CS29003 ALGORITHMS LABORATORY
Sorting
Last Date of Submission: 23 – August – 2018

General Instruction

1. Please do not use any global variable unless you are explicitly instructed so.
2. Please use proper indentation in your code.
3. Please name your file as `<roll_no>_<assignment_no>`. For example, if your roll number is 14CS10001 and you are submitting assignment 3, then name your file as `14CS10001_3.c or 14CS10001_3.cpp` as applicable.
4. Please write your name and roll number at the beginning of your program.

We have seen various comparison based sorting algorithms like bubble sort, insertion sort, quick sort, merge sort, etc. The good thing about bubble sort and insertion sort is that they are iterative algorithms. But they have $\mathcal{O}(n^2)$ running time. On the other hand, quick sort runs in time $\mathcal{O}(n \log n)$ in the average case and merge sort runs in time $\mathcal{O}(n \log n)$ in the worst case. However they are recursive algorithms. It would be really excellent to have a sorting algorithm which is iterative and runs in time $\mathcal{O}(n \log n)$. In this exercise, we will implement such an algorithm which we call **SUPERB SORTING**. Let $\mathcal{A}$ be an array of n integers. A crucial observation is that $\mathcal{A}$ can be viewed as a collection of n sorted subarrays each of size 1. In the first iteration, it pairs up these n subarrays and combine each of them into $\lceil n/2 \rceil$ sorted subarrays each of size 2 (except possibly one subarray if n is an odd integer). In general, at the $i-th$ iteration, we combine sorted subarrays each of size 2^i (except possibly one subarray if n is not divisible by 2^i) into sorted subarrays each of size 2^{i+1} (except possibly one subarray if n is not divisible by 2^{i+1}).

Part I: Implement Combine and SUPERB SORTING

Implement combine using the following prototype.

void combine(int $\mathcal{A}[]$, int left, int middle, int right)

It assumes that the subarrays A[left,..., middle] and A[middle+1,..., right] are sorted. Once it returns, the subarray A[left, ..., right] becomes sorted. Any temporary array which you may need within combine should be created using malloc and freed before returning from the function.

Now implement **SUPERB SORTING** by repeatedly calling combine. The prototype of **SUPERB SORTING** should be as follows.

void superbSorting(int $\mathcal{A}[]$, int sizeOfA)

It takes an array $\mathcal{A}$ of integers and its size as input, and it sorts the array $\mathcal{A}$.

Part II: Convert an Array into Another

Usually in sorting, we arrange the elements of an array either in non-decreasing or non-increasing order. In this part, we will use sorting algorithms to arrange numbers in an array $\mathcal{A}$ according to the order defined by another array $\mathcal{B}$ which contains all elements of $\mathcal{A}$ and possibly more. We assume that the elements of both $\mathcal{A}$ and $\mathcal{B}$ are all distinct. For comparing elements of $\mathcal{A}$ according to the order defined by $\mathcal{B}$, write a compare function using the following prototype.

int compare1(int $\mathcal{X}[]$, int sizeOfX, int x, int y)

The compare1 function assumes that both x and y are present in $\mathcal{X}$. It returns 1 if there exist $0 \leqslant i < j \leqslant n-1$ with $\mathcal{X}[i] = x$ and $\mathcal{X}[j] = y$. It returns -1 if there exist $0 \leqslant i < j \leqslant n-1$ with $\mathcal{X}[i] = y$ and $\mathcal{X}[j] = x$. Since x and y are assumed to exist in $\mathcal{X}$, the location of x and y can be found by making one scan of the array $\mathcal{X}$. So compare1 can be implemented in $\mathcal{O}(n)$ time.

Write another combine and sorting functions using the following prototypes which are the same as part I except they use compare1 for comparing two elements (instead of using relational operators like $<=, <, ==, >, >=$).

void combine1(int $\mathcal{A}[]$, int left, int middle, int right, int $\mathcal{B}[]$, int sizeOfB)

void superbSorting1(int $\mathcal{A}[]$, int sizeOfA, int $\mathcal{B}[]$, int sizeOfB)

It takes two arrays $\mathcal{A}$ and $\mathcal{B}$ of integers along with their size as input, and it rearranges the elements in the array $\mathcal{A}$ according to the order defined by $\mathcal{B}$.

Part III: A Faster Algorithm for Conversion

Even when size of $\mathcal{B}$ is $\mathcal{O}(n)$, the compare1 function, as described in part II, takes $\mathcal{O}(n)$ time in the worst case which makes the worst case running time of the algorithm $\mathcal{O}(n^2 \log n)$. Now assume that $\mathcal{B}$ has $2n$ numbers for some positive integer n, they all belong to $\{1, 2, \ldots, 10n\}$, and they are all distinct as before. The array $\mathcal{A}$ has n numbers and they are all distinct. Every number in $\mathcal{A}$ is present in $\mathcal{B}$. Can you now design an algorithm for rearranging $\mathcal{A}$ according to the order defined by $\mathcal{B}$ which runs in time $\mathcal{O}(n \log n)$ and implement it? You are allowed to use $\mathcal{O}(n)$ extra space.

main()

In main() make the following declarations.

Take as input the number of numbers n in the array $\mathcal{A}$. Then dynamically create an array $\mathcal{A}$ of size n and read n integers in the array $\mathcal{A}$.

Sort the array $\mathcal{A}[]$ by calling the function **SUPERB SORTING**. Then print the sorted array.

Next take another n integers as input and store it in $\mathcal{A}$. Assume that the numbers in $\mathcal{A}$ are all distinct.

Next take as input the number of numbers m in the array $\mathcal{B}$. Assume $m \geqslant n$. Then dynamically create an array $\mathcal{B}$ of size m and read m integers in the input array $\mathcal{B}$. Again assume that the numbers in $\mathcal{B}$ are all distinct and every number in $\mathcal{A}$ is present in $\mathcal{B}$.

Using the superbSorting1 function of part II, rearrange the numbers in the array $\mathcal{A}$ according to the order defined by $\mathcal{B}$. Then print $\mathcal{A}[]$ after conversion.

Next, take one permutation of 1 to n as input and store it in $\mathcal{A}[]$.

Next take as input the number of numbers m in the array $\mathcal{B}$. Assume $10n \geqslant m \geqslant n$. Then, read m numbers one by one and create the input array $\mathcal{B}[]$. Again assume that the numbers in $\mathcal{B}$ are all distinct and every number in $\mathcal{A}$ is present in $\mathcal{B}$. Using the function you have written in part III, rearrange the numbers in the array $\mathcal{A}$ according to the order defined by $\mathcal{B}$. Then print $\mathcal{A}[]$ after conversion.

Sample Output

Enter number of numbers in the array $\mathcal{A}$: 10
Enter numbers in the array $\mathcal{A}$: 8 1 4 2 7 10 17 300 20 500

Array $\mathcal{A}$ sorted in non-decreasing order: 1 2 4 7 8 10 17 20 300 500

Enter numbers in the array $\mathcal{A}$: 8 1 4 2 7 10 17 30 20 50

Enter number of numbers in the array $\mathcal{B}$: 15
Enter numbers in the array $\mathcal{B}$: 790 7 30 119 2 1 13 8 50 10 57 4 31 17 20

Array $\mathcal{A}$ after rearranging: 7 30 2 1 8 50 10 4 17 20

Enter numbers in the array $\mathcal{A}$: 8 3 1 5 4 7 9 10 2 6

Enter number of numbers in the array $\mathcal{B}$: 17
Enter numbers in the array $\mathcal{B}$: 5 17 9 31 41 3 4 59 8 93 61 73 1 7 2 6 10

Array $\mathcal{A}$ after rearranging: 5 9 3 4 8 1 7 2 6 10