CS29003 Algorithms Laboratory

Assignment No: 2

Last date of submission: 23–January–2018

Suppose that you want to sort an array. You are not allowed to modify the array. This may be because the array resides in read-only memory, or because you do not have write access to the array. You can still implement the standard sorting algorithms by manipulating arrays of indices, to which you have read and write accesses. This assignment deals with a couple of such applications. This assignment also demonstrates that merge sort and quick sort are *comparison-based* sorting algorithms.

Part 1: Indexed Merge Sort

A black box contains n balls $B_0, B_1, B_2, \ldots, B_{n-1}$. Suppose that no two balls are of the same size. You do not know the size of any of the B_i's, because the balls are hidden inside the black box. The black box is, however, equipped to answer queries of the form **`compareballs(i,j)`** which returns $0, 1, -1$ according as whether B_i and B_j are of the same size (in the case $i = j$), B_i is bigger than B_j, or B_i is smaller than B_j, respectively. Your task is to prepare an array $S = [s_0, s_1, s_2, \ldots, s_{n-1}]$ of indices such that $B_{s_0}, B_{s_1}, B_{s_2}, \ldots, B_{s_{n-1}}$ is the sorted sequence of the balls in the ascending order of their sizes.

Modify merge sort to solve your problem. Your modification should use the array S to look into the actual sizes of the balls. Instead of moving/swapping/copying the sizes themselves, you move/swap/copy the indices. For example, if $s_i = j$, then B_{s_i} is the j-th ball B_j (or the size of the j-th ball). But you cannot access B_j or its size. However, you are free to make any modification to your own array S.

Part 2: Two-Way Indexed Quick Sort

The black box also contains n boxes $X_0, X_1, X_2, \ldots, X_{n-1}$. It is given that for every ball B_i there is a unique box X_j such that B_i snugly fits in X_j. Definitely, B_i fits in a bigger box, but does not fit in a smaller box. Your task is to find out which ball fits snugly in which box, that is, you need to prepare an array $M = [m_0, m_1, m_2, \ldots, m_{n-1}]$ such that the ball B_i is the exact match for the box X_{m_i}.

In Part 1, you have sorted the balls in the ascending order of their sizes. Likewise, if you can sort the boxes, you are done. Unfortunately, the black box is not willing to answer any query on comparisons of box sizes. It instead is equipped to answer queries of the form **`fitsin(i,j)`**. The return value is 0 if the box X_j is the exact match for the ball B_i. If B_i fits in X_j but it is not a snug fitting, the return value is 1. Finally, if B_i is too large to fit in X_j, then -1 is returned.

Modify quick sort to prepare M. You may or may not use the array S available from Part 1. Partitioning may use additional arrays. Now, make **`fitsin`** queries, but **`compareballs`** queries are not allowed. The balls and boxes arrays are sufficiently randomized, so you can expect an $O(n \log n)$ performance of quick sort. If you make more (like $\Theta(n^2)$) queries, the black box becomes very angry, and refuses to continue talking with your program.

Handling the Black Box

Download and store the compiler-specific binary file **`blackbox2.o`** in the <u>same</u> directory where your program resides. Compile as follows.

```
gcc myprog.c blackbox2.o
g++ myprog.cpp blackbox2.o
```

At the top of your code, write these **`extern`** prototypes.

```
extern int registerme ( int );
extern void startsort ( );
extern int compareballs ( int, int );
extern void verifysort ( int * );
extern void startmatch ( int );
extern int fitsin ( int i, int j );
extern void verifymatch ( int * );
```

Register yourself by calling **`registerme(n)`** at the beginning of your **`main()`** function. Here, n is the count of balls (or boxes) in the black box. You may read n from the user. The recommended values are in the range $10^3 \leqslant n \leqslant 10^7$. However, you can debug with smaller examples. All other black-box calls assume this value of n. You do not need to pass n explicitly to these calls. Your own functions may, however, need the value of n as parameters.

The call **`startsort()`** lets the black box prepare to help you in solving Part 1. Call **`compareballs()`** as many times as your merge sort implementation wants. When you have prepared the final sorting index array S, call **`verifysort(S)`** to verify the correctness of your implementation.

Start with the call **`startmatch(flag)`** for solving Part 2. If you want to use the array S of Part 1 in the matching problem, pass 1 as **`flag`**, else pass 0 as **`flag`**. You may call **`fitsin()`** at most a limited number of times. Verify the correctness of the final matching array M produced by your quick sort implementation, by calling **`verifymatch(M)`**.

Here is how your **`main()`** function would look like.

```
int n = 1000000;

registerme(n);

printf("\n+++ Sorting the balls...\n");
startsort();
...
verifysort(S);

printf("\n+++ Finding the matching boxes...\n");
startmatch(0);
...
verifymatch(M);
```

Successful output

```
+++ Sorting the balls...
*** Great! You cracked it.

+++ Finding the matching boxes...
*** Great! You cracked it.
```

Submit a single C/C++ source file. Do not use global/static variables.