CS 228: Introduction to Data Structures
Lecture 13
Wednesday, September 21, 2016

Merge Sort (Continued)

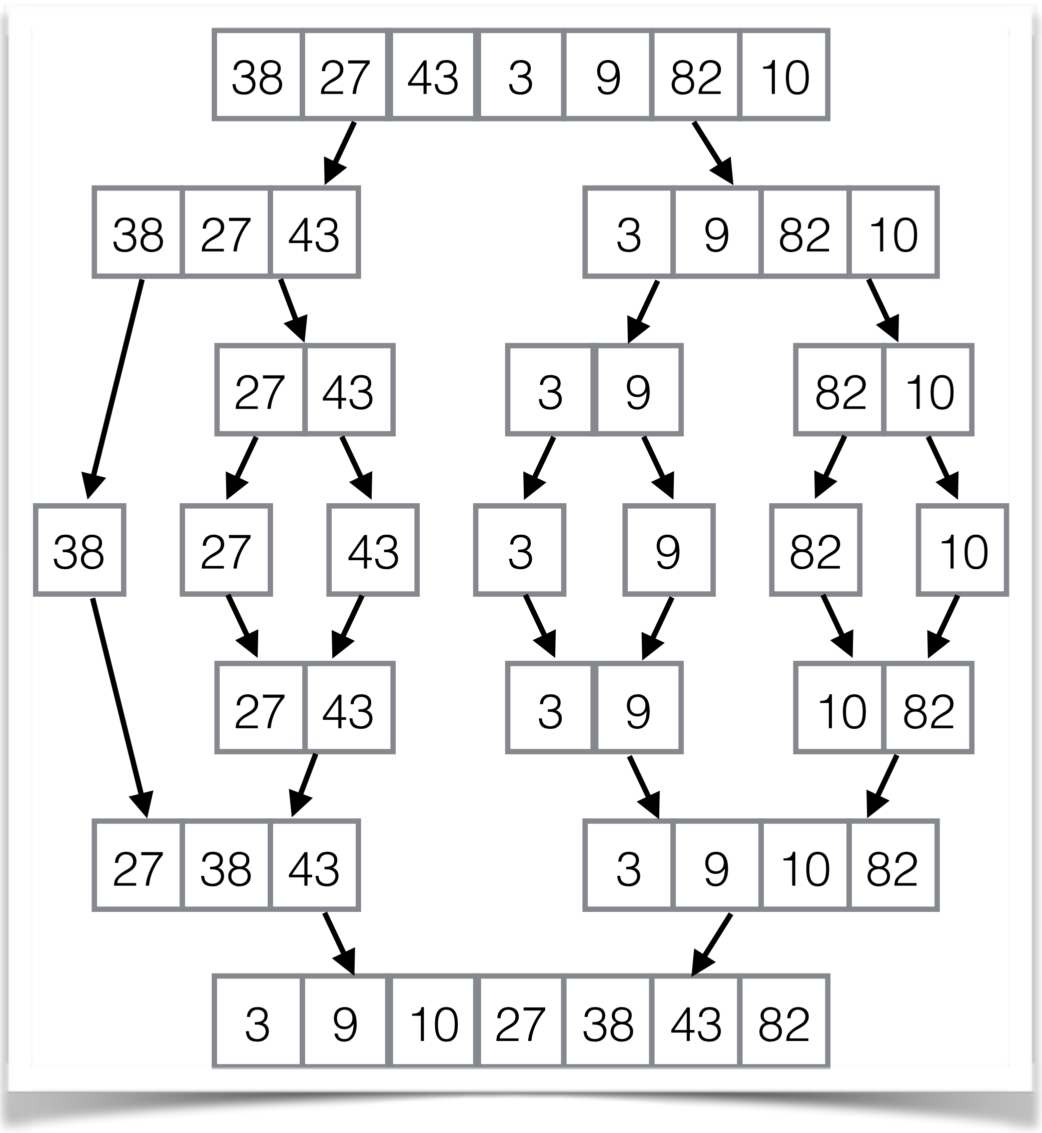
The key part of merge sort is the merge algorithm, which takes two sorted arrays B and C and combines their elements into a single sorted array. The merge algorithm works roughly as follows. It first iterates through the two input arrays, B and C, at each step extracting the smallest remaining item from among them, and appending it to the output array. When the elements in one of the two arrays are exhausted, the algorithm appends the leftover elements from the other array to the output.

The following pseudocode gives the details of the merge algorithm. It 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

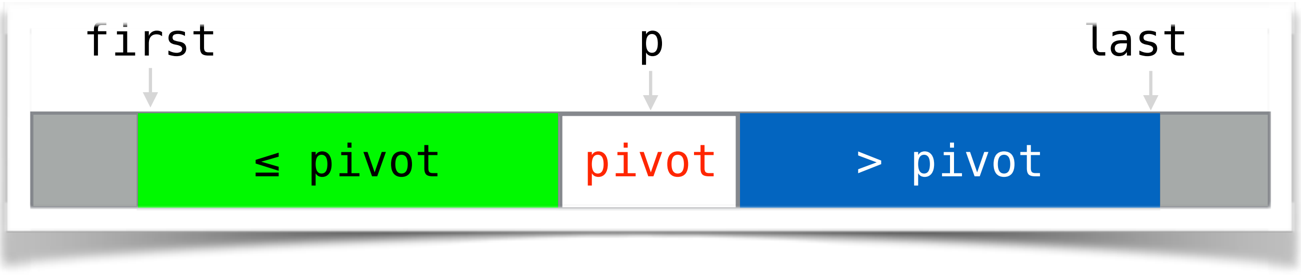
The key component of quicksort is the **`partition`** algorithm. `partition` takes as arguments an array `arr` and two indices `first` and `last` into `arr`. These arguments must satisfy the following.

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

`partition` 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; j < last; j++)
        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
```

Example. The figure on the next page shows the execution of PARTITION on the array

`arr = (4, 9, 1, 5, 2, 3).`

The pivot is shown in red, elements known to be greater than the pivot are blue, elements known to be less than or equal to the pivot are green, and elements that have not yet been considered are grey. Each iteration compares `arr[j]` against the pivot. If `arr[j]` is less than or equal to the pivot, `arr[j]` is swapped with the first blue element. The final step moves the pivot to position `i+1`.

```
i j                 i   j
  4 9 1 5 2 3    ->   4 9 1 5 2 3

i     j               i     j
  4 9 1 5 2 3    ->   1 9 4 5 2 3

  i       j             i       j
  1 9 4 5 2 3    ->   1 2 4 5 9 3

    i       j
  1 2 3 5 9 4
```