Lecture 15: MergeSort

Doina Precup
With many thanks to Prakash Panagaden and Mathieu Blanchette

February 7, 2014

When we started talking about sorting, we discussed an algorithm called Selection Sort, an in-place sorting algorithm (i.e. which does not allocate new memory) whose time complexity is $O(n^2)$. Can we do any better in terms of $O()$? The answer is yes, using an algorithm called MergeSort.

1 Main idea

MergeSort is based on the idea of dividing the problem into independent subproblems, solving the problems separately, and then combining the subproblem solutions to find the overall solution of the algorithm. The algorithm has three parts:

1. **Divide** the array into two halves
2. **Conquer** each half by sorting it recursively
3. **Combine** the results obtained, by merging the two halves to obtain one fully sorted array.

This **divide-and-conquer** idea is very powerful, and as we will see, it is a standard approach used throughout computer science. To illustrate how this works, consider the following array:
5 3 8 2 1 2 7 6
First, we split the array into two parts:
5 3 8 2 | 1 2 7 6
Then we sort recursively each half. If this works, we will have:
2 3 5 8 | 1 2 6 7
Finally, we will merge the two halves. To do this, we will use a temporary array, in which we copy the elements from both halves in order, using two indices ("fingers") to keep track of where we are. The result will be:
1 2 2 3 5 6 7 8
If we want the result in the original array (which is usually the case) we will have to copy it back. Hence, MergeSort is not an in-place algorithm, like selection sort.

2 Pseudocode

The pseudocode of the algorithm is as follows:

```
**Algorithm** mergeSort($a$,$p$,$q$)
**Input:** An array $a$ and two indices $p$ and $q$ between which we want to do the sorting.
**Output:** The array $a$ will be sorted between $p$ and $q$ as a side effect
**if** $p < q$ **then**
    **int** $m \leftarrow \lfloor \frac{p+q}{2} \rfloor$//this is the middle of the part of interest
    mergeSort($a$,$p$,$m$)
    mergeSort($a$,$m+1$,$q$)
    merge($a$,$p$,$m$,$q$)
```

Note that if we want to call the algorithm on an array of size n, the call will be: mergeSort(a,0,$n-1$)

The pseudocode of the merging routine is as follows:

```
**Algorithm** merge($a$,$p$,$m$,$q$)
**Input:** An array $a$, in which we assume that the halves from $p \ldots m$ and $(m+1) \ldots q$ are each sorted
**Output:** The array should be sorted between $p$ and $q$
**Array** $tmp$ of size $q-p+1$ // this array will hold the temporary result
**int** $i \leftarrow p$ //these are the two indices in the two halves of the array
**int** $j \leftarrow m+1$
**int** $k \leftarrow 0$ //this is the index we will use in the tmp array
**while** ($i \leq m$ **or** $j \leq q$) **do**
    **if** ($j = q+1$ **or** $a[i] \leq a[j]$) **then**
        $tmp[k] \leftarrow a[i]$
        $i \leftarrow i+1$
    **else if** ($i = m+1$ **or** $a[i] > a[j]$) **then**
        $tmp[k] \leftarrow a[j]$
        $j \leftarrow j+1$
    $k \leftarrow k+1$
**for** $k = 1$ **to** $q-p+1$ **do**
    $a[k+p-1] \leftarrow tmp[k-1]$
```

3 Tracing the execution on an example

Let us now trace the above example in more detail, based on the pseudocode. The first call is mergeSort(a,0,7), which is on the whole array:

```
 5  3  8  2  1  2  7  6
```

This will cause a recursive call: mergeSort(a,0,3), which works on the array:

```
 5  3  8  2
```

Inside this, there is another recursive call: mergeSort(a,0,1), which works on:

```
5   3
```
Now, the two next recursive calls: mergeSort(a,0,0) and mergeSort(a,1,1) meet the base case, so they will cause no more recursion. So the next step is to call merge(a,0,0,1). This will *swap the two elements*, creating the array:
```
3   5
```
So at this point, after the first merge call has executed, the whole array a is:
```
3   5   8   2   1   2   7   6
```
Similarly, we will have a call to mergeSort(a,2,3), working on:
```
8   2
```
This generates two next recursive calls: mergeSort(a,2,2) and mergeSort(a,3,3) meet the base case, so they will cause no more recursion. So the next step is to call merge(a,2,2,3). This will swap the two elements, creating the array:
```
2   8
```
So at this point, after the first merge call has executed, the whole array a is:
```
3   5   2   8   1   2   7   6
```
At this point, the next call is to a merge routine: merge(a,0,1,3), working on:
```
3   5   2   8
```
We will trace the movement of the indices here. We start with $i = 0$ and $j = 2$, in the positions below:
```
3   5   2   8
i       j
```
The array tmp has size 4 and no content yet (indicated by a dash - below), the index k is on its first position:
```
-   -   -   -
k
```
The if condition indicates that we should copy $a[j]$, and this index would be moved. After this, the positions of the indices in a are:
```
3   5   2   8
i           j
```
The tmp array is as follows:
```
2   -   -   -
    k
```
Next, we will copy $a[i]$, so in the original array we have:
```
3   5   2   8
    i       j
```
and the tmp array is:
```
2   3   -   -
        k
```
Next we will copy 5, so we have:
```
3   5   2   8
        i   j
```
and the tmp array is:
```
2   3   5   -
            k
```
Now, because i has gone out of bounds (it is $m + 1$), we will copy the rest of the elements from the

second part of the array, which gives the following tmp array: 2 3 5 8
Next we will copy this into a, which now looks as follows:
2 3 5 8 1 2 7 6
Note that all the “work” of shifting around array elements is actually done in the merge routine. You can similarly trace the evolution of the algorithm on the second half of the array (left as an exercise).