5SENG001W - Algorithms, Week 4

Dr. Klaus Draeger

February 5, 2021

RECAP

Last week…

- We talked about the relationship between algorithms and data structures
  - Choice of data structure influences complexity of basic operations like indexed access or insertion
  - This affects suitability of algorithms
- We compared sequential data structures (arrays and lists)
- We covered searching on sequential data structures
  - Linear search ($O(n)$ complexity)
  - Binary search ($O(\log n)$ complexity; requires indexed access and **sorted** data)
- This week, we will compare some algorithms for sorting.

Divide and Conquer

- We have encountered the Divide and Conquer strategy last week
  - Binary Search was an example
  - We will see more today
- In a bit more detail, the steps are:
  - If the problem is trivial (like searching in an empty set or sorting a single value), solve it directly. Otherwise:
  - Split the data into smaller parts
    - Ideally of equal size

    Solve the problem on each part
    - Usually recursively
    - In cases like Binary Search (where we only have to consider one part and can ingore the others), this can be transformed into a loop as we saw.

    Combine the partial solutions into a complete solution

The sorting problem

- We will be dealing with the following problem:
  - Input: a sequential data structure
    - We will be working with arrays
    - The data will be integers for simplicity
    - It is always worth thinking about how much of this also works for lists
  - Goal: permute the contents such that they are in increasing order
- We will be using this array as a running example:

```java
int[] values = {91, 32, 92, 13, 73, 14};
```

The sorting problem

- The main questions are:
  - What algorithms exist?
    - Quite a few; we will only look at a small sample
  - What is their time complexity?
  - What is the best we can hope for?
- For the last two, we first define our atomic operations:
  - Pairwise comparison: `if(a[i] < a[j]) { ... }`
  - Swap: `swap(a, i, j)` exchanges `a[i]` and `a[j]`
  - Usually every swap will be preceded by at least one comparison, so the number of comparisons will be the dominating factor.

A simple algorithm: Selection Sort

- In **Selection Sort** the array consists of
  - An unsorted section at the beginning (initially the whole array)
  - A sorted section at the end (initially empty)
- We grow the sorted section in each iteration using these steps:
  1. Find the maximal element of the unsorted section
  2. Move it to the end of the unsorted section; this becomes part of the sorted section
  3. Repeat until the unsorted section is empty.

Selection Sort: example

- We begin with our example array:
  91, 32, 92, 13, 73, 14
- The maximal unsorted element is 92, move it to the end:
  91, 32, 14, 13, 73, **92**
  - We are writing the sorted section in **bold**
  - We moved the element by **swapping** it with the last one.

Selection Sort: example

- The next iterations are:
  91, 32, 14, 13, 73, **92**
- The maximal unsorted element is 91, move it to the end:
  73, 32, 14, 13, **91, 92**
- The maximal unsorted element is 73, move it to the end:
  13, 32, 14, **73, 91, 92**
- The maximal unsorted element is 32, move it to the end:
  13, 14, **32, 73, 91, 92**
- The maximal unsorted element is 14, move it to the end:
  13, **14, 32, 73, 91, 92**
- The maximal unsorted element is 13, move it to the end:
  13, 14, 32, 73, 91, 92

Selection Sort: implementation

```java
public class SelectionSort{
    public static void sort(int[] values){
        int lastUnsorted = values.length - 1; // end of the unsorted section
        while(lastUnsorted > 0){
            // find the maximal unsorted element...
            int maxIndex = 0;               // this will be its index
            int maxValue = values[0];       // and this will be its value
            for(int i=1; i<=lastUnsorted; i++)
                if(values[i] > maxValue){ // new maximal value
                    maxIndex = i;
                    maxValue = values[i];
                }
            // then swap it with the last one, and add it to the sorted section
            values[maxIndex] = values[lastUnsorted];
            values[lastUnsorted] = maxValue;
            lastUnsorted--;
        }
    }
}
```

Selection Sort: analysis

- How many operations does this algorithm perform on an array of size N?
- There are N iterations of the main loop, each involving
  - Finding the maximal unsorted element
  - A single swap
- Finding the maximal element requires $K - 1$ comparisons, where K is the current size of the unsorted section
- The unsorted section shrinks by 1 element each iteration, so $1 + 2 + \ldots + (N-1) = (N^2 - N)/2$ total comparisons
- The Big-O notation ignores lower degree terms (like N) and constant factors (like $1/2$), so this is $\mathbf{O(N^2)}$.

Bubble Sort

- **Bubble Sort** is similar to Selection Sort
- While going through the unsorted section, the highest value found so far "bubbles" up
- Example: starting again with 91, 32, 92, 13, 73, 14.
- Main loop, iteration 1:
  - At first, 91 is the bubble. Compare it with the next element.
    91, 32, 92, 13, 73, 14
  - 91 is greater than 32; swap them, 91 is still the bubble.
    32, **91**, 92, 13, 73, 14
  - 91 is less than 92, so it stays where it is and 92 becomes the next bubble.
    32, 91, **92**, 13, 73, 14
  - 92 is greater than everything else, so it keeps bubbling up.
    32, 91, 13, **92**, 73, 14
    32, 91, 13, 73, **92**, 14
    32, 91, 13, 73, 14, **92**

Bubble Sort

- Iteration 2:
  - **32**, 91, 13, 73, 14, **92**
    32, **91**, 13, 73, 14, **92**
    32, 13, **91**, 73, 14, **92**
    32, 13, 73, **91**, 14, **92**
    32, 13, 73, 14, **91, 92**
- Iteration 3:
  - **32**, 13, 73, 14, **91, 92**
    13, **32**, 73, 14, **91, 92**
    13, 32, **73**, 14, **91, 92**
    13, 32, 14, **73, 91, 92**
- Iteration 4:
  - **13**, 32, 14, **73, 91, 92**
    13, **32**, 14, **73, 91, 92**
    13, 14, **32, 73, 91, 92**

Bubble Sort

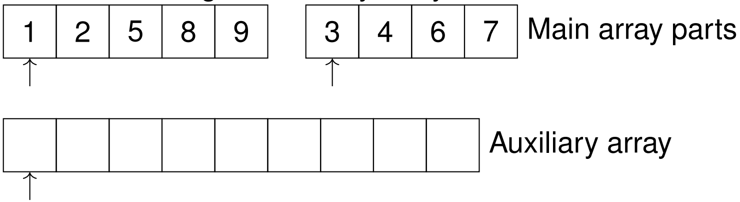
- Implementation of Bubble Sort is an exercise!
- The complexity analysis looks just like for Selection Sort: Each iteration of the main loop
  - shrinks the unsorted region by 1
  - requires $K - 1$ comparisons and up to $K - 1$ swaps, where K is the size of the unsorted region
- So complexity is again $O(N^2)$. Can we do better?

Merge Sort

- Suppose we want to apply the Divide and Conquer strategy to the sorting problem.
- A straightforward translation of the steps is:
  - If there are 0 or 1 values, there is nothing to do. Otherwise:
  - Split the array into two equal halves
Recursively sort each half
Combine the sorted halves by **merging** them
- This is known as **Merge Sort**.

Merge Sort: the merging step

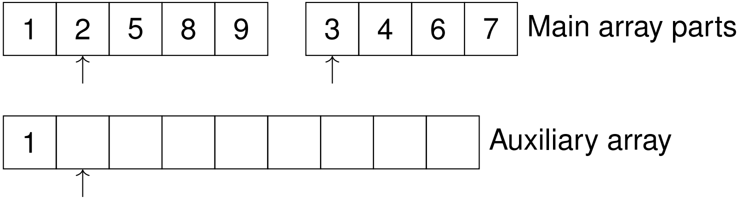
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - **Start at the beginning of both parts**
  - As long as there are unused values in both parts:
    - Compare the two indexed values
    - Copy the smaller one into the temporary array
    - Increase that index
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

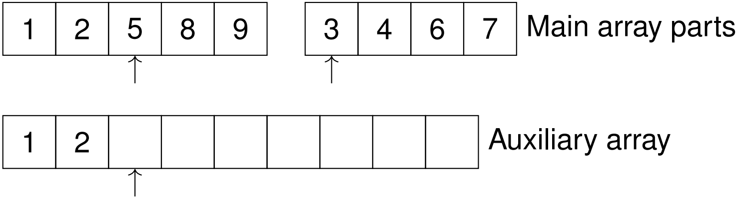
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

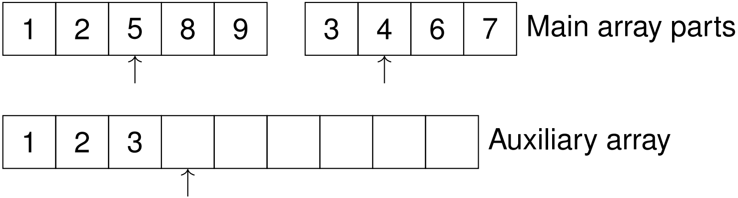
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

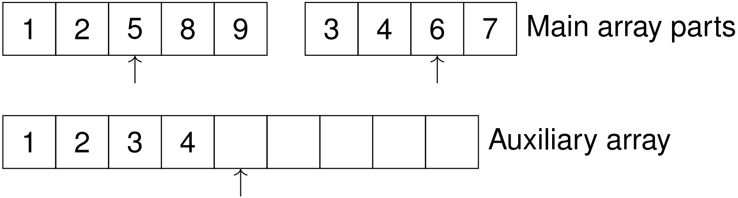
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

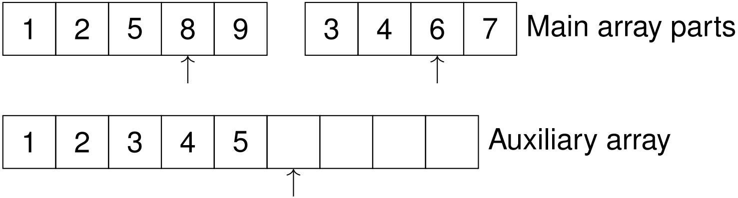
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

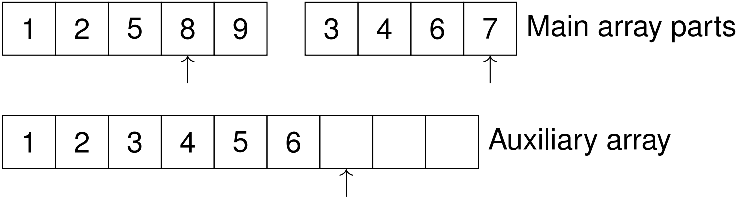
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

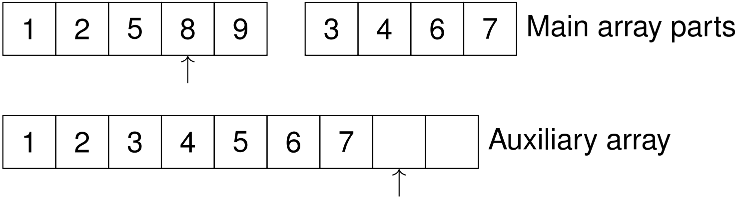
- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:



  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - **Compare the two indexed values**
    - **Copy the smaller one into the temporary array**
    - **Increase that index**
  - Copy the remaining unused values
  - Then copy the sorted values back to the main array

Merge Sort: the merging step

- In the final step of Merge Sort, we need to merge two sorted parts into one.
- We do this using an auxiliary array:

1	2	5	8	9		3	4	6	7	Main array parts

1	2	3	4	5	6	7	8	9	Auxiliary array

  - Start at the beginning of both parts
  - As long as there are unused values in both parts:
    - Compare the two indexed values
    - Copy the smaller one into the temporary array
    - Increase that index
  - **Copy the remaining unused values**
  - Then copy the sorted values back to the main array

Merge Sort: implementation

```java
public class MergeSort{
    // Merge (sorted) ranges values[first]...values[mid] and values[mid+1]...values[last]
    private static void mergeRanges(int[] values, int first, int mid, int last){
        // Exercise!
    }


    // Auxiliary recursive function
    // Sorts the range values[first]...values[last]
    private static void sortRange(int[] values, int first, int last){
        if(last > first){    // Otherwise there is nothing to do (single value)
            int mid = (first + last) / 2;
            sortRange(values, first, mid);      // Recursively sort first half
            sortRange(values, mid + 1, last);   // Recursively sort second half
            mergeRanges(values, first, mid, last); // Merge sorted halves
        }
    }

    public static void sort(int[] values){
        sortRange(values, 0, values.length - 1);
    }
}
```

Merge Sort: analysis

- Finding the complexity of recursive algorithms can be hard
- For many Divide and Conquer algorithms it can be done using the **Master Theorem**
- In the case of Merge Sort on an array of size n, we get:
  - The cost of **merging** two ranges of total size n is in $O(n)$
  - Suppose $T(n)$ is the cost of using Merge Sort.
  - Then $T(n) = 0$ if $n = 1$ (the trivial case)
  - Otherwise, we create 2 sub-ranges of size $\frac{n}{2}$, sort them, and then merge them
    So in this case $T(n) = 2 \cdot T(\frac{n}{2}) + O(n)$
  - By the Master Theorem, in this case $T(n)$ is in $O(n \log n)$.
  - If we simplify things by assuming that the cost of merging is exactly n, you can even directly check that $T(n) = n \log n$ satisfies the equation $T(n) = 2 \cdot T(\frac{n}{2}) + n$ using the fact that $\log(\frac{n}{2}) = \log(n) - 1$

Sorting: analysis

- The complexity of Merge Sort is $O(n \log n)$
- How much better can we get?
- Suppose we have n values, stored in an array **`a`** in a random order. Then
  - There are n possibilities for **`a[0]`**
  - For each of those, there are $n - 1$ possibilities for **`a[1]`**
  - The total number of permutations is given by the **factorial** $n! = n \cdot (n - 1) \cdot \ldots \cdot 2 \cdot 1$
- We must re-order values differently for each permutation
- Each comparison (ideally) cuts their number in half
- We need at least $log(n!)$ comparisons; this is in $\Theta(n \log n)$
- So we cannot do better than $O(n \log n)$