Sorting

Concepts and definitions

Sorting may seem a trivial exercise, but the discipline holds a rather strong position in computer science. In the old days when computers were much slower, it was obviously sensible to spend some time optimizing sorting algorithms with respect to time and space. Nowadays it may not seem so relevant, but since new tasks pushing computers to their limits keep popping up, it remains useful to try to improve the algorithms.

With respect to future groundbreaking innovations in the field of sorting, they seem rather unlikely to occur. The area seems quite well researched. But on the operational level, it makes a lot of sense to select the most appropriate sorting algorithm for the task at hand to the extent that knowledge is available. There is no guarantee that the sorting algorithm that seems the best from a theoretical point of view, also turns out to be so in a practical context. You may choose to use different sorting algorithms depending on the following characteristics:

1. The numbers are almost sorted
2. Big lumps of the numbers are sorted
3. The numbers are completely random
4. The numbers are almost in reverse order

Let's first take a look at what sorting[1] is really about from a theoretical point of view. Take a look at the following list of numbers:

34	8	64	51	32	21

The numbers are clearly unsorted, and let's see if we can measure out to what extent they are unsorted.
To assist us we can define a concept called *inversion*. An inversion is an ordered pair of numbers (i,j), having the property that i < j but a[i] > a[j]. In the above example the following inversions occur:

(34,8) (34,32) (34,21) (64,51) (64,32) (64,21) (51,32) (51,21) (32,21)

That is a total of nine, and it seems fair to assume that the more inversions the list contains, the more unsorted it is, and that the discipline of sorting is about reducing the number of inversions in a list of numbers to zero.

[1] For simplicity reasons it is assumed that sorting only concerns sorting of natural numbers in ascending order.

About the number of inversions in a list (of numbers) the following can be proved:

The average number of inversions in a list of N different numbers is N(N-1)/4.

The rationale can be seen in the following way: Let's look at the total number of ordered pairs in the above list. It must be 5+4+3+2+1, that is 15 or N(N-1)/2. Assuming that half of them are inversions, we end up with N(N-1)/4. A list of 6 elements will on the average contain 7.5 inversions. With 9 inversions our list is relatively unsorted.

This suggests that the number of inversions is of a quadratic 'nature'[2], which could indicate that sorting can be solved in quadratic time, so that we may write sorting algorithms with a time complexity of $O(N^2)$. We must be able to remove one inversion using one operation (or, more likely in constant time O(1)), and this sounds plausible since it can be done by swapping two elements.

Perhaps we can even do it better. Perhaps it could be possible to write an algorithm that could eliminate more than one inversion in constant time, so we might be able to achieve a lower time complexity than $O(N^2)$. If we for example swap 21 and 34 in our list, we have

21	8	64	51	32	34

Which contains the following inversions

(21,8) (64,51) (64,32) (64,34) (51,32) (51,34)

In one operation (constant time) we have reduced the number of inversions from nine to six.
This quite trivial observation combined with a great deal of creativity has made it possible for smarter heads to write sorting algorithms with a time complexity of O(N log N).
That this really means something, also for relatively small amounts of data, is shown by experiments by M.A. Weiss that a sorting of 100.000 numbers with an $O(N^2)$-sort takes about 59 seconds, whereas it can be completed with an O(N log N)-sort in less than 0.6 seconds.

Sorting algorithms

There are quite a few sorting algorithms around, and we will describe some of them in more or less detail.

$O(N^2)$

- SelectionSort
- InsertionSort
- BubbleSort

[2] With the reduction that a division with 4 implicates – a reduction which becomes insignificant for high values of N.

O(N log N)

- HeapSort
- MergeSort
- QuickSort

Sorting algorithms with O(N^2) time complexity

In **SelectionSort** you first find the smallest element in the list and place in the first position using a simple swap-operation[3]. Then you find the second-smallest element and swap it into the second position and so on.

BubbleSort is a SelectionSort *in reverse* meaning that you after one pass have placed the largest element in the last position of the list. Let us take a look at the code.

```
1   public void bubbleSort(int array[])
2   {
3       for (int i = 0; i < array.length - 1; i++)
4       {
5           boolean swapped = false;
6           for (int j = 0; j < array.length – i - 1; j++)
7           {
8               if (array[j] > array[j+1])
9               {
10                  swap(array,j,j+1);
11                  swapped = true;
12              }
13      }
14          if (!swapped)
15              return;
16      }
17  }
```

In the outer for-loop we run through the list N-1 times, and for each pass the relatively largest number has been placed in its proper position by the inner for-loop which compares neighbors and swap them if the left neighbor is larger than the right neighbor. The inner loop starts with N-1

[3]
```
public void swap(int[] array, int a, int b)
{
    int temp = array[a];
    array[a] = array[b];
    array[b] = temp;
}
```

comparisons of neighbors. In the next iteration only N-2, since the largest number is now in place in the N'th element. The third time only N-3 comparisons and so on. As mentioned earlier a swap of neighbors is performed by the method *swap* and implicates the removal of exactly one inversion. The structure of *BubbleSort* is such that for certain lists of numbers it is actually faster than the O(N log N)-sorts. That is the case if the list is almost sorted and ensured by the construction with the *swapped*-variable, which is set to true (line 11), each time two neighbors are swapped. If we have performed the inner loop without having set *swapped* to true, it can only mean that the list is now sorted. Thus the time complexity is close to O(N) for lists with few inversions.

InsertionSort acts as when you sort a deck of playing cards picking up one card at a time. This means that the part of the list you have seen so far is always sorted. InsertionSort is considered to be the most efficient of the $O(N^2)$-algorithms for smaller values of N and also good at almost-sorted lists. The code is brief and compact:

```
 1      public void insertionSort(int array[])
 2      {
 3              int j;
 4              for (int i = 1; i < array.length; i++)
 5              {
 6                      int tmp =  array [i];
 7                      for (j = i; j > 0 && tmp < array[j-1]; j--)
 8                              array[j] = array[j-1]
 9                      array[j] = tmp;
10              }
11  }
```

In the loop starting at line 4, we move forward in the list, and in the inner loop we place the element we have reached, in its, so far, correct position.

Sorting algorithms with O(N log N) time complexity

HeapSort is based on a data structure called a priority queue or *heap* (not to be confused with the heap that is used when executing programs). The priority queue is a bit more advanced than your ordinary queue in that elements are inserted into the queue with a pre-defined priority and executed in the order of the priorities (ascending or descending). The clever part is that the structure can be implemented using a simple array and that the operations *insert* (adding an element) and *deleteMin* (executing or removing the element with the highest priority) belong to O(log N). To create a priority queue can be done using an algorithm that runs in O(N) time.

Thus it seems rather obvious that an algorithm that can create a priority queue and output the elements in a sorted order is O(N log N).

MergeSort is a so-called"divide and conquer" algorithm and is most elegantly implemented as recursive. The idea is that you split the list into halves until you reach a point where you have lists containing just one element, which obviously are sorted. That is the divide part. The conquer part then merges the sorted tables. Divide costs log N operations[4], and since the merge has linear time complexity, we may conclude (without offering actual proof) that MergeSort is O(N log N).

The disadvantage of this algorithm is that it is rather space consuming, since it takes double space – 2N – to perform it, and consequently it is hardly ever used for sorting taking place in memory.

Below is given a draft of MergeSort:

```
 1    private void mergeSort(int a[], int left, int right)
 2    {
 3        if (left < right)
 4        {
 5            int center = (left + right) / 2;
 6            mergeSort(a, left, center);
 7            mergeSort(a, center + 1, right);
 8            merge(a, left, center + 1, right);
 9        }
10    }
```

In merge the two subtables going from left to center and center+1 to right are merged. An external call of MergeSort could be like this:

mergeSort(array);

QuickSort is likewise a divide and conquer algorithm, and just like MergeSort it aims at halving the list, not by size though, but by value. This implicates two things, firstly, you must choose a strategy for choosing the middle value (pivot) and secondly, you cannot be certain that the lists are exactly halved. Let's take a look at a list with an odd number of elements:

31	81	92	43	13	65	57	26	75	49	2

[4] This is of course the base 2 logarithm, so we should actually write $\log_2$ but since other logarithms than base 2 logarithms are rarely used in computer science, they will simply be denoted with *log*.

The perfect pivot by value is 49, but to find it you need to perform a function that at least has a linear time complexity, O(N), and that is probably not worthwhile. Clever heads have found out that if you pick the first, the middle, and the last elements of the list and choose the one with "middle" value, you most likely have a good pivot. In our case it is 31 (middle value of 31, 65 and 2). You the place your pivot in the last element of the list and a pointer at the first (pL) and second-to-last (pR) element, leading to:

2	81	92	43	13	65	57	26	75	49	31
pL									pR	

The pointers are now moved towards one another, and elements are swapped when pL points at an element larger that pivot and pR at an element smaller than pivot. This occurs for the first time here:

2	81	92	43	13	65	57	26	75	49	31
	pL						pR			

When the pointers meet the list looks like this:

2	26	13	43	92	65	57	81	75	49	31
		pL/R								

Our choice of pivot was not particularly fortunate, since the elements were distributed with 30 % and 70 % respectively in the two halves.
The next step will be to swap pivot and the first ellement of the upper list:

2	26	13	31	92	65	57	81	75	49	43

And we now know that the first four elements are smaller than the last seven elements. The process so far is called *partitioning.*

The same exercise is now repeated for each of the sub-tables, and we keep going until the tables are of a size where the algorithm is no longer efficient. This is the case for tables of less than ten elements, where after the rest of the sorting is done by using for example InsertionSort.

This all sounds very cumbersome, but it is a well-documented fact that QuickSort is an extremely efficient sorting algorithm, and for instance clearly faster than HeapSort. This may not seem that obvious since it often happens that sub-tables are far from equal in size, and we must assume that this fact deceases efficiency considerably.

This inconvenience is, however, clearly counterbalanced by the unrivalled efficiency of the *partitioning*, where literally thousands of inversions can be eliminated by one swap.

Similar to MergeSort you can add an extra method, which wraps QuickSort nicely as seen from the calling program.

```
public void quickSort(int[] array)
{
    quickSort(array, 0, array.length – 1);

 }
```

Let us take a look at the code for QuickSort:

```
 1    private void quickSort(int array[], int left, int right)
 2    {
 3            if (left + CUTOFF < right)
 4            {
 5                    int pivot = median3(array, left, right);
 6                    int i = left, j = right -1;
 7                    for ( ; ; )
 8                    {
 9                            while( array[i] < pivot) i++;
10                            while( array[j] > pivot) j--;
11                            if (i < j)
12                                    swap(array, i, j);
13                            else
14                                    break;
15                    }
16                    swap(array, i, right - 1);
17
18                    quickSort(array, left, i - 1);
19                    quickSort(array, i + 1, right);
20            }
21            else
22                    insertionSort(array, left, right);
23    }
```

Remarks:

1. CUTOFF (3) is a class variable that stops the recursive part – recommended value is 10.
2. median3 (5) finds pivot and places it in the last element of the table.
3. Lines 7-15 perform the partitioning (moving pointers and possible swaps).
4. Line 16 puts pivot in its correct position.
5. Lines 18 and 19 are the recursive calls.
6. Line 22: call of simple sorting algorithm for the small tables.

Choosing the right sorting algorithm - best case, average case og worst case

As earlier mentioned it is not irrelevant how you choose your sorting algorithm. We saw how BubbleSort in tables with few inversions comes close to O(N) – an example of *best case*.

We must also, perhaps reluctantly, admit that if choosing pivot goes maximum wrong each time in QuickSort, then it goes $O(N^2)$ – an example of *worst case*. For big values of N this is, however, more unlikely than winning in the lottery, so if you have a rather big table with unknown distribution, QuickSort will always be a fair choice. It can be proven that both average case and best case for QuickSort er O(N log N).

Not all sorting algorithms are equipped with a different worst and/or best case. MergeSort is always O(N log N). We may very well avoid swap operations if the table is pre-sorted, but we still need to do N log N comparisons.

In Windows-applications you might have the need for sorting large amounts of objects/tuples. It is my experience that even if we are talking several thousand elements to be sorted in memory, it gives no significant delays in response times to use $O(N^2)$-algorithms as for instance BubbleSort.

At one point I experienced with generating random numbers to see how they could possibly solve other problems, e.g. the well-known 8-queen problem, but also sorting. I wrote an algorithm, *ShuffleSort*, with a rather peculiar time complexity. Best case is O(N), average case is O(N!) and in worst case it never finishes! But since best case has a probability of far less than a lottery win, and the average case's factorial time complexity means that it takes 20 times longer to sort 20 integers that 19, my product will probably never become a commercial success.

The area of sorting is, as earlier mentioned, rather well-researched. Thousands of research-hours have been spent exploring and developing the area over the course of time. But that does not mean that the area is fully explored. For instance it is commonly recognized that the average case for comparisons in HeapSort is 2N log N-O(N); but it remains undecided if it can be proved, so it may not even be true.

Litterature

This paper is mainly inspired by Mark Allen Weiss. "Data Structures & Algorithm Analysis in Java", 3rd ed., Pearson, 2012.

Ole Dolriis, SDU, October 2016