Every day algorithms

Short cuts to algorithms

Author:
Høgni Beinisson
(hogni@it4.fo)

May 28, 2015

Contents

1 Introduction

These pages are primarily written for myself to better understand algorithms.

I'm starting to like Open Source better and better for each day that goes by, and there for I have decided to host these notes and java programs on github, for everyone to share and use.

All the code is published with the *GNU General Public License* - so are these pages.

Hopefully someone out there will benefit of these few pages, and the sample java code these pages are based on.

I'm not providing any proofs for these algorithms. These proofs can be found in well written text books. These pages simply try to describe some of the most general algorithms in some easy steps.

2 Insertion Sort

Insertion sort works by taking an array of elements - in our case integers - which then are sorted. The sorting algorithm works, by maintaining an sorted list on the left side of an array pointer, and moving the array pointer to the right - one index at the time, while maintaining the sorted list on the left side.

Every time the pointer moves to the right, the element that the pointer points to, is compared to the previous element. If these elements should swap place, they swap. If necessary, the swap is done again and again, until this new element in the left list has found it's final position.

We now have a larger, but still sorted, list on the left side and a smaller unordered list on the right side. When the algorithm is done, the whole list is ordered.

When an element is added to the list, the element is given the position farthest to the right, and then the swap is initiated.

Step 1: [2][5][1][10][4] → [2][5][1][10][4]
Step 2: [2][5][1][10][4] → [2][1][5][10][4]
Step 2: [2][1][5][10][4] → [1][2][5][10][4]
Step 3: [1][2][5][10][4] → [1][2][5][10][4]
Step 4: [1][2][5][10][4] → [1][2][5][4][10]
Step 5: [1][2][5][4][10] → [1][2][4][5][10]
Step 5: [1][2][4][5][10] → [1][2][4][5][10]

The result is the array [1][2][4][5][10]
Note: Insertion sort has a worst-case running time of $\Theta(n^2)$

3 Merge Sort

Merge sort is a divide and conquer algorithm. We divide the initial unsorted array into smaller parts, until we reach a state where we only have sorted arrays. In most cases (and in our code) we simply assume that an array is sorted, when the size of the array is 1.

There for we simply tread each array element as an array. Two and two of the arrays are then merged. When merging two arrays, we simply compare the first elements of both of the arrays. The chosen one is removed from its original array and put in our new result array. When both the input arrays have a size of 0 (zero), we have a new and larger sorted array.

[10] - [5] - [3] - [12] - [6] - [2]
[10] - [5] → [5][10]
[3] - [12] → [3][12]
[6] - [2] → [2][6]

[5][10] - [3][12] - [2][6]
[5][10] - [3][12]
[5]([10]) - [3]([12]) → [3] - [5][10] - [12]
[5]([10]) - [12] → [3][5] - [10] - [12]
[10] - [12] → [3][5][10] - [12]
[12] → [3][5][10][12]

[3][5][10][12] - [2][6]
[3]([5][10][12]) - [2]([6]) → [2] - [3][5][10][12] - [6]
[3]([5][10][12]) - [6] → [2][3] - [5][10][12] - [6]
[5]([10][12]) - [6] → [2][3][5] - [10][12] - [6]
[10][12] - [6] → [2][3][5][6] - [10][12]
[10]([12]) → [2][3][5][6][10] - [12]
[12] → [2][3][5][6][10][12]

The result is the array [2][3][5][6][10][12]
Note: Merge sort has a worst-case time of $\Theta(n \lg n)$

4 Quick Sort

Quick sort is a divide and conquer algorithm. The algorithm is a combination of two parts. A partition method and a recursive call.

In the partition method a pivot is chosen from the array and all the elements in the array that are less or equal to the pivot are placed on the left side of the pivot, while all the elements that are larger or equal to the pivot are placed on the right side of the pivot.

The pivot is in our implementation the first element in the array to be sorted.

The recursive parts then calls the partition method again, but this time with the left side of the array and the right side of the array - omitting the pivot.

When the recursion is done, the array is ordered.

Initial call: Array = [4][10][5][3][12][6][2], p = 0, q = 8
In the first partition call, the result array is:
[2][3][4][10][12][6][5] - Where 4 was the pivot.

The next recursive call will be: Array [2][3][4][10][12][6][5], p = 0, q = 3
The result array is: [2][3][4][10][12][6][5] - Where 2 was the pivot.

The next recursive call will be: Array [2][3][4][10][12][6][5], p = 3, q = 8
The result array is: [2][3][4][5][6][10][12] - Where 10 was the pivot.

The result is the array [2][3][4][5][6][10][12]
Note: Quick sort has a worst-case time of $\Theta(n \lg n)$