A linear time algorithm for partial sorting

James A. Riechel

January 23, 2009

Abstract

A linear time algorithm for partial sorting is presented which is appropriate for partially sorting long lists where sorting is computationally prohibitive, or which is appropriate for real-time applications where sorting is impossible. In a list of length *n*, whether sorted, partially sorted, or not, there are on the order of $O(n^2)$ possible inversions. In the algorithm presented, the number of inversions is significantly reduced – on the order of $O(n^2)$ – but, in the general case, the number of inversions is still on the order of $O(n^2)$. However, the ratio of inversions in completely random data to inversions in partially sorted completely random data, appears constant.

1 Algorithm

The partial sorting algorithm presented consists of three functions: *classified_sort*, *classified_swap*, and *classified_merge*. All three functions are given below in C or C++ code.

1.1 *classified_sort*

To partially sort a list of length *n*, *data[0]*, ..., *data[n − 1]*, a call to *classified_sort(n, data)* is made. Obviously, *classified_sort()* first calls *classified_swap()*, followed by *classified_merge()*. classified_sort(n, data) assumes *n* is even, *n/2* is even, and *n/4* is even.

```
void classified_sort(int n, int* data) {
  // assume n is even, (n / 2) is even, and (n / 4) is even.
  classified_swap(n, data);
  classified_merge(n, data);
}
```

Figure 1: *classified_sort*

1.2 classified_swap

The author considers the swaps performed by *classified_swap()* to be classified. To perform classified swaps on a list of length *n*, *data[0]*, ..., *data[n − 1]*, a call to *classified_swap(n, data)* is made. *classified_swap(n, data)* makes on the order of *O(n)* insertion sorts on lists of length four (*4*), which amounts to any number of swaps between zero (*0*) (on already sorted data) to *3n/4* swaps. The insertion sort algorithm can be found in any computer science textbook on algorithms. *classified_swap(n, data)* assumes *n is even*, *n/2 is even*, and *n/4* is even.

```
void classified_swap(int n, int* data) {
  // assume n is even, (n / 2) is even, and (n / 4) is even
  for (int i = 0; i < (n / 4); i++) {
    int temp[4];
    temp[0] = data[i];
    temp[1] = data[n / 2 - i - 1];
    temp[2] = data[i + n / 2];
    temp[3] = data[n - i - 1];
    insertion_sort(4, temp);
    data[i] = temp[0];
    data[n / 2 - i - 1] = temp[1];
    data[i + n / 2] = temp[2];
    data[n - i - 1] = temp[3];
  }
}
```

Figure 2: *classified_swap*()

1.3 *classified_merge*

classified_merge() merges the first half of a list of length *n*, *data[0]*, ..., *data[n/2 − 1]*, with the second half of the same list, *data[n/2]*, ..., *data[n − 1]*. To do this, a call to *classified_merge(n, data)* is made. *classified_merge(n, data)* calls the standard computer science merge algorithm to merge two lists together, in this case, the first half and the second half of the same list. The merge algorithm can be found in any computer science textbook on algorithms. *classified_merge(n, data)* assumes *n* is even, and *n/2* is even.

```
void classified_merge(int n, int* data) {
   // assume n is even, and (n / 2) is even
   merge(n / 2, data, n / 2, &(data[n / 2]));
}
```

Figure 3: *classified_merge()*

2 Time complexity

The partial sorting algorithm presented is a linear time algorithm. In other words, it requires on the order of *O(n)* operations to partially sort a list of length *n*.

2.1 *classified_swap*

classified_swap() is also a linear time algorithm. It requires on the order of *O(n)* operations to make a call to *classified_swap(n, data)*. There are *n/4* iterations of the for loop in *classified_swap()*. Inside the for loop, a constant, or *O(1)*, amount of work is done. Note that insertion sort on a list of length four (*4*) does a constant amount of work, since insertion sort is an $O(n^2)$ algorithm, and for $n = 4$, the amount of

work done is $O(n^2) = O(4^2) = O(16) = O(1)$. Therefore, as mentioned before, *classified_swap(n, data)* is a linear time algorithm which requires on the order of *O(n)* operations to complete.

2.2 *classified_merge*

classified_merge(n, data) is a linear time algorithm which requires on the order of *O(n)* operations to complete. *classified_merge(n, data)* calls *merge(n/2, data, n/2, &(data[n/2]))*, merge being the standard computer science merge algorithm, which is also a linear time algorithm requiring on the order of *O(n)* operations to complete. Therefore, *classified_merge(n, data)* is a linear time algorithm requiring on the order of *O(n)* operations to complete.

2.3 *classified_sort*

classified_sort(n, data) calls *classified_swap(n, data)*, and *classified_merge(n, data)*. Both *classified_swap(n, data)*, and *classified_merge(n, data)* do a linear amount of work, or require on the order of *O(n)* operations to complete. Therefore, *classified_sort(n, data)* requires a linear amount of time to complete, or on the order of *O(n)* operations to complete.

3 Special cases

There are two special cases that should be mentioned.

3.1 Already sorted data

If the partial sorting algorithm is given a list which is already sorted, the partial sorting algorithm maintains the sorted order. For example, if *data[0], ..., data[n − 1]* is:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

the partial sorting algorithm maintains the sorted order, and the output of the algorithm, or *data[0], ..., data[n − 1]* is:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

3.2 Reverse sorted data

If the partial sorting algorithm is given a list in reverse sorted order, the partial sorting algorithm completely sorts the list! For example, if *data[0], ..., data[n−1]* is:

12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

the partial sorting algorithm completely sorts the list, and the output of the algorithm, or *data[0], ..., data[n − 1]* is:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

4 Inversions

The maximum number of inversions in a list of length n occurs when the list is in reverse sorted order, and is given by $n^2/2 - n/2$. If $f(n)$ is the number of inversions in completely random data, and $g(n)$ is the number of inversions in the same completely random data but which has been partially sorted, $g(n) / f(n)$ appears constant. Also $f(n) \epsilon O(n^2)$, $g(n) \epsilon O(n^2)$, and $[f(n) - g(n)] \epsilon O(n^2)$.

5 Search

In my next paper, "A recursive binary search algorithm for partially sorted data" (2009), I present an efficient algorithm for searching partially sorted data.

6 Conclusion / summary

An efficient linear time algorithm for partial sorting has been presented. It is appropriate for use on huge lists where sorting is impossible, and is appropriate for real-time applications where sorting is also impossible.