1]
        # Modify the head and the tail node of the initial linked list
        ll.head = sentinel.next
        ll.tail = buckets[-1][1]


def minmax(ll):
    global_min = global_max = ll.tail.val
    curr = ll.head

    while curr and curr.next:
        if curr.val > curr.next.val:
            if curr.val > global_max:      global_max = curr.val
            if curr.next.val < global_min: global_min = curr.next.val
        else:
            if curr.next.val > global_max: global_max = curr.next.val
            if curr.val< global_min:       global_min = curr.val
        curr = curr.next.next

    return global_min, global_max


def insert_node(ll_head: 'linked list head (sentinel)', node):  # Inserts node in a right position maintaining
    # Insert the node before a greater one
    curr = ll_head
    while node.val > curr.next.val:
        curr = curr.next
    node.next = curr.next
    curr.next = node


def insertion_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next: return ll_head
    if not ll_head.next.next: return ll_head.next

    prev = ll_head.next
    while prev.next: # We start from the second node (prev.next)
        # If a current node (prev.next) has a value lower than a prev node, we have to
        # shift this node to a right position before.
        if prev.next.val < prev.val:
            # Remove a current node
            removed = prev.next
            prev.next = prev.next.next
            # Now we insert this node in a right position
            insert_node(ll_head, removed)
        # We can skip a current node otherwise.
        else:
            prev = prev.next

    return prev
```

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

In [56]:
```python
sorting_functions['Bucket Sort #1 (objective with Insertion Sort)'] = (bucket_sort, LinkedList, {
    'minmax': minmax,
    'insert_node': insert_node,
    'insertion_sort': insertion_sort
})
```

Kilka testów

In [57]:
```python
# test_sort_obj(bucket_sort, range_=(-1_000, 1_000), val_counts=(0, 1_000), samples=1_000, failed_only=True)
# test_sort_obj(bucket_sort, val_counts=(0, 1000), samples=1000, failed_only=True)
test_sort_obj(bucket_sort, val_counts=(100, 200), samples=25, failed_only=True)
```

```
Total tests passed: 25/25
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #2 (z pomocą Insertion Sorta) (dla funkcyjnej implementacji listy)

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. 64 elementów (w przypadku implementacji dla list odsyłaczowych) - patrz porównanie wydajności dla takich danych) jest bardzo szybki. W poniższej implementacji wykorzystuję liczbę wiaderek, która jest równa 1/48 wielkości danych 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 48, chcę, aby w jak największej liczbie wiaderek, liczba elementów nie przekroczyła zbyt mocno liczby 64.

In [58]:
```python
def bucket_sort(ll_head: 'linked list head (sentinel)', k: 'threshold' = 32):
    length = linked_list_length(ll_head)
    if length < k:
        insertion_sort(ll_head)
    else:
        # Perform sorting if only there are at least 2 values in a list
        if ll_head.next and ll_head.next.next:
            m = int(2/3 * k)
            buckets_count = length // m + 1
            # Create buckets (a list of sentinel nodes)
            buckets = []
            for _ in range(buckets_count):
                sentinel = tail = Node()
                buckets.append([sentinel, tail])
            # Calculate an interval in order to store values in proper buckets
            min_val, max_val = minmax(ll_head)
            val_interval = (max_val - min_val) / buckets_count
            # Distribute values to the proper buckets
            curr = ll_head.next
            while curr:
                # Calculate the bucket's index depending on how much the
                # current value is greater than the lowest one
                bucket_idx = int((curr.val - min_val) / val_interval - .5)
                tail = buckets[bucket_idx][1]
                tail.next = curr
                buckets[bucket_idx][1] = tail.next
                curr = curr.next
            # Sort each bucket separately
            for bucket in buckets:
                # Remove tail's next node and update a tail after sorting
                bucket[1].next = None
                bucket[1] = insertion_sort(bucket[0])
            # Link buckets together to create a sorted list
            sorted_tail = ll_head
            for bucket in buckets:
                # If a bucket is not empty, link a list to he result
                if bucket[0].next:
                    sorted_tail.next = bucket[0].next
                    sorted_tail = bucket[1]


def linked_list_length(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    length = 0
    while curr:
        length += 1
        curr = curr.next
    return length


def minmax(ll_head):
    curr = ll_head.next
    global_min = global_max = curr.val

    while curr and curr.next:
        if curr.val > curr.next.val:
            if curr.val > global_max:      global_max = curr.val
            if curr.next.val < global_min: global_min = curr.next.val
        else:
            if curr.next.val > global_max: global_max = curr.next.val
            if curr.val< global_min:       global_min = curr.val
        curr = curr.next.next

    if curr:
        if curr.val > global_max:
            global_max = curr.val
        elif curr.val < global_min:
            global_min = curr.val

    return global_min, global_max


def insert_node(ll_head: 'linked list head (sentinel)', node):  # Inserts node in a right position maintaining
    # Insert the node before a greater one
    curr = ll_head
    while node.val > curr.next.val:
        curr = curr.next
    node.next = curr.next
    curr.next = node


def insertion_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next: return ll_head
    if not ll_head.next.next: return ll_head.next

    prev = ll_head.next
    while prev.next: # We start from the second node (prev.next)
        # If a current node (prev.next) has a value lower than a prev node, we have to
        # shift this node to a right position before.
        if prev.next.val < prev.val:
            # Remove a current node
            removed = prev.next
            prev.next = prev.next.next
            # Now we insert this node in a right position
            insert_node(ll_head, removed)
        # We can skip a current node otherwise.
        else:
            prev = prev.next

    return prev
```

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

In [59]:
```
sorting_functions['Bucket Sort #2 (functional with Insertion Sort)'] = (bucket_sort, create_linked_list, {
    'minmax': minmax,
    'insert_node': insert_node,
    'insertion_sort': insertion_sort
})
```

Kilka testów

In [60]:
```
# test_sort_func(bucket_sort, val_counts=(20, 2000), samples=10000, failed_only=True)
test_sort_func(bucket_sort, val_counts=(20, 200), samples=25, failed_only=True)
```

```
Total tests passed: 25/25
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #3 (z pomocą Counting Sorta) (dla obiektowej implementacji listy)

(Wersja dla WSZYSTKICH liczb CAŁKOWITYCH)

Tak naprawdę ten algorytm nie wymaga żadnego sortowania ani zliczania wartości, a jedynie ich włożenie do odpowiednich wiaderek, a następnie ich połączenie (złączenie list odsyłaczowych). (Oczywiście ten, jak i poniższy algorytm sortowania (dla funkcyjnej implementacji listy), jest niewydajny pamięciowo, jeśli sortowana jest tablica bardzo różniących się między sobą wartości).

In [61]:
```
def bucket_counting_sort(ll: LinkedList):
    if len(ll) > 1:
        min_val, max_val = minmax(ll)
        # Allocate memory (create sentinel nodes) for values (our buckets)
        buckets = []
        for _ in range(max_val - min_val + 1):
            sentinel = tail = Node()
            buckets.append([sentinel, tail])
        # Store values in the proper buckets
        curr = ll.head
        while curr:
            idx = curr.val - min_val
            tail = buckets[idx][1]
            tail.next = curr
            buckets[idx][1] = curr
            curr = curr.next

        # Link all linked lists together
        tail = Node()
        sentinel = tail
        for bucket in buckets:
            # If a bucket is not empty, link a list to he result
            if bucket[0].next:
                tail.next = bucket[0].next
                tail = bucket[1]

        # Update the pointers of the initial linked list
        tail.next = None
        ll.head = sentinel.next
        ll.tail = tail


def minmax(ll):
    global_min = global_max = ll.tail.val
    curr = ll.head

    while curr and curr.next:
        if curr.val > curr.next.val:
            if curr.val > global_max:      global_max = curr.val
            if curr.next.val < global_min: global_min = curr.next.val
        else:
            if curr.next.val > global_max: global_max = curr.next.val
            if curr.val< global_min:       global_min = curr.val
        curr = curr.next.next

    return global_min, global_max
```

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

In [62]:
```
sorting_functions['Bucket Sort #3 (objective with Counting Sort)'] = (bucket_counting_sort, LinkedList, {
    'minmax': minmax
})
```

Kilka testów

In [63]:
```
test_sort_obj(bucket_counting_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #4 (z pomocą Counting Sorta) (dla funkcyjnej implementacji listy)

(Wersja dla WSZYSTKICH liczb CAŁKOWITYCH)

In [64]:
```
def bucket_counting_sort(ll_head: 'linked list head (sentinel)'):
    if ll_head.next and ll_head.next.next:
        min_val, max_val = minmax(ll_head)
        # Allocate memory (create sentinel nodes) for values (our buckets)
        buckets = []
        for _ in range(max_val - min_val + 1):
            sentinel = tail = Node()
            buckets.append([sentinel, tail])
        # Store values in the proper buckets
        curr = ll_head.next
        while curr:
            idx = curr.val - min_val
            tail = buckets[idx][1]
            tail.next = curr
            buckets[idx][1] = curr
            curr = curr.next

        # Link all linked lists together
        tail = Node()
        sentinel = tail
        for bucket in buckets:
            # If a bucket is not empty, link a list to he result
            if bucket[0].next:
                tail.next = bucket[0].next
                tail = bucket[1]

        # Update the pointers of the initial linked list
        tail.next = None
        ll_head.next = sentinel.next


def minmax(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    global_min = global_max = curr.val

    while curr and curr.next:
        if curr.val > curr.next.val:
            if curr.val > global_max:      global_max = curr.val
            if curr.next.val < global_min: global_min = curr.next.val
        else:
            if curr.next.val > global_max: global_max = curr.next.val
            if curr.val< global_min:       global_min = curr.val
        curr = curr.next.next

    if curr:
        if curr.val > global_max:
            global_max = curr.val
        elif curr.val < global_min:
            global_min = curr.val

    return global_min, global_max
```

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

In [65]:
```
sorting_functions['Bucket Sort #4 (functional with Counting Sort)'] = (bucket_counting_sort, create_linked_list, {
    'minmax': minmax
})
```

Kilka testów

In [66]:
```
test_sort_func(bucket_counting_sort, samples=100, failed_only=True)
```

```
Total tests passed: 100/100
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #5 (z pomocą Bucket Sorta) (dla obiektowej implementacji listy)

Wersja dla DOWOLNYCH liczb RZECZYWISTYCH

Ulepszona wersja Bucket Sorta, która sortuje duże wiaderka ponownie Bucket Sortem. Poniższa wersja algorytmu jest wersją iteracyjną, ale łatwo zaimplementować wersję rekurencyjną.

In [67]:
```
def bucket_sort(ll: LinkedList, k: 'threshold' = 48):
    if len(ll) < 2: return
    sentinel = Node()
    sentinel.next = ll.head
    bucket_stack = [[sentinel, ll.tail, len(ll)]]
    res_curr = res_sentinel = Node()
    while bucket_stack:
        # Get the last bucket from a stack
        bucket = bucket_stack.pop()
        # If it is small enough, sort it and link to the result linked list
        if bucket[2] < k:
            # Link a sorted part to the result linked list
            tail = insertion_sort(bucket[0])
            res_curr.next = bucket[0].next
            res_curr = tail
        # Split a bucket into smaller buckets otherwise
        else:
            min_val, max_val = minmax(bucket[0])
            # If a bucket contains all the same values, link it to the result linked list
            if max_val == min_val:
                res_curr.next = bucket[0].next
                res_curr = bucket[1]
            # If not, divide this bucket into smaller buckets
            else:
                buckets = _divide_into_buckets(bucket, min_val, max_val, k)
                for i in range(len(buckets)-1, -1, -1):
                    if buckets[i][2]: # If is not empty
                        bucket_stack.append(buckets[i])
    # Fix linked list pointers
    ll.head = res_sentinel.next
    ll.tail = res_curr


def _divide_into_buckets(bucket, min_val, max_val, k: 'threshold'):
    m = int(2/3 * k)
    buckets_count = bucket[2] // m + 1
    val_interval = (max_val - min_val) / buckets_count
    # Create buckets
    buckets = []
    for _ in range(buckets_count):
        sentinel = tail = Node()
        buckets.append([sentinel, tail, 0])
    # Distribute values to the proper buckets
    curr = bucket[0].next
    bucket[1].next = None
    while curr:
        # Calculate the bucket's index depending on how much the
        # current value is greater than the lowest one
        bucket_idx = int((curr.val - min_val) / val_interval - .5)
        buckets[bucket_idx][1].next = curr
        buckets[bucket_idx][1] = curr
        buckets[bucket_idx][2] += 1
        curr = curr.next
    # Unlink next nodes of tails
    for bucket in buckets:
        bucket[1].next = None

    return buckets


def minmax(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    global_min = global_max = curr.val

    while curr and curr.next:
        if curr.val > curr.next.val:
            if curr.val > global_max:      global_max = curr.val
            if curr.next.val < global_min: global_min = curr.next.val
        else:
            if curr.next.val > global_max: global_max = curr.next.val
            if curr.val< global_min:       global_min = curr.val
        curr = curr.next.next

    if curr:
        if curr.val > global_max:
            global_max = curr.val
        elif curr.val < global_min:
            global_min = curr.val

    return global_min, global_max


def insert_node(ll_head: 'linked list head (sentinel)', node):  # Inserts node in a right position maintaining
    # Insert the node before a greater one
    curr = ll_head
    while node.val > curr.next.val:
        curr = curr.next
    node.next = curr.next
    curr.next = node


def insertion_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next: return ll_head
    if not ll_head.next.next: return ll_head.next

    prev = ll_head.next
    while prev.next: # We start from the second node (prev.next)
        # If a current node (prev.next) has a value lower than a prev node, we have to
        # shift this node to a right position before.
        if prev.next.val < prev.val:
            # Remove a current node
            removed = prev.next
            prev.next = prev.next.next
            # Now we insert this node in a right position
            insert_node(ll_head, removed)
        # We can skip a current node otherwise.
        else:
            prev = prev.next

    return prev
```

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

In [68]:
```
sorting_functions['Bucket Sort #5 (objective iterative with Bucket Sort)'] = (bucket_sort, LinkedList, {
    'minmax': minmax,
    'insert_node': insert_node,
    'insertion_sort': insertion_sort,
    '_divide_into_buckets': _divide_into_buckets
})
```

Kilka testów

In [69]:
```
# test_sort_obj(bucket_sort, val_counts=(100, 1000), samples=1000, failed_only=True)
test_sort_obj(bucket_sort, val_counts=(100, 200), samples=25, failed_only=True)
```

```
Total tests passed: 25/25
A sorting algorithm is implemented correctly
```

Implementacja algorytmu #6 (z pomocą Bucket Sorta) (dla funkcyjnej implementacji listy)

Wersja dla DOWOLNYCH liczb RZECZYWISTYCH

Ulepszona wersja Bucket Sorta, która sortuje duże wiaderka ponownie Bucket Sortem. Poniższa wersja algorytmu jest wersją iteracyjną, ale łatwo zaimplementować wersję rekurencyjną.

In [70]:
```
def bucket_sort(ll_head: 'linked list head (sentinel)', k: 'threshold' = 32):
    # If a linked list has no more than one elemnt, don't perform sorting
    if not ll_head.next or not ll_head.next.next: return
    # Get linked list tail and length
    length = 1
    curr = ll_head.next
    while curr.next:
        length += 1
        curr = curr.next
    # Create the bucket_stack
    bucket_stack = [[ll_head, curr, length]]
    res_curr = res_sentinel = Node()

    while bucket_stack:
        # Get the last bucket from a stack
        bucket = bucket_stack.pop()
        # If it is small enough, sort it and link to the result linked list
        if bucket[2] < k:
            # Link a sorted part to the result linked list
            tail = insertion_sort(bucket[0])
            res_curr.next = bucket[0].next
            res_curr = tail
        # Split a bucket into smaller buckets otherwise
        else:
            min_val, max_val = minmax(bucket[0])
            # If a bucket contains all the same values, link it to the result linked list
            if max_val == min_val:
                res_curr.next = bucket[0].next
                res_curr = bucket[1]
            # If not, divide this bucket into smaller buckets
            else:
                buckets = _divide_into_buckets(bucket, min_val, max_val, k)
                for i in range(len(buckets)-1, -1, -1):
                    if buckets[i][2]: # If is not empty
                        bucket_stack.append(buckets[i])
    # Fix linked list pointers
    ll_head.next = res_sentinel.next


def _divide_into_buckets(bucket, min_val, max_val, k: 'threshold'):
    m = int(2/3 * k)
    buckets_count = bucket[2] // m + 1
    val_interval = (max_val - min_val) / buckets_count
    # Create buckets
    buckets = []
    for _ in range(buckets_count):
        sentinel = tail = Node()
        buckets.append([sentinel, tail, 0])
    # Distribute values to the proper buckets
    curr = bucket[0].next
    bucket[1].next = None
    while curr:
        # Calculate the bucket's index depending on how much the
        # current value is greater than the lowest one
        bucket_idx = int((curr.val - min_val) / val_interval - .5)
        buckets[bucket_idx][1].next = curr
        buckets[bucket_idx][1] = curr
        buckets[bucket_idx][2] += 1
        curr = curr.next
    # Unlink next nodes of tails
    for bucket in buckets:
        bucket[1].next = None

    return buckets


def minmax(ll_head: 'linked list head (sentinel)'):
    curr = ll_head.next
    global_min = global_max = curr.val

    while curr and curr.next:
        if curr.val > curr.next.val:
            if curr.val > global_max:      global_max = curr.val
            if curr.next.val < global_min: global_min = curr.next.val
        else:
            if curr.next.val > global_max: global_max = curr.next.val
            if curr.val< global_min:       global_min = curr.val
        curr = curr.next.next

    if curr:
        if curr.val > global_max:
            global_max = curr.val
        elif curr.val < global_min:
            global_min = curr.val

    return global_min, global_max


def insert_node(ll_head: 'linked list head (sentinel)', node):  # Inserts node in a right position maintaining
    # Insert the node before a greater one
    curr = ll_head
    while node.val > curr.next.val:
        curr = curr.next
    node.next = curr.next
    curr.next = node


def insertion_sort(ll_head: 'linked list head (sentinel)'):
    if not ll_head.next: return ll_head
    if not ll_head.next.next: return ll_head.next

    prev = ll_head.next
    while prev.next: # We start from the second node (prev.next)
        # If a current node (prev.next) has a value lower than a prev node, we have to
        # shift this node to a right position before.
        if prev.next.val < prev.val:
            # Remove a current node
            removed = prev.next
            prev.next = prev.next.next
            # Now we insert this node in a right position
            insert_node(ll_head, removed)
        # We can skip a current node otherwise.
        else:
            prev = prev.next

    return prev
```

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

In [71]:
```
sorting_functions['Bucket Sort #6 (functional iterative with Bucket Sort)'] = (bucket_sort, create_linked_list,
    'minmax': minmax,
    'insert_node': insert_node,
    'insertion_sort': insertion_sort,
    '_divide_into_buckets': _divide_into_buckets
})
```

Kilka testów

In [72]:
```
# test_sort_func(bucket_sort, val_counts=(100, 200), samples=10000, failed_only=True)
test_sort_func(bucket_sort, val_counts=(100, 200), samples=25, failed_only=True)
```

```
Total tests passed: 25/25
A sorting algorithm is implemented correctly
```

Porównanie szybkości

In [73]:
```
for i, (name, value) in enumerate(sorting_functions.items()):
    print(f'[{i}] {name}')
```

```
[0] Insertion Sort #1 (objective)
[1] Insertion Sort #2 (functional)
[2] Selection Sort #1 (objective)
[3] Selection Sort #2 (functional)
[4] Bubble Sort #1 (objective)
[5] Bubble Sort #2 (functional)
[6] Merge Sort #1 (objective, recursive)
[7] Merge Sort #2 (objective, recursive optimized)
[8] Merge Sort #3 (objective, iterative, natural series)
[9] Merge Sort #4 (functional, recursive)
[10] Merge Sort #5 (functional, recursive optimized)
[11] Merge Sort #6 (functional, iterative, natural series)
[12] Quick Sort #1 (objective, recursive)
[13] Quick Sort #2 (objective, iterative)
[14] Quick Sort #3 (functional, recursive)
[15] Quick Sort #4 (functional, iterative)
[16] Bucket Sort #1 (objective with Insertion Sort)
[17] Bucket Sort #2 (functional with Insertion Sort)
[18] Bucket Sort #3 (objective with Counting Sort)
[19] Bucket Sort #4 (functional with Counting Sort)
[20] Bucket Sort #5 (objective iterative with Bucket Sort)
[21] Bucket Sort #6 (functional iterative with Bucket Sort)
```

In [74]:
```
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
```

Rozkład jednostajny - tylko sortowania $O(n \cdot log(n))$ i $O(n)$

Pomijam algorytmy o numerach 6 i 9, ponieważ jądro Jupytera zdecha zamiast wyrzucić wyjątku RecursionError. Nie wiem, dlaczego tak się dzieje, ponieważ Pycharm obsługuje prawidłowo wyjątek RecursionError podczas używania tych funkcji dla dużych list odsyłaczowych.

In [75]:
```
compare_performance(choose(*(set(range(7, 22)) - {9})),
                    samples=100,
                    val_counts=(0, 100_000),
                    progress_interval=25,
                    uniform_distribution=True
                   )
```

```
===== Results after 25 tests: =====
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 1.6400     Average: 0.0656
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 1.7320     Average: 0.0693
(1.06x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 2.6519     Average: 0.1061
(1.62x slower)
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 2.6650     Average: 0.1066
(1.62x slower)
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 2.8209     Average: 0.1128
(1.72x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 3.1450     Average: 0.1258
(1.92x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 6.7101     Average: 0.2684
(4.09x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 7.2090     Average: 0.2884
(4.40x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 7.4990     Average: 0.3000
(4.57x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 7.6430     Average: 0.3057
(4.66x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 7.6549     Average: 0.3062
(4.67x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 7.9249     Average: 0.3170
(4.83x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 10.4890    Average: 0.4196
(6.40x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 10.8340    Average: 0.4334
(6.61x slower)

===== Results after 50 tests: =====
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 3.3150     Average: 0.0663
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 3.5030     Average: 0.0701
(1.06x slower)
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 5.1879     Average: 0.1038
(1.56x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 5.2069     Average: 0.1041
(1.57x slower)
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 5.5339     Average: 0.1107
(1.67x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 6.1110     Average: 0.1222
(1.84x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 13.9111    Average: 0.2782
(4.20x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 14.2759    Average: 0.2855
(4.31x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 14.9049    Average: 0.2981
(4.50x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 14.9700    Average: 0.2994
(4.52x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 15.4729    Average: 0.3095
(4.67x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 15.6950    Average: 0.3139
(4.73x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 20.7199    Average: 0.4144
(6.25x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 21.3509    Average: 0.4270
(6.44x slower)

===== Results after 75 tests: =====
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 5.1820     Average: 0.0691
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 5.3850     Average: 0.0718
(1.04x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 7.5228     Average: 0.1003
(1.45x slower)
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 7.6699     Average: 0.1023
(1.48x slower)
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 8.0969     Average: 0.1080
(1.56x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 8.8670     Average: 0.1182
(1.71x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 20.2261    Average: 0.2697
(3.90x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 20.8789    Average: 0.2784
(4.03x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 21.8709    Average: 0.2916
(4.22x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 22.0500    Average: 0.2940
(4.26x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 22.7519    Average: 0.3034
(4.39x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 23.1320    Average: 0.3084
(4.46x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 30.6210    Average: 0.4083
(5.91x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 31.1148    Average: 0.4149
(6.00x slower)

===== Results after 100 tests: =====
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 6.5701     Average: 0.0657
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 6.8529     Average: 0.0685
(1.04x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 9.4077     Average: 0.0941
(1.43x slower)
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 9.5269     Average: 0.0953
(1.45x slower)
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 9.9759     Average: 0.0998
(1.52x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 11.0159    Average: 0.1102
(1.68x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 25.0310    Average: 0.2503
(3.81x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 25.8079    Average: 0.2581
(3.93x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 26.9429    Average: 0.2694
(4.10x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 27.0610    Average: 0.2706
(4.12x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 27.9010    Average: 0.2790
(4.25x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 28.4530    Average: 0.2845
(4.33x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 37.8139    Average: 0.3781
(5.76x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 38.4438    Average: 0.3844
(5.85x slower)

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

Total times (in seconds):
        Bucket Sort #4 (functional with Counting Sort)          :  6.5700738
        Bucket Sort #3 (objective with Counting Sort)           :  6.8529048
        Bucket Sort #2 (functional with Insertion Sort)         :  9.4077418
        Bucket Sort #6 (functional iterative with Bucket Sort)  :  9.5268528
        Bucket Sort #5 (objective iterative with Bucket Sort)   :  9.9758525
        Bucket Sort #1 (objective with Insertion Sort)          :  11.0158870
        Quick Sort #3 (functional, recursive)                   :  25.0310299
        Quick Sort #1 (objective, recursive)                    :  25.8078816
        Merge Sort #5 (functional, recursive optimized)         :  26.9428749
        Merge Sort #2 (objective, recursive optimized)          :  27.0609658
        Quick Sort #4 (functional, iterative)                   :  27.9009700
        Quick Sort #2 (objective, iterative)                    :  28.4530010
        Merge Sort #3 (objective, iterative, natural series)    :  37.8139405
        Merge Sort #6 (functional, iterative, natural series)   :  38.4438114

Average times (in seconds):
        Bucket Sort #4 (functional with Counting Sort)          :  0.0657007
        Bucket Sort #3 (objective with Counting Sort)           :  0.0685290    (1.04x slower)
        Bucket Sort #2 (functional with Insertion Sort)         :  0.0940774    (1.43x slower)
        Bucket Sort #6 (functional iterative with Bucket Sort)  :  0.0952685    (1.45x slower)
        Bucket Sort #5 (objective iterative with Bucket Sort)   :  0.0997585    (1.52x slower)
        Bucket Sort #1 (objective with Insertion Sort)          :  0.1101589    (1.68x slower)
        Quick Sort #3 (functional, recursive)                   :  0.2503103    (3.81x slower)
        Quick Sort #1 (objective, recursive)                    :  0.2580788    (3.93x slower)
        Merge Sort #5 (functional, recursive optimized)         :  0.2694287    (4.10x slower)
        Merge Sort #2 (objective, recursive optimized)          :  0.2706097    (4.12x slower)
        Quick Sort #4 (functional, iterative)                   :  0.2790097    (4.25x slower)
        Quick Sort #2 (objective, iterative)                    :  0.2845300    (4.33x slower)
        Merge Sort #3 (objective, iterative, natural series)    :  0.3781394    (5.76x slower)
        Merge Sort #6 (functional, iterative, natural series)   :  0.3844381    (5.85x slower)
```

Rozkład niejednostajny - wszystkie sortowania (poza 6 i 9)

In [76]:
```
compare_performance(choose(*set(range(0, 22)) - {6, 9}),
                    samples=40,
                    val_counts=(0, 10_000),
                    progress_interval=10,
                    uniform_distribution=False
                   )
```

```
===== Results after 10 tests: =====
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 0.0860     Average: 0.0086
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 0.1170     Average: 0.0117
(1.36x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 0.1710     Average: 0.0171
(1.99x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 0.1730     Average: 0.0173
(2.01x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 0.1750     Average: 0.0175
(2.03x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 0.1830     Average: 0.0183
(2.13x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 0.1960     Average: 0.0196
(2.28x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 0.1980     Average: 0.0198
(2.30x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 0.2000     Average: 0.0200
(2.32x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 0.2010     Average: 0.0201
(2.34x slower)
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 0.2030     Average: 0.0203
(2.36x slower)
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 0.2330     Average: 0.0233
(2.71x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 2.6660     Average: 0.2666
(31.00x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 2.9790     Average: 0.2979
(34.63x slower)
Insertion Sort #2 (functional)                          Total (in seconds): 7.8240     Average: 0.7824
(90.96x slower)
Insertion Sort #1 (objective)                           Total (in seconds): 8.4550     Average: 0.8455
(98.30x slower)
Selection Sort #1 (objective)                           Total (in seconds): 14.9710    Average: 1.4971
(174.06x slower)
Selection Sort #2 (functional)                          Total (in seconds): 15.4550    Average: 1.5455
(179.69x slower)
Bubble Sort #2 (functional)                             Total (in seconds): 55.6494    Average: 5.5649
(647.00x slower)
Bubble Sort #1 (objective)                              Total (in seconds): 56.2280    Average: 5.6228
(653.73x slower)

===== Results after 20 tests: =====
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 0.1590     Average: 0.0080
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 0.1910     Average: 0.0095
(1.20x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 0.3260     Average: 0.0163
(2.05x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 0.3310     Average: 0.0165
(2.08x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 0.3480     Average: 0.0174
(2.19x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 0.3590     Average: 0.0179
(2.26x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 0.3670     Average: 0.0183
(2.31x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 0.3710     Average: 0.0186
(2.33x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 0.3970     Average: 0.0199
(2.50x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 0.4009     Average: 0.0200
(2.52x slower)
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 0.4010     Average: 0.0200
(2.52x slower)
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 0.4570     Average: 0.0229
(2.87x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 4.4330     Average: 0.2216
(27.88x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 5.7260     Average: 0.2863
(36.01x slower)
Insertion Sort #2 (functional)                          Total (in seconds): 14.3050    Average: 0.7153
(89.96x slower)
Insertion Sort #1 (objective)                           Total (in seconds): 15.5870    Average: 0.7793
(98.03x slower)
Selection Sort #1 (objective)                           Total (in seconds): 27.1641    Average: 1.3582
(170.83x slower)
Selection Sort #2 (functional)                          Total (in seconds): 27.7660    Average: 1.3883
(174.62x slower)
Bubble Sort #2 (functional)                             Total (in seconds): 100.0164   Average: 5.0008
(629.00x slower)
Bubble Sort #1 (objective)                              Total (in seconds): 101.0780   Average: 5.0539
(635.68x slower)

===== Results after 30 tests: =====
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 0.2420     Average: 0.0081
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 0.2710     Average: 0.0090
(1.12x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 0.4939     Average: 0.0165
(2.04x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 0.5010     Average: 0.0167
(2.07x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 0.5280     Average: 0.0176
(2.18x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 0.5400     Average: 0.0180
(2.23x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 0.5540     Average: 0.0185
(2.29x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 0.5640     Average: 0.0188
(2.33x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 0.5989     Average: 0.0200
(2.47x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 0.6040     Average: 0.0201
(2.50x slower)
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 0.6150     Average: 0.0205
(2.54x slower)
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 0.6380     Average: 0.0213
(2.64x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 6.3600     Average: 0.2120
(26.28x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 7.6520     Average: 0.2551
(31.62x slower)
Insertion Sort #2 (functional)                          Total (in seconds): 21.9031    Average: 0.7301
(90.51x slower)
Insertion Sort #1 (objective)                           Total (in seconds): 23.6330    Average: 0.7878
(97.66x slower)
Selection Sort #1 (objective)                           Total (in seconds): 41.2231    Average: 1.3741
(170.34x slower)
Selection Sort #2 (functional)                          Total (in seconds): 42.0430    Average: 1.4014
(173.73x slower)
Bubble Sort #2 (functional)                             Total (in seconds): 151.4724   Average: 5.0491
(625.91x slower)
Bubble Sort #1 (objective)                              Total (in seconds): 152.3750   Average: 5.0792
(629.64x slower)

===== Results after 40 tests: =====
Bucket Sort #5 (objective iterative with Bucket Sort)   Total (in seconds): 0.3250     Average: 0.0081
Bucket Sort #6 (functional iterative with Bucket Sort)  Total (in seconds): 0.3471     Average: 0.0087
(1.07x slower)
Merge Sort #2 (objective, recursive optimized)          Total (in seconds): 0.6510     Average: 0.0163
(2.00x slower)
Merge Sort #5 (functional, recursive optimized)         Total (in seconds): 0.6630     Average: 0.0166
(2.04x slower)
Quick Sort #3 (functional, recursive)                   Total (in seconds): 0.6990     Average: 0.0175
(2.15x slower)
Quick Sort #1 (objective, recursive)                    Total (in seconds): 0.7130     Average: 0.0178
(2.19x slower)
Merge Sort #6 (functional, iterative, natural series)   Total (in seconds): 0.7359     Average: 0.0184
(2.26x slower)
Merge Sort #3 (objective, iterative, natural series)    Total (in seconds): 0.7450     Average: 0.0186
(2.29x slower)
Quick Sort #4 (functional, iterative)                   Total (in seconds): 0.7929     Average: 0.0198
(2.44x slower)
Quick Sort #2 (objective, iterative)                    Total (in seconds): 0.8010     Average: 0.0200
(2.46x slower)
Bucket Sort #3 (objective with Counting Sort)           Total (in seconds): 0.8080     Average: 0.0202
(2.49x slower)
Bucket Sort #4 (functional with Counting Sort)          Total (in seconds): 0.8140     Average: 0.0203
(2.50x slower)
Bucket Sort #2 (functional with Insertion Sort)         Total (in seconds): 7.7650     Average: 0.1941
(23.89x slower)
Bucket Sort #1 (objective with Insertion Sort)          Total (in seconds): 9.2929     Average: 0.2323
(28.59x slower)
Insertion Sort #2 (functional)                          Total (in seconds): 29.5810    Average: 0.7395
(91.02x slower)
Insertion Sort #1 (objective)                           Total (in seconds): 31.7930    Average: 0.7948
(97.83x slower)
Selection Sort #1 (objective)                           Total (in seconds): 54.2040    Average: 1.3551
(166.79x slower)
Selection Sort #2 (functional)                          Total (in seconds): 55.4020    Average: 1.3851
(170.48x slower)
Bubble Sort #2 (functional)                             Total (in seconds): 200.1164   Average: 5.0029
(615.77x slower)
Bubble Sort #1 (objective)                              Total (in seconds): 201.2550   Average: 5.0314
(619.27x slower)

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

Total times (in seconds):
        Bucket Sort #5 (objective iterative with Bucket Sort)   :  0.3249853
        Bucket Sort #6 (functional iterative with Bucket Sort)  :  0.3470583
        Merge Sort #2 (objective, recursive optimized)          :  0.6509788
        Merge Sort #5 (functional, recursive optimized)         :  0.6629732
        Quick Sort #3 (functional, recursive)                   :  0.6990416
        Quick Sort #1 (objective, recursive)                    :  0.7129865
        Merge Sort #6 (functional, iterative, natural series)   :  0.7359385
        Merge Sort #3 (objective, iterative, natural series)    :  0.7449694
        Quick Sort #4 (functional, iterative)                   :  0.7929204
        Quick Sort #2 (objective, iterative)                    :  0.8010280
        Bucket Sort #3 (objective with Counting Sort)           :  0.8079662
        Bucket Sort #4 (functional with Counting Sort)          :  0.8139906
        Bucket Sort #2 (functional with Insertion Sort)         :  7.7650254
        Bucket Sort #1 (objective with Insertion Sort)          :  9.2929296
        Insertion Sort #2 (functional)                          :  29.5810430
        Insertion Sort #1 (objective)                           :  31.7930119
        Selection Sort #1 (objective)                           :  54.2040355
        Selection Sort #2 (functional)                          :  55.4020355
        Bubble Sort #2 (functional)                             :  200.1164174
        Bubble Sort #1 (objective)                              :  201.2550139

Average times (in seconds):
        Bucket Sort #5 (objective iterative with Bucket Sort)   :  0.0081246
        Bucket Sort #6 (functional iterative with Bucket Sort)  :  0.0086765    (1.07x slower)
        Merge Sort #2 (objective, recursive optimized)          :  0.0162745    (2.00x slower)
        Merge Sort #5 (functional, recursive optimized)         :  0.0165743    (2.04x slower)
        Quick Sort #3 (functional, recursive)                   :  0.0174760    (2.15x slower)
        Quick Sort #1 (objective, recursive)                    :  0.0178247    (2.19x slower)
        Merge Sort #6 (functional, iterative, natural series)   :  0.0183985    (2.26x slower)
        Merge Sort #3 (objective, iterative, natural series)    :  0.0186242    (2.29x slower)
        Quick Sort #4 (functional, iterative)                   :  0.0198230    (2.44x slower)
        Quick Sort #2 (objective, iterative)                    :  0.0200257    (2.46x slower)
        Bucket Sort #3 (objective with Counting Sort)           :  0.0201992    (2.49x slower)
        Bucket Sort #4 (functional with Counting Sort)          :  0.0203498    (2.50x slower)
        Bucket Sort #2 (functional with Insertion Sort)         :  0.1941256    (23.89x slower)
        Bucket Sort #1 (objective with Insertion Sort)          :  0.2323232    (28.59x slower)
        Insertion Sort #2 (functional)                          :  0.7395261    (91.02x slower)
        Insertion Sort #1 (objective)                           :  0.7948253    (97.83x slower)
        Selection Sort #1 (objective)                           :  1.3551009    (166.79x slower)
        Selection Sort #2 (functional)                          :  1.3850509    (170.48x slower)
        Bubble Sort #2 (functional)                             :  5.0029104    (615.77x slower)
        Bubble Sort #1 (objective)                              :  5.0313753    (619.27x slower)
```