Sorting Algorithms

Joseph Randall Hunt
Tyler McKinney
Bobby Wertman
Marco Anton
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 $nlog(n)$, 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. [**?**]

2.1.2 Efficiency

Merge sort's complexity is $nlog(n)$ for all cases, but the number of computations performed changes between best and worst cases. In the all cases, Merge Sort performs $log(n)$ splits of n elements, putting the efficiency of this phase at $nlog(n)$. However, when merging, the number of operations varies based on the data.

Worst Case In the worst case, the input data is interleaved in such a way that for each step of the merging process, both lists of elements are traversed in full before the merge is complete. This results in n comparisons over $log(n)$ levels (complexity $nlog(n)$, and brings the total number of operations for the sort to $2nlog(n)$, so the efficiency class is $O(nlog(n))$.

Best Case In the best case, the input data is already in order, allowing the merge operations to traverse only the first list, ignoring the second. This cuts the number of operations in half compared to the worst case, so it performs $\frac{n}{2}$ comparisons on $log(n)$ levels, which is $\frac{nlog(n)}{2}$ operations for the merge phase and $\frac{3nlog(n)}{2}$ for the sort, giving us the complexity of $\Omega(nlog(n))$. Supposedly, a variation of Merge Sort exists with efficiency class $O(n)$ for already sorted data.

Code The actual code for the algorithm is available in **Listing ??**. 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 ??** 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

Quicksort is a fast algorithm, developed by Tony Hoare, that has a best and average running time of $O(nlog(n))$ and a worst-case of $O(n^2)$.
Quicksort first divides a large list into two smaller lists with respect to a "pivot" (a pivot is a random item selected from the list). The first list contains items that are smaller and the second list contains items that are larger than the pivot. Finally, apply quicksort to the smaller and larger lists and return the ordered list. According to Weiss [**?**] the basic algorithm to sort an array S consist of the following four steps:

1. If the number of elements in S is 0 or 1, then return.
2. Pick any element v in S. This is called the **pivot**.
3. **Partition** $S - \{v\}$ (the remaining elements in S) into two disjoint groups: $S_1 = \{x \in S - \{v\} | x \leq v\}$, and $S_2 = \{x \in S - \{v\} | x \geq v\}$.
4. Return $\{\text{quicksort}(S_1)$ followed by v followed by $\text{quicksort}(S_2)\}$

2.2.2 Efficiency

Like mergesort, quicksort is recursive; therefore, its analysis requires solving a recurrence formula.

- *Worst Case Analysis*
  To set the recurrence relation on this case, one assumes that the pivot that is chosen is always

the smallest in the list. Therefore, there is only going to be one list (the list containing the largest elements). The recurrence relation on the worst case is:

$$T(1) = 1$$

$$T(n) = T(n-1) + Cn,\ n > 1$$

Where $T(n-1)$ are the recursions and $c*n$ is the linear time spent in the partition (selecting a pivot); moreover, in the base case, we just sort 0 or 1 item, we can do this in constant time. To solve this recursion, the method of Backward substitution is used:

$$\begin{array}{lll} & & substitute: T(n-1) = T(n-2) + C(n-1) \\ T(n-2) + C(n-1) + Cn & ; & substitute: T(n-2) = T(n-3) + C(n-2) \\ T(n-3)) + C(n-2) + C(n-1) + Cn & ; & substitute: T(n-3) = T(n-4) \\ \vdots & & \end{array}$$

$$=> T(n-(n-1)) = C(n-n) + C(n-(n-1)) + ... + Cn$$

$$= \sum_{i=2}^{n} Ci = C\left(\frac{n(n+1)}{2}\right)$$ Therefore, in this case quicksort is of $O(n^2)$

- *Best Case Analysis*
  For the best case analysis we have to choose the pivot such that it is always in the middle. Therefore our new recurrence relation is:

$$T(n) = 2T(n/2) + Cn$$

$$T(1) = 1$$

  Note that the recursion went from $T(n-1)$ to $T(n/2)$. This is because of the selection of the pivot; the pivot has been selected such that the list is divided into two equal and smaller lists. These smaller lists are performed recursively (this is the reason for the 2 in front of the recurence relation) until $T(1)$ where we only return the element.
  This recurrence relation could be solved using the "Master Theorem" [**?**].

$$a = 2; b = 2; d = 1$$

$$\text{CASE II: } a = b^d => 2 = 2^1$$

  Therefore the efficiency class in this case is: $\Theta(n^d logn) = \Theta(nlogn)$

As analysed before, the pivot can be chosen in many ways; however, this might make the quicksort algorithm to perform poorly. As stated in the *Best Case Analysis*, in order for quicksort to perform better, one has to select a pivot to be in the center of the array, but this might be an issue when we have a list of odd elements. An efficient way of choosing the pivot is choosing the element of intermediate value between three elements in the list, more specifically, the three elements chosen from the list have to be the first, last and middle possition (middle element would be $\lceil N/2 \rceil$ where N = number of elements in a list). After selecting them, they need to be sorted and the pivot selected would be the middle element. This method is called *Median-of-Three partitioning*. Not

only this method will help choose a good pivot, but also, will help reduce the number of elements to check. The elments in the first and last position are in the correct side of their corresponding lists; therefore, they don't have to be checked when performing quicksort.

- QuickSort Example.
  The following list is Sorted using the quicksort algorithm:

$$S = 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5$$

  - Find "Pivot"
    Left index = 0; right index = 10; center = 5
    Sort left, right and center (compare the elements of these indexes and put the largest on the right index, and the smallest on the left index).
    The pivot is the element in position "center". For the next step, one needs to put the pivot on position $(n-1)$ of the array (n is the size of the array). This yields the following array 31415326545(5)9. The item between parentheses is the pivot.

  - Partition array into two groups: Set i to 1 and j to $n-2$ (note that left and right items are on the correct side of the array).
    Transverse through the array until i is greater or equal to the pivot and j is smaller or equal to the pivot; in this case until $i = 4$ and $j = 8$, and swap the items in i and j. This produces the following array: 314154265(5)9
    On the next iteration i and j have crossed. No swap is performed; however, we need to restore the pivot to its correct position (i).
    Pivot in correct position: 3141532(5)569
    After this step, it is known that the pivot is sorted (smaller elements are on left of pivot and bigger elements are on right of pivot).

  - Quicksort the remaining two groups
    We now proceed to sort elements to the left of the pivot (smaller elements) and elements to the right of the pivot (bigger elements).

3	1	4	1	5	3	2	(5)	5	6	9
3	1	4	(1)	5	3	2	|5|	5	6	9
1	1	4	(3)	5	3	2	|5|	5	6	9
1	1	4	(2)	5	3	3	|5|	5	6	9
1	1	4	3	5	(2)	3	|5|	5	6	9
1	1	|2|	3	5	4	3	|5|	5	6	9
|1|	|1|	|2|	3	5	4	3	|5|	5	6	9
|1|	|1|	|2|	3	(5)	4	3	|5|	5	6	9
|1|	|1|	|2|	3	(3)	4	5	|5|	5	6	9
|1|	|1|	|2|	3	4	(3)	5	|5|	5	6	9
|1|	|1|	|2|	|3|	|3|	|4|	|5|	|5|	|5|	|6|	|9|

2.2.3 Applications

2.3 Shell Sort

2.3.1 Algorithm

The Shell Sort was developed by Donald Shell in 1959. Formally caled Shell's Sort, the algorithm was the first sorting method to break the quadratic time, however this was not proven unitil years later. The Shell Sort is made faster my comparing elements at a distance rather than onl comparing elements side by side. This sorting method has several pros and cons [**?**]. Mainly, Shell Sort is the fastest N^2 based sort,however, its computational complexity can be difficult to calculate [**?**]. Shell Sort is reasonably easy to implement and also easy to comprehend. Donald Shell's sorting algorithm can be relativly easy to visualize.

1. Take the first element, h_0.
2. Compare it to the $h_{k_{th}}$ element. For example, if the gap is 5, compare to the 5^{th} element.
3. If the $h_{k_{th}}$ is smaller, switch the elements.
4. Advance the first element by the gap and compare to the next element one gap ahead.
5. Repeat the above, and, if the element one gap BELOW the first element is more, switch the elements and repeat.

2.3.2 Efficiency

The Shell Sort can sort an array of elements faster than the insertion sort by comparing elements across a gap as opposed to only comparing adjacent elements. This gap is reduced and the set is compared repeatedly until the gap equals 1. At this point the set can be sorted with a simple, and fast, insertion sort. It is because of this that the Shell Sort is known as the "diminishing increment sort " [**?**]. This gap is also the reason that the computational complexity is difficult to find, it varies based on the gap. Several mathmeticians have attempted to find the best "gap sequence ". With the discovery of the algorithm, Donald Shell used to floor function of $gap/2$. This is also the gap sequence used in the implementation for this project. The worst case complexity for this gap sequence is $\Theta(N/2^k)$. Vaughan Pratt, who developed "Pratt's theorem "for tree spanning developed his own gap sequence beginning with $h = \{1, 3, 7, 15, 31, 63, 127, 255, 511\}$. Pratt's gap sequence has a running time of $\Theta(N^{3/2})$. Several others have their own gap sequence and experts in the mathematical field continue to look for the fastest gap sequence [**?**].

2.3.3 Applications

2.4 Comparing and Contrasting

3 Conclusions

A Code Examples

Listing 1: Go implementation of MergeSort

```go
package main

var optimize int = 10

func Merge(arr []int) {
  if len(arr) < optimize && len(arr) != 1 {
    InsertionSort(arr)
  } else if len(arr) > 1 {
    // Start at the midpoint
    size := len(arr) / 2

    left  := arr[:size]
    right := arr[size:]

    // Perform sub-merges
    Merge(left)
    Merge(right)

    // For holding results
    data := make([]int, len(arr))

    i := 0
    for len(left) > 0 && len(right) > 0 {
      // Decide which list to pull from
      if left[0] > right[0] {
        // Put the first value into the results and shrink the array
        data[i], right = right[0], right[1:]
      } else {
        // Same thing for the left
        data[i], left = left[0], left[1:]
      }
      i++ // Next element
    }

    // Figure out which list isn't empty
    var remainder []int

    if len(left) > 0 {
      remainder = left
    } else {
      remainder = right
    }

    // Empty remaining values
    for len(remainder) > 0 {
      data[i], remainder = remainder[0], remainder[1:]
      i++
    }

    // Copy result to our answer
    for i := 0; i < len(arr); i++ {
      arr[i] = data[i]
    }
  }
}

func InsertionSort(arr []int) {
```

```
  for i, j := 1, 1; i < len(arr); i, j = i+1, i+1 {
    for j > 0 && arr[j] < arr[j-1] {
      arr[j-1], arr[j] = arr[j], arr[j-1]
      j--
    }
  }
}
```

Listing 2: Parallel implementation of MergeSort in Go

```
package main

import "sync"

var optimize int = 25

func Merge(arr []int, group *sync.WaitGroup) {
  if len(arr) < optimize && len(arr) != 1 {
    InsertionSort(arr)
  } else if len(arr) > 1 {
    // Start at the midpoint
    size := len(arr) / 2

    left  := arr[:size]
    right := arr[size:]

    // Perform sub-merges
    group := new(sync.WaitGroup)
    group.Add(2)
      go Merge(left,  group)
      go Merge(right, group)
    group.Wait()

    // For holding results
    data := make([]int, len(arr))

    i := 0
    for len(left) > 0 && len(right) > 0 {
      // Decide which list to pull from
      if left[0] > right[0] {
        // Put the first value into the results and shrink the array
        data[i], right = right[0], right[1:]
      } else {
        // Same thing for the left
        data[i], left = left[0], left[1:]
      }
      i++ // Next element
    }

    // Figure out which list isn't empty
    var remainder []int

    if len(left) > 0 {
      remainder = left
    } else {
      remainder = right
    }

```

```
    // Empty remaining values
    for len(remainder) > 0 {
      data[i], remainder = remainder[0], remainder[1:]
      i++
    }

    // Copy result to our answer
    for i := 0; i < len(arr); i++ {
      arr[i] = data[i]
    }
  }
  group.Done()
}

func InsertionSort(arr []int) {
  for i, j := 1, 1; i < len(arr); i, j = i+1, i+1 {
    for j > 0 && arr[j] < arr[j-1] {
      arr[j-1], arr[j] = arr[j], arr[j-1]
      j--
    }
  }
}
```