CS 228: Introduction to Data Structures
Lecture 11
Friday, February 6, 2015

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:

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

The recursion bottoms out when we get to a one-element subarray, since there is nothing to sort. 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($A_{left}$, $A_{right}$)
```

Merging. The key part of merge sort is the merging algorithm:

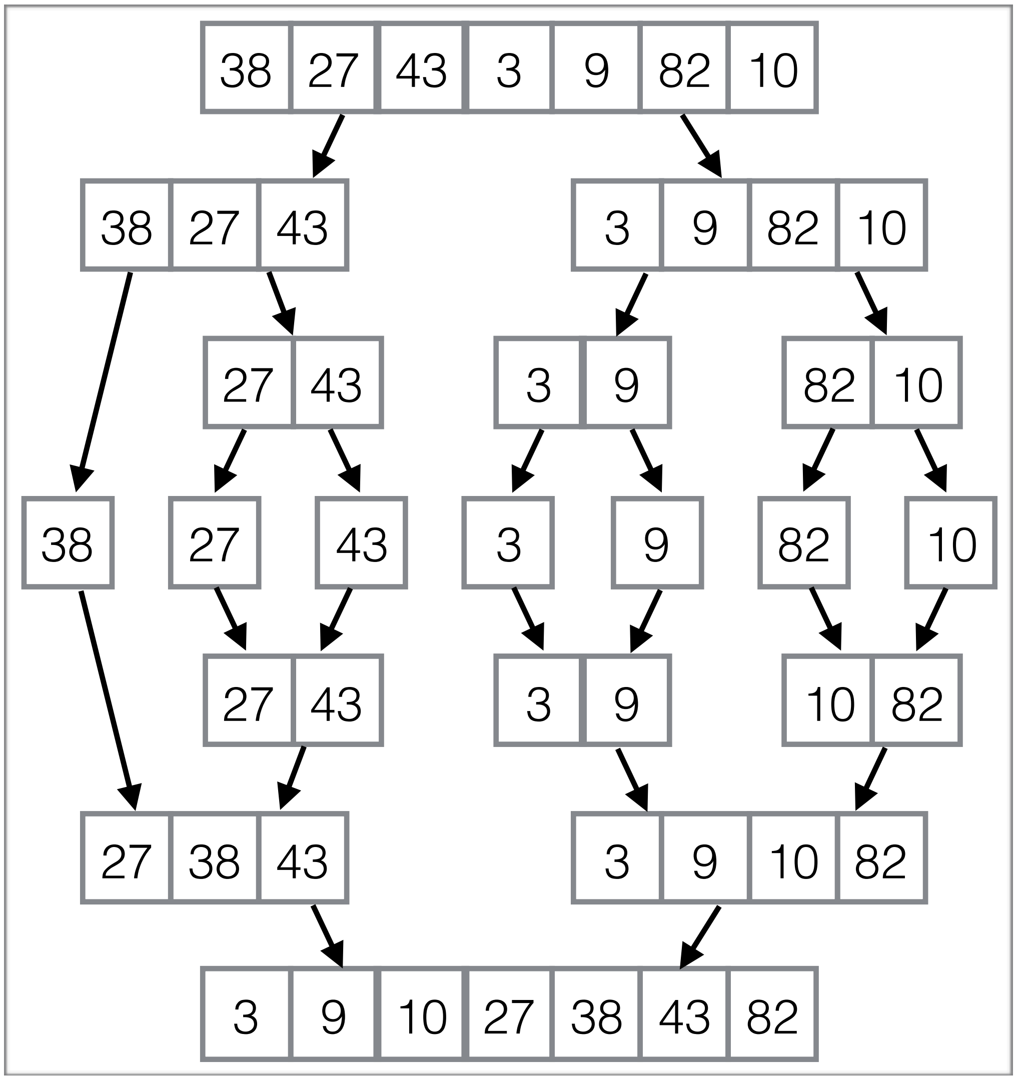
At each iteration, choose the smallest remaining item from the two input arrays, and append it to the output array. If there are any leftover elements in either array, append them to the output array

The pseudocode below assumes that the arguments B and C are arrays sorted in increasing order.

```
MERGE(B,C)
  p = B.length, q = C.length
  create an empty array D of length p+q
  i=0, j=0
  while i < p && j < q
    if B[i] ≤ C[j]
      append B[i] to D
      i++
    else
      append C[j] to D
      j++
  if i ≥ p
    append C[j],...,C[q−1] to D
  else
    append B[i],...,B[p−1] to D
  return D
```

Let us analyze MERGE(B,C). Let n = p+q, where p = B.length and q = C.length; i.e., n is the total number of items in the arrays to be merged. Since each iteration of the **while** takes constant time, the loop takes O(n) time. Appending the remainder of B or C to D also takes O(n) time. Thus, MERGE takes O(n) time. Note that MERGE requires O(n) additional space to temporarily hold the result of a merge.

Time complexity of MergeSort. Consider mergesort's ***recursion tree***, illustrated below.



Each level of the tree involves O(n) operations, and there are O(log n) levels. Hence, merge sort runs in O(n log n) time, making it asymptotically faster that either insertion sort or selection sort, which are $O(n^2)$. This big-O bound again does not tell the whole story, however. In fact, insertion sort is faster than merge sort on a sorted or nearly sorted array. Can you see why?

Quicksort

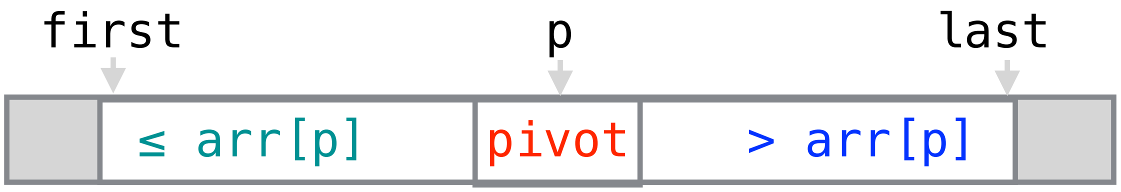
The key component of quicksort is the ***partition*** algorithm. This algorithm takes as arguments an array `arr` and two indices `first` and `last` into `arr`. These arguments must satisfy the following.

Precondition: `0 ≤ first ≤ last ≤ arr.length`

The partition algorithm rearranges `arr` and returns an integer p, such that the following condition is satisfied:

Postcondition:
(i) `arr[k] ≤ arr[p]` for all k with `first ≤ k < p`
(ii) `arr[k] > arr[p]` for all k with `p < k ≤ last`,
where p is the returned value

That is, `partition` rearranges `arr[first..last]` like this:



Then, it returns p. The value `arr[p]` is called the ***pivot***; i.e., `partition` rearranges `arr[first..last]` around the pivot.

Suppose that the signature of `partition` is

```
private static int partition
   (int[] arr, int first, int last)
```

Quicksort sorts `arr[first..last]` by first invoking `partition` and then recursively sorting `arr[first..p–1]` and `arr[p+1..last]`, where p is the index of the pivot returned by `partition`.

```
private static void quickSortRec
   (int[] arr, int first, int last)
{
   if (first >= last) return;
   int p = partition(arr, first, last);
   quickSortRec(arr, first, p – 1);
   quickSortRec(arr, p + 1, last);
}
```

This method is private, to hide the details of the recursion. Users would invoke a public method such as this one:

```
public static void quickSort(int[] arr)
{
   quickSortRec(arr, 0, arr.length-1);
}
```

Example.

```
               4 9 1 5 2 3
              /     |     \
           1 2      3      5 9 4
          /  |               |  \
         1   2               4   9 5
                                 |  \
                                 5   9
```

Notice that quicksort sorts as it goes down the recursion tree. Unlike merge sort, there is no “combine” step going up the tree.

Partitioning

There are many ways to implement partition. The following algorithm is due to Nick Lomuto.

```
PARTITION(arr,first,last)
    // Use the last element as the pivot.
    pivot = arr[last]
    i = first - 1
    for j = first to last - 1
        if arr[j] ≤ pivot
            i++
            swap arr[i] and arr[j]
    // Now put pivot in position i+1.
    swap arr[i+1] and arr[last]
    return i + 1
```

Next time, we will study PARTITION in depth (before that, though, you might want to try the algorithm out on a few examples). We will then analyze the best- and worst-case behavior of quicksort.