15 Merge sort and Quicksort

15.1 Introduction

The sorting algorithms we have looked at so far are not very fast, except for Shell sort and insertion sort on nearly-sorted lists. In this chapter we consider two of the fastest sorting algorithms known: merge sort and quicksort.

15.2 Merge Sort

Merge sort is a **divide and conquer algorithm** that solves a large problem by dividing it into parts, solving the resulting smaller problems, and then combining these solutions into a solution to the original problem. The strategy of merge sort is to sort halves of a list (recursively) then merge the results into the final sorted list. Merging is a pretty fast operation, and breaking a problem in half repeatedly quickly gets down to lists that are already sorted (lists of length one or zero), so this algorithm performs well. A Ruby implementation of merge sort appears in Figure 1 below.

Potential
for exploration
Potential
for development
ENGINEERS, UNIVERSITY GRADUATES & SALES PROFESSIONALS
Junior and experienced F/M
Total will hire 10,000 people in 2014. Why not you?
Are you looking for work in process, electrical or other types of engineering, R&D, sales & marketing or support professions such as information technology?
We're interested in your skills.
Join an international leader in the oil, gas and chemical industry by applying at
www.careers.total.com
More than 700 job openings are now online!
orc.fr Copyright : Total/Corbis
TOTAL
COMMITTED TO BETTER ENERGY

```
def merge_sort(array)
  merge_into(array.dup, array, 0, array.size)
  return array
end

def merge_into(src, dst, lo, hi)
  return if hi-lo < 2
  m = (lo+hi)/2
  merge_into(dst, src, lo, m)
  merge_into(dst, src, m, hi)
  j, k = lo, m
  (lo..hi-1).each do |i|
    if j < m and k < hi
      if src[j] < src[k]
        dst[i] = src[j]; j += 1
      else
        dst[i] = src[k]; k += 1
      end
    elsif j < m
      dst[i] = src[j]; j += 1
    else # k < hi
      dst[i] = src[k]; k += 1
    end
  end
end
```

Figure 1: Merge Sort

Merging requires a place to store the result of merging two lists: duplicating the original list provides space for merging. Hence this algorithm duplicates the original list and passes the duplicate to `merge_into()`. This operation recursively sorts the two halves of the auxiliary list and then merges them back into the original list. Although it is possible to sort and merge in place, or to merge using a list only half the size of the original, the algorithms to do merge sort this way are complicated and have a lot of overhead—it is simpler and faster to use an auxiliary list the size of the original, even though it requires a lot of extra space.

In analyzing this algorithm, the measure of the size of the input is, of course, the length of the list sorted, and the operations counted are key comparisons. Key comparison occurs in the merging step: the smallest items in the merged sub-lists are compared, and the smallest is moved into the target list. This step is repeated until one of the sub-lists is exhausted, in which case the remainder of the other sub-list is copied into the target list.

Merging does not always take the same amount of effort: it depends on the contents of the sub-lists. In the best case, the largest element in one sub-list is always smaller than the smallest element in the other (which occurs, for example, when the input list is already sorted). If we make this assumption, along with the simplifying assumption that $n = 2^k$, then the recurrence relation for the number of comparisons in the best case is

$$\begin{aligned}
B(n) &= n/2 + 2 \cdot B(n/2) \\
&= n/2 + 2 \cdot (n/4 + 2 \cdot B(n/4)) = 2 \cdot n/2 + 4 \cdot B(n/4) \\
&= 2 \cdot n/2 + 4 \cdot (n/8 + 2 \cdot B(n/8) = 3 \cdot n/2 + 8 \cdot B(n/8) \\
&= \ldots \\
&= i \cdot n/2 + 2^i \cdot B(n/2^i)
\end{aligned}$$

The initial condition for the best case occurs when n is one or zero, in which case no comparisons are made. If we let $n/2^i = 1$, then $i = k = \lg n$. Substituting this into the equation above, we have

$$B(n) = \lg n \cdot n/2 + n \cdot B(1) = (n \lg n)/2$$

Thus, in the best case, merge sort makes only about $(n \lg n)/2$ key comparisons, which is quite fast. It is also obviously in $O(n \lg n)$.

In the worst case, making the most comparisons occurs when merging two sub-lists such that one is exhausted when there is only one element left in the other. In this case, every merge operation for a target list of size n requires n-1 comparisons. We thus have the following recurrence relation:

$$\begin{aligned}
W(n) &= n\text{-}1 + 2 \cdot W(n/2) \\
&= n\text{-}1 + 2 \cdot (n/2\text{-}1 + 2 \cdot W(n/4)) = n\text{-}1 + n\text{-}2 + 4 \cdot W(n/4) \\
&= n\text{-}1 + n\text{-}2 + 4 \cdot (n/4\text{-}1 + 2 \cdot W(n/8) = n\text{-}1 + n\text{-}2 + n\text{-}4 + 8 \cdot W(n/8) \\
&= \ldots \\
&= n\text{-}1 + n\text{-}2 + n\text{-}4 + \ldots + n\text{-}2^{i\text{-}1} + 2^i \cdot W(n/2^i)
\end{aligned}$$

The initial conditions are the same as before, so we may again let $i = \lg n$ to solve this recurrence.

$$\begin{aligned}
W(n) &= n\text{-}1 + n\text{-}2 + n\text{-}4 + \ldots + n\text{-}2^{i\text{-}1} + 2^i \cdot W(n/2^i) \\
&= n\text{-}1 + n\text{-}2 + n\text{-}4 + \ldots + n\text{-}2^{\lg n - 1} \\
&= \Sigma_{j = 0 \text{ to } \lg n - 1}\, n - 2^j \\
&= \Sigma_{j = 0 \text{ to } \lg n - 1}\, n - \Sigma_{j = 0 \text{ to } \lg n - 1}\, 2^j \\
&= n\, \Sigma_{j = 0 \text{ to } \lg n - 1}\, 1 - (2^{\lg n - 1 + 1} - 1) \\
&= n \lg n - n + 1
\end{aligned}$$

The worst case behavior of merge sort is thus also in $O(n \lg n)$.

As an average case, lets suppose that each comparison of keys from the two sub-lists is equally likely to result in an element from one sub-list being moved into the target list as from the other. This is like flipping coins: it is as likely that the element moved from one sub-list will win the comparison as an element from the other. And like flipping coins, we expect that in the long run, the elements chosen from one list will be about the same as the elements chosen from the other, so that the sub-lists will run out at about the same time. This situation is about the same as the worst case behavior, so on average, merge sort will make about the same number of comparisons as in the worst case.

Thus, in all cases, merge sort runs in $O(n \lg n)$ time, which means that it is significantly faster than the other sorts we have seen so far. Its major drawback is that it uses $O(n)$ extra memory locations to do its work.

15.3 Quicksort

Quicksort was invented by C.A.R. Hoare in 1960, and it is still the fastest algorithm for sorting random data by comparison of keys. A Ruby implementation of quicksort appears in Figure 2 below.

Quicksort is a divide and conquer algorithm. It works by selecting a single element in the list, called the *pivot element*, and rearranging the list so that all elements less than or equal to the pivot are to its left, and all elements greater than or equal to it are to its right. This operation is called *partitioning*. Once a list is partitioned, the algorithm calls itself recursively to sort the sub-lists left and right of the pivot. Eventually, the sub-lists have length one or zero, at which point they are sorted, ending the recursion.

The heart of quicksort is the partitioning algorithm. This algorithm must choose a pivot element and then rearrange the list as quickly as possible so that the pivot element is in its final position, all values greater than the pivot are to its right, and all values less than it are to its left. Although there are many variations of this algorithm, the general approach is to choose an arbitrary element as the pivot, scan from the left until a value greater than the pivot is found, and from the right until a value less than the pivot is found. These values are then swapped, and the scans resume. The pivot element belongs in the position where the scans meet. Although it seems very simple, the quicksort partitioning algorithm is quite subtle and hard to get right. For this reason, it is generally a good idea to copy it from a source that has tested it extensively.

```
def quick(array, lb, ub)
  return if ub <= lb
  pivot = array[ub]
  i, j = lb-1, ub
  loop do
    loop do i += 1; break if pivot <= array[i]; end
    loop do j -= 1; break if j <= lb || array[j] <= pivot; end
    array[i], array[j] = array[j], array[i]
    break if j <= i
  end
  array[j], array[i], array[ub] = array[i], pivot, array[j]
  quick(array,lb,i-1)
  quick(array,i+1,ub)
end

def Quicksort(array)
  quick(array, 0, array.size-1)
  return array
end
```

Figure 2: Quicksort

We analyze this algorithm using the list size as the measure of the size of the input, and using comparisons as the basic operation. Quicksort behaves very differently depending on the contents of the list it sorts. In the best case, the pivot always ends up right in the middle of the partitioned sub-lists. We assume, for simplicity, that the original list has 2^k-1 elements. The partitioning algorithm compares the pivot value to every other value, so it makes n-1 comparisons on a list of size n. This means that the recurrence relation for the number of comparison is

$$B(n) = n-1 + 2 \cdot B((n-1)/2)$$

The initial condition is $B(n) = 0$ for $n = 0$ or 1 because no comparisons are made on lists of size one or empty lists. We may solve this recurrence as follows:

$$\begin{aligned}
B(n) &= n\text{-}1 + 2 \cdot B((n\text{-}1)/2) \\
&= n\text{-}1 + 2 \cdot ((n\text{-}1)/2 - 1 + 2 \cdot B(((n\text{-}1)/2 - 1)/2)) \\
&= n\text{-}1 + n\text{-}3 + 4 \cdot B((n\text{-}3)/4) \\
&= n\text{-}1 + n\text{-}3 + 4 \cdot ((n\text{-}3)/4 - 1 + 2 \cdot B(((n\text{-}3)/4 - 1)/2)) \\
&= n\text{-}1 + n\text{-}3 + n\text{-}7 + 8 \cdot B((n\text{-}7)/8) \\
&= \ldots \\
&= n\text{-}1 + n\text{-}3 + n\text{-}7 + \ldots + (n\text{-}(2^i\text{-}1) + 2^i \cdot B((n\text{-}(2^i\text{-}1))/2^i)
\end{aligned}$$

If we let $(n\text{-}(2^i\text{-}1))/2^i = 1$ and solve for i, we get $i = k\text{-}1$. Substituting, we have

$$\begin{aligned}
B(n) &= n\text{-}1 + n\text{-}3 + n\text{-}7 + \ldots + n\text{-}(2^{k\text{-}1}\text{-}1) \\
&= \Sigma_{\,i = 0 \text{ to } k\text{-}1}\ n - (2^i - 1) \\
&= \Sigma_{\,i = 0 \text{ to k-1}}\ n{+}1 - \Sigma_{\,i = 0 \text{ to k-1}}\ 2^i \\
&= k \cdot (n{+}1) - (2^k - 1) \\
&= (n{+}1) \lg (n{+}1) - n
\end{aligned}$$

Thus the best case complexity of quicksort is in $O(n \lg n)$.

Quicksort's worst case behavior occurs when the pivot element always ends up at one end of the sub-list, meaning that sub-lists are not divided in half when they are partitioned, but instead one sub-list is empty and the other has one less element than the sub-list before partitioning. If the first or last value in the list is used as the pivot, this occurs when the original list is already in order or in reverse order. In this case the recurrence relation is

$$W(n) = n\text{-}1 + W(n\text{-}1)$$

This recurrence relation is easily solved and turns out to be $W(n) = n(n\text{-}1)/2$, which of course we know to be $O(n^2)$!

The average case complexity of quicksort involves a recurrence that is somewhat hard to solve, so we simply present the solution: $A(n) = 2(n{+}1) \cdot \ln 2 \cdot \lg n \approx 1.39\ (n{+}1) \lg n$. This is not far from quicksort's best case complexity. So in the best and average cases, quicksort is very fast, performing $O(n \lg n)$ comparisons; but in the worst case, quicksort is very slow, performing $O(n^2)$ comparisons.

15.4 Improvements to Quicksort

Quicksort’s worst case behavior is abysmal, and because it occurs for sorted or nearly sorted lists, which are often encountered, this is a big problem. Many solutions to this problem have been proposed, but perhaps the best is called the *median-of-three improvement*, and it consists of using the median of the first, last, and middle values in each sub-list as the pivot element.

Except in rare cases, this technique produces a pivot value that ends up near the middle of the sub-list when it is partitioned, especially if the sub-list is sorted or nearly sorted. A version of quicksort with the median-of-three improvement appears in Figure 3 below. The median finding process also allows sentinel values to be placed at the ends of the sub-list, which speeds up the partitioning algorithm a little bit as well because array indices need not be checked.

Sentinel value: a special value placed in a data structure to mark a boundary.

From now on, we will assume that quicksort includes the median-of-three improvement.

```
def quick_m3(array, lb, ub)
  return if ub <= lb

  # find sentinels and the median for the pivot
  m = (lb+ub)/2
  array[lb],array[m]=array[m],array[lb] if array[m] < array[lb]
  array[m],array[ub]=array[ub],array[m] if array[ub] < array[m]
  array[lb],array[m]=array[m],array[lb] if array[m] < array[lb]

  # if the sub-array is size 3 or less, it is now sorted
  return if ub-lb < 3

  # put the median just shy of the end of the list
  array[ub-1], array[m] = array[m], array[ub-1]

  pivot = array[ub-1]
  i, j = lb, ub-1
  loop do
    loop do i += 1; break if pivot <= array[i]; end
    loop do j -= 1; break if j <= lb || array[j] <= pivot; end
    array[i], array[j] = array[j], array[i]
    break if j <= i
  end
  array[j], array[i], array[ub-1] = array[i], pivot, array[j]
  quick_m3(array,lb,i-1)
  quick_m3(array,i+1,ub)
end
private :quick_m3

def Quicksort_m3(array)
  quick_m3(array, 0, array.size-1)
  return array
end
```

Figure 3: Quicksort with the Median-of-Three Improvement

Other improvement to quicksort have been proposed, and each speeds it up slightly at the expense of making it a bit more complicated. Among the suggested improvement are the following:

- Use insertion sort for small sub-lists (ten to fifteen elements). This eliminates a lot of recursive calls on small sub-lists and takes advantage of insertion sort's linear behavior on nearly sorted lists. Generally this is implemented by having quicksort stop when it gets down to lists of less than ten or fifteen elements, and then insertion sorting the whole list at the end.
- Remove recursion and use a stack to keep track of sub-lists yet to be sorted. This removes function calling overhead.
- Partition smaller sub-lists first, which keeps the stack a little smaller.

Despite all these improvement, there are still many cases where quicksort does poorly on data that has some degree of order, which is characteristic of much real-world data. Recent efforts to find sorting algorithms that take advantage of order in the data have produced algorithms (usually based on merge sort) that use little extra space and are far faster than quicksort on data with some order. Such algorithms are considerably faster than quicksort in many real-world cases.

15.5 Summary and Conclusion

Merge sort is a fast sorting algorithm whose best, worst, and average case complexity are all in $O(n \lg n)$, but unfortunately it uses $O(n)$ extra space to do its work. Quicksort has best and average case complexity in $O(n \lg n)$, but unfortunately its worst case complexity is in $O(n^2)$. The median-of-three improvement makes quicksort's worst case behavior less likely, but quicksort is still vulnerable to poor performance on data with some order. Nevertheless, quicksort is still the fastest algorithm known for sorting random data by comparison of keys.

15.6 Review Questions

1. Why does merge sort need extra space?
2. What stops recursion in merge sort and quicksort?
3. What is a pivot value in quicksort?
4. What changes have been suggested to improve quicksort?
5. If quicksort has such bad worst case behavior, why is it still used?

15.7 Exercises

1. Explain why the merge sort algorithm first copies the original list into the auxiliary array.
2. Write a non-recursive version of merge sort that uses a stack to keep track of sub-lists that have yet to be sorted. Time your implementation against the unmodified merge sort algorithm and summarize the results.
3. The merge sort algorithm presented above does not take advantage of all the features of Ruby. Write a version of merge sort that uses features of Ruby to make it simpler or faster than the version in the text. Time your algorithm against the one in the text to determine which is faster.
4. Modify the quicksort algorithm with the median-of-three improvement so that it does not sort lists smaller than a dozen elements and calls insertion sort to finish sorting at the end. Time your implementation against the unmodified quicksort with the median-of-three improvement and summarize the results.
5. Modify the quicksort algorithm with the median-of-three improvement so that it uses a stack rather than recursion and works on smaller sub-lists first. Time your implementation against the unmodified quicksort with the median-of-three improvement and summarize the results.

6. The quicksort algorithm presented above does not take advantage of all the features of Ruby. Write a version of quicksort that takes advantage of Ruby's features to make the algorithm shorter or faster. Time your version against the one in the text to determine which one is faster.
7. Write the fastest quicksort you can. Time your implementation against the unmodified quicksort and summarize the results.
8. The Ruby `Array` class has a `sort()` method that uses a version of the quicksort algorithm. If you time any of the quicksort algorithms we have presented, or one you write yourself, against the `Array.sort()` method, you will find that the latter is much faster. Why is this so?

15.8 Review Question Answers

1. Merge sort uses extra space because it is awkward and slow to merge lists without using extra space.
2. Recursion in merge sort and quicksort stops when the sub-lists being sorted are either empty or of size one—such lists are already sorted, so no work needs to be done on them.
3. A pivot value in quicksort is an element of the list being sorted that is chosen as the basis for rearranging (partitioning) the list: all elements less than the pivot are placed to the left of it, and all elements greater than the pivot are placed to the right of it. (Equal values may be placed either left or right of the pivot, and different partitioning algorithms may make different choices).
4. Among the changes that have been suggested to improve quicksort are (a) using the median of the first, last, and middle elements in the list as the pivot value, (b) using insertion sort for small sub-lists, (c) removing recursion in favor of a stack, and (d) sorting small sub-lists first to reduce the depth of recursion (or the size of the stack).
5. Quicksort is still used because its performance is so good on average: quicksort usually runs in about half the time of other sorting algorithms, especially when it has been improved in the ways discussed in the chapter. Its worst case behavior is relatively rare if it incorporates the median-of-three improvement.