Funkcja testująca poprawność algorytmu wyszukiwania wskazanej wartości

```python
In [2]: import random


def test_search(searching_fn, *,
                samples=20,
                search_first=True,
                search_last=False,
                sorted_input=False,
                failed_only=False
               ):
    # search_first and search_last cannot be set to True at the same time
    if search_first and search_last:
        raise ValueError('Cannot search the first and the last element at the same time')

    passed = 0
    for i in range(samples):
        random_lst = [random.randint(-100, 100) for _ in range(random.randint(0, 40))]
        searched_val = random.choice(random_lst) if random.random() > .5 and random_lst else random.randint(-10
        if sorted_input: random_lst.sort()
        result = searching_fn(random_lst, searched_val)

        is_correct = False
        if search_first is search_last is False:
            searched_occurrence = "NOT SPECIFIED"
            if searched_val not in random_lst:
                is_correct = result == -1
            else:
                is_correct = random_lst[result] == searched_val
        else:
            searched_occurrence = "FIRST"
            if searched_val not in random_lst:
                expected_result = -1
            elif search_last:
                searched_occurrence = "LAST"
                for j in range(len(random_lst)-1, -1, -1):
                    if random_lst[j] == searched_val:
                        expected_result = j
                        break
            else:
                expected_result = random_lst.index(searched_val)
            is_correct = expected_result == result

        passed += is_correct

        if not failed_only or (failed_only and not is_correct):
            print(f'TEST #{i+1}:')
            print('Input:', random_lst)
            print('Searched value:', searched_val)
            print('Searched occurrence:', searched_occurrence)
            print('Result:', result)
            print('Expected result:', expected_result)
            print(f'Test {"PASSED" if is_correct else "FAILED"}')
            print(f'Current passed-to-tested ratio: {passed}/{i+1}')
            print()

    print(f'Total tests passed {passed}/{samples}')
    print(f'An algorithm is {"CORRECT" if passed == samples else "WRONG"}')
```

Funkcja testująca poprawność algorytmu wskazywania wartości na danej pozycji posortowanej tablicy (dla tablic)

```python
In [3]: import random


def test_select(select_fn, *,
                samples=20,
                values_count=(1, 100),
                range_=(-100, 100),
                unique_only=False,
                failed_only=False
               ):
    passed = 0
    for i in range(1, samples + 1):
        random_lst = [random.randint(*range_) for _ in range(random.randint(*values_count))]
        if unique_only:
            random_lst = list(set(random_lst))
        k = random.randint(0, len(random_lst)-1)
        sorted_lst = sorted(random_lst)
        expected = sorted_lst[k]
        result = select_fn(random_lst, k)
        is_correct = expected == result
        passed += is_correct
        if not failed_only or (failed_only and not is_correct):
            print(f'TEST #{i}:')
            print('k:', k)
            print('Input arr: ', random_lst)
            print('Sorted arr:', sorted_lst)
            print('Expected:', expected)
            print('Result:', result)
            print(f'Test {"PASSED" if is_correct else "FAILED"}')
            print(f'Passed-to-tested ratio: {passed}/{i}')
            print()
    print(f'===== Final results: =====')
    print(f'Final passed-to-tested ratio: {passed}/{samples}')
    print(f'An algorithm is {"CORRECT" if passed == samples else "WRONG"}')
```

Funkcja testująca poprawność algorytmu wskazywania wartości na danej pozycji posortowanej tablicy (dla list odsyłaczowych)

```python
In [4]: import random


def test_select_ll(select_fn, ll_creation, print_fn, *,
                   samples=20,
                   values_count=(1, 100),
                   range_=(-100, 100),
                   unique_only=False,
                   failed_only=False
                  ):
    passed = 0
    for i in range(1, samples + 1):
        random_lst = [random.randint(*range_) for _ in range(random.randint(*values_count))]
        if unique_only:
            random_lst = list(set(random_lst))
        k = random.randint(0, len(random_lst)-1)
        sorted_lst = sorted(random_lst)
        expected = sorted_lst[k]
        ll = ll_creation(random_lst)
        result = select_fn(ll, k)
        is_correct = expected == result
        passed += is_correct
        if not failed_only or (failed_only and not is_correct):
            print(f'TEST #{i}:')
            print('k:', k)
            print('Input arr: ', random_lst)
            print('Sorted arr:', sorted_lst)
            print('linked list after:', end=' ')
            print_fn(ll)
            print('Expected:', expected)
            print('Result:', result)
            print(f'Test {"PASSED" if is_correct else "FAILED"}')
            print(f'Passed-to-tested ratio: {passed}/{i}')
            print()
    print(f'===== Final results: =====')
    print(f'Final passed-to-tested ratio: {passed}/{samples}')
    print(f'An algorithm is {"CORRECT" if passed == samples else "WRONG"}')
```

» Wyszukiwanie połówkowe (binarne)

wskazanej wartości

› Złożoność wyszukiwania

Złożoność czasowa

Najgorszy przypadek

$O(log(n))$

Najlepszy przypadek

$O(log(n))$

Złożoność pamięciowa

Najgorszy przypadek

$O(1)$

Najlepszy przypadek

$O(1)$

› Implementacja algorytmu #1 (zwraca pierwsze wystąpienie)

!!! UWAGA !!!

Ten algorytm może zostać użyty jedynie dla indeksowalnej sekwencji posortowanej niemalejąco. Zwracaną wartością jest nieujemna liczba całkowita, oznaczająca indeks znalezionego elementu. Jeżeli dany element nie występuje w przeszukiwanej sekwencji, zwrócona zostanie wartość -1.

Wersja z poszukiwaniem wartości na całym zakresie tablicy

```python
In [5]: def binary_search_first(arr: 'sorted sequence', el: 'searched element') -> int:
    left_idx = 0
    right_idx = len(arr)-1

    while left_idx <= right_idx:
        mid_idx = (left_idx + right_idx) // 2
        if el > arr[mid_idx]:
            left_idx = mid_idx + 1
        else:
            right_idx = mid_idx - 1

    return left_idx if left_idx < len(arr) and arr[left_idx] == el else -1
```

Kilka testów

```python
In [6]: test_search(binary_search_first, sorted_input=True, samples=100, failed_only=True)
```

```
Total tests passed 100/100
An algorithm is CORRECT
```

Wersja z poszukiwaniem wartości na wyszczególnionym zakresie tablicy

```python
In [7]: def binary_search_first(arr, begin_idx, end_idx, val):
    l = begin_idx
    r = end_idx

    while l <= r:
        mid = (l + r) // 2
        if val <= arr[mid]:
            r = mid - 1
        else:
            l = mid + 1

    return l if l < end_idx and arr[l] == val else -1
```

› Implementacja algorytmu #2 (zwraca ostatnie wystąpienie)

!!! UWAGA !!!

Ten algorytm może zostać użyty jedynie dla indeksowalnej sekwencji posortowanej niemalejąco. Zwracaną wartością jest nieujemna liczba całkowita, oznaczająca indeks znalezionego elementu. Jeżeli dany element nie występuje w przeszukiwanej sekwencji, zwrócona zostanie wartość -1.

```python
In [14]: def binary_search_last(arr: 'sorted sequence', el: 'searched element') -> int:
    left_idx = 0
    right_idx = len(arr)-1

    while left_idx <= right_idx:
        mid_idx = (left_idx + right_idx) // 2
        if el < arr[mid_idx]:
            right_idx = mid_idx - 1
        else:
            left_idx = mid_idx + 1

    return right_idx if right_idx >= 0 and arr[right_idx] == el else -1
```

Kilka testów

```python
In [15]: test_search(binary_search_last, samples=100, sorted_input=True, search_last=True, search_first=False, failed_on
```

```
Total tests passed 100/100
An algorithm is CORRECT
```

Wersja z poszukiwaniem wartości na wyszczególnionym zakresie tablicy

```python
In [10]: def binary_search_last(arr, begin_idx, end_idx, val):
    l = begin_idx
    r = end_idx

    while l <= r:
        mid = (l + r) // 2
        if val < arr[mid]:
            r = mid - 1
        else:
            l = mid + 1

    return r if r >= begin_idx and arr[r] == val else -1
```

› Zastosowania algorytmu

- Liczba wystąpień wskazanej wartości w posortowanej tablicy

Oczywiście możliwe jest zaimplementowanie funkcji działającej w czasie liniowym $O(n)$. Ponadto, taka funkcja działałaby również na nieposortowanych sekwencjach, a nie tylko na tych, które są posortowane niemalejąco. Mimo wszystko, czasem mamy posortowane dane, wiec można znaleźć szukaną wartość w czasie $O(log(n))$ (przydatne jest to szczególnie w przypadku dużych sekwencji danych, które są posortowane).

W poniższej implementacji korzystamy z obu powyżej zadeklarowanych funkcji (działających na całych tablicach)

Implementacja algorytmu

```python
In [18]: def binary_search_first(arr: 'sorted sequence', el: 'searched element') -> int:
    left_idx = 0
    right_idx = len(arr)-1

    while left_idx <= right_idx:
        mid_idx = (left_idx + right_idx) // 2
        if el > arr[mid_idx]:
            left_idx = mid_idx + 1
        else:
            right_idx = mid_idx - 1

    return left_idx if left_idx < len(arr) and arr[left_idx] == el else -1


def binary_search_last(arr: 'sorted sequence', el: 'searched element') -> int:
    left_idx = 0
    right_idx = len(arr)-1

    while left_idx <= right_idx:
        mid_idx = (left_idx + right_idx) // 2
        if el < arr[mid_idx]:
            right_idx = mid_idx - 1
        else:
            left_idx = mid_idx + 1

    return right_idx if right_idx >= 0 and arr[right_idx] == el else -1


def count_occurrences(arr: 'sorted sequence', el: 'element to count occurrences of') -> int:
    if arr:
        begin_idx = binary_search_first(arr, el)
        if begin_idx >= 0:  # That means there is at least one occurrence of the specified element
            end_idx = binary_search_last(arr, el)
            return end_idx - begin_idx + 1
    return 0
```

Kilka testów

```python
In [21]: for _ in range(5):
    random_lst = sorted(random.randint(-7, 7) for _ in range(random.randint(0, 40)))
    to_count = random.randint(-7, 7)
    expected = random_lst.count(to_count)
    result = count_occurrences(random_lst, to_count)
    print('Input:', random_lst)
    print('Value to count:', to_count)
    print('Expected result:', expected)
    print('Result:', result)
    print()
```

```
Input: [-6, -3, -3, -3, 0, 2, 3, 3, 5, 6]
Value to count: 2
Expected result: 1
Result: 1

Input: [-7, -6, -6, -6, -5, -5, -5, -5, -5, -5, -4, -3, -3, -3, -3, -1, 0, 0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3,
4, 4, 4, 4, 5, 5, 6, 7, 7]
Value to count: -3
Expected result: 4
Result: 4

Input: [-3, 2, 6]
Value to count: -2
Expected result: 0
Result: 0

Input: [-7, -7, -7, -7, -6, -6, -5, -5, -5, -5, -5, -4, -2, -2, -2, -2, -2, -2, -1, 0, 1, 1, 1, 1, 2, 2, 2, 2,
4, 4, 4, 4, 4, 5, 6, 6, 6, 7]
Value to count: 4
Expected result: 5
Result: 5

Input: [-7, -7, -6, -6, -6, -5, -5, -4, -3, -3, -3, -2, -2, -1, -1, -1, 2, 3, 4, 4, 5, 5, 7, 7]
Value to count: 1
Expected result: 0
Result: 0
```

» Quick Select

Zwraca wskazaną w kolejności wartość (liczbę, która znajdowała by się na wskazanej pozycji posortowanej tablicy)

Ten algorytm bazuje na zmodyfikowanej funkcji partition z algorytmu sortowania Quick Sort (w wersji Lomuto, ponieważ istotne jest to, aby pivot został umieszczony na swojej końcowej pozycji). Zatem wszelkie wady i korzyści, jakie ma ta funkcja, mają odzwierciedlenie w przypadku tego algorytmu. Takie wyszukiwanie jest niestabilne, więc jeżeli istnieje kilka wartości o tym samym kluczu, według którego sortujemy, nie mamy pewności, że uzyskana przy pomocy tej funkcji wartość będzie taka sama, jaka stałaby na wskazanej pozycji po przesortowaniu tablicy stabilnym algorytmem sortowania (wynika to bezpośrednio z faktu, iż algorytm ten jest podobny do Quick Sorta, lecz sortowanie ogranicza się do fragmentów tablicy, które musimy przesortować, aby otrzymać szukaną wartość).

UWAGA

Konieczne jest użycie funkcji partition Lomuto. Funkcja Hoare'a nie zwraca finalnej pozycji pivota, a jedynie indeks, który dzieli tablicę na wartości mniejsze lub równe od pivota oraz wartości od niego większe lub równe.

› Złożoność wyszukiwania

Złożoność czasowa

Najgorszy przypadek

$O(n^2)$

Aby zminimalizować ryzyko wystąpienia najgorszego przypadku, należy wybierać losowo pivota.

Najlepszy przypadek

$O(n)$

Złożoność pamięciowa

Najgorszy przypadek

$O(1)$

Najlepszy przypadek

$O(1)$

Algorytm dla tablic

› Implementacja algorytmu #1 (rekurencyjna)

(Ze sztywno ustalonym pivotem)

```python
In [22]: def quick_select(arr, k: 'index of a value'):
    if not 0 <= k < len(arr):
        raise IndexError(f'index too {"small" if k < 0 else "large"}')
    if len(arr) == 1:
        return arr[0]
    return _quick_select(arr, k, 0, len(arr)-1)


def _quick_select(arr, k, left_idx, right_idx):
    pivot_position = _partition(arr, left_idx, right_idx)

    if pivot_position > k:
        return _quick_select(arr, k, left_idx, pivot_position - 1)
    elif pivot_position < k:
        return _quick_select(arr, k, pivot_position + 1, right_idx)
    else:
        return arr[pivot_position]


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

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

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

    return i  # Return a pivot position after the last swap

""" Modified lomuto partition function (choosing left element as a pivot) below: """
# def _partition(arr, left_idx, right_idx):
#     pivot = arr[left_idx]

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

#     # Place a pivot element on its destination index
#     swap(arr, i - 1, left_idx)

#     return i - 1  # Return a pivot position after the last swap
""" End of a partition function """

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

Kilka testów

```python
In [23]: test_select(quick_select, samples=1000, values_count=(100, 1000), range_=(-1000, 1000), failed_only=True)
```

```
===== Final results: =====
Final passed-to-tested ratio: 1000/1000
An algorithm is CORRECT
```

› Implementacja algorytmu #2 (rekurencyjna)

(Z losowo wybieranym pivotem)

```python
In [24]: import random


def quick_select(arr, k: 'index of a value'):
    if not 0 <= k < len(arr):
        raise IndexError(f'index too {"small" if k < 0 else "large"}')
    if len(arr) == 1:
        return arr[0]
    return _quick_select(arr, k, 0, len(arr)-1)


def _quick_select(arr, k, left_idx, right_idx):
    pivot_position = _partition(arr, left_idx, right_idx)

    if pivot_position > k:
        return _quick_select(arr, k, left_idx, pivot_position - 1)
    elif pivot_position < k:
        return _quick_select(arr, k, pivot_position + 1, right_idx)
    else:
        return arr[pivot_position]


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

    # Swap a pivot with the last element
    swap(arr, right_idx, pivot_idx)

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

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

    return i  # Return a pivot position after the last swap


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

Kilka testów

```python
In [26]: test_select(quick_select, samples=1000, values_count=(100, 1000), range_=(-1000, 1000), failed_only=True)
```

```
===== Final results: =====
Final passed-to-tested ratio: 1000/1000
An algorithm is CORRECT
```

› Implementacja algorytmu #3 (iteracyjna) (NAJLEPSZA)

(Z losowo wybieranym pivotem)

```python
In [28]: import random


def quick_select(arr, k: 'index of a value'):
    if not 0 <= k < len(arr):
        raise IndexError(f'index too {"small" if k < 0 else "large"}')
    if len(arr) == 1:
        return arr[0]

    left_idx = 0
    right_idx = len(arr) - 1
    pivot_position = -1
    while k != pivot_position:
        pivot_position = _partition(arr, left_idx, right_idx)

        if pivot_position > k:
            right_idx = pivot_position - 1
        else:
            left_idx = pivot_position + 1

    return arr[k]


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

    # Swap a pivot with the last element
    swap(arr, right_idx, pivot_idx)

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

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

    return i  # Return a pivot position after the last swap


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

Kilka testów

```python
In [29]: test_select(quick_select, samples=1000, values_count=(100, 1000), range_=(-1000, 1000), failed_only=True)
```

```
===== Final results: =====
Final passed-to-tested ratio: 1000/1000
An algorithm is CORRECT
```

Algorytm dla list odsyłaczowych jednokierunkowych

Implementacja obiektowa listy odsyłaczowej

```python
In [30]: 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)
```

Implementacja funkcyjna listy odsyłaczowej

```python
In [31]: 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 print_linked_list_part(begin_prev_node: 'previous node to the beginning of a part to print',
                           end_node: 'last node of a sublist to print'):
    prev = begin_prev_node.next
    print(prev.val, end=' ')
    while prev is not end_node.next:
        print('->', prev.next.val, end=' ')
        prev = prev.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
```

› Implementacja algorytmu #1 (rekurencyjna) (dla implementacji obiektowej listy)

```python
In [32]: def quick_select(ll: LinkedList, k: 'index of a value'):
    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
        res = _quick_select(ll.head, None, 0, k)
        # Remove a sentinel node which was added
        ll.head = ll.head.next
        return res
    elif ll and k == 0:
        return ll.head.val
    else:
        raise IndexError(f'index too {"small" if k < 0 else "large"}')


def _quick_select(begin_prev_node, end_node, pivot_idx, k):
    res = _partition(begin_prev_node, end_node, pivot_idx)
    first_end_node, second_begin_prev_node, pivot_head_idx, pivot_tail_idx = res

    # If we placed a span of values equal to the pivot value on they final
    # position and the k index is somewhere within the bounds of this span.
    # we can return the last pivot value
    if pivot_head_idx <= k <= pivot_tail_idx:
        return second_begin_prev_node.val
    # If the last pivot value was placed before the k index, we have
    # to search a right part of a linked list for a desired value
    if pivot_tail_idx < k:
        return _quick_select(second_begin_prev_node, end_node, pivot_tail_idx + 1, k)
    # Otherwise we have to search the left part
    return _quick_select(begin_prev_node, first_end_node, pivot_idx, k)


def _partition(begin_prev_node, end_node, begin_idx):
    # Store a pivot node and a current node pointers in variables
    pivot_node = begin_prev_node.next
    curr_node = pivot_node.next

    # Store indices of the beginning and the end of values equal to pivot span
    eq_pivot_head_idx = eq_pivot_tail_idx = begin_idx

    # 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
            eq_pivot_head_idx += 1
            eq_pivot_tail_idx += 1
        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
            eq_pivot_tail_idx += 1
        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, eq_pivot_head_idx, eq_pivot_tail_idx
```

Kilka testów

```python
In [36]: # test_select_ll(quick_select, LinkedList, print,
#                samples=1000, values_count=(1, 10000), range_=(-1000, 1000), failed_only=True)
test_select_ll(quick_select, LinkedList, print, samples=3)
```

```
TEST #1:
k: 16
Input arr:  [92, 81, -71, 34, -78, -2, -11, 65, 27, -76, 21, -97, 31, 18, 99, 92, -50, 55, -57, 98, 38, 57, -9
4, 35, -44, 36, -55, 0, -49, -13, 59, 94, 63, 84, 90, -80, -45, -60, -74, -83, -73, 10, 11, -13, 9, -76, -78, -
90, 66, 55, 34, -23, -58, -9, 8, 7, -98, -78, -12, 88, -13, 8, -8, -44, 67, -53, -99, -85, -84, -56, -28, 61, 1
3, 60, -79, 14, -84, 61, 11, -18, -94, 22, -55, 8]
Sorted arr: [-99, -98, -97, -94, -94, -90, -85, -84, -84, -83, -80, -79, -78, -78, -78, -76, -76, -74, -73, -7
1, -60, -58, -57, -56, -55, -55, -53, -50, -49, -45, -44, -44, -28, -23, -18, -13, -13, -13, -12, -11, -9, -8,
-2, 0, 7, 8, 8, 8, 9, 10, 11, 11, 13, 14, 18, 21, 22, 27, 31, 34, 34, 35, 36, 38, 55, 55, 57, 59, 60, 61, 61, 6
3, 65, 66, 67, 81, 84, 88, 90, 92, 92, 94, 98, 99]
linked list after: -97 -> -94 -> -80 -> -83 -> -90 -> -98 -> -99 -> -85 -> -84 -> -79 -> -84 -> -94 -> -78 -> -
78 -> -78 -> -76 -> -76 -> -74 -> -73 -> -71 -> 34 -> -2 -> -11 -> 65 -> 27 -> 21 -> 31 -> 18 -> -50 -> 55 -> -
57 -> 38 -> 57 -> 35 -> -44 -> 36 -> -55 -> 0 -> -49 -> -13 -> 59 -> 63 -> -45 -> -60 -> 10 -> 11 -> -13 -> 9 -
> 66 -> 55 -> 34 -> -23 -> -58 -> -9 -> 8 -> 7 -> -12 -> -13 -> 8 -> -8 -> -44 -> 67 -> -53 -> -56 -> -28 -> 61
-> 13 -> 60 -> 14 -> 61 -> 11 -> -18 -> 22 -> -55 -> 8 -> 81 -> 84 -> 90 -> 88 -> 92 -> 92 -> 99 -> 98 -> 94
Expected: -76
Result: -76
Test PASSED
Passed-to-tested ratio: 1/1

TEST #2:
k: 40
Input arr:  [50, -50, -3, -68, -63, -56, 49, -97, 58, 99, -16, -30, -74, 82, 76, 92, 2, 90, 53, 83, 45, -5, 91,
37, -2, -74, 22, 67, 1, 87, 2, 75, 84, -38, 100, -17, -49, -15, 95, -26, -87, -60]
Sorted arr: [-97, -87, -74, -74, -68, -63, -60, -56, -50, -49, -38, -30, -26, -17, -16, -15, -5, -3, -2, 1, 2,
2, 22, 37, 45, 49, 50, 53, 58, 67, 75, 76, 82, 83, 84, 87, 90, 91, 92, 95, 99, 100]
linked list after: -50 -> -3 -> -68 -> -63 -> -56 -> 49 -> -97 -> -16 -> -30 -> -74 -> 2 -> 45 -> -5 -> 37 -> -
2 -> -74 -> 22 -> 1 -> 2 -> -38 -> -17 -> -49 -> -15 -> -26 -> -87 -> -60 -> 50 -> 53 -> 58 -> 82 -> 76 -> 92 -
> 90 -> 83 -> 91 -> 67 -> 87 -> 75 -> 84 -> 95 -> 99 -> 100
Expected: 99
Result: 99
Test PASSED
Passed-to-tested ratio: 2/2

TEST #3:
k: 12
Input arr:  [25, 62, -2, -89, -88, 14, -37, -80, 94, -55, -2, 81, -11, -49, 63, 72, 81, 27, -68, 98, 39, 31, -5
5, 31, -14, 16, 66, -92]
Sorted arr: [-92, -89, -88, -80, -68, -55, -55, -49, -37, -14, -11, -2, -2, 14, 16, 25, 27, 31, 31, 39, 62, 63,
66, 72, 81, 81, 94, 98]
linked list after: -89 -> -88 -> -37 -> -80 -> -55 -> -11 -> -49 -> -68 -> -55 -> -14 -> -92 -> -2 -> -2 -> 14
-> 16 -> 25 -> 62 -> 94 -> 81 -> 63 -> 72 -> 81 -> 27 -> 98 -> 39 -> 31 -> 31 -> 66
Expected: -2
Result: -2
Test PASSED
Passed-to-tested ratio: 3/3

===== Final results: =====
Final passed-to-tested ratio: 3/3
An algorithm is CORRECT
```

› Implementacja algorytmu #2 (rekurencyjna) (dla implementacji funkcyjnej listy)

Cały algorytm poza funkcją

```python
def quick_select(ll_head: 'linked list head (sentinel)', k: 'index of a value'):
```

jest identyczny do powyższego. Wynika to z faktu, iż, w przypadku takich algorytmów, listę obiektową traktujemy identycznie jak zwykły ciąg węzłów (tzn. nie zwracamy uwagi na jej obiektowość, i zaimplementowane dla niej metody, ponieważ działamy tylko na węzłach).

In [37]:

```python
def quick_select(ll_head: 'linked list head (sentinel)', k: 'index of a value'):
    # If a linked list is not empty
    if ll_head.next and ll_head.next.next:
        # Perform sorting on a linked list
        return _quick_select(ll_head, None, 0, k)
    # If has exactly one element and k is equal to 0
    elif ll_head.next and k == 0:
        return ll_head.next.val
    # In other cases when wrong k value was passed
    else:
        raise IndexError(f'index too {"small" if k < 0 else "large"}')


def _quick_select(begin_prev_node, end_node, pivot_idx, k):
    res = _partition(begin_prev_node, end_node, pivot_idx)
    first_end_node, second_begin_prev_node, pivot_head_idx, pivot_tail_idx = res

    # If we placed a span of values equal to the pivot value on they final
    # position and the k index is somewhere within the bounds of this span.
    # we can return the last pivot value
    if pivot_head_idx <= k <= pivot_tail_idx:
        return second_begin_prev_node.val
    # If the last pivot value was placed before the k index, we have
    # to search a right part of a linked list for a desired value
    if pivot_tail_idx < k:
        return _quick_select(second_begin_prev_node, end_node, pivot_tail_idx + 1, k)
    # Otherwise we have to search the left part
    return _quick_select(begin_prev_node, first_end_node, pivot_idx, k)


def _partition(begin_prev_node, end_node, begin_idx):
    # Store a pivot node and a current node pointers in variables
    pivot_node = begin_prev_node.next
    curr_node = pivot_node.next

    # Store indices of the beginning and the end of values equal to pivot span
    eq_pivot_head_idx = eq_pivot_tail_idx = begin_idx

    # 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
            eq_pivot_head_idx += 1
            eq_pivot_tail_idx += 1
        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
            eq_pivot_tail_idx += 1
        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, eq_pivot_head_idx, eq_pivot_tail_idx
```

Kilka testów

In [38]:

```python
# test_select_ll(quick_select, create_linked_list, print_linked_list,
#                samples=1000, values_count=(1, 10000), range_=(-1000, 1000), failed_only=True)
test_select_ll(quick_select, create_linked_list, print_linked_list, samples=3)
```

```
TEST #1:
k: 3
Input arr:  [-98, -12, 98, -46, 93]
Sorted arr: [-98, -46, -12, 93, 98]
linked list after: None -> -98 -> -46 -> -12 -> 93 -> 98
Expected: 93
Result: 93
Test PASSED
Passed-to-tested ratio: 1/1

TEST #2:
k: 10
Input arr:  [83, 8, 44, -82, -8, -63, 69, -37, -41, 42, 62, -64, 88, -78, -98, 28, -4, -39, -86, 88, 78, 73, 2, -65, -61, -6, 10, 66, 28, 44, 46, -93, -25, -94, 76, 68, -36, 72, -40, 60, 59, -76, 47, -30, -45, 45, -44, -60, -21, -13, 40, 31, -80]
Sorted arr: [-98, -94, -93, -86, -82, -80, -78, -76, -65, -64, -63, -61, -60, -45, -44, -41, -40, -39, -37, -36, -30, -25, -21, -13, -8, -6, -4, 2, 8, 10, 28, 28, 31, 40, 42, 44, 44, 45, 46, 47, 59, 60, 62, 66, 68, 69, 72, 73, 76, 78, 83, 88, 88]
linked list after: None -> -98 -> -86 -> -93 -> -94 -> -82 -> -64 -> -78 -> -65 -> -76 -> -80 -> -63 -> -37 -> -41 -> -39 -> -61 -> -25 -> -36 -> -40 -> -30 -> -45 -> -44 -> -60 -> -21 -> -13 -> -8 -> -4 -> 2 -> -6 -> 8 -> 44 -> 69 -> 42 -> 62 -> 28 -> 78 -> 73 -> 10 -> 66 -> 28 -> 44 -> 46 -> 76 -> 68 -> 72 -> 60 -> 59 -> 47 -> 45 -> 40 -> 31 -> 83 -> 88 -> 88
Expected: -63
Result: -63
Test PASSED
Passed-to-tested ratio: 2/2

TEST #3:
k: 2
Input arr:  [-7, 63, -80, 88, 23, 88, -15, -3, -82]
Sorted arr: [-82, -80, -15, -7, -3, 23, 63, 88, 88]
linked list after: None -> -82 -> -80 -> -15 -> -7 -> 63 -> 88 -> 23 -> 88 -> -3
Expected: -15
Result: -15
Test PASSED
Passed-to-tested ratio: 3/3

===== Final results: =====
Final passed-to-tested ratio: 3/3
An algorithm is CORRECT
```

» Median of Medians

Algorytm, który wyznacza w czasie liniowym przybliżoną medianę liczb z pewnego zbioru danych

› Złożoność algorytmu

Złożoność czasowa

Każdy przypadek

$O(n)$

Złożoność pamięciowa

Każdy przypadek

$O(1)$

UWAGA: Taka złożoność pamięciowa dotyczy poniższej implementacji, gdzie wszystkie operacje odbywają się w miejscu. Wiele implementacji dostępnych w Internecie wykorzystuje dodatkowe tablice do przechowywania median i wówczas taka złożoność wynosi zazwyczaj $O(log(n))$.

Ponieważ sam algorytm wyznaczania przybliżonej wartości mediany jest trochę bezużyteczny, przejdziemy od razu do jego praktycznych zastosowań niżej.

› Quick Select

Zwraca wskazaną w kolejności wartość (liczbę, która znajdowała by się na wskazanej pozycji posortowanej tablicy). Jest to udoskonalona wersja, która gwarantuje zadziałanie w czasie liniowym w każdym przypadku.

Implementacje dla tablic

Implementacja algorytmu #1

(iteracyjna z iteracyjnym wybieraniem pivota, bez tworzenia dodatkowych tablic)

In [39]:

```python
def linear_select(arr: list, k: 'index of element'):
    if not 0 <= k < len(arr):
        raise IndexError(f'array index too {"small" if k < 0 else "large"}')
    if len(arr) == 1:
        return arr[0]

    # Prepare variables which indicate the bounds of the subarray searched
    left_idx = 0
    right_idx = len(arr) - 1

    # Loop till the subarray is not empty
    while left_idx <= right_idx:
        # Calculate a median of medians and store this value on the left_idx
        median_of_medians(arr, left_idx, right_idx)
        # Partition the current subarray using a median calculated above
        # as a pivot value
        pivot_idx = _partition(arr, left_idx, right_idx)

        # If a pivot was placed before the index desired, we have to look for
        # a desired value int the right part of the current subarray
        if pivot_idx < k:
            left_idx = pivot_idx + 1
        # If a pivot was placed after the index desired, we have to search
        # for a value in the left part of the current subarray
        elif pivot_idx > k:
            right_idx = pivot_idx - 1
        # Otherwise, (if k == pivot_idx) return a value which was searched
        else:
            return arr[k]


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

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

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

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

    # Finally, swap a median of the subarray of medians (which has no more than
    # k elements) with the first (leftmost) value of the subarray
    _swap(arr, left_idx, _select_median(arr, left_idx, right_idx))
    # Return a value of a median
    return arr[left_idx]


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


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


def _partition(arr: list, left_idx: int, right_idx: int) -> int:
    # After running the median of medians function a pivot (this median of medians)
    # will be placed on the left_idx of the subarray
    pivot = arr[left_idx]

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

    # Place a pivot element on its destination index
    _swap(arr, i - 1, left_idx)

    return i - 1  # Return a pivot position after the last swap
```

Kilka testów

In [40]:

```python
# test_select(linear_select, range_=(-10_000, 10_000), values_count=(1, 10_000), samples=10000, failed_only=Tru
test_select(linear_select, range_=(-10_000, 10_000), values_count=(1, 100), samples=25, failed_only=True)
```

```
===== Final results: =====
Final passed-to-tested ratio: 25/25
An algorithm is CORRECT
```

Implementacja algorytmu #2

(rekurencyjna z rekurencyjnym wybieraniem pivota, bez tworzenia dodatkowych tablic)

In [41]:

```python
def linear_select(arr: list, k: 'index of element'):
    if not 0 <= k < len(arr):
        raise IndexError(f'array index too {"small" if k < 0 else "large"}')
    if len(arr) == 1:
        return arr[0]
    return _linear_select(arr, k, 0, len(arr) - 1)


def _linear_select(arr, k, left_idx, right_idx):
    # Calculate a median of medians and store this value on the left_idx
    median_of_medians(arr, left_idx, right_idx)
    # Partition the current subarray using a median calculated above
    # as a pivot value
    pivot_idx = _partition(arr, left_idx, right_idx)

    # Return a value selected if pivot_idx is equal to the desired position
    if pivot_idx == k: return arr[k]
    # If a pivot was placed before the index desired, we have to look for
    # a desired value int the right part of the current subarray
    elif pivot_idx < k:
        return _linear_select(arr, k, pivot_idx + 1, right_idx)
    # If a pivot was placed after the index desired, we have to search
    # for a value in the left part of the current subarray
    else:
        return _linear_select(arr, k, left_idx, pivot_idx - 1)


def median_of_medians(arr: list, left_idx: int, right_idx: int, k: int = 5) -> 'median of medians':
    if right_idx - left_idx < k:
        # Finally, swap a median of the subarray of medians (which has no more than
        # k elements) with the first (leftmost) value of the subarray
        _swap(arr, left_idx, _select_median(arr, left_idx, right_idx))
        # Return a value of a median
        return arr[left_idx]
    else:
        # Store the position on which the next median will be stored
        # (we will store each median of current k-element subarrays one
        # after another at the beginning of the subarray which begins
        # on the left_index and ends on the right_idx (inclusive)
        next_swap_idx = left_idx

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

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

        # Search for an approximate median of medians recursively
        return median_of_medians(arr, left_idx, next_swap_idx - 1, k)


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


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


def _partition(arr: list, left_idx: int, right_idx: int) -> int:
    # After running the median of medians function a pivot (this median of medians)
    # will be placed on the left_idx of the subarray
    pivot = arr[left_idx]

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

    # Place a pivot element on its destination index
    _swap(arr, i - 1, left_idx)

    return i - 1  # Return a pivot position after the last swap
```

Kilka testów

In [42]:

```python
# test_select(linear_select, range_=(-10_000, 10_000), values_count=(1, 10_000), samples=10_000, failed_only=Tr
test_select(linear_select, range_=(-10_000, 10_000), values_count=(1, 100), samples=25, failed_only=True)
```

```
===== Final results: =====
Final passed-to-tested ratio: 25/25
An algorithm is CORRECT
```

› Median Select

Jest to drobna modyfikacja powyższego algorytmu, z tym, że od razu poszukujemy elementu ze środka tablicy, więc wartość k jest odgórnie ustalona.

Implementacja algorytmu #1

(iteracyjna z iteracyjnym wybieraniem pivota, bez tworzenia dodatkowych tablic) (wersji rekurencyjnej nie implementuję, bo jest analogiczna)

W poniższej deklaracji funkcji

```
def median_select(arr: list, _even_arr_case=0):
```

używamy parametru _even_arr_case, który ma na celu wskazanie zachowania w przypadku, gdy liczba elementów w tablicy jest parzysta, przez co wybranie właściwej mediany jest niemożliwe. Domyślnie parametr ten ma przypisaną wartość 0, co będziemy interpretować jako obliczenie średniej arytmetycznej z dwóch środkowych wartości posortowanej tablicy. Łatwo można zauważyć, że wówczas konieczne jest znalezienie dwóch wartości, więc 2 razy musimy skorzystać z algorytmu szybkiego (liniowego) wybierania. Jeżeli ustawimy wartość tego parametru na -1, wówczas zwrócona zostanie wcześniej występująca wartość (ta, znajdująca się z lewej strony środka tablicy), natomiast dla wartości 1, zwrócona zostanie prawa mediana, czyli wartość znajdująca się z prawej strony środka tablicy (oczywiście jest to wartość, która znalazłaby się na tej pozycji, po uprzednim posortowaniu tablicy niemalejąco, ale my tego sortowania nie przeprowadzamy).

In [43]:

```python
def median_select(arr: list, _even_arr_case=0):
    if not arr: return None
    if len(arr) == 1: return arr[0]

    mid_idx = len(arr) // 2
    # If an array has odd number of elements, select the middle value
    if len(arr) % 2: return linear_select(arr, mid_idx)
    # If a number of array values is even and _even_arr_case is set to
    # 0, return an average of two elements selected on the indices
    # surrounding the middle of an array
    if _even_arr_case == 0:
        res = (linear_select(arr, mid_idx) + linear_select(arr, mid_idx - 1)) / 2
        return int(res) if int(res) == res else res
    # Return the left median
    if _even_arr_case == -1: return linear_select(arr, mid_idx - 1)
    # Return the left right
    if _even_arr_case == 1: return linear_select(arr, mid_idx)
    # If _even_arr_case has not a valid value, raise an exception
    raise ValueError(f'Wrong value of _even_arr_case. Expected -1, 0 or 1, got {_even_arr_case}')


def linear_select(arr: list, k: 'index of element'):
    # Prepare variables which indicate the bounds of the subarray searched
    left_idx = 0
    right_idx = len(arr) - 1

    # Loop till the subarray is not empty
    while left_idx <= right_idx:
        # Calculate a median of medians and store this value on the left_idx
        median_of_medians(arr, left_idx, right_idx)
        # Partition the current subarray using a median calculated above
        # as a pivot value
        pivot_idx = _partition(arr, left_idx, right_idx)

        # If a pivot was placed before the index desired, we have to look for
        # a desired value int the right part of the current subarray
        if pivot_idx < k:
            left_idx = pivot_idx + 1
        # If a pivot was placed after the index desired, we have to search
        # for a value in the left part of the current subarray
        elif pivot_idx > k:
            right_idx = pivot_idx - 1
        # Otherwise, (if k == pivot_idx) return a value which was searched
        else:
            return arr[k]


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

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

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

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

    # Finally, swap a median of the subarray of medians (which has no more than
    # k elements) with the first (leftmost) value of the subarray
    _swap(arr, left_idx, _select_median(arr, left_idx, right_idx))
    # Return a value of a median
    return arr[left_idx]


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


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


def _partition(arr: list, left_idx: int, right_idx: int) -> int:
    # After running the median of medians function a pivot (this median of medians)
    # will be placed on the left_idx of the subarray
    pivot = arr[left_idx]

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

    # Place a pivot element on its destination index
    _swap(arr, i - 1, left_idx)

    return i - 1  # Return a pivot position after the last swap
```

Kilka testów

In [55]:

```python
import random

even_case = -1

arr = [random.randint(0, 100) for _ in range(random.randint(0, 25))]
print('Input array:', arr, sep='\n')
sorted_arr = sorted(arr)
print('Sorted array:', sorted_arr, sep='\n')
print('Array length:', len(arr))
print('Result:', median_select(arr, even_case))
mid_idx = len(sorted_arr) // 2
res = None
if len(sorted_arr) % 2 or (even_case == 1 and sorted_arr):
    res = sorted_arr[mid_idx]
elif sorted_arr:
    if even_case == 0:
        res = sum(sorted_arr[mid_idx-1:mid_idx+1])/2
        if int(res) == res: res = int(res)
    elif even_case == -1:
        res = sorted_arr[mid_idx-1]
print('Expected:', res)
```

```
Input array:
[90, 38, 94, 87, 6, 78, 33, 4, 92, 94, 39, 72, 25, 95, 48, 51, 85]
Sorted array:
[4, 6, 25, 33, 38, 39, 48, 51, 72, 78, 85, 87, 90, 92, 94, 94, 95]
Array length: 17
Result: 72
Expected: 72
```