in Sort algorithm

Sort algorithms Java implementation

In this post I will show you the Java implementation of the most common sort algorithms with their basic properties like stability, best , average and worst case.

Stability is the property of the search algorithm that guaranties that two elements that are same (A1 and A2) will keep their order after the sort is completed.

For example if we have array like this A1, B1, A2, C1,D1 , after sorting process we wil have A1, A2, B1, C1,D1 in **stable** algorithm and may result A2, A1, B1, C1,D1 in **unstable** one.

There are several types of algorithms:

Simple Algorithms

- Insertion sort
- Selection sort

Efficient Algorithms

- Merge sort
- Heap sort
- Quick sort

Bubble sort and variants

- Bubble
- Shell

Insertion Sort

Insertion sort is simple algorithm, efficient on small number of items or when array is presorted. It is usually used as a part of more advanced algorithms like **shell sort**.

Key properties:

- Stable: YES
- Best case : N, when sorting almost sorted data
- Worst and average case : N^2
- Memory: 1, does not require additional memory

```
1  public class InsertionAlgorithm implements Sort {
2
3      public int[] sort(int[] array) {
4          for (int i = 1; i < array.length; i++) {
5              int currentIndex = i;
6              for (int j = i - 1; j >= 0; j--) {
7                  int temp = array[currentIndex];
8                  int inner = array[j];
9
10                 if (temp < inner) {
11                     array[currentIndex] = array[j];
12                     array[j] = temp;
13                     currentIndex--;
14                 }
15             }
16         }
17
18         return array;
19     }
20
21     static void print(String prefix, int[] array) {
22         System.out.println(prefix + " " + Arrays.toString(array));
23     }
24
25     public static void main(String[] args) {
26         Sort algorithm = new InsertionAlgorithm();
27         print("final", algorithm.sort(new int[] { 8, 1, 6, 4, 3 }));
28     }
29 }
```

We have two loops, one outer loop(indexed by **i**) goes from index 1 to N and inner that goes from i to 0.

Inner loop is the key of the algorithm, it moves **i-th** element back to its correct position or to the start of the array (basically this is the same since this element is the smallest).

```
1 int temp = array[currentIndex];
2 int inner = array[j];
3
4 if (temp < inner) {
5     array[currentIndex] = array[j];
6     array[j] = temp;
7     currentIndex--;
8 }
```

If i-th element is smaller then the j-th element then swap them. Do that until you reach start of the array or element is not smaller. This way element is either the smallest in the array or it is in its correct location.

Observe the **currentIndex–**, every time we swap and the outer loop element moves to the left of the array we need to update the current index so that it points to correct element. Correct element is the original i-th element that got moved left.

Selection sort

Selection sort is in-place comparison sorting algorithm that usually performs worse then insertion sort, but it uses less swaps , so when swapping is a problem this is good solution.

Key properties:

- Stable: NO
- Best case : N^2
- Worst and average case : N^2
- Memory: 1, does not require additional memory

```
1  public class SelectionAlgoritham implements Sort {
2
3      public int[] sort(int[] array) {
4          for (int i = 0; i < array.length - 1; i++) {
5
6              int maxIndex = i;
7              int min = array[i];
8              for (int j = i; j < array.length; j++) {
9                  int minCandidate = array[j];
10                 if (minCandidate < min) {
11                     min = minCandidate;
12                     minIndex = j;
13                 }
14             }
15
16             int temp = array[i];
17             array[i] = array[minIndex];
18             array[minIndex] = temp;
19         }
20         return array;
21     }
22 //main and print methods are removed
23 }
```

We have two loops outer and inner, for every element of the outer loop go the the end of the array and find the minimum. Then swap the outer element with the minimum. Key advantage is that when element jumps 5 places we swap only once.

Merge sort

Merge sort sort by splitting the array into smaller and smaller arrays until we reach atomic level, atomic level is simply sorted and then we go up and finish the sort.

Key properties:

- Stable: YES
- Best case : n log(n)
- Worst and average case : n log(n)
- Memory: N, does require additional memory

Original 8, 1, 6, 4, 3 ,7

Split 1: (8,1,6) (4,3,7)

Split 2 (8, 1) (6) (4,3) (7)

Note that we have reached atomic level, now we order the items

8,1 becomes 1,8

1,8,6 becomes 1,6,8

4,3 becomes 3,4

4,3, 7 becomes 3,4, 7

1,6,8,3,4,7 becomes 1,3,4,6,7,8

```java
public class MergeSortAlgorithm implements Sort {

    public int[] sort(int[] array) {

        int[] tmp = new int[array.length];

        mergeSort(array, tmp, 0, array.length - 1);
        print(array, 0, array.length - 1);
        return null;
    }

    private void print(int[] array, int start, int end) {
        for (int i = start; i <= end; i++) {
            System.out.print(array[i]);
        }
    }

    private void mergeSort(int[] array, int[] tmp, int start, int end) {
                if (end <= start) {
            return;
        }

        int middle = (end + start) / 2;
        mergeSort(array, tmp, start, middle);
        mergeSort(array, tmp, middle + 1, end);

        int i = start;
        int j = middle + 1;

        int resultIdx = 0;

        while (i <= middle && j <= end) {
            if (array[i] < array[j]) {
                tmp[resultIdx] = array[i];

                i++;
            } else {
                tmp[resultIdx] = array[j];
                j++;
            }
            resultIdx++;
        }

        for (int iCopy = i; iCopy <= middle; iCopy++) {
            tmp[resultIdx++] = array[iCopy];
        }
        for (int iCopy = j; iCopy <= end; iCopy++) {
            tmp[resultIdx++] = array[iCopy];
        }

        System.arraycopy(tmp, 0, array, start, (end - start) + 1);
    }

    public static void main(String[] args) {
        Sort algorithm = new MergeSortAlgorithm();
        algorithm.sort(new int[] { 2, 6, 3, 6, 8, 9, 0, 6, 4, 4, 3, 7, 5, 1 });
    }
}
```

Observe the exit of the recursion, here we exit when atomic level is reached and there is only 1 element in the section.

```
1 if (end <= start) {
2     return;
3 }
```

In above example we have 8, 1 of indexes 0 and 1.

int middle = (0+1)/2;// equals 0

mergeSort(array, tmp, 0, 0); //start and end are the same
mergeSort(array, tmp, 0 + 1, 1); //start and end are the same

Heap Sort

Key properties:

- Stable: NO
- Best case : n log(n)
- Worst and average case : n log(n)
- Memory: 1, does not require additional memory

Heap sort uses a tree like structure called heap , first it arranges the array into a heap. For example 8, 1, 6, 4, 3 ,7 is 8 7 6 4 3 1. Observe that we have a array representation of the tree, this means that first element is array[1], his children are array[2] and array[3], children of element two are array[4] and array[5]. You might guess the pattern here children of any element are calculated by

Child 1: index * 2

Child 2: index * 2 +1

This is very important math since it allows us to go up and down the tree.

After the heap is arranged it moves the first element (that is the biggest) and moves it to the end of the array, then it rearranges the tree again as a result we have sorted array.

First phase is executed in the look starting from N/2 to 1, reason for this is that we want to start working from the bottom of the tree up to the top.

```
1 6, 7, 3, 5, 1, 2, 4
```

1. Initially maxSort is called with parameters array, 3, 7. index 3 references the children 2 and 4.
2. while 2 * 3 < 7 do
3. first child has index 6

4. if index is not illegal check if first child is lesser than second child
5. use the bigger child (this is done in j++)
6. if p is > child this means that part satisfies the heap
7. p < child this means that we must move the child up
8. Use next element with index 2 and children 1 and 2 ...

```
private void maxSort(int[] array, int p, int n) {
        while ((2 * p) <= n) {
            int j = 2 * p;

            if (j < n && less(array, j, j + 1))
                j++;

            if (!less(array, p, j))
                break;

            exchange(array, p, j);
            p = j;
        }
    }
```

After the heap is in order we must order the array, this is done using the while loop,

1. We move the top element to the end of the array `exchange(array, 1, N--);`
2. Then we order the heap again putting the max element on the top (observe that we are excluding the area of the array that is sorted.
3. Repeat

```
public class HeapSort implements Sort {

    public int[] sort(int[] array) {
        System.out.println(Arrays.toString(array));
        int N = array.length;

        for (int i = N / 2; i >= 1; i--) {
            maxSort(array, i, N);
        }

        System.out.println(Arrays.toString(array));
        while (N > 1) {
            exchange(array, 1, N--);
            maxSort(array, 1, N);
        }

        return array;
    }

    private void maxSort(int[] array, int p, int n) {
        while ((2 * p) <= n) {
            int j = 2 * p;

            if (j < n && less(array, j, j + 1))
                j++;

            if (!less(array, p, j))
                break;

            exchange(array, p, j);
            p = j;
        }
    }

    private boolean less(int[] array, int i, int j) {
        System.out.println("less " + array[i - 1] + " " + array[j - 1]);
        return array[i - 1] < array[j - 1];
    }

    private void exchange(int[] array, int p, int k) {
        System.out.println("Exchange " + array[p - 1] + " " + array[k - 1]);
```

```
        int temp = array[p - 1];
        array[p - 1] = array[k - 1];
        array[k - 1] = temp;
    }

    public static void main(String[] args) {
        HeapSort heap = new HeapSort();
        System.out.println(Arrays.toString(heap.sort(new int[] { 6, 7, 3, 5, 1, 2, 4 })));

    }
}
```

Quick Sort

Quick sort is sort algorithm that uses partitioning , we will observe the the code later on. Algorithm splits the array into two parts separated by a pivot, then moves all elements in the right side that are greater then the pivot and moves all elements that are smaller to the left side. Again and again it splits the parts into two smaller parts and repeats the arranging until the array is sorted.

Key properties:

- Stable: NO
- Best and average case : n log(n)
- Worst case : n^2
- Memory: 1, does not require additional memory

```
public class QuickSort implements Sort {

    public int[] sort(int[] array) {

        if (array == null || array.length == 0) {
            throw new IllegalArgumentException();
        }
        sort(array, 0, array.length - 1);

        return array;
    }

    private void sort(int[] array, int start, int end) {
        System.out.println(Arrays.toString(array));
        int low = start;
        int high = end;
        int pivot = (start + end) / 2;

        while (low <= high) {
            while (array[low] < array[pivot]) {
                low++;
            }

            while (array[high] > array[pivot]) {
                high--;
            }
            if (low <= high)
                exchange(array, low++, high--);
        }
        if (start < high)
            sort(array, start, high);
        if (low < end)
            sort(array, low, end);

    }

    private void exchange(int[] array, int start, int end) {
```

```
38          int temp = array[start];
39          array[start] = array[end];
40          array[end] = temp;
41      }
42
43      public static void main(String[] args) {
44          Sort sort = new QuickSort();
45          System.out.println(Arrays.toString(sort.sort(new int[] { 8, 1, 6, 4, 3})));
46      }
47
48  }
```

1. Pivot is element with value 6
2. Low is element with value 8
3. High is element with value 3
4. Now we go to while loop and check is 8< 6 -> NO
5. Now we go to next while loop and check if 3>6 -> NO
6. Check if Low < Hight -> exchange 8 and 3, move low to element with value 1 and move the hight to element with value 4
7. First while loop will jump over the element 1
8. Second while loop will stay on element 4
9. Exchange will happen on 4 and 6
10. Result is now 3 1 4 6 8, low is 6, high is 4
11. Next we go to recursion, from 0 to high and from low to end.
12. First recursion is interesting here, 3 1 4
13. Pivot is 1
14. is 3 < then 1 NO
15. is 4 > then 1 YES, High is 1
16. Exchange 3 and 1
17. Result is now 1 3 4 6 8
18. END

Bubble sort

Key properties:

- Stable: YES
- Best case : n
- Worst and average case : n^2
- Memory: 1 , does require additional memory

Sorting algorithm that goes through the list comparing the elements and putting the wrong elements into position. It is called bubble sort because the biggest elements tend to bubble first to the top. It is simple to implement but it is not so efficient.

Using two loops, for each element compare it to the all next elements and if element is bigger swap it. When no swaps are made exit the loop.

```java
public class BubbleSort implements Sort {

    public int[] sort(int[] array) {
        int temp;

        for (int i = 0; i < array.length; i++) {
            boolean swapped = false;
            for (int j = 1; j < array.length; j++) {
                if (array[j - 1] < array[j]) {
                    temp = array[j - 1];
                    array[j - 1] = array[j];
                    array[j] = temp;
                    swapped = true;
                }
            }
            if(!swapped)break;
        }

        return array;
    }
    static void print(String prefix, int[] array) {
        System.out.println(prefix + " " + Arrays.toString(array));
    }

    public static void main(String[] args) {
        BubbleSort algorithm = new BubbleSort();
        print("final", algorithm.sort(new int[] { 8, 1, 3, 4, 6 }));
    }
}
```

Shell sort

Key properties:

- Stable: NO
- Best case : n log n
- Worst case : n log ^2 n
- Average case : n log ^2 n
- Memory: 1 , does require additional memory

Main property of shell sort is that it allows elements to jump from one end of the array to the other easily. This is achieved by sorting the array in big gaps at first and going down to the 1. When we reach size of the step 1 array is already partly sorted and using the insertion sort we are able to quickly sort the array.

Before starting the sort one must decide the gap sequence, this can be tricky since there are multiple gap sequences to be selected where each one has especial properties. In this implementation I have used Knuth's sequence, you can see more about it here.

```java
package ps.pscode.algorithms;

import java.util.Arrays;
import java.util.Date;
import java.util.Random;

import org.junit.rules.Stopwatch;

public class ShellSort implements Sort {

    public int[] sort(int[] array) {

        int interval = 1;
```

```
14          while (interval <= array.length / 3) {
15              interval = (interval * 3) + 1;
16          }
17
18          while (interval > 0) {
19              for (int i = interval; i < array.length; i++) {
20
21                  int currentIndex = i;
22
23                  for (int j = i - interval; j >= 0 && array[j] < array[currentIndex]; j -= int
24                      int temp = array[j];
25                      array[j] = array[currentIndex];
26                      array[currentIndex] = temp;
27
28                      currentIndex -= interval;
29                  }
30              }
31
32              interval = (interval - 1) / 3;
33          }
34
35          return array;
36      }
37
38      public static void main(String[] args) {
39          int[] array = new Random().ints(100).toArray();
40                  new ShellSort().sort(array);
41         }
42  }
```