Technical University of Moldova
Department of Software Engineering and Automatics

Analysis of sorting algorithms

FAF193: Miron Cristian Catalin
FAF.CC16.1 Calculability and Complexity
Professor: Mihai Gaidau
25th of February, 2021

Content

Laboratory objective

- To implement quicksort, mergesort, heap sort and bubble sort in a programming language;
- Empiric analysis of these algorithms.

Prerequisites

- Python 3.9;
- PyCharm IDE;
- Matplotlib 3.3.4 library;
- Browser and internet;

Base tasks

- Implement the algorithms in a programming language;
- Determine the properties of the input data which will be analysed;
- Choose a metric for comparing the algorithms;
- Perform the empiric analysis on the algorithms;
- Make a graph out of the output data;
- Elaborate the conclusion over the work done;

1.Introduction

In computer science, a sorting algorithm is an algorithm that puts elements of a list in a certain order. The most frequently used orders are numerical order and lexicographical order. Efficient sorting is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input data to be in sorted lists. Sorting is also often useful for canonicalizing data and for producing human-readable output. More formally, the output of any sorting algorithm must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation (a reordering, yet retaining all of the original elements) of the input.

For optimum efficiency, the input data in fast memory should be stored in a data structure which allows random access rather than one that allows only sequential access.

2. Implementation of the algorithms

All the algorithms will be implemented in python 3.9 as generators, to be used later in charts representing methods for visualization purposes.

For visualization, I used matplotlib 3.3.4.

2.1 Quick sort

QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in 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.

There can be many ways to do partition. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise we ignore the current element.

In worst case scenarios, its time complexity is O(n^2). But the average time complexity is O(nlogn).

Space complexity is O(logn):

O(1) as it has in-place partitioning;

O(logn) because of the recursive calls.

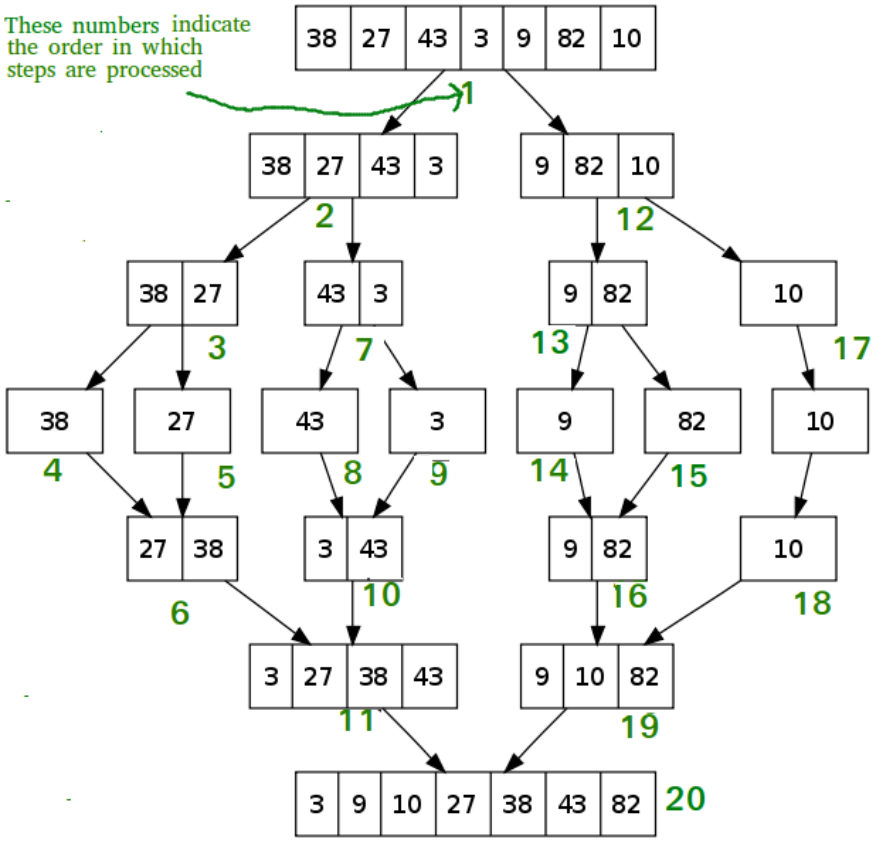
The following is my implementation:

```python
def quickSort(array, start, end):
    if start >= end:
        return

    pivot = array[end]
    pivotIndex = start

    for i in range(start, end):
        if array[i] < pivot:
            swap(array, i, pivotIndex)
            pivotIndex += 1
        yield array
    swap(array, end, pivotIndex)
    yield array

    yield from quickSort(array, start, pivotIndex - 1)
    yield from quickSort(array, pivotIndex + 1, end)
# end of quickSort function
```

2.2 Merge sort

Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted subarrays into one.

The following diagram from wikipedia shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Merge sort's best case and worst case scenario do not differ, because the number of operations depends only from the number of elements on the input.

Time complexity: O(nlogn)

Space complexity: O(n)

And my implementation:

```
def mergeSort(array, start, end):
    if end <= start:
        return

    mid = start + ((end - start + 1) // 2) - 1
    yield from mergeSort(array, start, mid)
    yield from mergeSort(array, mid + 1, end)
    yield from merge(array, start, mid, end)
    yield array
# end of mergeSort function
```

And the merge utility method:

```
def merge(array, start, mid, end):
    merged = []
    leftIdx = start
    rightIdx = mid + 1

    while leftIdx <= mid and rightIdx <= end:
        if array[leftIdx] < array[rightIdx]:
            merged.append(array[leftIdx])
            leftIdx += 1
        else:
            merged.append(array[rightIdx])
            rightIdx += 1

    while leftIdx <= mid:
        merged.append(array[leftIdx])
        leftIdx += 1

    while rightIdx <= end:
        merged.append(array[rightIdx])
        rightIdx += 1

    for i, sorted_val in enumerate(merged):
        array[start + i] = sorted_val
        yield array
# end of merge function
```

2.3 Heap sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of the tree.

3. Repeat step 2 while the size of the heap is greater than 1.

Time complexity of heapify is O(logn). Time complexity of heapSort() is O(n)

and the overall time complexity of heap sort is O(nlogn).

Space complexity: O(1) - in place heap rearrangement

And following is my implementation:

```python
def heapify(array, n, i):
    largest = i
    left = i * 2 + 1
    right = i * 2 + 2
    while left < n and array[left] > array[largest]:
        largest = left
    while right < n and array[right] > array[largest]:
        largest = right
    if largest != i:
        swap(array, i, largest)
        yield array
        yield from heapify(array, n, largest)
# end of heapify function


def heapSort(array):
    size = len(array)
    for i in range(size, -1, -1):
        yield from heapify(array, size, i)
    for i in range(size - 1, 0, -1):
        swap(array, 0, i)
        yield array
        yield from heapify(array, i, 0)
# end of heapSort function
```

2.4 Comb sort

Comb Sort is mainly an improvement over Bubble sort. Bubble sort always compares adjacent values. So all inversions are removed one by one. Comb sort improves on Bubble sort by using gaps of size more than 1. The gap starts with a large value and shrinks by a factor of 1.3 in every iteration until it reaches the value 1. Thus Comb sort removes more than one inversion counts with one swap and performs better than Bubble sort.

On average, it works better than Bubble sort but still, the worst time complexity is O(n^2)

Best time complexity: O(nlogn)

Space complexity: O(1)

```
def combSort(array):

    n = len(array)
    gap = n
    swapped = True

    # Keep running while gap is more than 1 and last
    # iteration caused a swap
    while gap != 1 or swapped == 1:

        # Find next gap
        gap = getNextGap(gap)

        # Initialize swapped as false so that we can
        # check if swap happened or not
        swapped = False

        # Compare all elements with current gap
        gap = int(gap)
        for i in range(0, n - gap):
            if array[i] > array[i + gap]:
                array[i], array[i + gap] = array[i + gap], array[i]
                swapped = True
            yield array
```

3. Analyzing the data

function/ complexity	Quicksort	Mergesort	Heapsort	Bubblesort	Combsort
Time(worst)	O(n^2)	O(nlogn)	O(nlogn)	O(n^2)	O(n^2)
Time(avg)	O(nlogn)	O(nlogn)	O(nlogn)	O(n^2)	O(nlogn)
Time(best)	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	O(n)
Space	O(logn)	O(n)	O(1)	O(1)	O(1)

function	in-place	stable
Bubble	yes	yes
Comb	yes	yes
Quick	yes	no
Merge	no(extra arrays)	yes
Heap	yes	no

4. Conclusion

Upon working on this laboratory work, I researched 4 different sorting algorithms and implemented them in code, also I added the possibility of viewing them working on bar graphs.

I analyzed them in-depth, and found out which algorithm fits the best for certain tasks: to sort a small input of data, quicksort and heapsort would fit the best, but for big amounts of data, mergesort will work better, because it is a stable algorithm.