a)

The asymptomatic upperbound for heapsort is nlogn because:

The heapsort pseudocode is something like this:

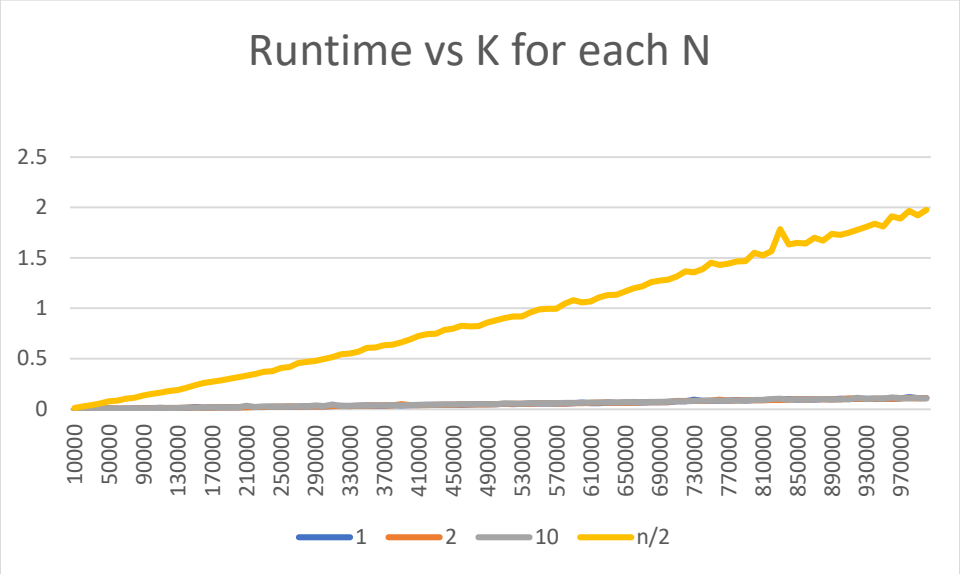
```
Heapsort (arr):
    heap_build(arr)
    sorted_arr=[]
    from i <- 0 to i = arr.length:
        sorted_arr[i]=arr.pop()
    arr=sorted_arr
```

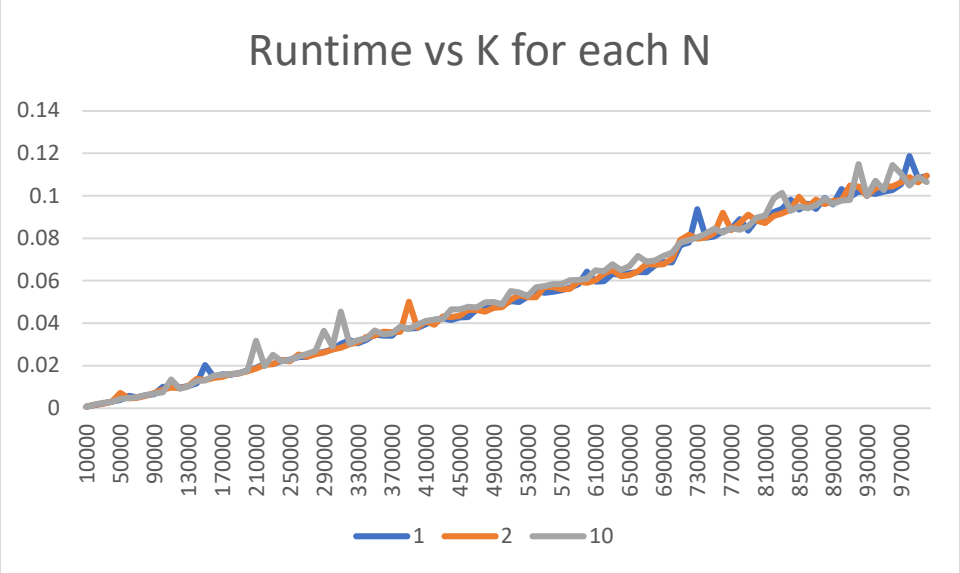
the heapbuild operation does “heapify” operation N/2 times and every time we call arr.pop() it also calls heapify once. Heapify operation starts from a node of a tree and sifts down until heap property is satisfied and because the depth of a binary tree is log(n) this operation takes logn time at most but because we already have the lower part of the tree mostly sifted down when we do the lower nodes in the tree it doesn’t go as deep as the upper nodes so the heapify operation actually takes closer to constant time in this case. From this we can say that the heap_build operation takes O(n) time. For the loop we are repeating the arr.pop() n times and it calls heapify once and in this case heapify actually does take log(n) time so we can also assume that the loop in the function takes O(nlog(n)). Because nlog(n) dominates n we can say that complexity of heapsort is nlog(n).

b)

N \ K	1	2	10	n/2
10	2.50E-05	4.00E-06	1.40E-05	7.00E-06
100	1.60E-05	1.20E-05	2.80E-05	9.90E-05
1000	7.00E-05	8.40E-05	8.50E-05	0.00108
1000000	0.094094	0.097746	0.101518	1.9604

c)





For this problem we can actually say that the complexity is O(N+K*log(N)). This is because loop part of the heapsort function will repeat K times for us and it will pull the value from a heap of depth n. Here from this graph we see that the increase in the value of k affects the runtime more because building the heap takes O(N) time but O(K(log(N))) dominates it as the value of K increases. we can see that from the graph as lines with constant K values of 1 2 and 10 increase linearly but the k=n/2 line diverges much faster. Therefore we can say that the complexity is T(N + Klog(N)) and the Klogn dominates N when the value of k increases.

```
Heapsort (arr):
        heap_build(arr) // O(N)
        sorted_arr=[]
        from i <- 0 to i = arr.length: // loops for K times
                sorted_arr[i]=arr.pop() //takes log(N) time
        arr=sorted_arr
```