YASA
Yet Another Sorting Algorithm

Lorenzo Cioni

Department of Computer Science, University of Pisa

e-mail: lcioni@di.unipi.it

keywords: algorithms, data sorting, real time sorting, data structures

Abstract

The paper presents an algorithm for real time sorting of a stream of data. It is based of an abstract data type called **procession** in which the elements are inserted from the head, from the tail and between such extremes but are extracted only from the head when they have been fully collected (and ordered). The paper contains a high level description of the algorithm together with a discussion of its computational complexity and closes with some possible variations.

1 The problem

This paper addresses a very classical problem, that of sorting a stream of n data, with an hopefully new algorithm. We show the algorithm applied to a stream of n integers (on which a total order is defined) that can assume any value but it is easy to generalise it on condition that a comparator[1] is defined on the class to which the data belong[2].

[1] With this term we identify an operator that, given a class of elements, allows the definition on that class of a total ordering among any set of its elements. In case of integers, classical comparators are $<$, $>$, $\leq$, $\geq$. Each of such comparators defines a binary relation on integers, each with its own properties ([Ros99]).

[2] The basic implicit hypothesis is that the n data can assume any value so that we can imagine to have n distinct data to order. The algorithm should work pretty well also in presence of repetitions. We note that if the data are bounded within min and Max values they can assume $l = Max - min + 1$ values so that, if $n > l$, it is impossible to avoid repetitions.

2 The structure of the paper

The paper is structured as follows. In the next section a set of traditional algorithms for sorting is shortly presented together with a discussion of the computational complexity of each ([Dro01], [GT97]). Next we describe the basic idea of $YASA$, present its structure with the use of a pseudo-code and examine its working with an example. Then we face the problem of how we can adjust the middle insertion point. The paper closes with some comments on the computational complexity of the proposed algorithm and a section where some variations are shortly examined.

3 Some classical solutions

In scientific literature the problem of sorting a set of n integers is fully described and many solutions are available that form the basic culture of any person working in the fields of Computer Science, Data Structures and Algorithms.
For our purposes we present here some of the classics ([Dro01], [GT97]) such as: bubble sort, heap sort, merge sort, quick sort, bucket sort and radix sort. We note that all the aforesaid methods but the last two are based on comparisons between the elements of a sequence that can assume any value. In these cases ([GT97]) we have that for sorting a sequence of n elements we need a running time that is $\Omega(nlog\, n)$ in the worst case. As to the last two algorithms we note that they are based on special assumptions on the data to be sorted ([GT97]). **Bucket sort** assumes that the n data are characterised by a key that is an integer in the range $[0, N-1]$. In this case, with the use of a bucket array and without executing comparisons between the data, the algorithm runs in $O(n+N)$ and uses $O(n+N)$ space. Under similar hypotheses **radix sort** sorts lexicographically a sequence of data in $O(d(n+N))$ time, where d represents the number of the keys , each assuming a value in the range $[0, N-1]$. These algorithms, however, represent a particular case owing to the assumption they make on the input data and that does not hold both in the other algorithms and in $YASA$. As to the other algorithms we have:

1. **bubble sort** ([GT97]) executes a series of passes over the sequence of n elements performing pairwise comparisons and possible swaps and executes in $O(n^2)$ in the worst case, provided that accesses and swaps are implemented so to execute in $O(1)$ time;

2. **heap sort** ([GT97]) uses a heap to implement a priority queue and sort a sequence of n elements in $O(nlog\,n)$ time;

3. **merge sort** ([GT97]) is based on the principle of **divide-and-conquer** to sort a sequence S of n elements by splitting it in two subsequences S_1 and S_2, each containing about half of the elements, and recursively sorting them and runs in $O(nlog\,n)$ time in the worst case;

4. also **quick sort** ([GT97]) is based on the principle of **divide-and-conquer** to sort a sequence S of n elements by using an element as a pivot p to split the sequence in three parts L (the elements $i < p$), E (those equal to p) and G (those $i > p$), recursively sort L and G and put back the elements in an ordered sequence as first L then E and after G and all this in $O(n^2)$ time in the worst case.

4 The basic idea of $YASA$

The algorithm is based on a very simple idea. Given a sequence on n integers to be ordered in non decreasing order using the classical comparator[3] $\leq$ we use the first value i_0 as the initial *pivot* and, for $j = 1, \ldots, n-1$, we consider all the other values i_j and try to insert each of them in the proper position as fast as we can.
For that purpose we define three pointers on the newly created ordered sequence[4]: H, to the head of the procession, T, to its tail, and M to its "middle". Initially all such pointers refers to[5] i_0.
Now we can have:

1. if $i_j \leq H()$ we update the pointer H and insert i_j there (i.e. at the head of the procession);

[3]For our purposes we consider also other comparators such as $<$, $>$ and $\geq$ with the usual meanings, in case of integer values.

[4]We nickname such a structure **procession** since the elements enter from both ends and in the amidst but they proceed from the head only when they have been collected and ordered.

[5]We use the notation $H()$, $M()$ and $T()$ to denote the elements located, respectively, at the beginning, in the "middle" and at the end of the procession. With $--H$ and $H--$ we denote an unitary $[pre \mid post]$ decrement of H. In similar ways we define a $[pre \mid post]$ increment and/or decrement of T and of M. Other operations are introduced as needed and must be considered as primitive operations of the data type that we will use to implement the procession. All of them are correctly supposed to execute in $O(1)$ time.

2. if $T() \leq i_j$ we update the pointer T and insert i_j there (i.e. at the tail of the procession);

3. if $H() < i_j < T()$ we have to insert the value i_j somewhere in the middle of the procession, between the two other pointers.

In the last case we take into consideration the third pointer M. For this purpose we define a function distance[6] $d(x, y)$ that returns the distance of x from y and a function $proxy(x, y, z)$ that returns a pointer to x if $d(x, z) < d(y, z)$ otherwise it returns a pointer to y. At this point we have the following three cases.

1. If $i_j = M()$ we insert the value either on the left (ad execute $M--$) or on the right (and execute $M++$)[7] and update the pointer, if it is the case, as it is shown in the next point.

2. If $i_j < M()$ we have to insert i_j in the first part of the procession. The next step is to find the right place so that the ordering is guaranteed. For this purpose we use $proxy(H(), M(), i_j)$ and get either H or M. In the former case we scan the procession from H with a temporary pointer until we find the element after which we have to insert i_j and insert it. In the latter case we work similarly from M but stop when we find the element before (looking at the head of the procession) which we have to insert i_j. At this point we have to decide if we have to move M or not. To do so we can use two counters c_h, to count the number of elements between H and M, and c_t, to count the number of elements between M and T. After each insertion we evaluate $c_h - c_t$. If such a number exceeds the value of a given threshold τ we step M toward H by decrementing it otherwise we do nothing on M. The threshold can be set either statically at a constant value or dynamically at a value that depends on the values inserted up to that point. With a static τ we simply execute[8] $M = M - \lceil d \rceil$, $c_h = c_h - \lceil d \rceil$ and $c_t = c_t + \lfloor d \rfloor$ with $d = \tau/2$. If τ is dynamically defined we act on M in a more complex way, to be shortly described in section 7.

3. If $i_j > M()$ we behave as in the previous case but with respect to T instead of H. In this case M is incremented, if it is the case. We note

[6] In our context we have $d(x, y) =\mid x - y \mid$. In other cases such a function depends on the type of the data to be sorted.

[7] We note that if use a doubly linked list to implement our procession all these operations cost $O(1)$ in time.

[8] With $\lceil a/b \rceil$ we mean the smaller integer greater than or equal to a/b whereas with $\lfloor a/b \rfloor$ we mean the smaller integer lower than or equal to a/b. Of course, if $\tau = 0$ pointer M stays fixed in its initial position.

that we have to check if $c_t - c_h > \tau$ in order to update the position of M.

5 The structure of $YASA$ and its pseudo-code

In this section we present the structure of the algorithm in the base case of a constant threshold and postpone a brief examination of the dynamic case to section 8.
We start with the definition of the abstract data type (ADT) **Procession** with its basic methods and implement it with a structure that allows the execution of the required operations with the required time complexity[9]. Since we need a structure that can be easily and dynamically extended allowing easy (and constant time) insertions at the head, at the tail and in the middle but extractions only from the head (when all the insertions are over) we use a doubly linked list with the aforesaid three pointers, some other auxiliary variables and a set of primitive operations.

```
public abstract class Procession {
//public methods
public abstract Procession(){};
public abstract void insert(int el){};
public abstract int extract();
//private methods
private abstract int distance(int x, int y){};
private abstract ProcessionEl proxy(int x, int y, int z){};
//data structures
private ProcessionEl H, M, T;
private int c_h, c_t; //counters of the elements on the
                      //left and on the right of M
private int tau; //threshold value
}
```

At this level a *Procession* is a set of *ProcessionEl* elements, each with three fields: a field containing the value to be sorted and two pointers, to the previous and next elements, respectively.
Once we have defined the ADT and we have established that we implement it with a doubly linked list we can show the high level structure of the algorithm. We think we have an input stream of integers in, a pointer m through which

[9] In what follows we are going to use a pseudo-Java coding scheme so to make the pseudo-code more easily readable ([Dro01] and [GT97]).

we access the stream and an output stream *out* on which the algorithm puts the ordered elements of the procession.

```
procedure YASA()
{
  P=new Procession(); //(1)
  m=0;
  count=0;
  i=read(in, m);
  while (i !EOF)
  {
    P.insert(i);
    count++;
    i=read(in, ++m);
  }
  for(i=0;i<count;i++)
  {
    write(out, P.extract());
  }
}
```

where $P.insert(i)$ inserts element i in the proper position within the procession and $P.extract()$ pops each element of the procession P from P's head until P is empty. Instruction (1) requires some comments. To understand it we must consider that we are using an algorithm at an abstract level that uses an instance of an ADT. In a low level algorithm, written in real Java, we would use a class[10] $DLLProcession$ that implements the abstract class $Procession$ and all its methods (constructors, methods for both insertion and extraction and pointers management).
Going back to the pseudo-code of $YASA$ we note that:

1. the method $extract()$ can be easily implemented as a method on a doubly linked list since it involves the access to $H()$ and an increment of H after each extraction until procession gets emptied;

2. the method $insert(i)$ uses a certain number of private methods that implement the mathematical operators $d(x, y)$ and $proxy(x, y, z)$, the updating of one of the three pointer and the effective insertion.

As to $insert(i)$ we can devise the following high level algorithm[11]:

[10] Where DLL stands for **Doubly Linked List**.

[11] We note that if we use a doubly linked list to implement procession P the instructions that are executed follow a distinct procedure since we have, first, to create a new element

```
procedure insert(int el)
{
  ProcessionEl pt;
  if(el <= H())
  {
     --H;
     H()=el;
     return;
  }
  if(el >= T())
  {
     ++T;
     T()=el;
     return;
  }
  if(el <= M()) //insert in the higher part of procession
     pt = proxy(H(), M(), el);
  else //insert in the lower part of procession
     pt = proxy(T(), M(), el);
  P.scan_insert(pt, el);
  return;
}
```

where $scan\_insert(pt, el)$ scans the procession P from pt and insert el in the proper position. We note that procedure $P.scan\_insert(pt, el)$ represents the time consuming step of the algorithm but can be easily translated in a low level version on a doubly linked list.

6 One example

We give now one example of a short sequence to which the algorithm is applied.
Let use suppose to have the following short sequence[12]:

$$7, 4, 8, 2, 5, 3, 9 \tag{1}$$

to be inserted in the list with the proper value and then to insert it in the proper position. Instructions such as $--H$ and $H() = el$, therefore, must be translated in the proper instructions on a doubly linked list. The same holds for the other instructions that we use in *procedure insert(int el)*.

[12]For a better readability we use commas to separate the elements of the sequence that can be thought either as being read form the input stream or as being stored in the cells of an array.

The algorithm executes the following steps:

current element	conditions	pointer	procession	complexity
7	empty	H,M,T	7	$O(1)$
4	$4 < 7$	H	$4, 7$	$O(1)$
8	$8 > 7$	T	$4, 7, 8$	$O(1)$
2	$2 < 4$	H	$2, 4, 7, 8$	$O(1)$
5	$5 > 2, 5 < 7$	M	$2, 4, 5, 7, 8$	$O(?)$
3	$3 > 2, 3 < 5$	M	$2, 3, 4, 5, 7, 8$	$O(?)$
9	$9 > 8$	T	$2, 3, 4, 5, 7, 8, 9$	$O(1)$

Table 1: Example 1

In the example shown in Table 1 we have the following steps[13]:

1. when the first element 7 is read in, it is inserted in the empty procession P and the pointers H, M, T point to it;

2. then follows $4 < 7$ and we have $P = 4, 7$ $H \to 4$ and $T = M \to 7$, $c_t = 0$ and $c_h = 1$;

3. then follows $8 > 7$ and we have $P = 4, 7, 8$ $H \to 4$, $M \to 7$ and $T \to 8$, $c_t = 1$ and $c_h = 1$;

4. then follows $2 < 4$ and we have $P = 2, 4, 7, 8$ $H \to 2$, $M \to 7$ and $T \to 8$, $c_t = 1$ and $c_h = 2$;

5. then follows $2 < 5 < 7$ with $proxy(2, 7, 5) = 7$ so the insertion occurs examining the procession from M, we have $P = 2, 4, 5, 7, 8$ $H \to 2$, $M \to 7$ and $T \to 8$, $c_t = 1$ and $c_h = 3$;

6. we have an update of M so that $M \to 5$ and $c_t = 2$ and $c_h = 2$

7. then follows $2 < 3 < 5$ with $proxy(2, 5, 3) = 2$ so the insertion occurs examining the procession from H, we have $P = 2, 3, 4, 5, 7, 8$ $H \to 2$, $M \to 5$ and $T \to 8$, $c_t = 2$ and $c_h = 3$;

8. then follows $9 > 8$ and we have $P = 2, 3, 4, 5, 7, 8, 9$ $H \to 2$, $M \to 5$ and $T \to 9$, $c_t = 3$ and $c_h = 3$.

[13]We remind that with c_h and c_t we define two counters of the elements, respectively, on the left of M and on its right.

At step 5, if we have a static threshold $\tau = 2$, we have a shift of M as it is showed by the following step.
When the algorithm inserts 5 it performs the *proxy* test to define a direction of scanning from M so that it immediately finds the place where such a value must be inserted. Similar considerations holds when the algorithm has to insert 3 (in this case it works on the updated middle pointer M).
When algorithm inserts 5, in this case, the operation costs $O(1)$ and the same holds when it inserts 3. This occurs mainly because the procession is very short and cannot be generalised.

7 Dynamic thresholds

We have already seen how to use a static threshold to keep almost balanced the two halves of the procession. In this section we examine very briefly the use of a dynamic threshold τ so to define a rule for the updating of M after a certain number of insertions. For this purpose we have to define both the amount and the direction of the updating.
At the beginning (we have $i = 0$) τ assumes an initial low value τ_0. After each insertion we can have:

1. $\mid c_h - c_t \mid \leq \tau_i$
2. $\mid c_h - c_t \mid > \tau_i$

In the former case we put:

$$\tau_i = \tau_{i-1} + 1 \tag{2}$$

for $i = 1, \ldots n-2$. In the latter case we have to balance the two halves, reset τ to its initial value τ_0 and restart the process of updating the threshold τ. The balancing requires:

1. the definition of the direction of the updating of M as $sign(c_h - c_t)$;
2. its entity as $\lceil \tau_i/2 \rceil$.

Besides the sizes of the two halves of the procession for the updating of the threshold we can use the **density** of each half that is defined as[14]:

$$\delta_h = \frac{\sum_{pt=H}^{M} pt()}{c_h} \tag{3}$$

[14] According to an already used convention with $pt()$ we denote the value pointed by pt.

$$\delta_t = \frac{\sum_{pt=M}^{T} pt()}{c_h} \tag{4}$$

The basic idea is to use the densities instead of the sizes. Apart from this all the other computations follow analogous lines, mutatis mutandis. Both these variations, however, must be carefully evaluated to see if they can influence positively someway the computation complexity and are not a pure waste of code if not a useless complication.

8 Computational complexity

Termination of the algorithm is out of the question so let us say something about its computational complexity. From a classical perspective we should define its best, average and worst cases. The **best case** occurs whenever the succession to be ordered requires insertions only in H or in T (or also in M). If we implement the procession with a doubly linked list the insertions at T, H and M cost $O(1)$ in time so that, e. g., a sequence of n integers $i_0 \ldots i_{n-1}$ formed by two subsequences[15] i_{2k}, increasing, and i_{2k+1}, decreasing, can be sorted in $O(n)$ time. The same is true also for an already sorted sequence or for a sequence sorted in inverse order.
Time consuming operations are those marked as $O(?)$ in Table 1 so that as a **worst case** we can imagine successions such as the followings[16]: $min, Max, a_2, \ldots, a_{n-1}$ or $Max, min, a_2, \ldots, a_{n-1}$ where the a_i are in any order. In such cases we have $n-2$ insertions in the inner positions of the procession.
If the algorithm succeeds in keeping the two halves of the procession of almost the same size at the l-th step we have[17] l elements in P so that we have $(l-1)/2$ elements on both sides of M. Now we have to insert element $l+1-$th. In the worst cases we start scanning the procession from H to insert it just before M or from M to insert if just after H, the same holds for T and M. In such cases algorithm scans $((l-1)/2)-1$ positions. If we evaluate:

$$\sum_{l=3}^{n-1} \frac{l-3}{2} \tag{5}$$

[15] Obviously $k \in [0, \frac{n-2}{2}]$.

[16] With min and Max we denote, respectively, the smaller and the bigger value of a succession on n integers that, anyway, can assume any value.

[17] We assume l is odd so to simplify notation. The reasoning holds also without such an assumption but notation would be more cluttered with symbols.

we can easily see that we get a complexity of $O(n^2)$, as for quick sort and bubble sort[18].
The **average case** can be estimated by assuming an uniform probability distribution of the input values. In this case after having inserted l values we have, roughly speaking:

$$\frac{l-3}{2} \tag{6}$$

values on each half of the procession (we count l values but $H()$, $M()$ and $T()$ and divide by 2). When we insert the $l+1-$th value we insert it with probability:

$$y = \frac{1}{n}\frac{l-3}{2} \tag{7}$$

in either the first part (between H and M) or the second part (between M and T) of the procession and:

$$x = \frac{n-l-3}{2n} \tag{8}$$

before H and after T[19]. In these extreme cases we have a cost $O(1)$ in time. In the other two cases we can use (7) to evaluate the overall complexity as:

$$\frac{1}{n}\sum_{l=3}^{n-1}\frac{l-3}{2} = \frac{1}{2n}\sum_{l=3}^{n-1}(l-3) = \frac{1}{2n}\sum_{i=0}^{n-4} i = \frac{(n-4)(n-3)}{2n} \tag{9}$$

(with a change of variable $i = l-3$) and so $O(n)$.

9 Variations on the theme

YASA has been designed under the assumption that sorting is performed in real time while input data are read in and so without any preliminary phase of data loading.
If we discard such assumption and let the algorithm pre load the data in $O(n)$ time, the main variations on the theme, yet to be explored, involve mainly the selection of the initial pivot from the sequence of input data, the selection of the next element at each step of the algorithm and the way we move the "middle" pointer M.

[18] We put $l-3 = i$ so that the summation of $i/2$ is from 0 to $n-4$ and so it is equal to $(n-4)(n-3)/4$.

[19] If we fix l and let n go to infinity such probabilities tend, respectively, to 0 and to $1/2$ so that $2x = 1$. If, on the other hand, we fix n and let $l-3$ tend to n we have that x tends to 0 and y to $1/2$ so that $2y = 1$.

Instead of choosing the first element of the sequence as the initial pivot we could use a **randomised quick select** ([GT97]) to select the median value of the given sequence of n elements with an expected running time of $O(n)$[20]. If we make this choice we may think that M stays fixed on that value for the whole execution of the algorithm. Obviously, in this case, we must take care of not inserting the pivot twice in the ordered sequence.
As to the way we pass from one element of the original sequence S to the next we note that any choice we make is bound to cost $O(1)$ time so that it cannot involve any search, with or without comparisons, among the data.
Possible solutions, that can be easily implemented, include: the scanning of S from the end toward the beginning; the scanning of S first on the even numbered subsequence i_{2k} and then on the odd numbered subsequence i_{2k+1} or vice versa; the scanning of S from the borders to the centre according to the following succession i_0, i_{n-1}, i_1, i_{n-2} and so on; the scanning of S from the centre to the borders according to the following succession[21] $i_{n/2}$, $i_{n/2+1}$, $i_{n/2-1}$, $i_{n/2+2}$, $i_{n/2-2}$ and so on.
An interesting point to address is to understand if such variations reflect in some positive way on the computational complexity of the algorithm or if they are simply a waste of code.
As to the third point we have already made some comments in the past sections of the paper. Here we only note that a preliminary examination of the data, on condition that it costs $O(n)$ in time, can be useful to understand how data are distributed and so how the threshold τ can be appropriately updated.

References

[Dro01] Adam Drozdek. *Algoritmi e strutture dati in Java.* Apogeo, 2001. Italian version of "Data Structures and Algorithms in Java", Brooks/Cole 2001.

[GT97] Michael T. Goodrich and Roberto Tamassia. *Data Structures and Algorithms in Java.* John Wiley & Sons, 1997.

[Ros99] Kenneth H. Rosen. *Discrete Mathematics and Its Applications.* WCB/McGraw-Hill, 1999.

[20] We remind that the median value of a sequence is an element such that half of the other values are smaller and the remaining half are higher ([GT97]).

[21] This if n is even. It is easy to modify the succession if n is odd.