Ministerul Educaţiei și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 2:

Study and empirical analysis of sorting algorithms

Analysis of quickSort, mergeSort, heapSort, (one of your choice)

Elaborated:
st. gr. FAF-211

Spătaru Dionisie

Verified:

asist. univ.

Fiștic Cristofor

Chişinău - 2023

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of sorting algorithms.

Tasks

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical Notes:

Sorting algorithms are fundamental tools in computer science that are used to arrange a collection of data in a specific order. The goal of sorting algorithms is to arrange the elements of an array or list in ascending or descending order, based on a comparison of the values of these elements.

Sorting algorithms are commonly used in a variety of applications, including database systems, search algorithms, and statistical analysis. There are several important factors to consider when evaluating the performance of a sorting algorithm, including its efficiency, simplicity, and stability.

Efficiency is a critical consideration when choosing a sorting algorithm, as it determines how quickly the algorithm can process large sets of data. The efficiency of an algorithm is typically measured by its time complexity, which describes how the algorithm's runtime increases as the size of the input data increases. Sorting algorithms can have different time complexities, ranging from O(n^2) for simple algorithms like bubble sort and insertion sort, to O(n log n) for more advanced algorithms like merge sort and quicksort.

Simplicity is also an important consideration when evaluating sorting algorithms, as it can affect how easy it is to implement and maintain the algorithm. Simple algorithms are typically easier to understand and modify, but may not always provide the best performance.

Stability is a characteristic of sorting algorithms that ensures that elements with equal values retain their relative order after the sorting process. Stable algorithms are important in applications where the original ordering of equal elements is significant.

Overall, the goal of sorting algorithms is to provide an efficient and effective method for arranging collections of data in a specific order, based on the specific needs of the application. The choice of sorting algorithm will depend on the size of the input data, the desired level of performance, and other considerations such as simplicity and stability.

Comparison Metric:

Efficiency and speed of sorting

IMPLEMENTATION

Quick sort

```
def quick_sort(vector):
   """
   Sorts a vector using the quicksort algorithm.
   """
   if len(vector) <= 1:
       return vector
   else:
       pivot = vector[0]
       left = []
       right = []
       for i in range(1, len(vector)):
           if vector[i] < pivot:
               left.append(vector[i])
           else:
               right.append(vector[i])
       return quick_sort(left) + [pivot] + quick_sort(right)
```

Quick sort is a sorting algorithm that selects a pivot element, partitions the array into two sub-arrays, places the pivot in its final position, and recursively applies this process to the sub-arrays until the entire array is sorted. It has an average-case time complexity of O(n log n) but can have a worst-case time complexity of O(n^2).

Merge sort

```
def merge_sort(vector):
   """
   Sorts a vector using the merge sort algorithm.
   """
   if len(vector) > 1:
       mid = len(vector) // 2
       left_half = vector[:mid]
       right_half = vector[mid:]

       merge_sort(left_half)
       merge_sort(right_half)

       i = j = k = 0
       while i < len(left_half) and j < len(right_half):
           if left_half[i] < right_half[j]:
               vector[k] = left_half[i]
               i += 1
           else:
               vector[k] = right_half[j]
               j += 1
           k += 1
```

```
        while i < len(left_half):
            vector[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            vector[k] = right_half[j]
            j += 1
            k += 1
```

Merge sort is a divide-and-conquer sorting algorithm that works by recursively splitting an array into two halves, sorting each half, and then merging the two sorted halves into a single sorted array. It has a time complexity of O(n log n) and is generally considered to be a highly efficient and stable sorting algorithm.

Heap sort

```
def heap_sort(vector):
    """
    Sorts a vector using the heapsort algorithm.
    """
    n = len(vector)
    for i in range(n // 2 - 1, -1, -1):
        heapify(vector, n, i)
    for i in range(n - 1, 0, -1):
        vector[0], vector[i] = vector[i], vector[0]
        heapify(vector, i, 0)

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

Heap sort is a sorting algorithm that works by first creating a binary heap data structure from the array to be sorted, and then repeatedly extracting the maximum element from the heap and placing it at the end of the sorted portion of the array. In Python, the heap module provides a heap implementation that can be used for heap sort. The basic steps are:

1. Convert the array to a heap using the heapify() function from the heapq module.
2. While the heap is not empty, repeatedly extract the maximum element from the heap using the heappop() function and place it at the end of the array.
3. Reverse the order of the array to get it in ascending order.

Overall, heap sort has a time complexity of O(n log n) and can be an efficient sorting algorithm for certain types of data.

Selection sort

```
def selection_sort(vector):
    """
    Sorts a vector using the selection sort algorithm.
    """
    n = len(vector)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if vector[j] < vector[min_idx]:
                min_idx = j
        vector[i], vector[min_idx] = vector[min_idx], vector[i]
```

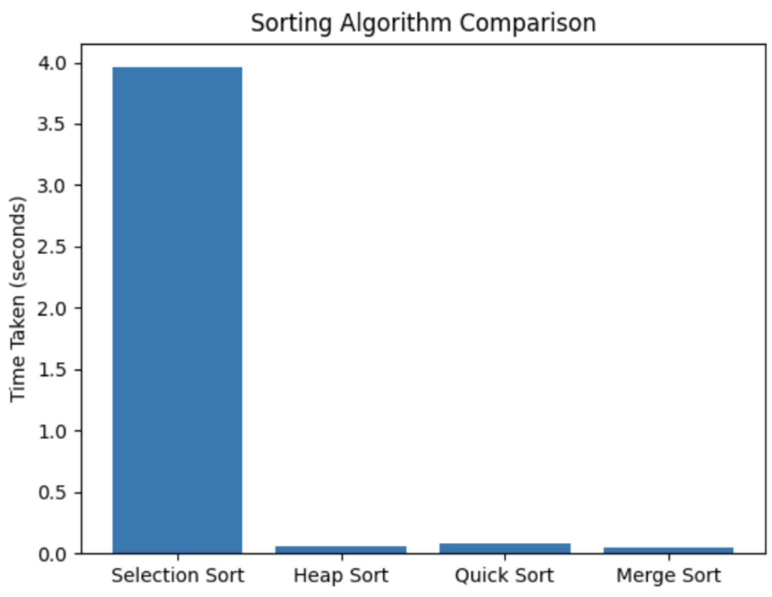
Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted portion of an array and swapping it with the first unsorted element. In Python, selection sort can be implemented using a nested loop and a temporary variable to hold the minimum value. The basic steps are:

1. Iterate over the unsorted portion of the array.
2. Find the minimum element in the unsorted portion of the array.
3. Swap the minimum element with the first unsorted element.
4. Repeat steps 1-3 until the entire array is sorted.

Selection sort has a time complexity of O(n^2), which makes it inefficient for large datasets. However, it is a simple and easy-to-implement sorting algorithm and can be useful for small datasets or as a building block for more complex sorting algorithms.

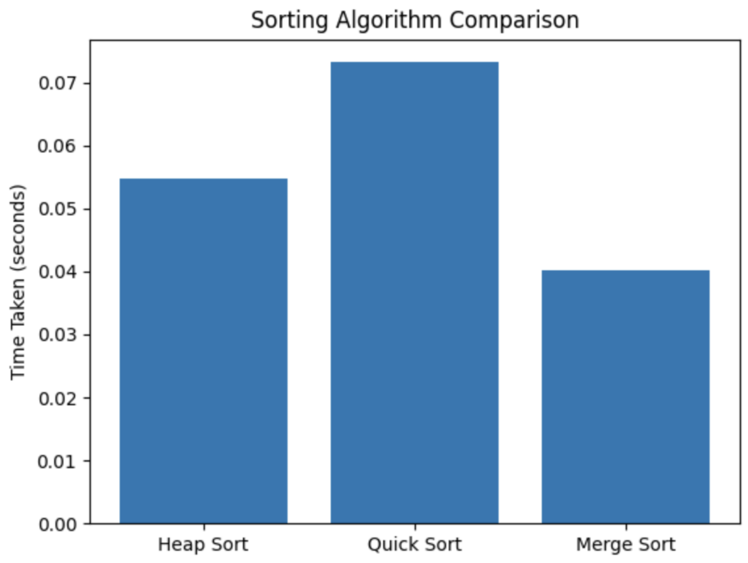
COMPARISON AND RESULTS

Firstly I tried to compare all my implemented algorithms together on a plotted graph.

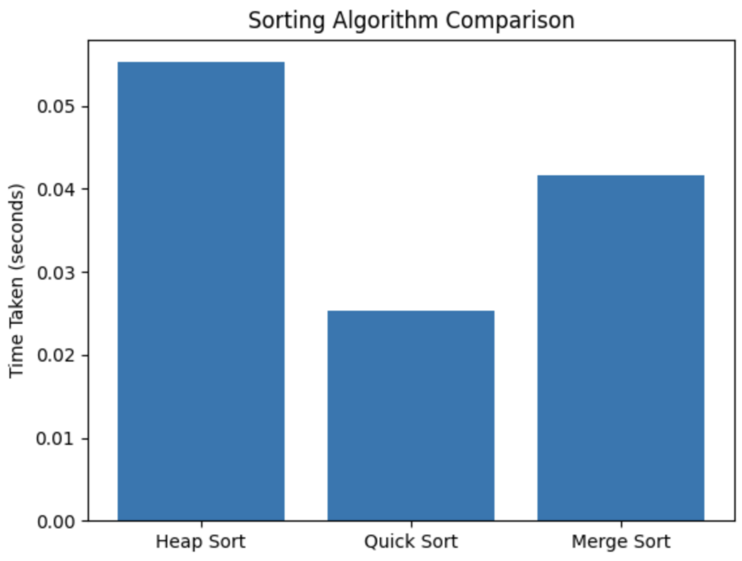


But as we can see, the execution time to sort a vector with 10,000 random int numbers for Selection sort tooks much longer time, so I had to compare other 3 main sorting algorithms(Heap, Quick, Merge).

Firstly I tried to compare this algorithms using an array with 10,000 integers in range **[0 to 100] for each int**.(values)



And we can observe that in this case Quick sort is a bit slower then Heap and Merge sort.
But if we change the range of values for each int from the vector from [0 to 100] to, for example, `[-1000000, 1000000]...`



We observe that Quick sort is faster then Merge and Heap.

CONCLUSION

When working with large datasets, Merge sort and Heap sort are often preferred over Quick sort and Selection sort due to their O(n log n) time complexity. Merge sort is a stable sorting algorithm that works well with various types of data, while Heap sort is an efficient in-place sorting algorithm that works well with numerical data.

Quick sort can be highly efficient in practice, especially when using randomized pivot selection or other optimizations. However, it has a worst-case time complexity of O(n^2) if the pivot is not chosen well, which can be problematic for certain types of datasets.

Selection sort, on the other hand, is a simple and easy-to-implement algorithm that works well for small datasets or as a building block for more complex sorting algorithms. However, its O(n^2) time complexity makes it inefficient for larger datasets.

Overall, the choice of sorting algorithm depends on the specific requirements of the application. For example, if performance is critical and the dataset is large and varied, Merge sort or Heap sort may be the best choice. If simplicity is a priority and the dataset is small, Selection sort may be sufficient. If efficiency is important and the dataset is well-suited to it, Quick sort may be a good choice.

REPOSITORY

https://github.com/denyred/AA_Lab2