CS 228: Introduction to Data Structures
Lecture 12

Insertion Sort

Here is the idea:

```
for i = 1 to n - 1:
    insert A[i] in its proper place amongst A[0..i].
```

For example,

```
4 3 2 1 --> 3 4 2 1 --> 2 3 4 1 --> 1 2 3 4
```

Note that inserting `A[i]` in the correct place implies shifting up some, maybe all, elements of `A[0..i-1]` to make room for `A[i]`. All this is handled by the pseudocode below.

```
INSERTIONSORT(A)
    n = A.length
    for (i = 1; i < n; i++)
        temp = A[i]
        j = i - 1
        while j > -1 && A[j] > temp
            A[j+1] = A[j]
            --j
        A[j+1] = temp
```

The analysis is very similar to that of selection sort: There are n iterations of the outer loop and the inner loop iterates no more than n times, doing a constant amount of work per iteration. Thus, the algorithm is also $O(n^2)$.

Insertion Sort versus Selection Sort

Both algorithms take $O(n^2)$ time, so which should you use?

Here is why insertion sort might be preferable:

- In selection sort, iteration i of the outer loop always scans elements `A[i]` through `A[n-1]`. This takes time proportional to `i`, which means that the total time

is at least cn^2 for some c.

- The running time of insertion sort is proportional to n plus the number of ***inversions***. An inversion is a pair of keys `j` < k such that `j` appears after k in A. There are anywhere between zero and n·(n – 1) / 2 inversions in A. If A has few inversions (i.e., it is "almost" sorted), insertion sort can be as fast as O(n).

Moral of the Story: The O-bound tells only part of the story.

Loop Invariants

Loop invariants are statements that remain true during the execution of a loop; they provide a way to prove the correctness of algorithms. For example, insertion sort maintains the following:

> **Invariant.** At the start of iteration `i` of the outer (**`for`**) loop, subarray `A[0..i–1]` consists of the elements originally in `A[0..i–1]`, but in sorted order.

Now:

- *(Initialization)* The invariant is true at the outset.

- *(Maintenance)* The loop maintains the invariant through shifting and insertion.

- *(Termination)* At termination, `i = n`, so the invariant implies that subarray `A[0..n-1]` — *the whole array* — consists of the elements originally in `A[0..n-1]`, but in sorted order.

The last statement proves the correctness of insertion sort.

Exercise. What loop invariant does selection sort maintain?

Merge Sort

Merge sort relies on the observation that it is possible to merge two sorted arrays into one sorted array in linear time. The algorithm is recursive:

```
if n == 1  // there is nothing to sort!
    return

split A down the middle into two subarrays
    A_left and A_right

recursively sort A_left
recursively sort A_right

merge A_left and A_right into a sorted array
```

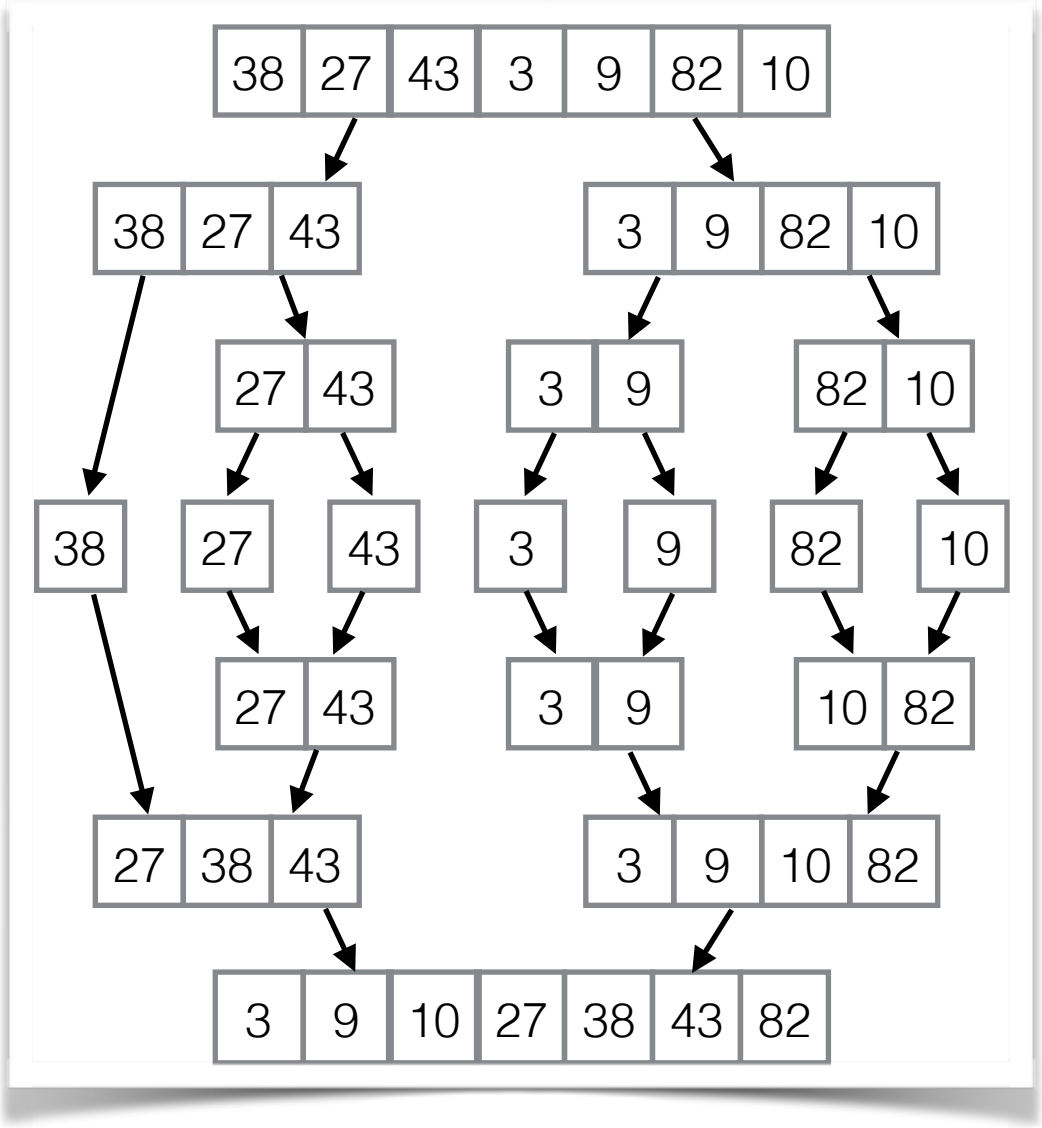
For example:

Input	8	7	3	5	2	1
Divide array into halves	8	7	3	5	2	1
Recursively sort left half	3	7	8	5	2	1
Recursively sort right half	3	7	8	1	2	5
Merge two halves to make a sorted whole	1	2	3	5	7	8

Here is the pseudocode for the algorithm.

```
MERGESORT(A)
    n = A.length
    if n ≤ 1 return
    m = n/2
    A_left = (A[0], . . . , A[m−1])
    A_right = (A[m], . . . , A[n−1])
    MERGESORT(A_left)
    MERGESORT(A_right)
    A = MERGE(A_left, A_right)
```

We will describe the details of the MERGE algorithm next time. Note that the MERGESORT pseudocode hides the details of the recursion. The figure on the next page depicts the entire execution of the algorithm, including all recursive calls.



Let us define the ***depth*** of the recursion to be the number of successive recursive calls until the recursion bottoms out; i.e., until we get to a sub-array of length one. If we start with an array of length n, then, after k successive recursive calls, we will have a sub-array of length roughly $n/2^k$. Now, it must be the case that $n/2^k \geq 1$, so $k \leq \log_2 n$. That is, the depth of the recursion is O(log n). (Notice the

similarity with the analysis of binary search.) On Wednesday, we will use this fact to show that the time complexity of MERGESORT is O(n log n).