Practice Arena

Practice problems aimed to improve your coding skills.

PRACTICE-02_SCAN-PRINT
PRACTICE-03_TYPES
LAB-PRAC-02_SCAN-PRINT
LAB-PRAC-01
PRACTICE-04_COND
BONUS-PRAC-02
LAB-PRAC-03_TYPES
PRACTICE-05_COND-LOOPS
LAB-PRAC-04_COND
LAB-PRAC-05_CONDLOOPS
PRACTICE-07_LOOPS-ARR
LAB-PRAC-06_LOOPS
LAB-PRAC-07_LOOPS-ARR
LABEXAM-PRAC-01_MIDSEM
PRACTICE-09_PTR-MAT
LAB-PRAC-08_ARR-STR
PRACTICE-10_MAT-FUN
LAB-PRAC-09_PTR-MAT
LAB-PRAC-10_MAT-FUN
PRACTICE-11_FUN-PTR
LAB-PRAC-11_FUN-PTR
LAB-PRAC-12_FUN-STRUC
LABEXAM-PRAC-02_ENDSEM
LAB-PRAC-13_STRUC-NUM
LAB-PRAC-14_SORT-MISC
- Predecessor and Successor
- Insertion Sort
- Link a List
- The United Sums of Arrays
- Bubble Sort
- Pretty Queues Revisited
- Just About Sorted
- Brick Sort
- All My Descendants
- Mr C likes a Majority
- Cocktail Sort
- All My Descendants - Part II

Brick Sort

LAB-PRAC-14_SORT-MISC

Brick Sort [20 marks]

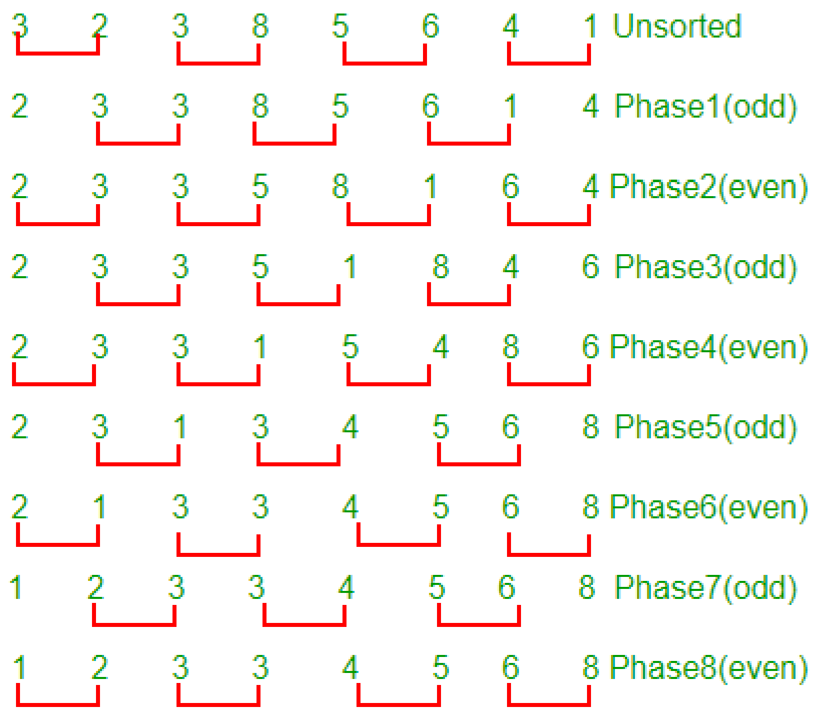
Problem Statement
There are several sorting algorithms that have been developed over the years. We saw some during the lectures and will explore others here. One simple algorithm is called Brick Sort. The algorithm got its name since it sorts alternating elements of the array in subsequent phases, so that the whole arrangement (see figure below) looks like bricks have been laid down. This variant was originally designed for parallel processing and may work very well on parallel computing architectures such as GPUs.

Brick sort proceeds in n phases. In the first phase, even indices of the array get compared to the element to their immediate right and if the pair is out of order (i.e. the element at the even index is strictly greater than the element to its immediate right), then the pair is swapped. In the second phase, odd indices of the array get compared (and swapped if out of order) to the element to their immediate right. The third phase again considers the even indices and so on. If the array has n elements, then the algorithm works for n phases. The array is guaranteed to be sorted at the end of the n-th phase.

The first line of the input will give you n, a strictly positive integer and the second line will give you n integers, separated by a space. Store these numbers in an array of size n. In your output, you have to print the array after each phase of brick sort, on a separate line. Print the array by printing each number in the array from left to right with two numbers separated by a single space. However, there should be no space at the end of a line. The image below shows you how brick sort works. Also notice that at some iterations (e.g. phase 7), nothing needs to be done.

Now, we agree that at first it is not entirely clear why this algorithm should even be correct (i.e. why should it completely sort the array in n phases). The proof of correctness of this algorithm is a bit involved and is based on a powerful meta-theorem by Donald Knuth called the 0-1 principle. Check it out if you are interested.

Image courtesy: geeksforgeeks.org (image modified for better clarity)


3 2 3 8 5 6 4 1 Unsorted
2 3 3 8 5 6 1 4 Phase1(odd)
2 3 3 5 8 1 6 4 Phase2(even)
2 3 3 5 1 8 4 6 Phase3(odd)
2 3 3 1 5 4 8 6 Phase4(even)
2 3 1 3 4 5 6 8 Phase5(odd)
2 1 3 3 4 5 6 8 Phase6(even)
1 2 3 3 4 5 6 8 Phase7(odd)
1 2 3 3 4 5 6 8 Phase8(even)

Caution

1. If the array has an odd number of elements, say 5, then the index 4 element will have no element to its right in the phases when even elements are being compared to the element to their immediate right (for example the first phase). So nothing will need to be done. However, the index 4 element will get compared to the index 3 element in the 2nd pass when odd index elements are being compared to their immediate neighbors to the right.
2. Please do not try to cheat by using library functions like qsort(). These will not sort the array in the order insertion sort will and hence you will not get partial marks for printing the intermediate steps of the algorithm.
3. The n numbers we give you may be positive, negative or zero. The same number may occur twice in the list too.
4. The number of elements n can be any strictly positive number, even 1. Your output must have exactly n lines.
5. Some phases may not require any work. However, you must still print the array after those phases. Your output must have exactly n lines.
6. Be careful about extra/missing lines and extra/missing spaces in your output. There should be no space at the end of any line in your output, nor should there be any extra newlines at the end of your output.

EXAMPLE:
INPUT

```
4
3 4 1 2
```

OUTPUT:

```
3 4 1 2
3 1 4 2
1 3 2 4
1 2 3 4
```

Explanation:
First phase
3 4 1 2 (Checking for swap between 3 & 4 - nothing to be done)
3 4 1 2 (Checking for swap between 1 & 2 - nothing to be done)
No other even index elements left.

Second phase
3 1 4 2 (Checking for swap between 4 & 1 - swap them)
No other odd index elements left.

Third phase
1 3 4 2 (Checking for swap between 3 & 1 - swap them)
1 3 2 4 (Checking for swap between 4 & 2 - swap them)
No other even index elements left.

Fourth phase
1 2 3 4 (Checking for swap between 3 & 2 - swap them)
No other odd index elements left.

Grading Scheme:
Total marks: **[20 Points]**

There will be partial grading in this question. There are several lines in your output. Printing each line correctly, in the correct order, carries equal weightage. Each visible test case is worth 2 points and each hidden test case is worth 4 points. There are 2 visible and 4 hidden test cases.

Please remember, however, that when you press Submit/Evaluate, you will get a green bar only if all parts of your answer are correct. Thus, if your answer is only partly correct, Prutor will say that you have not passed that test case completely, but when we do autograding afterwards, you will get partial marks.

Start Solving! (/editor/practice/6292)