

Awaking drenched in sweat one night, you clearly see your path to fame and fortune. You will build a robotic rhinoceros and tour the country singing songs about nature to children, who will be allowed to play and interact with the rhinoceros. While a real rhinoceros would be too dangerous, you believe a robotic rhinoceros can be kept in check. In each of the situations below, which sort would you use? In all cases, assume memory is not an issue, and that the goal is to minimize run time so that the rhino can react as quickly as possible to any potential trouble. Answers may be used many times.

- | | | |
|-------|---|--|
| ----- | The rhinoceros is outfitted with a large number of sensors, each of which generates objects of type <code>Observation</code> . Observations include many instance variables, including <code>importance</code> , <code>timestamp</code> , <code>pressure</code> , <code>temperature</code> , <code>light intensity</code> , etc. These are placed in an unsorted array, and every time 1000000 <code>Observations</code> are generated, they are delivered to a central processing unit that sorts the <code>Observations</code> by the <code>importance</code> field, which is of type <code>double</code> . What sort should you use to minimize the run time required to sort all <code>Observations</code> by <code>importance</code> ? | A. Quicksort
B. Mergesort
C. Insertion sort
D. Selection sort
E. Knuth shuffle |
| ----- | Due to some close calls, you're going to refactor the sorting process to deal with a rare but dangerous situation where some <code>Observations</code> are generated with an incorrect <code>importance</code> value. For engineering reasons not described here, you can detect these by sorting by the <code>timestamp</code> and <code>importance</code> of each <code>Observation</code> .

Instead of <code>importance</code> , you first want to sort by the <code>timestamp</code> of each <code>Observation</code> . The <code>timestamp</code> is of a comparable type called <code>DateTime</code> . What sort should you use to minimize the run time required to sort all 1000000 <code>Observations</code> by <code>timestamp</code> ? | |
| ----- | After sorting by <code>timestamp</code> , you want to sort by <code>importance</code> such that all the objects of the same <code>timestamp</code> stay clustered. What sort should you use to minimize the run time while maintaining this clustering? | |
| ----- | You iterate through the array, update the <code>importance</code> of the very rare bad <code>Observations</code> with a new value, and sort once more. What sort do you use to put items in order of <code>importance</code> while minimizing run time? | |

Answers to above:

A. If we want to sort a set of randomly ordered items such that we get the best performance and we don't care about stability, we should use quicksort.

A or C. Again, we just want speed, but don't care about stability. If the Observations are randomly ordered, quicksort is the winner. It was also reasonable to assume that the unsorted Observation array was filled in roughly by timestamp, in which case we'd want to use insertion sort to take advantage of the partially ordered nature of the array.

B. In this case, we want speed and stability, and our objects are randomly ordered with respect to importance. The winning sort here is mergesort.

C. Here we have an array that is almost perfectly ordered, so we should use insertion sort.

Inversions. Design a subquadratic algorithm that counts the number of inversions in an array.

Answerish: Use a modified version of mergesort that counts the number of inversions fixed as it sorts. See <http://algs4.cs.princeton.edu/22mergesort/Inversions.java.html> for an example.

Linked List Scramble. Given a singly-linked list containing N items, rearrange the items uniformly at random. Your algorithm should consume a logarithmic (or constant) amount of extra memory and run in time proportional to $N \log N$ in the worst case.

Hint (but not full solution): design a linear-time subroutine that can take two uniformly shuffled linked lists of sizes N_1 and N_2 and combined them into a uniformly shuffled linked lists of size N_1+N_2 .