

Sorting Algorithms

Joseph Randall Hunt
Bobby Wertman
Western Carolina University,
Cullowhee, North Carolina

December 3, 2011

1 Project Proposal

1.1 Goals

1.2 Plan

2 Sorts

2.1 Merge Sort

2.1.1 Algorithm

Merge sort is a comparison-based sorting algorithm, based on the divide-and-conquer design. Its average and worst cases are both $n\log(n)$, and its best case is $\Omega(n)$. Invented in 1945 by John von Neumann, it exploits the fact that combining two lists of sorted data is a linear-time process. [1]

2.1.2 Efficiency

The actual code for the algorithm is available in **Listing 1**. This particular implementation uses an optimization that switches to insertion sort on small arrays. This speeds up the algorithm because it allows the small data set to fit entirely in cache along with the small amount of code associated with the insertion sort algorithm. See **Listing 2** for a parallel implementation of merge sort.

2.1.3 Applications

Merge sort is useful in applications where the data set will not fit entirely into memory. This allows for the data to be read in from disk and sorted as it is read, thus requiring a very small memory footprint. In addition, when time complexity needs to be guaranteed, merge sort is preferable over quicksort, as quicksort's worst case is $O(n^2)$.

2.2 Quick Sort

2.2.1 Algorithm

2.2.2 Efficiency

2.2.3 Applications

2.3 Shell Sort

2.3.1 Algorithm

2.3.2 Efficiency

2.3.3 Applications

2.4 Comparing and Contrasting

3 Conclusions

A Code Examples

Listing 1: Go implementation of MergeSort

```
1 package main
2
3 var optimize int = 10
4
5 func Merge(arr []int) {
6     if len(arr) < optimize && len(arr) != 1 {
7         InsertionSort(arr)
8     } else if len(arr) > 1 {
9         // Start at the midpoint
10        size := len(arr) / 2
11
12        left := arr[:size]
13        right := arr[size:]
14
15        // Perform sub-merges
16        Merge(left)
17        Merge(right)
18
19        // For holding results
20        data := make([]int, len(arr))
21
22        i := 0
23        for len(left) > 0 && len(right) > 0 {
24            // Decide which list to pull from
25            if left[0] < right[0] {
26                // Put the first value into the results and shrink the array
27                data[i], right = right[0], right[1:]
28            } else {
29                // Same thing for the left
30                data[i], left = left[0], left[1:]
31            }
32        }
33    }
34}
```

```

32     i++ // Next element
33 }
34
35 // Figure out which list isn't empty
36 var remainder []int
37
38 if len(left) > 0 {
39     remainder = left
40 } else {
41     remainder = right
42 }
43
44 // Empty remaining values
45 for len(remainder) > 0 {
46     data[i], remainder = remainder[0], remainder[1:]
47     i++
48 }
49
50 // Copy result to our answer
51 for i := 0; i < len(arr); i++ {
52     arr[i] = data[i]
53 }
54 }
55 }
56
57 func InsertionSort(arr []int) {
58     for i, j := 1, 1; i < len(arr); i, j = i+1, i+1 {
59         for j > 0 && arr[j] < arr[j-1] {
60             arr[j-1], arr[j] = arr[j], arr[j-1]
61             j--
62         }
63     }
64 }

```

Listing 2: Parallel implementation of MergeSort in Go

```

1 package main
2
3 import "sync"
4
5 var optimize int = 25
6
7 func Merge(arr []int, group *sync.WaitGroup) {
8     if len(arr) < optimize && len(arr) != 1 {
9         InsertionSort(arr)
10    } else if len(arr) > 1 {
11        // Start at the midpoint
12        size := len(arr) / 2
13
14        left := arr[:size]
15        right := arr[size:]
16
17        // Perform sub-merges
18        group := new(sync.WaitGroup)
19        group.Add(2)
20        go Merge(left, group)
21        go Merge(right, group)

```

```

22     group.Wait()
23
24     // For holding results
25     data := make([]int, len(arr))
26
27     i := 0
28     for len(left) > 0 && len(right) > 0 {
29         // Decide which list to pull from
30         if left[0] > right[0] {
31             // Put the first value into the results and shrink the array
32             data[i], right = right[0], right[1:]
33         } else {
34             // Same thing for the left
35             data[i], left = left[0], left[1:]
36         }
37         i++ // Next element
38     }
39
40     // Figure out which list isn't empty
41     var remainder []int
42
43     if len(left) > 0 {
44         remainder = left
45     } else {
46         remainder = right
47     }
48
49     // Empty remaining values
50     for len(remainder) > 0 {
51         data[i], remainder = remainder[0], remainder[1:]
52         i++
53     }
54
55     // Copy result to our answer
56     for i := 0; i < len(arr); i++ {
57         arr[i] = data[i]
58     }
59 }
60 group.Done()
61 }
62
63 func InsertionSort(arr []int) {
64     for i, j := 1, 1; i < len(arr); i, j = i+1, i+1 {
65         for j > 0 && arr[j] < arr[j-1] {
66             arr[j-1], arr[j] = arr[j], arr[j-1]
67             j--
68         }
69     }
70 }

```

References

- [1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 2nd edition,

2001.