Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 2:
Heapsort,Mergesort,Quick sort and Radixsort

Elaborated:
FAF-233 — Mohamed Dhiaeddine Hassine

Verified:
asist. univ. — Fiștic Cristofor

Chişinău - 2025

So I implemented 4 different algorithms

Quicksort:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

Heapsort:

```
def heapsort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        _heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        _heapify(arr, i, 0)
    return arr

def _heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left] > arr[largest]:
        largest = left
    if right < n and arr[right] > arr[largest]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        _heapify(arr, n, largest)
```

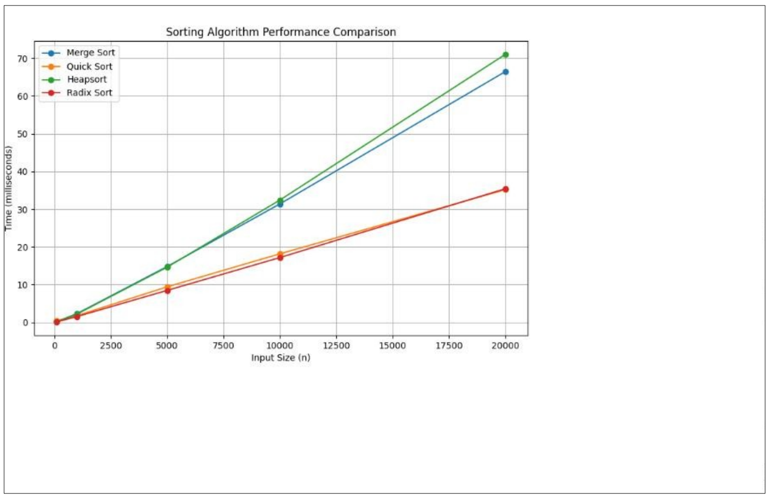
Radixsort:

```
def radix_sort(arr):
    if not arr:
        return arr
    max_val = max(arr)
    exp = 1
    while max_val // exp > 0:
        _counting_sort(arr, exp)
        exp *= 10
    return arr
```

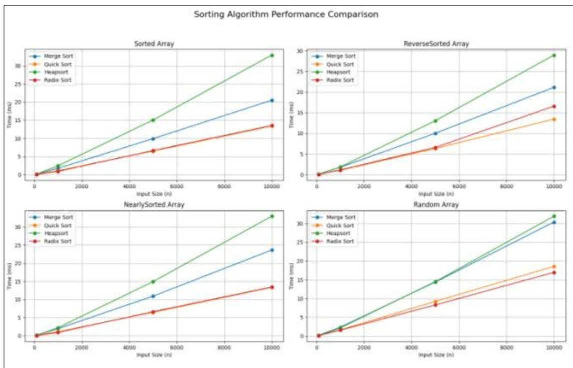
Mergesort:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return _merge(left, right)

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



A detailed comparative evaluation of the performance characteristics of each sorting algorithm, encompassing not only their execution time efficiency and algorithmic complexity but also their memory usage, stability, and responsiveness to varying input distributions



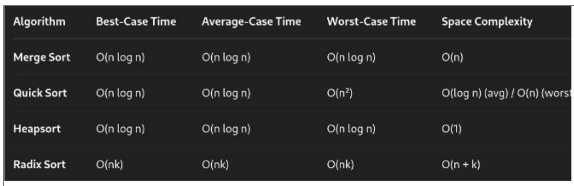
A comprehensive comparative analysis of Merge Sort, Quick Sort, Heap Sort, and Radix Sort reveals distinct performance characteristics across best, average, and worst-case scenarios, as well as varying space complexities:

Merge Sort: Exhibits consistent time complexity of O(n log n) in all cases, with a space complexity of O(n) due to the need for auxiliary arrays.

Quick Sort: Offers an average and best-case time complexity of O(n log n), but can degrade to O(n^2) in the worst case, particularly with poor pivot choices; it has a space complexity of O(log n) due to recursive calls.

Heap Sort: Maintains a steady time complexity of O(n log n) across all cases and is space-efficient with a complexity of O(1), as it sorts in place without additional memory.

Radix Sort: Achieves linear time complexity of O(nk), where k is the number of digits in the largest number, making it efficient for fixed-length integers; however, it requires additional space of O(n + k) for counting sort operations used in each digit's processing.



Algorithm	Best-Case Time	Average-Case Time	Worst-Case Time	Space Complexity
Merge Sort	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)	O(log n) (avg) / O(n) (worst
Heapsort	O(n log n)	O(n log n)	O(n log n)	O(1)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)

Conclusion:

Merge Sort guarantees $O(n \log n)$ performance in all cases and is stable, which is advantageous for preserving the order of equal elements and for use with linked lists or parallel environments. Its principal drawback is the $O(n)$ extra space required for merging. Quick Sort, with randomized pivot selection, typically achieves $O(n \log n)$ expected time and benefits from excellent cache locality and low overhead due to its in-place partitioning; however, its worst-case $O(n^2)$ behavior (if not carefully implemented) can be problematic on certain inputs. Heap Sort offers a robust worst-case $O(n \log n)$ guarantee in constant extra space ($O(1)$), making it reliable where memory is limited, though its scattered memory accesses can slow its practical performance compared to Quick Sort. Radix Sort—by processing keys digit by digit—can achieve near-linear time $O(n \cdot k)$ when k (the number of digits) is small, but this efficiency is data-specific and comes with additional space overhead of $O(n + k)$.

In summary, no comparison-based sort can surpass the $\Omega(n \log n)$ lower bound, so choosing the “better” algorithm depends on context: Quick Sort is preferred for in-memory, average-case speed; Merge Sort is ideal where stability is essential; Heap Sort is useful when worst-case guarantees and minimal additional space are required; and Radix Sort can outperform the others for suitable numerical or fixed-key data despite its specialized nature.