CSC345 Discussion 7

Sorting

Insertion Sort

Idea

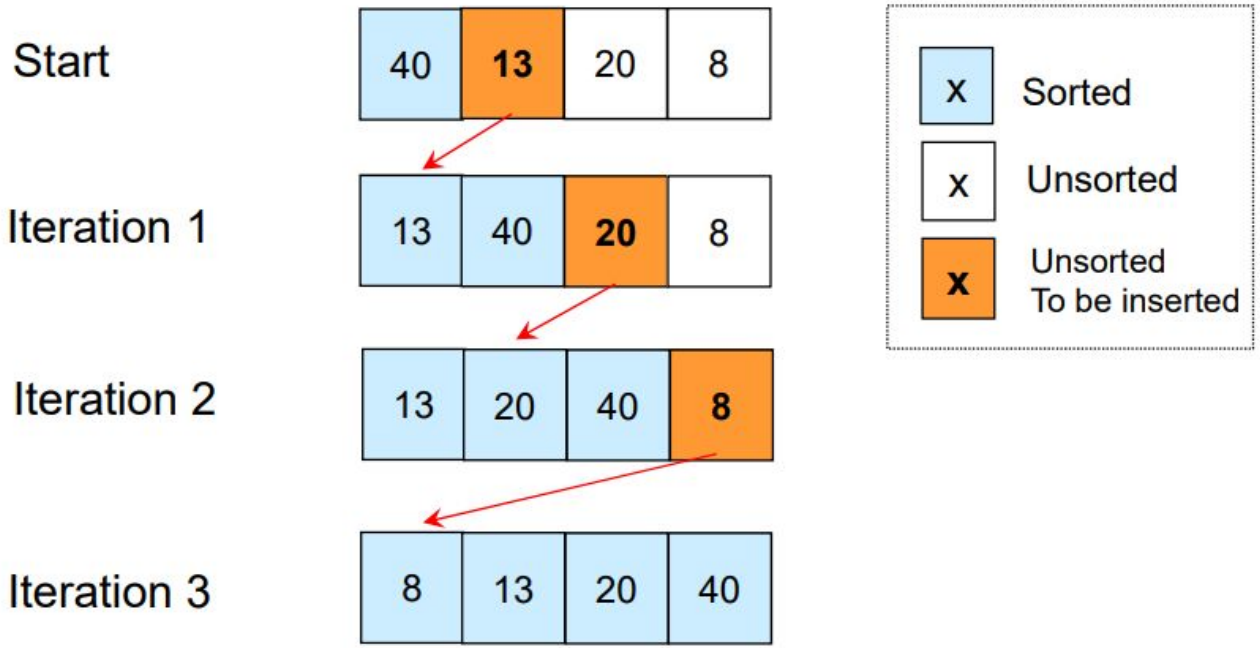
Similar to how most people arrange a hand of poker cards

Start with one card in your hand

Pick the next card and insert it into its proper sorted order

Repeat previous step for all cards

Insertion Sort



Insertion Sort

Outer-loop executes (n−1) times

Number of times inner-loop is executed depends on the input
Best-case: the array is already sorted and (a[j] > next) is always false No shifting of data is necessary
Worst-case: the array is reversely sorted and (a[j] > next) is always true

Insertion always occur at the front

Therefore, the best-case time is O(n) And the worst-case time is O(n2)

Merge Sort

Suppose we only know how to merge two sorted sets of elements into one
Merge {1, 5, 9} {1, 5, 9} with {2, 11} {2, 11} {1, 2, 5, 9, 11}

Idea (use merge to sort n items)
Merge each pair of elements into sets of 2
Merge each pair of sets of 2 into sets of 4
Repeat previous step for sets of 4 …
Final step: merge 2 sets of n/2 elements to obtain a fully sorted set

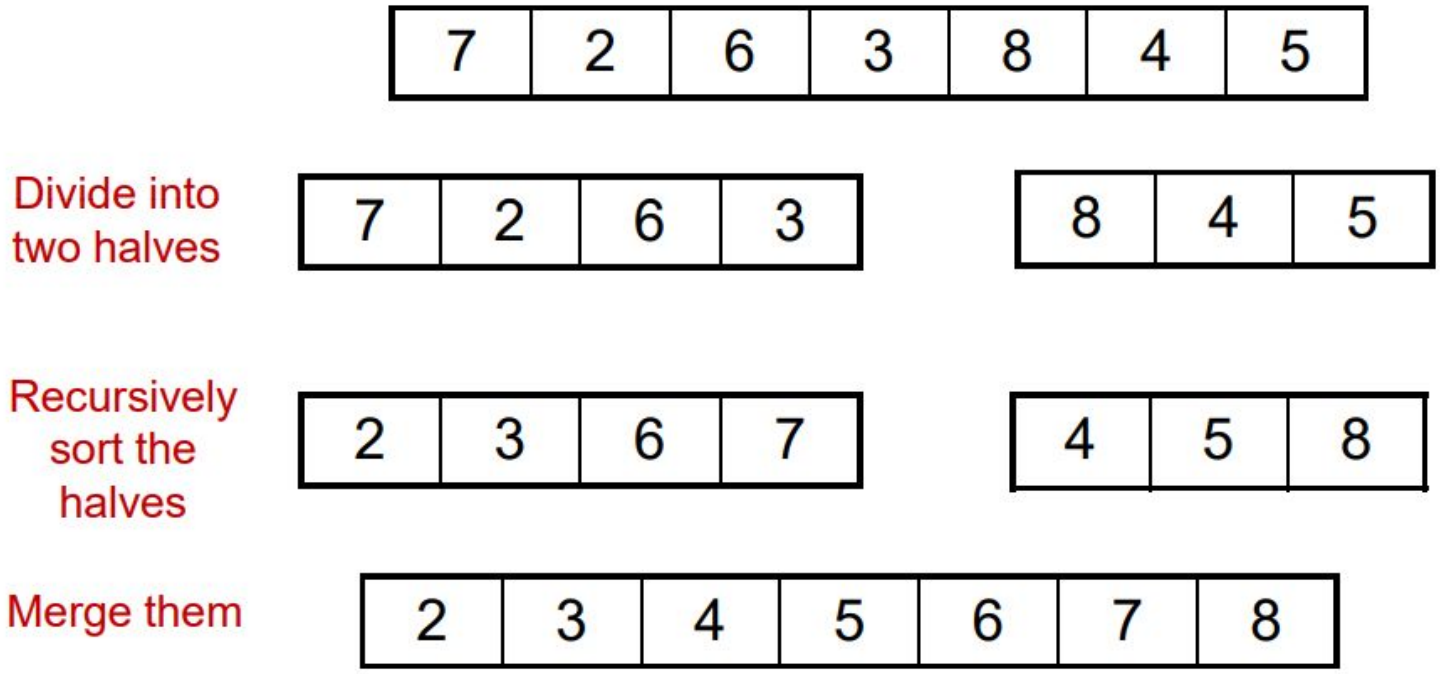
Merge Sort

Merge Sort is a divide-and-conquer sorting algorithm

Divide step
Divide the array into two (equal) halves
Recursively sort the two halves

Conquer step
Merge the two halves to form a sorted array

Merge Sort


7 2 6 3 8 4 5
Divide into two halves
7 2 6 3
8 4 5
Recursively sort the halves
2 3 6 7
4 5 8
Merge them
2 3 4 5 6 7 8

Merge Sort

Pros

The performance is guaranteed, i.e. unaffected by original ordering of the input

Suitable for extremely large number of inputs

Can operate on the input portion by portion

Cons

Not easy to implement

Requires additional storage during merging operation

O(n) extra memory storage needed

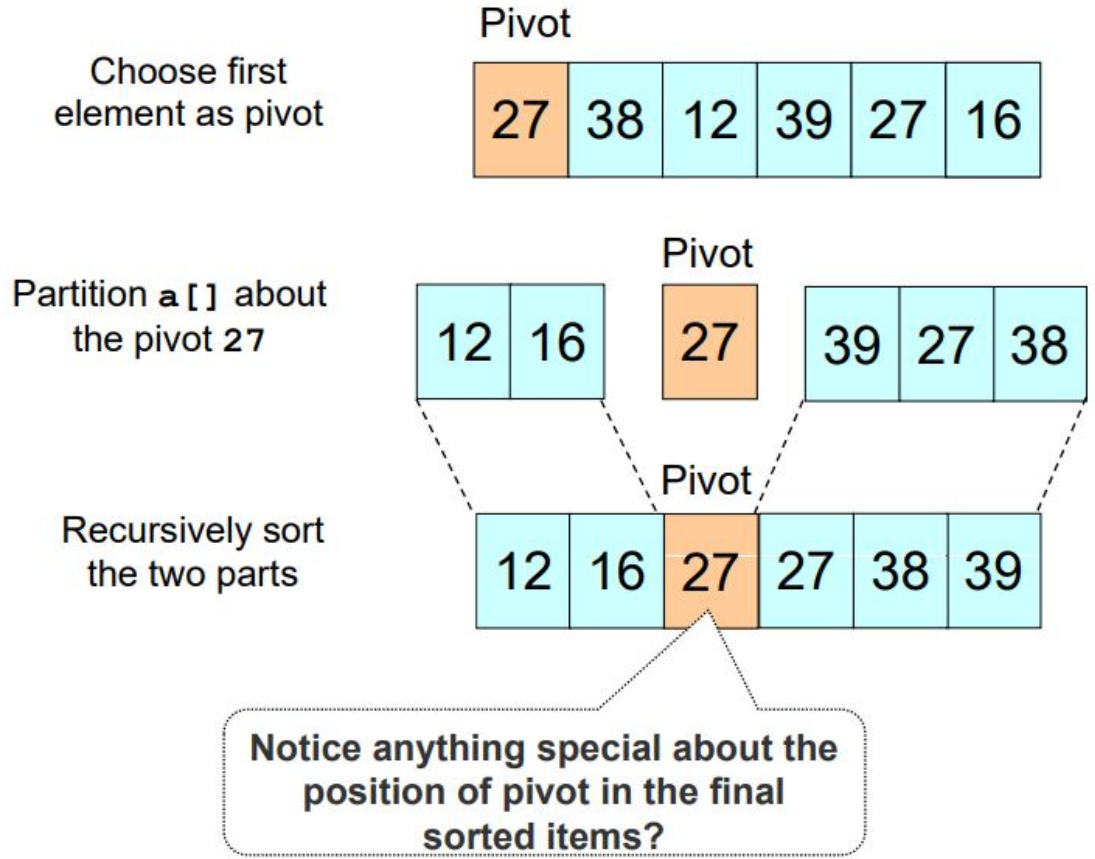
Quick Sort

Quick Sort is a divide-and-conquer algorithm

Divide step
Choose an item
Choose an item p (known as (known as pivot) and partition the) and partition the items of a[i...j] into two parts
Items that are smaller than p
Items that are greater than or equal to p
Recursively sort the two parts

Conquer step
Do nothing! In comparison, Merge Sort spends most of the time in conquer step but very little time in divide step

Quick Sort


Pivot
Choose first element as pivot
27 38 12 39 27 16
Pivot
Partition a[] about the pivot 27
12 16 27 39 27 38
Pivot
Recursively sort the two parts
12 16 27 27 38 39
Notice anything special about the position of pivot in the final sorted items?

Quick Sort

Best case
occurs when partition always splits the array into two equal halves
Depth of recursion is log n
Each level takes n or fewer comparisons, so the time complexity is O(n log n)

In practice, worst case is rare, and on the average we get some good splits and some bad ones

Average time is also O(n log n)

Stable Sorts

A sorting algorithm is stable if the relative order of elements with the same key value is preserved by the algorithm

Example application of stable sort
Assume that names have been sorted in alphabetical order
Now, if this list is sorted again by tutorial group number, a stable sort algorithm would ensure that all students in the same tutorial groups still appear in alphabetical order of their names

Sorts

	Worst Case	Best Case	In-place?	Stable?
`Selection Sort`	$O(n^2)$	$O(n^2)$	Yes	No
`Insertion Sort`	$O(n^2)$	$O(n)$	Yes	Yes
`Bubble Sort`	$O(n^2)$	$O(n^2)$	Yes	Yes
`Bubble Sort 2`	$O(n^2)$	$O(n)$	Yes	Yes
`Merge Sort`	$O(n \lg n)$	$O(n \lg n)$	No	Yes
`Quick Sort`	$O(n^2)$	$O(n \lg n)$	Yes	No
`Radix sort`	$O(dn)$	$O(dn)$	No	yes

Question 0

Sort the sequence 4, 3, 8, 4 by using

a) Insertion sort

b) Mergesort

Question 1

Using induction, prove that Insertion Sort will always produce a sorted array.

```
static <E extends Comparable<? super E>>
void inssort(E[] A) {
  for (int i=1; i<A.length; i++) // Insert i'th record
    for (int j=i; (j>0) && (A[j].compareTo(A[j-1])<0); j--)
      DSutil.swap(A, j, j-1);
}
```

Question 2

The Bubble Sort implementation has the following inner for loop:
for (int j=n-1; j>i; j--)
Consider the effect of replacing this with the following statement:
for (int j=n-1; j>0; j--)
Would the new implementation work correctly? Would the change affect the asymptotic complexity of the algorithm? How would the change affect the running time of the algorithm?

Question 3

Recall that a sorting algorithm is said to be stable if the original ordering for duplicate keys is preserved. Of the sorting algorithms Insertion Sort, Bubble Sort, Selection Sort, Mergesort, Quicksort, which of these are stable, and which are not? For each one, describe either why it is or is not stable. If a minor change to the implementation would make it stable, describe the change

Question 4

Here is a variation on sorting. The problem is to sort a collection of n nuts and n bolts by size. It is assumed that for each bolt in the collection, there is a corresponding nut of the same size, but initially we do not know which nut goes with which bolt. The differences in size between two nuts or two bolts can be too small to see by eye, so you cannot rely on comparing the sizes of two nuts or two bolts directly. Instead, you can only compare the sizes of a nut and a bolt by attempting to screw one into the other (assume this comparison to be a constant time operation). This operation tells you that either the nut is bigger than the bolt, the bolt is bigger than the nut, or they are the same size. What is the minimum number of comparisons needed to sort the nuts and bolts in the worst case?

Special Thanks

Steven Ha for diagrams and content