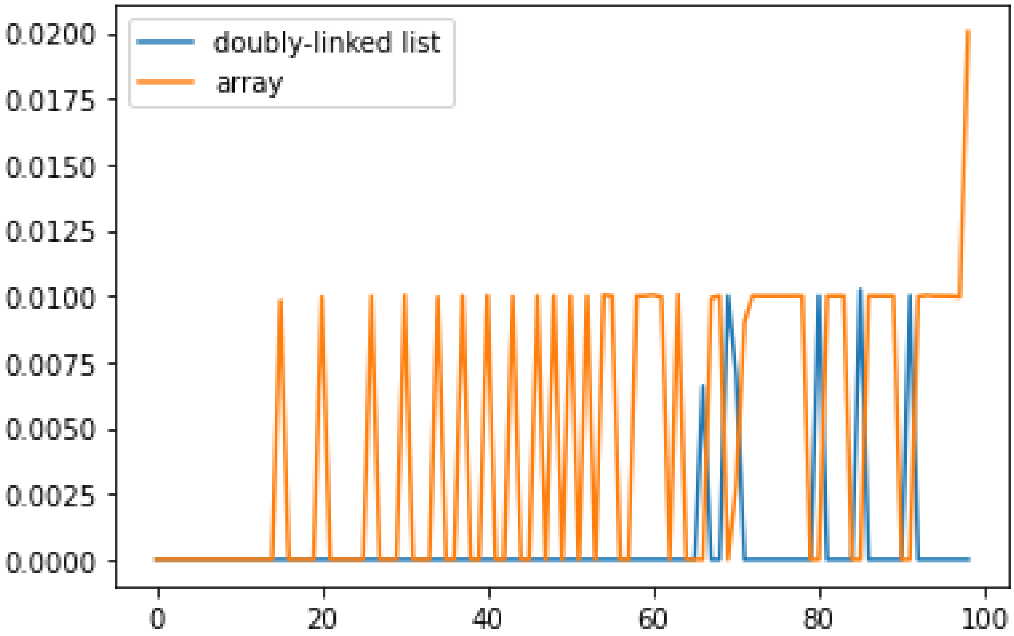

Merge sort on doubly linked list is more efficient than on array most of the time. This is caused by the amortisation during append operation in the merge process of array merge (as the array needs to change its length from time to time), while the merge on doubly linked list is merely redirections between the links.

```python
import time
import numpy as np
import matplotlib.pyplot as plt

class _DoublyLinkedBase:
    """A base class providing a doubly linked list representation."""

    class _Node:
        """Lightweight, nonpublic class for storing a doubly linked node."""
        __slots__ = '_element', '_prev', '_next'    # streamline memory

        def __init__(self, element, prev, next):    # initialize node's fields
            self._element = element                 # user's element
            self._prev = prev                       # previous node reference
            self._next = next                       # next node reference

    def __init__(self):
        """Create an empty list."""
        self._header = self._Node(None, None, None)
        self._trailer = self._Node(None, None, None)
        self._header._next = self._trailer          # trailer is after header
        self._trailer._prev = self._header          # header is before trailer
```

```python
        self._size = 0                              # number of elements

    def __len__(self):
        """Return the number of elements in the list."""
        return self._size

    def is_empty(self):
        """Return True if list is empty."""
        return self._size == 0

    def _insert_between(self, e, predecessor, successor):
        """Add element e between two existing nodes and return new node."""
        newest = self._Node(e, predecessor, successor) # linked to neighbours
        predecessor._next = newest
        successor._prev = newest
        self._size += 1
        return newest

    def _delete_node(self, node):
        """Delete nonsentinel node from the list and return its element."""
        predecessor = node._prev
        successor = node._next
```

```
        predecessor._next = successor
        successor._prev = predecessor
        self._size -= 1
        element = node._element                          # record deleted element
        node._prev = node._next = node._element = None   # depreciate node
        return element                                   # return deleted element

# Function to split nodes of the given doubly linked list into
# two halves using the fast/slow pointer strategy
def split(head):

    slow = head
    fast = head._next
    # advance `fast` by two nodes, and advance `slow` by a single node
    while fast:
        fast = fast._next
        if fast:
            slow = slow._next
            fast = fast._next

    return slow
```

```
# Recursive function to merge nodes of two sorted lists
# into a single sorted list
def d_merge(a, b):

    # base cases
    if a or a._element is None:
        return b

    if b or b._element is None:
        return a

    # pick either `a` or `b`, and recur
    if a._element <= b._element:
        a._next = d_merge(a._next, b)
        a._next._prev = a
        a._prev = None
        return a
    else:
        b._next = d_merge(a, b._next)
        b._next._prev = b
        b._prev = None
```

```
        return b

# Function to sort a doubly-linked list using merge sort algorithm
def d_mergesort(head):

    # base case: 0 or 1 node
    if head is None or head._next is None:
        return head

    # split head into `a` and `b` sublists
    a = head

    slow = split(head)
    b = slow._next
    slow._next = None

    # recursively sort the sublists
    a = d_mergesort(a)
    b = d_mergesort(b)

    # merge the two sorted lists
    head = d_merge(a, b)
```

```
    return head



def merge_a(A,p,q,r):
    n1 = q - p + 1
    n2 = r - q
   # print("\nmerging, p="+str(p)+' q='+str(q)+" r="+str(r)+" n1="+str(n1)+" n2="+str(n2) + ",
arr[p:r]="+str(arr[p:r+1]))
    L = []
    R = []
    for i in range(n1):
        L.append(A[p+i])
    for i in range(n2):
        R.append(A[q+i+1])
    L.append(99999)
    R.append(99999)
   # print('L = ' + str(L) + " R = " + str(R))

    i=0
    j=0
    for k in range(p,r+1):
        if L[i] <= R[j]:
```

```
            A[k] = L[i]
            i += 1
        else:
            A[k] = R[j]
            j += 1
    #print("MERGED = " + str(A)+"\n")

def mergesort(A,p,r):
   # if p<r:
       # print('p<r? True. p = ' + str(p) +" r = " + str(r))
   # else:
       # print('p<r? False. p = ' + str(p) +" r = " + str(r) +", RETURNING...")
    if p < r:
        q = (p + r) // 2
       # print("calling merge_sort (left half) with p=" + str(p) + ' r='+str(q))
        mergesort(A,p,q)
       # print("calling merge_sort(right half) with p="+str(q+1)+' r='+str(r))
        mergesort(A,q+1,r)
        merge_a(A,p,q,r)


if __name__ == '__main__':
```

```
times = []
times2 = []

for i in range(10,1000,10):
    keys = np.random.randint(0, 1000, i)
    d = _DoublyLinkedBase()
    for key in keys:
        d._insert_between(key, d._header, d._header._next)
    start_t = time.time()
    head = d_mergesort(d._header)
    times.append(time.time() - start_t)
    start_t = time.time()
    mergesort(keys,0,i-1)
    times2.append(time.time() - start_t)

plt.plot(times, label="doubly-linked list")
plt.plot(times2, label="array")
plt.legend()
```