[CENG 315 All Sections] Algorithms

Description Submission view

THE1

Available from: Friday, October 28, 2022, 11:59 AM
Due date: Saturday, October 29, 2022, 11:59 PM
Requested files: the1.cpp, test.cpp, the1_solution.cpp (Download)
Maximum number of files: 2
Type of work: Individual work

Specifications:

- There is **1 task** to be solved in **36 hours** in this take home exam.
- You will implement your solutions in **the1.cpp** file.
- You are free to add other functions to **the1.cpp**
- Do **not** change the first line of **the1.cpp**, which is **#include "the1.h"**
- Do **not** change the arguments and return value of the functions **kWayMergeSortWithHeap()** in the file **the1.cpp**
- Do **not** include any other library or write include anywhere in your **the1.cpp** file (not even in comments).

- You are given a test.cpp file to **test** your work on **Odtuclass** or your **locale**. You can and you are encouraged to modify this file to add different test cases.
- If you want to **test** your work and see your outputs you can **compile** your work on your locale as:

```
>g++ test.cpp the1.cpp -Wall -std=c++11 -o test
> ./test
```

- You can test your **the1.cpp** on virtual lab environment. If you click **run**, your function will be compiled and executed with test.cpp. If you click **evaluate**, you will get a feedback for your current work and your work will be **temporarly** graded for **limited** number of inputs.
- The grade you see in lab is **not** your final grade, your code will be reevaluated with **different** inputs after the exam.

The system has the following limits:

- a maximum execution time of 1 minute (your functions should return in less than 1 seconds for the largest inputs)
- a 256 MB maximum memory limit
- a stack size of 64 MB for function calls (ie. recursive solutions)

- Each task has a complexity constraint explained in respective sections.
- Solutions with longer running times will not be graded.

- If you are sure that your solution works in the expected complexity constrains but your evaluation fails due to limits in the lab environment, the constant factors may be the problem.

- If you solution is correct, the time and memory limits may be adjusted to accept your solution after the lab. Please send an email if that is the case for you.

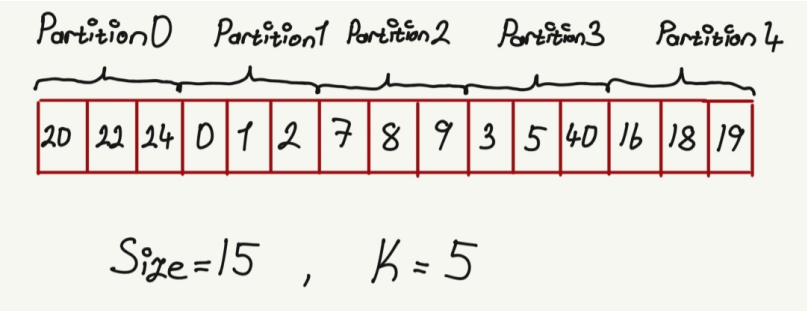
```
int kWayMergeSortWithHeap(int* arr, int K, int size, long& comparison,     long& swap);
```

In this exam, you are asked to complete the function definitions to sort the given array **arr** with **ascending** order.

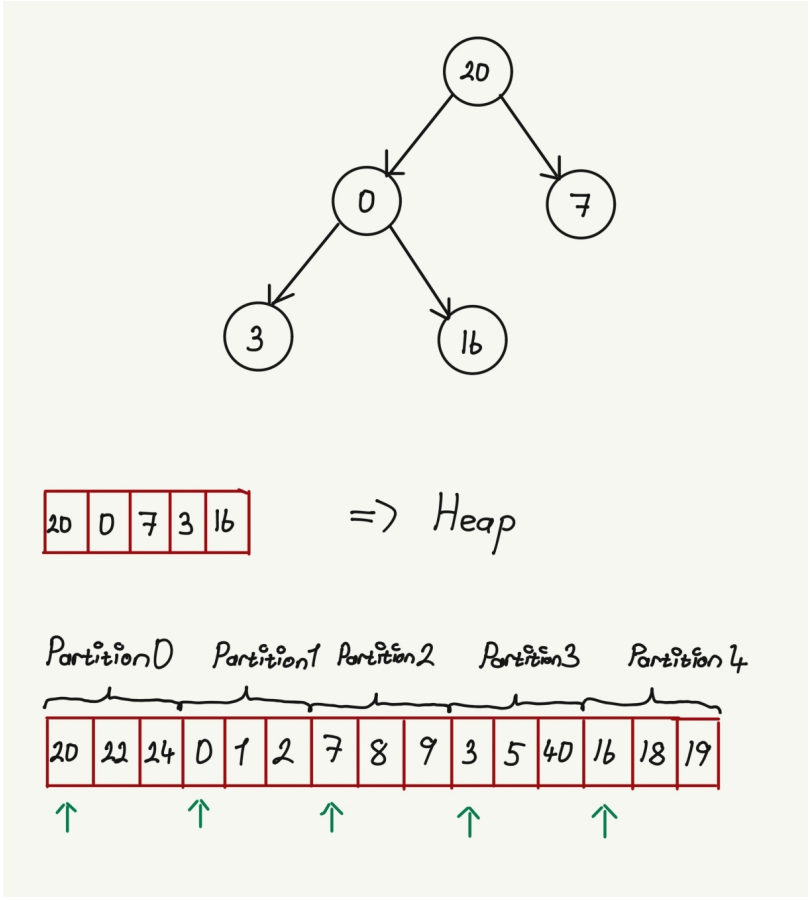
- **kWayMergeSortWithHeap()** should count the number of **comparison** and **swap** executed during sorting process (Comparisons are only between the values to be sorted during insertion sort and heapify process) and returns the total number of calls of **kWayMergeSortWithHeap()**.

K Way Merge Sort With Heap algortihm (**kWayMergeSortWithHeap()**) is as follows:

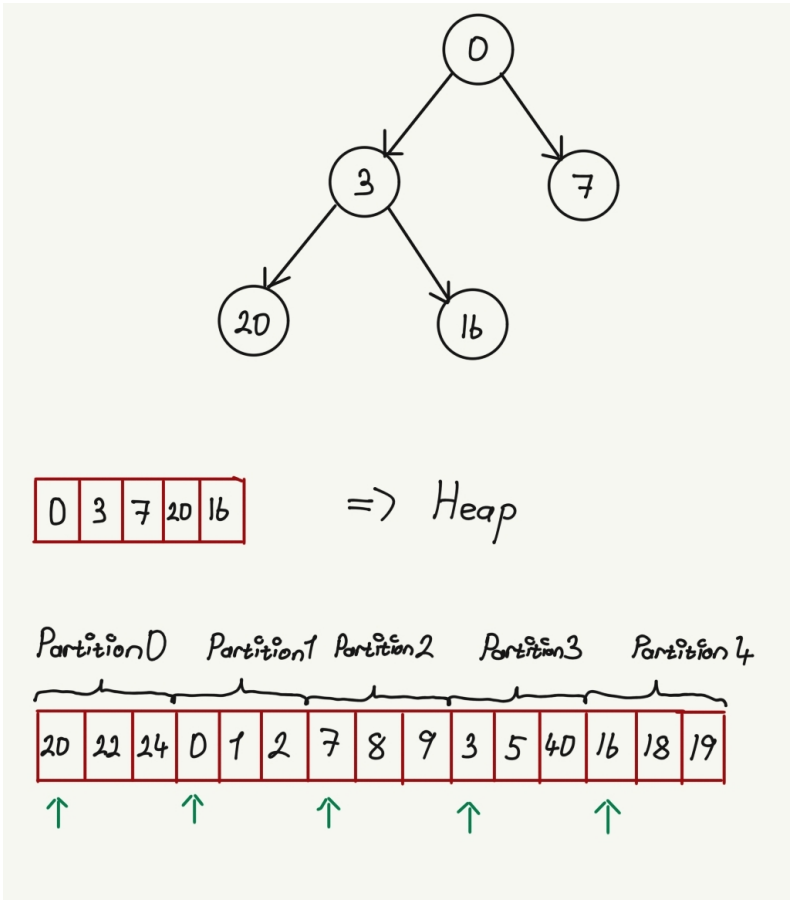
- If the size of the array is less than K, then sort the array by using insertion sort.(You can use the insertion sort algorithm given to you in THE0.)
- Otherwise, split the array into K sub-arrays and do K recursive calls to sort the partitions.
  - Then, merge K sorted arrays.
  - When merging K sorted-arrays, you should use a Binary Min Heap to select the minimum element between the minimum elements of K partition arrays.
  - When creating the array of the heap,
    - Firstly, generate a linear array whose elements are the minimum elements of the K partition arrays. At the beginning, the position of the each element is determined by the belonging partition. For example, the element coming from partition 0 is placed to heap_array[0] and the element coming from partition 1 is placed to heap_array[1] etc.
    - Then, heapify the initial array.
  - After finding the minimum element, you should insert a new element from the related partition to the Min Heap.
    - Read the minimum element in the heap and record it.
    - Then, replace the minimum element with a new element from the partition that has the last minimum element.(New element insertion is not a swap operation. Swap has to be counted only inside the heap or insertion sort.)
    - Then, heapify the current array.
- In case of equality during heapify and insertion sort, do not swap the elements.
- Count the comparison and swap between any 2 elements of the array H in both insertion sort and heapify, such as H[i]>H[j]
- Return the total number of **kWayMergeSortWithHeap()** calls.
- Let's have an example case:
  - Let's say in some point the array is as follows:



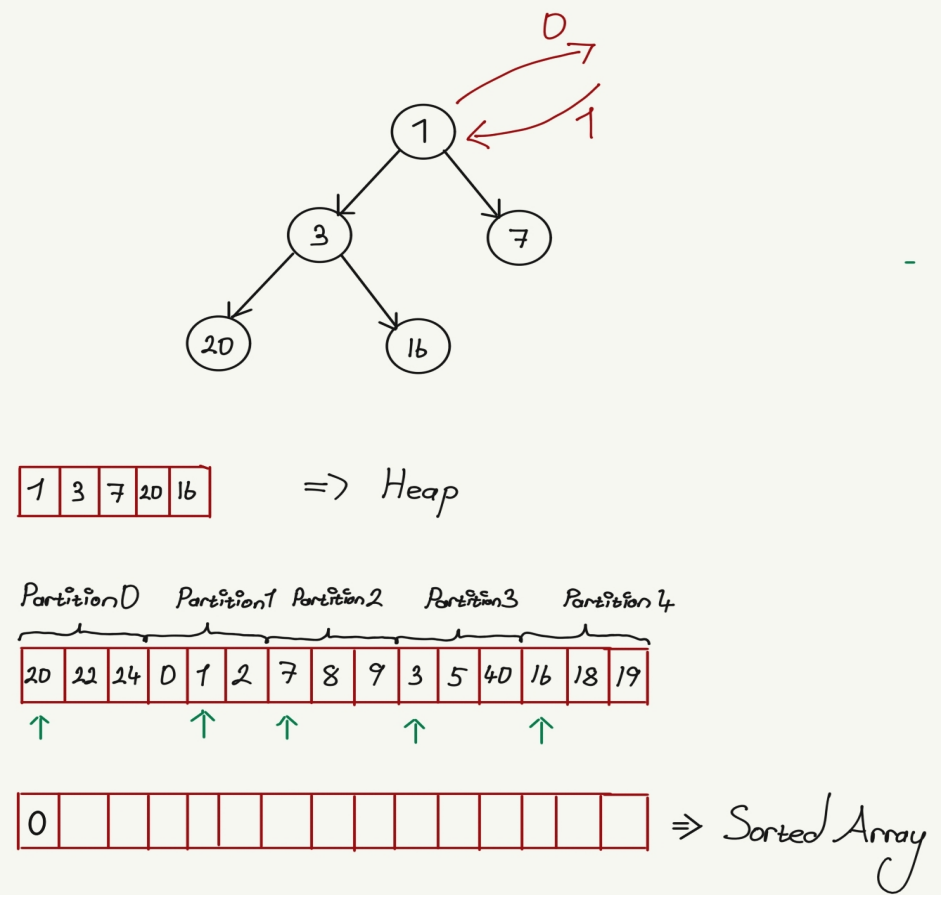
    - Create a heap array and place the first elements of the partitions



- Heapify the array(6 comparisons and 2 swaps are required.)



- Record the minimum and insert a new element(It is not counted as swap.)



- Then, heapify again.

Constraints:

- Maximum array size is 2^11.
- You can make sure that size of the array is $\beta K^{depth-1}$, where $\beta < K$ and depth is equal to recursion depth. That means, you can split the array into equal sized sub-arrays during recursive calls.
- Binary Min Heap should be implemented by using a linear array.
- $2 < K < 65$.
- The maximum element inside the list is INT_MAX-1 and all elements are integer. Therefore, you can insert INT_MAX to the heap as an empty location.

Evaluation:

- After your exam, black box evaluation will be carried out. You will get full points if you fill the **arr** variable as stated and return the number of comparisons, function calls and swaps correctly for the cases that will be tested.
- Because evaluation function checks the comparison and swap numbers, you will get zero point if you implement the merge function by using another way other than binary heap.

Example IO:

1)
Array size: 7, K: 7
Initial Array: {7, 6, 5, 4, 3, 2, 1}
Sorted Array: {1, 2, 3, 4, 5, 6, 7}

Number of comparison: 25
Number of swap: 14
Number of calls: 8

2)
Array size: 10, K: 15
Initial Array: {20, 45, 65, 78, 98, 65, 32, 74, 9, 1}
Sorted Array: {1, 9, 20, 32, 45, 65, 65, 74, 78, 98}

Number of comparison: 33
Number of swap: 26
Number of calls: 1

3)
Array size: 16, K: 4
Initial Array: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}
Sorted Array: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16}

Number of comparison: 60
Number of swap: 20
Number of calls: 21

4)
Array size: 20, K: 5
Initial Array: {79, 63, 21, 78, 52, 63, 45, 10, 0, 1, 22, 100, 89, 66, 2, 63, 89, 98, 99, 785}
Sorted Array: {0, 1, 2, 10, 21, 22, 45, 52, 63, 63, 63, 66, 78, 79, 89, 89, 98, 99, 100, 785}

Number of comparison: 72
Number of swap: 32
Number of calls: 6

Requested files

the1.cpp

```cpp
#include "the1.h"
#include <climits>


//You can add your own helper functions

int kWayMergeSortWithHeap(int* arr, int K, int size, long& comparison, long& swap){

  int number_of_calls = 1;

    //Your code here
    return number_of_calls;
}


```

test.cpp

```cpp
//This file is entirely for your test purposes.
//This will not be evaluated, you can change it and experiment with it as you want.
#include <iostream>
#include <fstream>
#include <random>
#include <ctime>
#include "the1.h"

// the1.h only contains declaration of the function:
// int kWayMergeSortWithHeap(int* arr, int K, int size, long& comparison, long& swap) ;

using namespace std;

void randomFill(int*& arr, int size, int minval, int interval){
    arr = new int [size];
    for (int i = 0; i < size; i++)
    {
        arr[i] = minval + (random() % interval);
    }
}

void print_to_file(int* arr, int size){
    ofstream ofile;
    ofile.open("sorted.txt");
    for(int i = 0; i < size; i++)
        ofile << arr[i] << endl;
}

void read_from_file(int*& arr, int& K, int& size){

    char addr[]= "input01.txt";
    ifstream infile (addr);
    if (!infile.is_open())
    {
        cout << "File \'"<< addr
            << "\' can not be opened. Make sure that this file exists." <<endl;
        return;
    }
  infile >> K;
  infile >> size;
    arr = new int [size];

    for (int i=0; i<size;i++) {

        infile >> arr[i];
    }
}


void test(int* arr, int K, int array_size){

    clock_t begin, end;
    double duration;

    //data generation and initialization- you may test with your own data
    long comparison = 0;
    long swap = 0;
    int calls;




    // Print initial array
    cout << "Array size: " << array_size << ", K: " << K << endl << endl;
    cout << "Initial Array: {";
    for(int i=0; i<array_size; i++){
        cout << arr[i];
        if(i != array_size-1) cout << ", ";
    }
    cout << "}" << endl;

    // Function call and and calculate the duration
    if ((begin = clock() ) ==-1)
        cerr << "clock error" << endl;

    calls = kWayMergeSortWithHeap(arr, K, array_size, comparison, swap);

    if ((end = clock() ) ==-1)
        cerr << "clock error" << endl;


    cout << "Sorted Array: {";
    for(int i=0; i<array_size; i++){
        cout << arr[i];
        if(i != array_size-1) cout << ", ";
    }
    cout << "}" << endl << endl;

    duration = ((double) end - begin) / CLOCKS_PER_SEC;
    cout << "Duration: " << duration << " seconds." <<endl;
    cout << "Number of comparison: " << comparison << endl <<
            "Number of swap: " << swap << endl <<
            "Number of calls: " << calls << endl;
    print_to_file(arr, array_size);
    // Calculation and output end
}

int main(){
    int size = 15;
    int K = 5;
    int minval = 0;
    int interval = 100;
    int *arr;
    // Randomly generate initial array:
    randomFill(arr, size, minval, interval);
    // Read the test inputs. input01.txt through input04.txt exists.
    // read_from_file(arr, K, size);
    srandom(time(0));
    test(arr, K, size);
```

```
110         cout << endl;
111         return 0;
112     }
113
```

the1_solution.cpp

```cpp
#include<iostream>
#include<cmath>
#include <climits>
#include "the1.h"

using namespace std;

struct HeapNode{
  int value;
  int array_index;
  int next_value;
};

void swapNode(HeapNode* a, HeapNode* b){
  HeapNode temp = *a;
  *a = *b;
  *b = temp;
}

class MinHeap{
  private:
    HeapNode* heap_array;
    int size_of_heap;
  public:
    MinHeap(HeapNode input_array[], int size, long& comparison, long& swap){
        heap_array = input_array;
        size_of_heap = size;

        for(int index=(size_of_heap - 1) / 2; index >= 0; index--){
          heapify(index, comparison, swap);
        }
    }

    void heapify(int starting_index, long& comparison, long& swap){
        int right_child = 2 * starting_index + 1;
        int left_child = 2 * starting_index + 2;
        int smallest = starting_index;

        if(left_child < size_of_heap && heap_array[left_child].value != INT_MAX){
          comparison++;
          if(heap_array[left_child].value < heap_array[smallest].value){
            smallest = left_child;
          }
        }

        if(right_child < size_of_heap && heap_array[right_child].value != INT_MAX){
          comparison++;
          if(heap_array[right_child].value < heap_array[smallest].value){
            smallest = right_child;
          }
        }

        if(smallest != starting_index){
          swap++;
          swapNode(&heap_array[smallest], &heap_array[starting_index]);
          heapify(smallest, comparison, swap);
        }
    }

    HeapNode getMin(){
       return heap_array[0];
    }

    void insert(HeapNode node, long& comparison, long& swap){
      heap_array[0] = node;
      heapify(0, comparison, swap);
    }

    HeapNode* get_array(){
      return heap_array;
    }
};

void print_heap(MinHeap& heap, int array_size){
  HeapNode* sorted_heap = heap.get_array();
  int depth = 0;
  for(int i=0, j=1; i < array_size; i++){
    cout << i << ": "<< sorted_heap[i].value << "  ";
    if(j == pow(2, depth)){
      cout << endl;
      j = 1;
      depth++;
      continue;
    }
    j++;
  }
  cout << endl;
}

void kWayMerge(int* arr, int K, int size, long& comparison, long& swap){
  int binsize = size / K;
  int output_array[size];
  HeapNode* node_array = new HeapNode[K];
  for(int i=0; i<K; i++){
    node_array[i].value = arr[i*binsize];
    node_array[i].array_index = i;
    node_array[i].next_value = 1;
  }

  MinHeap heap(node_array, K, comparison, swap);
  for(int i=0; i<size; i++){
    HeapNode min = heap.getMin();
    output_array[i] = min.value;
    if(min.next_value < binsize){
      min.value = arr[min.array_index*binsize + min.next_value];
      min.next_value++;
    }
    else{
      min.value = INT_MAX;
```

```
    }
    heap.insert(min, comparison, swap);
  }
  for(int i=0; i<size; i++){
    *(arr+i) = output_array[i];
  }
}

void insertionSort(int* arr, int size, long& comparison, long& swap){
  for(int i=1; i<size; i++){
    int j = i-1;
    int key = arr[i];
    while(j >= 0 && arr[j] > key){
      arr[j+1] = arr[j];
      swap++; comparison++;
      j--;
    }
    if(j != -1) comparison++;
    arr[j+1] = key;
  }
}

int kWayMergeSortWithHeap(int* arr, int K, int size, long& comparison, long& swap){
  int number_of_calls = 1;
  if(K > size){
    insertionSort(arr, size, comparison, swap);
  }else{
    int binsize = size / K;
    for(int i=0; i<K; i++)
      number_of_calls += kWayMergeSortWithHeap(arr+i*binsize, K, binsize, comparison, swap);
    kWayMerge(arr, K, size, comparison, swap);
  }
  return number_of_calls;
}

```

VPL