Assignment 1: Sorting Algorithms

Subject: Algorithms and Complexity

Semester: Spring 2020

Student: Niko Tili

Instructor: Elton Ballhysa­­

Contents

[1 Introduction 2](#_Toc38996609)

[2 Implementation 2](#_Toc38996610)

[2.1 Mergesort 2](#_Toc38996611)

[2.1.1 Description 2](#_Toc38996612)

[2.1.2 Pseudocode 2](#_Toc38996613)

[2.2 Heapsort 3](#_Toc38996614)

[2.2.1 Description 3](#_Toc38996615)

[2.2.2 Pseudocode 3](#_Toc38996616)

[2.3 Quicksort 4](#_Toc38996617)

[2.3.1 Description 4](#_Toc38996618)

[2.3.2 Pseudocode 4](#_Toc38996619)

[2.4 Ins-Mergesort 5](#_Toc38996620)

[2.4.1 Description 5](#_Toc38996621)

[2.4.2 Pseudocode 5](#_Toc38996622)

[3 Performance Comparison 6](#_Toc38996623)

[4 Analysis of Ins-Mergesort 6](#_Toc38996624)

[4.1 Proof 6](#_Toc38996625)

[4.1.1 Mergesort 6](#_Toc38996626)

[4.1.2 Insertionsort 7](#_Toc38996627)

[4.1.3 Ins-Mergesort 7](#_Toc38996628)

[4.2 Finding M 7](#_Toc38996629)

[4.3 Who the hell knows what else 7](#_Toc38996630)

# Introduction

In this assignment, I have implemented four sorting algorithms: Mergesort, Heapsort, Quicksort, Ins-Mergesort. Implemented in Java

# Implementation

## Mergesort

### Description

This is the out-of-place implementation of Mergesort. At first the array, is split in two halves recursively until the base case of one element long array. For every recursive call, the two halves are merged into one sorted array.

### Pseudocode

1. mergeSort(A):
2. n = |A|
3. **if** n > 1:
4. left = A[0..n/2]
5. right = A[n/2 + 1..n]
6. **return** merge(mergeSort(left), mergeSort(right))
7. end **if**
9. **return** A


13. merge(A, B):
14. m = |A|
15. n = |B|
16. C = array[m + n]
17. k = 0
18. i = 0
19. j = 0
20. **while** i < m **and** j < n:
21. **if** A[i] <= B[j]:
22. C[k] = A[i]
23. i += 1
24. end **if**
25. **else**:
26. C[k] = B[j]
27. j += 1
28. end **else**
29. k += 1
30. end **while**
32. **if** i = m:
33. **while** j < n:
34. C[k] = B[j]
35. k += 1
36. j += 1
37. end **while**
38. end **if**
39. **else**
40. **while** i < m:
41. C[k] = A[i]
42. k += 1
43. i += 1
44. end **while**
45. end **else**
47. **return** C

## Heapsort

### Description

Heapsort is implemented as an in-place algorithm. At first, a maximum heap is built by dividing logically the array into two sub-arrays, one being the max-heap and the other the remaining elements of the array. The elements of the array are inserted into the heap one by one in a way that the max-heap is still valid. After this procedure, the maximum of the heap is divided logically into two sub-arrays one being the max-heap and the other the sorted array. The current maximum of the heap is placed recursively to the next available sorted array index until the array is fully sorted.

### Pseudocode

1. heapSort(A):
2. buildMaxHeap(A)
3. doSort(A)
5. buildMaxHeap(A):
6. n = |A|
7. **for** heapSize = 1 to n - 1:
8. currPos = heapSize
9. **while** currPos > 0:
10. parentIndex = (currPos - 1) / 2
11. **if** A[currPos] < A[parentIndex]:
12. **break** **while**
13. end **if**
15. swap A[currPos] with A[parentIndex]
16. currPos = parentIndex
17. end **while**
18. end **for**

21. doSort(H):
22. n = |H|
23. **for** i = 0 to n - 2:
24. swap H[0] with H[n - 1 - i]
25. siftDown(H, n - 1 - i, 0)
26. end **for**
27. **if** H[0] > H[1]:
28. swap H[0] with H[1]
29. end **if**
31. siftDown(H, n, pos):
32. **if** pos > (n - 2) / 2:
33. **return**
34. end **if**
36. leftChildPos = pos \* 2 + 1
37. rightChildPos = leftChildPos + 1
39. siftPos = pos
40. **if** rightChildPos < n:
41. **if** H[rightChildPos] > h[siftPos]:
42. siftPos = rightChildPos
43. end **if**
44. end **if**
46. **if** H[leftChildPos] > H[siftPos]:
47. siftPos = leftChildPos
48. end **if**
50. **if** siftPos = pos
51. **return**
52. end **if**
54. swap H[pos] with H[siftPos]

## Quicksort

### Description

Quicksort also is implemented as an in-place algorithm. The first step of this algorithm is to do the partition of the array by selecting a random pivot value from the array, moving the elements less than the pivot to the left side and other elements to the right side of it. This step is repeated recursively on the left and the right part of the array until the base case (start position is greater than or equal to end position). The pivot selection is random in order to avoid worst-case scenario, the case when the pivot is always the max or min value of the array.

### Pseudocode

1. quickSort(A):
2. n = |A|
3. quickSort(A, 0, n - 1)

6. quickSort(A, start, end):
7. **if** end >= start:
8. **return**
9. end **if**
11. r = random **from** [start, end]
12. pivotPos = partition(arr, start, end, r)
14. quickSort(A, start, pivotPos - 1)
15. quickSort(A, pivotPos + 1, end)

18. partition(A, start, end, pivotPos):
19. pivotVal = A[pivotPos]
20. swap A[pivotPos] with A[end]
21. leftPos = start
23. **for** i = start to end - 1:
24. **if** A[i] < pivotVal:
25. swap A[leftPos] with A[i]
26. leftPos += 1
27. end **if**
28. end **for**
29. swap A[leftPos] with A[end]
31. **return** leftPos

## Ins-Mergesort

### Description

Ins-Mergesort behaves the same as Mergesort. The only difference is at the base case. Here, arrays of length less than or equal to six, are sorted using Insertionsort.

### Pseudocode

1. insMergeSort(A):
2. **return** doSort(A)
4. doSort(A):
5. n = |A|
6. **if** n > 6:
7. left = A[0..n/2]
8. right = A[n/2..n]
9. **return** merge(doSort(left), doSort(right))
10. end **if**
11. **else**
12. insertionSort(A)
13. end **else**
15. **return** A


19. merge(A, B):
20. m = |A|
21. n = |B|
22. C = array[m + n]
23. k = 0
24. i = 0
25. j = 0
26. **while** i < m **and** j < n:
27. **if** A[i] <= B[j]:
28. C[k] = A[i]
29. i += 1
30. end **if**
31. **else**:
32. C[k] = B[j]
33. j += 1
34. end **else**
35. k += 1
36. end **while**
38. **if** i = m:
39. **while** j < n:
40. C[k] = B[j]
41. k += 1
42. j += 1
43. end **while**
44. end **if**
45. **else**
46. **while** i < m:
47. C[k] = A[i]
48. k += 1
49. i += 1
50. end **while**
51. end **else**
53. **return** C

56. insertionSort(A):
57. n = |A|
59. **for** i = 0 to n - 2:
60. key = A[i + 1]
61. j = i
63. **while** j >= 0 **and** A[j] > key:
64. A[j + 1] = A[j]
65. j -= 1
66. end **while**
68. A[j + 1] = key
69. end **for**

# Performance Comparison

All four algorithms have average case time-complexity. The difference is at the multiplicative constants.

# Analysis of Ins-Mergesort

## Proof

### Mergesort

For an array containing elements, the output of the function is the sorted array.

1. The base case is for . An array of size 1 is already sorted because it contains zero inversions.
2. Suppose that for the inputted array of size, the function splits it into two arrays: and, and the output is:, the sorted array.
3. Since the function splits the inputted array into two halves and calls itself into both; this process is correct based on the supposition.
4. To complete the proof, it is sufficient to show that, which merges two sorted arrays into one sorted array, works correctly.

* **Initialization:** It creates where and. So we have: and which is empty.
* **Maintenance:** After each iteration.
* **Termination:** The loop terminates if or is fully iterated. contains and in a sorted order.

The remaining values from or are placed to. The result is containing values and in a sorted order.

### Insertionsort

1. insertionSort(A):
2. n = |A|
4. **for** i = 0 to n - 2:
5. key = A[i + 1]
6. j = i
8. **while** j >= 0 **and** A[j] > key:
9. A[j + 1] = A[j]
10. j -= 1
11. end **while**
13. A[j + 1] = key
14. end **for**

* **Initialization**: At the beginning of the for-loop, sub-array where is already sorted.
* **Maintenance**: For every iteration, while, is shifted one position to the right until. When the while loop terminates is placed right after. After each iteration is in sorted order.
* **Termination:** At the last iteration, is going to be inserted and since the sorted order is maintained, after the last iteration, the array is fully sorted.

### Ins-Mergesort

Ins-Mergesort is a combination of Mergesort and Insertionsort. For arrays of length, Insertionsort algorithm is used. Arrays of length, are processed via Mergesort. The difference is at the base case of it, when Insertionsort is called. Sorted arrays by Insertionsort are processed by Mergesort resulting in a sorted array.

## Finding M

For an array of length, Insertionsort on average does comparisons and on the worst case. While Mergesort does comparisons on average and worst case. For, Insertionsort in average case does less comparisons than Mergesort. Testing Ins-Mergesort with, the best results are for.

## Who the hell knows what else