Uğur AKYEL
2019080D8020

A)

```
Sort_kfactor(A, k, n)
1....  resulH[n]
2....  H[k+1]
3....  J, i = 0
4....  for a ← 1 to (k+1)
5....      H[a] = A[a]
6....  Build_Min_Heap(H, k+1)
7....  J = k+1
8....  while (k > 0
9....      result[i++] ← heap_extract_min(H, k+1)
10....     if J < n
11....         heap_insert(H, A[J++], k)
12....  return result
```

k: k index for our problem
n: A's length
A: unsorted array for k factor

```
Build_Min_Heap(H, n) ---> O(n)
1....  for i ← n/2 downto 1
2....      min_heapify(H, i, n)
```

```
heap_extract_min(A, n) ---> O(lgn)
1....  if n < 1
2....      return ∞
3....  min ← A[1]
4....  A[1] ← A[n-1]
5....  n ← n-1
6....  min_heapify(A, 1, n)
7....  return min
```

```
Heap_insert(A, key, n) ---> O(lgn)
    i = n
    while i > 1 and A[⌊i/2⌋] > key
        A[i] ← A[⌊i/2⌋]
        i ← ⌊i/2⌋
    A[i] ← key
```

I implemented code via the C/C++ and I used low level array based algorithm for the solution. In my code, some k and J values I changed for array consistency at the boundary condition. And I implemented some utilities fuction for debugging and testing for example; "print_array()", "write_extract_min()", "gettime_ksort()", "rand_arr_stdid" etc.

Uğur ARYEC
2019080020

B) for our algorithm line by line;

We know
heap-extract-min → $\Theta(\lg n)$
build-min-heap → $\Theta(n)$
heap-insert → $\Theta(\lg n)$

1.. c
2.. c
3.. c
4.. $c(k+2)$
5.. $c(k+1)$
6.. $c\,\Theta(k+1)$
7.. c
8.. $c(n+1)$
9.. $c\sum_{i=1}^{n-(k+1)} \lg(k+1) + \sum_{i=1}^{k+1} \lg(i)$

first part:
This line says that each A's element must arrive our Min-heap, and then extracting to our Result array

second part: so we have removing (k+1) element of min-heap, last operation

10.
11.. $c\sum_{k+2}^{n} \lg k$ ← This line walk until A's remaining elements goes to min-heap one by one
12.. c

$$\Rightarrow T(n) = 10c + 3kc + cn + c\sum_{i=1}^{n-k-1} \lg(k+1) + \sum_{i=1}^{k+1} \lg(i) + c\sum_{k+2}^{n} \lg k$$

$$= c\left(10 + 3k + n + (n-k-1)\lg(k+1) + \lg((k+1)!) + (n-k-2)\lg k\right)$$

$$= c\left(k + n + (n-k)\lg k + k\lg k + (n-k)\lg k\right)$$

$$= O\left(n + 2(n-k)\lg k + k\lg k\right)$$

$$= O\left(n + (2n-2k+k)\lg k\right) = O(n + (2n-k)\lg k)$$

$$= O(n + n\lg k)$$

$$\begin{cases} O(n\lg n), & \text{if } k = n-1 \\ O(n), & \text{if } k = 1 \end{cases}$$

We know Stirling Approximation
$\lg(n!) \simeq n\lg n$

Our worst case scenario asymptotic tight-bound is $O(n\lg n)$

Comparison: We know Randomized QuickSort is $O(n\lg n)$, so this two algorithms is similar. But if we could use sufficiently large enough "n" and large "k" we could beat Min-heap. We will show in experimentally Because good implemented QuickSort beats heapsort and MergeSort.

Now we also know ***normal QuickSort worst case complexity is O(n^2)*** if every partition is unlucky placed. But ***our algorithm says that in the worst case (k=n-1) we have O(nlgn) actually in the worst case our algortihmbeats quicksort in theory. But Well-builded Quicksort has a randomized and has average case time complexity is O(nlgn).*** So we could say that we had approximately similar growth rate.

C) Firstly, we want to find **appropriate k** for our algorithm experiment and I implemented for the optimized k function for k value. *This method firstly create a sorted array and compare for each array and sorted_array's index then if we find this two array value is equal we can say this index difference give a k value for us.* Also we search for the max k value for our array. You could see pseudocode in the below:

```
Find_Optimized_K(A, n)
    k = 0;
    B = sort(A, n)      .....this sort method returns a new sorted array.
    for i←1 to n
        for j←1 to n
            if A[i] == B[j] and k < abs(j-i+1)
                k = abs(j-i+1)
    if k!= 0
        return k
    else
        error("this array is sorted….")
```

For the time calculation we use timespec and clock_gettime method in C++, if you want to code compile and to show output in the terminal please use following command;

```
[kozan@kozan-pc    ]$ g++ -o a.exe heapsort_code_20190808020.cpp
[kozan@kozan-pc    ]$ ./a.exe
```

and then we have this outputs in the below;

```
--------------------------------
Min-Heap: A 100 elements array and k: 95 time calculation is: 3.9733e-05 ms
Quicksort: A 100 elements array  time calculation is: 2.5778e-05 ms
--------------------------------
--------------------------------
Min-Heap: A 1000 elements array and k: 983 time calculation is: 0.000602535 ms
Quicksort: A 1000 elements array  time calculation is: 0.000306381 ms
--------------------------------
--------------------------------
Min-Heap: A 10000 elements array and k: 9887 time calculation is: 0.00310939 ms
Quicksort: A 10000 elements array  time calculation is: 0.00133958 ms
--------------------------------
--------------------------------
Min-Heap: A 100000 elements array and k: 99666 time calculation is: 0.0411772 ms
Quicksort: A 100000 elements array  time calculation is: 0.0154427 ms
--------------------------------
```

We could see easily at the first three array, *quicksort beats our algortihms*, and secondly *we could see if we growh the array size multiply 10 times for each input we obtain a growth rate approximately O(nlgn) -⟶*10lg10 times grow our time complexity.*

Why Quicksort beat our algorithm because we have k approximately (n-1) so quickSort easily beat us. The student id randomized array's k value has a very high value above the output image we can see. So we could say our min-heap approach nearly in the worst case but quicksort works in the average case because of randomized array.

And also I added 2 important function screemshot for my report;

```
60 int * sort_kfactor(int A[], int k, int length){
61   int *result = new int[length];
62   int *H = new int[k+1];
63   int j, i = 0;
64
65   for(int a = 0; a < k+1; a++){
66     H[a] = A[a];
67   }
68   k = k+1;
69   build_min_heap(H, k);
70   j = k;
71   while(k > 0){
72     result[i++] = heap_extract_min(H, k);
73     k = k-1;
74     if(j < length){
75       heap_insert(H, A[j++], k);
76       k = k+1;
77     }
78   }
79   return result;
80 }
```

```
82 void gettime_ksort(int *arr, int k, int n){
83   struct timespec start, finish;
84   double elapsed;
85   clock_gettime(CLOCK_MONOTONIC, &start);
86
87   int * p100 = sort_kfactor(arr, k, n);
88
89   clock_gettime(CLOCK_MONOTONIC, &finish);
90
91   elapsed = (finish.tv_sec - start.tv_sec);
92   elapsed += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
93   cout << "A " << n << " elements array and k: " << k << " time calculation i
   s: "  << elapsed << " ms"  << endl;
94 }
```