

CSE 301 Algorithms Assignment Report: Heap Sort for k sorted arrays

We are given an unsorted array $A[1\dots n]$. Now, imagine its sorted version. The unsorted array has the property that each element has a distance of at most k positions, where $0 < k \leq n$, from its index in the sorted version. For example, when k is 2, an element at index 5 in the sorted array, can be at one of the indices $\{3, 4, 5, 6, 7\}$ in the unsorted array. The unsorted array can be sorted efficiently by utilizing a MinHeap data structure. The outline of the algorithm is given below:

- Create a Min Heap of size $k+1$ with first $k+1$ elements,
- One by one remove min element from the heap, put it in the result array, and add a new element to the heap from remaining elements.

a. Write down the complete algorithm in pseudocode convention to sort the array A.

- Create a min heap
- Add $k+1$ number of values of the array to the min heap you created
- One by one, remove min element from heap and add them to array, and add remaining element to the heap
- Pop all remaining elements from the min heap and assign them to the next available array index

b. Provide a tight asymptotic upper bound time complexity for this algorithm. Show your work.

- Defining variables
 - n = total items in the array
 - k = the distance that item can be from its final position

Every element is going to touch the min heap we are using. we are going to put every item in the array and they will run at the heap. How much work we do for an item?

We will be doing logarithmic amount of work with respect to the **maximum** elements we are keeping in the heap. So the maximum work we will do on insertion into our heap is going to be logarithmic with respect to k . Because our heap is going to hold $k + 1$ elements. So the insertion takes $\log k$ because we are going to do traverse of the height which is $\log k$ at worst case.

Since we are doing this for each element the time complexity will be $O(n * \log(k))$.

Another solution would be :

We have $arr[0 \dots n]$

Lets sort $\text{arr}[0..2k]$, we know that $\text{arr}[0..k]$ is sorted but $\text{arr}[k..2k]$ may be misplaced by k position

Then lets sort $[k..3k]$, we now sorted $\text{arr}[k..2k]$ but $\text{arr}[2k..3k]$ may be misplaced by k position

It goes like this

Until we sort $\text{arr}[ak..n]$

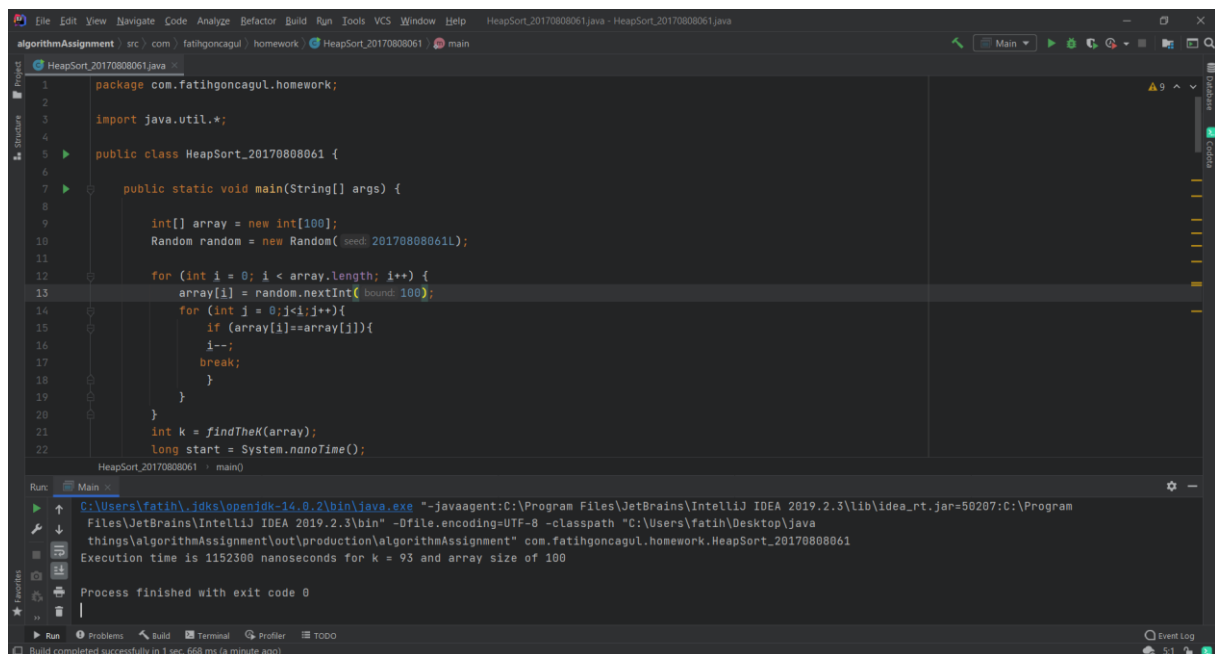
In each step we sort at most $2k$ elements the amount of work done is $O(k \cdot \log k)$ we put at least k elements in their final position. (inserting an element in heap takes $O(\log k)$)

Since we will have $O(n/k)$ steps

The complexity will be $O(n \cdot \log(k))$

c. Implement your solution in any language you prefer. Generate the array sizes of 100,1000,10000,100000 with using your student id as a seed. Run your solution and note the running times. Show your results.

- The code including the algorithm to find “ k ” is attached to the assignment as a .java document.
- *Here are all the results I have with correct k values for each array (has unique elements) with taking my student number as a seed.*
- Note: My algorithm to find k takes a lot of time when the array size is 100 000.
 - For the array size of 100.



The screenshot shows an IDE with a Java file named `HeapSort_20170808061.java`. The code implements a heap sort algorithm. It generates an array of 100 random integers using a seed of 20170808061L. It then finds a value k (93) and sorts the array in chunks of size $2k$. The execution time is 1152300 nanoseconds.

```
package com.fatihgocagul.homework;

import java.util.*;

public class HeapSort_20170808061 {

    public static void main(String[] args) {

        int[] array = new int[100];
        Random random = new Random(20170808061L);

        for (int i = 0; i < array.length; i++) {
            array[i] = random.nextInt(100);
        }

        for (int i = 0; i < array.length; i++) {
            for (int j = 0; j < i; j++) {
                if (array[i] < array[j]) {
                    int temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
            }
        }

        int k = findTheK(array);
        long start = System.nanoTime();

        // ... (rest of the code) ...

    }
}
```

Run: `C:\Users\fatih\jdk\openjdk-14.0.2\bin\java.exe -javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\lib\idea_rt.jar=50207:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\bin -Dfile.encoding=UTF-8 -classpath "C:\Users\fatih\Desktop\java things\algorithmAssignment\out\production\algorithmAssignment" com.fatihgocagul.homework.HeapSort_20170808061`

Execution time is 1152300 nanoseconds for $k = 93$ and array size of 100

Process finished with exit code 0

As we can see for an array size of 100 and $k = 93$. The running time of the algorithm is 1 152 300 nanoseconds.

- For the array size of 1 000.

```
1 package com.fatihguncagul.homework;
2
3 import java.util.*;
4
5 public class HeapSort_20170808061 {
6
7     public static void main(String[] args) {
8
9         int[] array = new int[1000];
10         Random random = new Random(seed: 20170808061L);
11
12         for (int i = 0; i < array.length; i++) {
13             array[i] = random.nextInt(bound: 1000);
14             for (int j = 0; j < i; j++) {
15                 if (array[i] < array[j]) {
16                     i--;
17                     break;
18                 }
19             }
20         }
21         int k = findTheK(array);
22         long start = System.nanoTime();
```

Run: Main

C:\Users\fatih\jdk\openjdk-14.0.2\bin\java.exe -javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\lib\idea_rt.jar=50235:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\bin -Dfile.encoding=UTF-8 -classpath "C:\Users\fatih\Desktop\java things\algorithmAssignment\out\production\algorithmAssignment" com.fatihguncagul.homework.HeapSort_20170808061

Execution time is 4167500 nanoseconds for k = 969 and array size of 1000

Process finished with exit code 0

The running time of the algorithm is 4 167 500 nanoseconds for k = 969 and array size of 1 000.

- For the array size of 10 000.

```
1 package com.fatihguncagul.homework;
2
3 import java.util.*;
4
5 public class HeapSort_20170808061 {
6
7     public static void main(String[] args) {
8
9         int[] array = new int[10000];
10         Random random = new Random(seed: 20170808061L);
11
12         for (int i = 0; i < array.length; i++) {
13             array[i] = random.nextInt(bound: 10000);
14             for (int j = 0; j < i; j++) {
15                 if (array[i] < array[j]) {
16                     i--;
17                     break;
18                 }
19             }
20         }
21         int k = findTheK(array);
22         long start = System.nanoTime();
```

Run: Main

C:\Users\fatih\jdk\openjdk-14.0.2\bin\java.exe -javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\lib\idea_rt.jar=50269:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\bin -Dfile.encoding=UTF-8 -classpath "C:\Users\fatih\Desktop\java things\algorithmAssignment\out\production\algorithmAssignment" com.fatihguncagul.homework.HeapSort_20170808061

Execution time is 7271300 nanoseconds for k = 9897 and array size of 10000

Process finished with exit code 0

It appears that the running time of the algorithm is 7 271 300 for k = 9897 and array size of 10 000.

- For the array size of 100 000.

```

public static void main(String[] args) {
    int[] array = new int[100000];
    Random random = new Random( seed: 20170808061);

    for (int i = 0; i < array.length; i++) {
        array[i] = random.nextInt( bound: 100000);
        for (int j = 0; j < i; j++){
            if (array[i] < array[j]){
                i--;
                break;
            }
        }
    }

    int k = findTheK(array);
    long start = System.nanoTime();
    kHeapSort(array, array.length - 1, k);
    //quickSort(array, 0, array.length-1);
    long duration = System.nanoTime() - start;
    System.out.println("Execution time is " + duration + " nanoseconds for k = " + k + " and array size of " + array.length);
}

```

Run: Main

```

C:\Users\fatih\jdk\openjdk-14.0.2\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2019.2.3\lib\idea_rt.jar=50282:C:\Program
Files\JetBrains\IntelliJ IDEA 2019.2.3\bin" -Dfile.encoding=UTF-8 -classpath "C:\Users\fatih\Desktop\java
things\algorithmAssignment\out\production\algorithmAssignment" com.fatihgocagul.homework.HeapSort_20170808061
Execution time is 32053100 nanoseconds for k = 99815 and array size of 100000
Process finished with exit code 0

```

We can see that the running time of this algorithm is 32 053 100 for k = 99660 and array size of 100 000.

- Results with correct k values :

Array size	k	Running time (nanoseconds)
100	93	1 152 300
1 000	969	4 167 500
10 000	9 897	6 477 800
100 000	99 815	32 053 100

As a result, if we investigate the increase in runtime, linear in array size, with a constant proportional to the $\log(k)$.

- Comparisonn of Quick Sort and Heap Sort.*

First of all, I investigate the Quick Sort algorithm asymptotically:

The worst case of quick sort happens (pivot is chosen as a last element) when we have an array which is already sorted in increasing or decreasing order. It uses divide and conquer strategy. Here we have the recurrence relation and solution :

- $T(0) = T(1) = 0$ this is the base case
- $T(n) = T(n-1) + n$

$$T(n) = T(n-1) + n$$

$$T(n-1) = T(n-2) + (n-1)$$

$$T(n-2) = T(n-3) + (n-2)$$

It goes like this

$$T(3) = T(2) + 3$$

$$T(2) = T(1) + 2$$

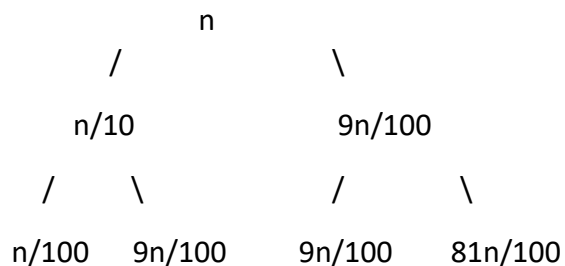
$$T(1) = 0$$

So we have $T(n) = n + (n-1) + (n-2) + \dots + 3 + 2$ which is approximately $n^2/2$

So the worst case is $O(n^2)$.

For the **average case of quick sort** let's say the pivot always ends up at least 10 percent from either edge.

The tree:



Work at each level is $O(n)$

Runtime will be $O(n \cdot \text{HEIGHT})$

Height is approximately $\log_{10/9} n$

Overall complexity will be $O(n \cdot \log n)$ for the average case

As I mentioned earlier the time complexity of the heap sort algorithm for k sorted arrays was $O(n \cdot \log(k))$.

If we compare, it is clear that quick sort's time complexity is more than heap sort's (for k sorted arrays) time complexity.

- Now, let's see compare the runtimes of quicksort and heap sort algorithms.

Array size	Heap sort (runtime ns)	Quick Sort (runtime ns)
100	1 152 300	278 100
1 000	4 167 500	495 500
10 000	6 477 800	2 545 800
100 000	32 053 100	11 430 700

However, if I compare the runtime results, quicksort is 2-3 times faster. One reason why heap sort appears to be slower than quick sort in practice is that, even if we have nearly sorted array we are going to swap all of the elements to order. Quick sort does not do too much unnecessary swaps.

But the main reason is that quick sort has better locality of reference. Locality means programs tends to use data and instruction with addresses near or equal to those they have used recently.

In quick sort we have good spatial locality in partitioning which means items with nearby addresses tend to be referenced close together in time. Caches take advantage of this by bringing blocks of data from memory rather than single byte at a time. When we need address "A" we don't only get address A we get A+1, A+2, A+3... up to a certain point. This is what quick sort does in partitioning part of the algorithm. On the other hand we have a good temporal locality in recursive subdivision which means recently referenced items are likely to be referenced again in the near future. Caches take advantage of this with storing recently accesses data which in our case it is arrays. In heap sort accesses to the data is widely scattered.

Fatih Goncagül