CS 61B Sorting and Hashing Fall 2014

1 Sorting I

Show the steps taken by each sort on the following unordered list:

`106, 351, 214, 873, 615, 172, 333, 564`

(a) Quicksort (assume the pivot is always the first item in the sublist being sorted and the array is sorted in place). At every step circle everything that will be a pivot on the next step and box all previous pivots.

106 351 214 873 615 172 333 564
106 351 214 873 615 172 333 564
106 214 172 333 351 873 615 564
106 172 214 333 351 615 564 873
106 172 214 333 351 564 615 873

(b) Merge sort. Show how the list is broken up at every step.

106 351 214 873 615 172 333 564
106 351 214 873 172 615 333 564
106 214 351 873 172 333 564 615
106 172 214 333 351 564 615 873

(c) LSD radix sort.

106 351 214 873 615 172 333 564
351 172 873 333 214 564 615 106
106 214 615 333 351 564 172 873
106 172 214 333 351 564 615 873

(d) Give an example of a situation where using insertion sort is more efficient than using merge sort.

Insertion sort performs better than merge sort for lists that are already almost in sorted order (i.e. if the list has only a few elements out of place or if all elements are within k positions of their proper place and $k < \log N$).

2 Sorting II

Match the sorting algorithms to the sequences, each of which represents several intermediate steps in the sorting of an array of integers.

Algorithms: Quicksort, merge sort, heap sort, MSD radix sort, insertion sort.

(a) 12, 7, 8, 4, 10, 2, 5, 34, 14
7, 8, 4, 10, 2, 5, 12, 34, 14
4, 2, 5, 7, 8, 10, 12, 14, 34
Quicksort

(b) 23, 45, 12, 4, 65, 34, 20, 43
12, 23, 45, 4, 65, 34, 20, 43
Insertion sort

(c) 12, 32, 14, 11, 17, 38, 23, 34
12, 14, 11, 17, 23, 32, 38, 34
MSD radix sort

(d) 45, 23, 5, 65, 34, 3, 76, 25
23, 45, 5, 65, 3, 34, 25, 76
5, 23, 45, 65, 3, 25, 34, 76
Merge sort

(e) 23, 44, 12, 11, 54, 33, 1, 41
54, 44, 33, 41, 23, 12, 1, 11
44, 41, 33, 11, 23, 12, 1, 54
Heap sort

3 Hash Codes

(a) Suppose that we represent Tic-Tac-Toe boards as 3 by 3 arrays of integers (each of which is in the range 0 to 2). Describe a good hash function for Tic-Tac-Toe boards that are represented in this manner. Try to come up with one such that boards that are not equal will never have the same hash code.
We can interpret the Tic-Tac-Toe board as a nine digit base 3 number, and use this as the hash code. More concretely, if the array used to store the Tic-Tac-Toe board was called board, then we could compute the hash code as follows:

$$\text{board}[0][0] + 3 \cdot \text{board}[0][1] + 3^2 \cdot \text{board}[0][2] + 3^3 \cdot \text{board}[1][0] + \ldots + 3^8 \cdot \text{board}[2][2]$$

This hash code actually guarantees that any two distinct Tic-Tac-Toe boards will always have distinct hash codes (in most situations this property is not feasible). Another thing to note is that if we used this same idea on boards of size $N \times N$ then it would take $\Theta(N^2)$ time to compute.

(b) Is it possible to add arbitrarily many Strings to a Java HashSet with no collisions? If not, what is the minimum number of distinct Strings you need to add to a HashSet to guarantee a collision?
No, it is not possible. Ideally, we should be able to make arbitrarily large hash codes and keep resizing the HashSet's underlying array as many times as necessary (which would mean we could add arbitrarily many Strings to a HashSet without collisions). However, in Java this is not possible. There are several reasons for this:

1) In Java, the hashCode method must return an **int**, which must have a value between -2^{31} and $2^{31}-1$. This means that there are only 2^{32} possible distinct hashCodes, so if we add $2^{32}+1$ distinct Strings then we are guaranteed that two of them will have the same hashCode.

2) In Java, arrays have a maximum size of $3^{31}-1$. So we cannot resize the HashSet's underlying array past this point. So if we add 2^{31} Strings then we are guaranteed that two of them will be put in the same bucket (though they might not have the same hashCode).

3) In Java's implementation of HashSet, the size of the underlying array is always a power of two. Thus the maximum size of the underlying array is 2^{30}, so if we add $2^{30}+1$ Strings then we are guaranteed that two of them will be put in the same bucket.

So the final answer is that $2^{30}+1$ is the minimum number of Strings required to guarantee a collision. You aren't expected to be able to come up with this exact number yourself, since it depends on the specific implementation details of Java's HashSet. Understanding the basic reasoning is enough (for instance, (1) is a good answer, though not technically correct).

4 Bonus Question

Describe a way to implement a linked list of Strings so that removing a String from the list takes constant time. You may assume that the list will never contain duplicates.

Use a doubly linked list and a HashMap whose keys are the Strings in the list and whose values are pointers to the nodes of the list. Then when removing a String, look up the corresponding node in the HashMap and delink that node from the list.