MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 2 : Study and empirical analysis of sorting algorithms.

Elaborated:
st.gr. FAF-211 Grama Alexandru
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Sorting algorithms are a fundamental concept in computer science and are essential for organizing and processing data efficiently. In computer science, sorting refers to the process of rearranging a collection of items in a specific order. Sorting algorithms are used to sort data structures like arrays, lists, and trees, and the efficiency of the sorting algorithm is determined by the number of comparisons and swaps it takes to sort the data.

There are numerous sorting algorithms, each with its own strengths and weaknesses. Some of the most popular sorting algorithms include bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort. Each algorithm has its own approach to sorting data, with different time and space complexity.

Understanding sorting algorithms is crucial for computer scientists, as efficient sorting can improve the performance of various applications such as databases, search engines, and operating systems. Moreover, a deeper understanding of sorting algorithms can help you become a better programmer and problem-solver, regardless of the field you're in. Sorting algorithms can be categorized as either internal or external, depending on how the data is stored during the sorting process. Internal sorting algorithms sort data that can fit into the main memory of a computer, while external sorting algorithms are used for data that is too large to fit into main memory and must be sorted on disk or other external storage media.

Efficient sorting algorithms are vital for handling large datasets, where the time and space complexity of the algorithm can make a significant impact on the performance. There are various ways to measure the efficiency of sorting algorithms, including the number of comparisons and swaps, the time taken to sort the data, and the amount of memory used.

Sorting algorithms also have numerous real-world applications, from sorting names in a phone book to sorting large datasets in scientific research. They are used in various industries, including finance, healthcare, and engineering. Additionally, sorting algorithms are often used in combination with other algorithms to optimize complex operations, such as data compression and searching.

Overall, sorting algorithms play a crucial role in computer science and data processing. Understanding how they work and their different approaches can help you choose the most efficient algorithm for a given task, optimize the performance of your code, and improve your problem-solving skills.

QuickSort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. Quicksort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. For this reason, it is sometimes called partition-exchange sort.[4] The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform thesorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Most implementations of quicksort are not stable, meaning that the relative order of equal sort items is not preserved.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) compar- isons to sort n items. In the worst case, it makes O(n^2) comparisons.

Code:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quicksort(left) + [pivot] + quicksort(right)
```

Mergesort

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

Code:

```python
def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result
```

HeapSort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable, but can be made stable
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

The heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

What is meant by Heapify?

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the last index of the non-leaf node whose index is given by n/2 – 1. Heapify uses recursion.

Code:

```
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)
```

```
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
```

BubbleSort

Bubble sort, sometimes referred to as sinking sort, is a simple sorting algorithm that repeatedly steps through the input list element by element, comparing the current element with the one after it, swapping their values if needed. These passes through the list are repeated until no swaps had to be performed during a pass, meaning that the list has become fully sorted. The algorithm, which is a comparison sort, is named for the way the larger elements "bubble" up to the top of the list.

This simple algorithm performs poorly in real world use and is used primarily as an educational tool. More efficient algorithms such as quicksort, timsort, or merge sort are used by the sorting libraries built into popular programming languages such as Python and Java.

Code:

```
def bubblesort(arr):
```

```python
    n = len(arr)
    swapped = False
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            return arr
```

Implementation

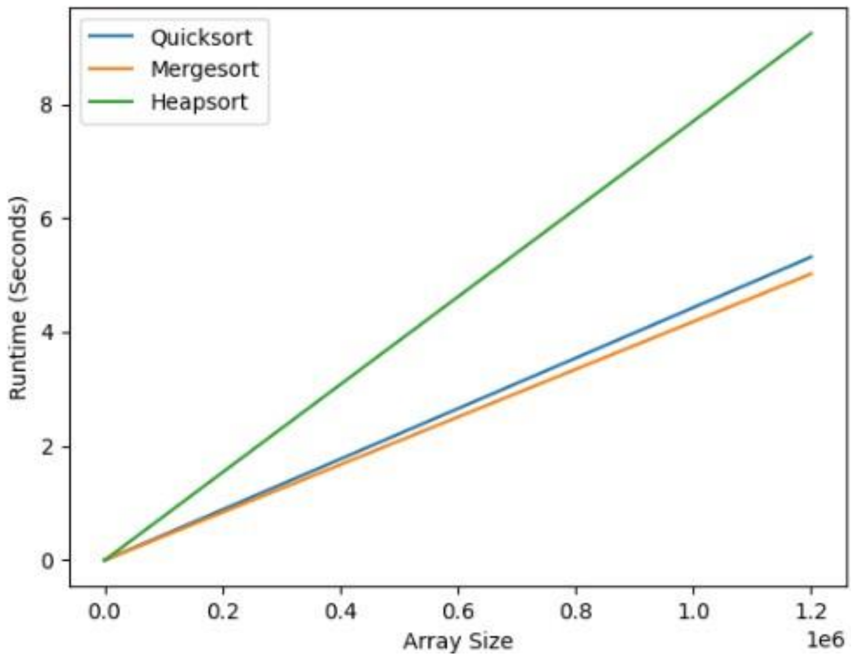
Code in python:

```python
import time
import random
import matplotlib.pyplot as plt

# Define the sorting algorithms

def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x < pivot]
    right = [x for x in arr[1:] if x >= pivot]
    return quicksort(left) + [pivot] + quicksort(right)

def mergesort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = mergesort(arr[:mid])
    right = mergesort(arr[mid:])
    result = []
    i, j = 0, 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result

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)

def heapsort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
```

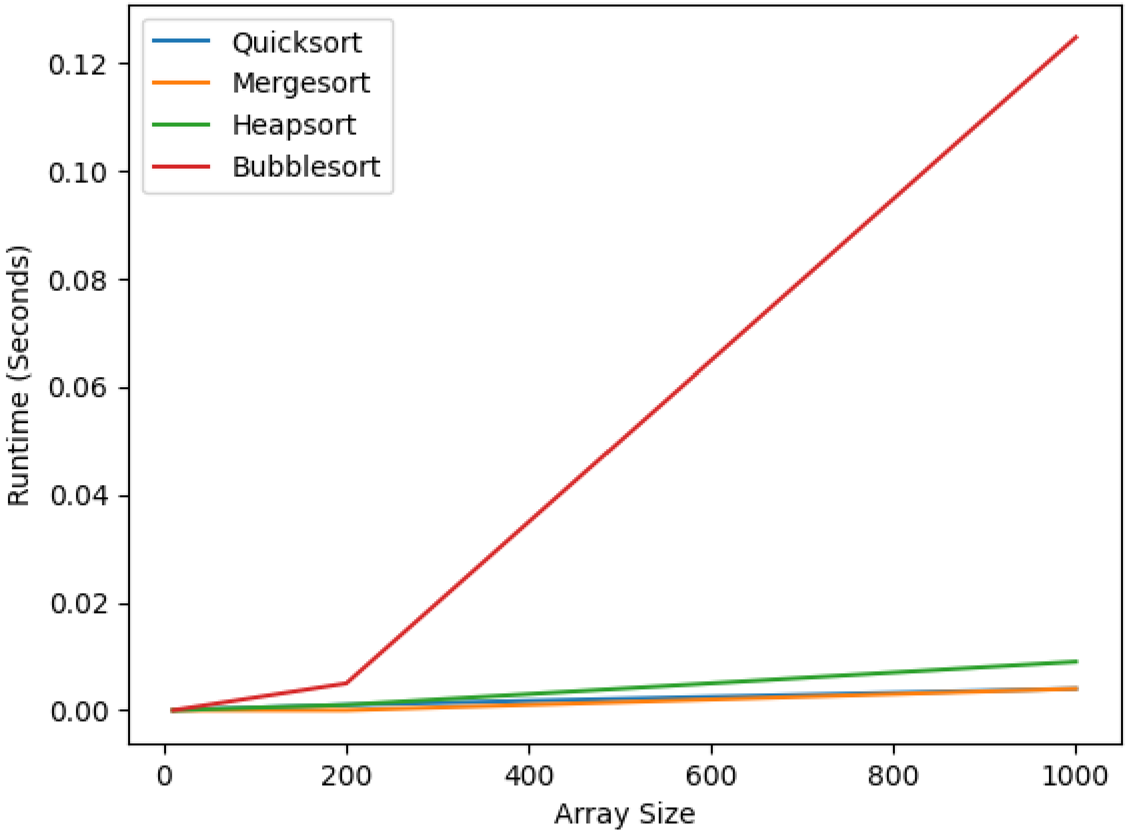
```
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr

def bubblesort(arr):
    n = len(arr)
    swapped = False
    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            return arr

# Generate random arrays to sort

sizes = [10, 200, 1000]
arrays = {}
for size in sizes:
    arrays[size] = [random.randint(1, size) for _ in range(size)]

# Sort the arrays with each algorithm and record the runtimes

quicksort_times = []
mergesort_times = []
heapsort_times = []
bubblesort_times = []

for size in sizes:
    array = arrays[size]
    start_time = time.time()
    quicksort(array.copy())
    end_time = time.time()
    quicksort_times.append(end_time - start_time)

    array = arrays[size]
    start_time = time.time()
    mergesort(array.copy())
    end_time = time.time()
    mergesort_times.append(end_time - start_time)

    array = arrays[size]
    start_time = time.time()
    heapsort(array.copy())
    end_time = time.time()
    heapsort_times.append(end_time - start_time)

    array = arrays[size]
    start_time = time.time()
```

```python
    bubblesort(array.copy())
    end_time = time.time()
    bubblesort_times.append(end_time - start_time)


plt.plot(sizes, quicksort_times, label="Quicksort")
plt.plot(sizes, mergesort_times, label="Mergesort")
plt.plot(sizes, heapsort_times, label="Heapsort")
plt.plot(sizes, bubblesort_times, label="Bubblesort")
plt.xlabel("Array Size")
plt.ylabel("Runtime (Seconds)")
plt.legend()
plt.show()
```

Screenshot:



Quicksort
Mergesort
Heapsort
Runtime (Seconds)
0
2
4
6
8
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e6
Array Size


Quicksort
Mergesort
Heapsort
Bubblesort
0.12
0.10
0.08
0.06
0.04
0.02
0.00
Runtime (Seconds)
0
200
400
600
800
1000
Array Size

Conclusion

To sum up, the performance of sorting algorithms such as quicksort, mergesort, heapsort, and bubblesort can vary when sorting an array of 1000 elements, depending on factors like input data, hardware and software environment, and implementation details. Quicksort and mergesort typically have an average time complexity of O(n log n) and are often preferred in practice, with quicksort being favored for its lower constant factors and better cache locality. Heapsort is a good option when stable sorting is not required, and space complexity is a concern. Bubblesort is a well-known but slow algorithm and is not used much.

Therefore, for sorting large arrays efficiently and flexibly, quicksort is a popular and reliable choice, but mergesort or heapsort may also be suitable depending on specific requirements and constraints. It's important to test and compare different algorithms on actual data and hardware to make an informed decision.

In general, when selecting a sorting algorithm for a large array, it's crucial to consider factors such as input data, hardware and software environment, and specific requirements and constraints. Quick-sort, mergesort, and heapsort are typically efficient and reliable sorting algorithms that can handle large datasets, while bubblesort may be more appropriate for smaller arrays or specialized applications.