Fast Sorting Algorithms

DANIEL ANDERSON [1]

Sorting is a fundamental and interesting problem in computer science. Many sorting algorithms exist which each have their own advantages and disadvantages. Sorting algorithms like Selection Sort and Insertion Sort take $O(N^2)$ time but are very easy to implement and run very fast for very small input sizes. Faster sorting algorithms exists which only take $O(N \log(N))$ time, but which may be more difficult to implement. We will analyse some of these algorithms and also explore some interesting theoretical results regarding sorting. We will also take a look at how when we have data of a specific variety, we can achieve even faster $O(N)$ sorts.

Summary: Fast Sorting Algorithms

In this lecture, we cover:

- Complexity lower bounds for sorting
- The Quicksort algorithm
- The Radix Sort algorithm

Recommended Resources: Fast Sorting Algorithms

- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Flag/`
- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Quick/`
- `http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix/`
- CLRS, Introduction to Algorithms, Chapters 6, 7, 8
- Weiss, Data Structures and Algorithm Analysis, Chapter 7

Revision: Fast Sorting Algorithms

You should be familiar from your previous studies with:

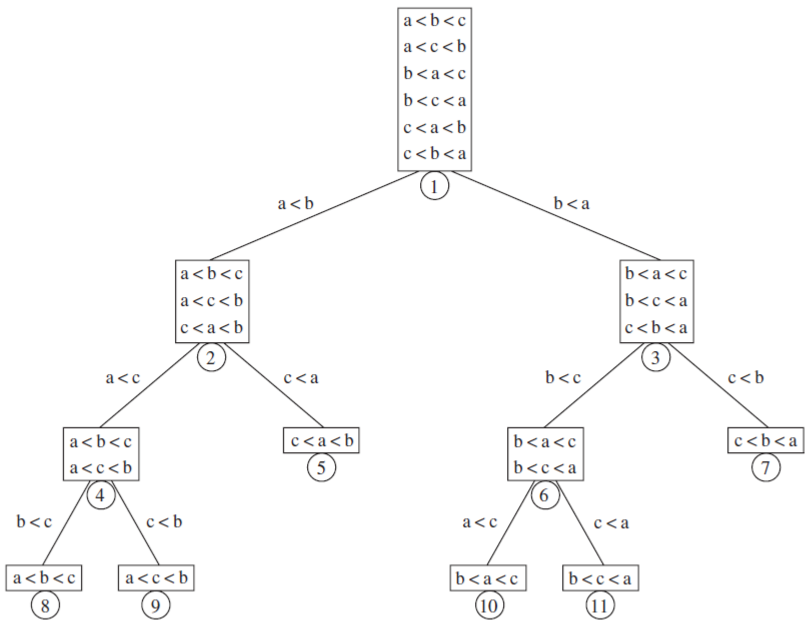
- The Heapsort algorithm (See appendix and additional preparatory documents on Moodle)
- The Merge Sort algorithm (See appendix and additional preparatory documents on Moodle)

How Fast Can We Really Sort Things?

Selection Sort and Insertion Sort take $O(N^2)$ time, which is fine for very small lists, but completely impractical when N grows large. Faster sorting algorithms such as Merge Sort, Heapsort and Quicksort run in $O(N \log(N))$ time, which is asymptotically a significant improvement. A fundamentally interesting question is can we do better than this? For specially structured data with properties that we can exploit, the answer is yes, but for completely general data, we can actually prove that comparison based sorting algorithms can not possibly be faster than $O(N \log(N))$. to do so, we consider a *decision tree* that models the knowledge we have about the order of a particular sequence of data after comparing individual elements.

A decision tree that represents the sorting process of a sequence of three elements in shown below. Each node corresponds to a set of potential sorted orders based on the comparisons performed so far. The leaf nodes of the tree correspond to states where we have enough information to know the fully sorted order of the sequence.

[1] Faculty of Information Technology, Monash University: `daniel.anderson@monash.edu`. These notes are based on the lecture slides developed by Arun Konagurthu for FIT2004.



A decision tree for comparison-based sorting.
Taken from Weiss, Data Structures and Algorithm Analysis in Java, 3rd ed, page 303.

Theorem: Lower bound on comparison-based sorting

Any comparison-based sorting algorithm can not run faster than $\Theta(N \log(N))$ in the worst case[a]

[a]Recall that Θ notation denotes an asymptotic upper **and** lower bound. In other words, the bound is tight. Refer to Lecture 1.

Proof: NOT EXAMINABLE

We appeal to the structure of the decision tree depicted above and observe that comparing elements until we deduce the fully sorted order of a sequence is equivalent to traversing the tree until reaching a leaf node. The worst-case behaviour of any comparison based sorting algorithm therefore corresponds to the depth of the deepest leaf in the tree, ie. the height of the tree. There are $N!$ total possible sorted orders for a sequence of N elements, each of which must appear as a leaf in the tree. Let's denote the height of the tree by h and observe that a binary tree of height h can not have more than 2^h leaves. Putting this together, we have that

$$N! \leq 2^h.$$

Taking the logarithm of both sides yields

$$h \geq \log_2(N!).$$

Intuitively, one can see that $N! = O(N^N)$ and hence $\log(N!) = O(N \log(N))$. This of course only establishes an upper bound on the worst-case behaviour, not a lower bound. Appealing to Stirling's formula, we get a proper bound of

$$\log(N!) = N \log(N) - N + O(\log(N)),$$

which establishes the desired lower bound

$$h = \Omega(N \log(N)).$$

We know that Heapsort and Merge Sort achieve this level of performance, hence we do have asymptotically optimal algorithms that run in

$$\Theta(N \log(N))$$

time.

Quicksort

Quicksort is one of many divide-and-conquer approaches to sorting an array of elements.

Key Ideas: Quicksort

- Select some element of the array, which we will call the *pivot*
- Partition the array so that all items less then the pivot are to its left, all elements equal to the pivot are in the middle, and all elements greater than the pivot are to its right
- Quicksort the left part of the array (elements less than the pivot)
- Quicksort the right part of the array (elements greater than the pivot)

The key ideas are very simple. The two major components to correctly (and efficiently) implementing Quicksort are the partitioning algorithm (how do we move elements lesser/greater than the pivot to the left/right efficiently?) and how we chose to select our pivot element.

The partitioning problem

The partitioning problem is central to Quicksort, and can be more easily understood in terms of a problem posed by Edsger Dijkstra, called the "Dutch National Flag (DNF) Problem."

Problem Statement

Given an array of N objects ($array[1..N]$) coloured **red**, **white** or **blue**, sort them so that all objects of the same colour are together, with the colours in the order **red**, **white** and **blue**.

The connection to the Quicksort partitioning problem should be quite clear: The **red** items correspond to elements less than the pivot, the **white** items correspond to elements equal to the pivot, and the **blue** items correspond to elements greater than the pivot. To solve the DNF problem, we maintain three pointers, `lo, mid, hi`, such that the following invariant is held.

Invariant: Dutch National Flag Partitioning Problem

Maintain three pointers `lo, mid, hi` such that,

- `array[1..lo-1]` contains the **red** items
- `array[lo..mid-1]` contains the **white** items
- `array[mid..hi]` contains the currently unknown items
- `array[hi+1..N]` contains the **blue** items

Initially, we set `lo = 1, mid = 1, hi = N`. This means that initially the entire array is the unknown section. We then iteratively move items from the unknown section into their corresponding sections while updating the pointers `lo, mid, hi`. At each iteration, we move the item at `array[mid]` (ie. the first unknown item). There are three cases to consider:

Case 1: `array[mid]` is Red:

If `array[mid]` is **red**, we'd like to move it into the first section. So, we will swap `array[mid]` with `array[lo]`. This means that the **red** section is extended by one element, so we increment the `lo` pointer. The item originally at `array[lo]` was either **white**, in which case it has been moved to the end of the **white** section, so we increment `mid`, or the **white** section was empty, meaning we simply swapped `array[lo]` with itself, then we increment `mid`. In both situations, we see that the invariant is maintained, and the unknown section has shrunk in size by one element.

Case 2: `array[mid]` is White:

If `array[mid]` is **white**, then it is already where we want it to be (right at the end of **white** section) so we simply have to increment the `mid` pointer. This also shrinks the unknown section by one element.

Case 3: `array[mid]` **is Blue:**

If `array[mid]` is **Blue**, we want to move it into the final section, so we will swap `array[mid]` with `array[hi]`. The **Blue** section is therefore extended so we decrement the `pointer`. We do not move the `mid` pointer in this case since the `array[mid]` now contains what was previously the final element in the unknown section.

Algorithm: Dutch National Flag (3 Colour Partitioning)

```
function PARTITION(array[1..N])
    Set lo = 1, mid = 1, hi = N
    while mid ≤ hi do
        if array[mid] = red then
            swap(array[mid], array[lo])
            lo += 1, mid += 1
        else if array[mid] = white then
            mid += 1
        else
            swap(array[mid], array[hi])
            hi -= 1
        end if
    end while
    return lo, mid
end function
```

Note that we return the final position of the `lo` and `mid` pointers so that we can quickly identify the locations of the three sections after the partitioning. Also notice that this partitioning process is not stable.

Implementing Quicksort

We now have the tools to implement Quicksort. Assume for now that we select the first element in the range `[lo..hi]` to be our pivot. We will explore more complicated pivot selection rules later.

Algorithm: Quicksort

```
function QUICKSORT(array[lo..hi])
    if hi > lo then
        Set pivot = array[lo]
        left, right = partition(array[lo..hi], pivot)
        quicksort(array[lo..left-1])
        quicksort(array[right..hi])
    end if
end function
```

Algorithm: Partition

```
function PARTITION(array[1..N], pivot)
    Set lo = 1, mid = 1, hi = N
    while mid ≤ hi do
        if array[mid] < pivot then
            swap(array[mid], array[lo])
            lo += 1, mid += 1
        else if array[mid] = pivot then
            mid += 1
        else
            swap(array[mid], array[hi])
            hi -= 1
        end if
    end while
    return lo, mid
end function
```

Time and space complexity of Quicksort

The running time of Quicksort is entirely dependent on the quality of the pivot that is selected at each stage. Ideally, the resulting partition will be balanced (contain roughly equal amounts of elements on each side) which will minimise the number of recursive calls required.

Best-case partition

In the ideal case, the pivot turns out to be the median of the array. Recall that the median is the element such that half of the array is less than it and half of the array is greater than it. This will result in us only having $\log_2(N)$ levels of recursion since the size of the array would be halved at each level. At each level, the partitioning takes $O(N)$ total time, and hence the total runtime in the best case for Quicksort is $O(N\log(N))$.

Worst-case partition

In the worst case, the pivot element might be the smallest or largest element of the array. Suppose we select the smallest element of the array as our pivot, then the size of the sub-problems solved recursively will be 0 and $N-1$. The size 0 problem requires no effort, but we will have only reduced the size of our remaining sub-problem by one. We will therefore have to endure $O(N)$ levels of recursion, on each of which we will perform $O(N)$ work to do the partition. In total, this yields a worst-case run time of $O(N^2)$ for Quicksort.

Average case partition

On average, the partitions obtained by an arbitrary pivot will be neither a perfect median nor the smallest or largest element. This means that on average, the splits should be “okay.” Our intuition should tell us that in the average case, Quicksort will take an expected $O(N\log(N))$ running time, since selecting a terrible pivot at every single level of recursion is extremely unlikely.

Theorem: Average case run time for Quicksort

The expected run time for Quicksort on an array that is randomly permuted with equal probability is $O(N\log(N))$. (This is equivalent to selecting a random pivot.)

Proof

Assume without loss of generality that the array contains no duplicate elements (duplicate elements only speed up the algorithm anyway since the size of the sub-problems is reduced by the number of elements equal to the pivot). Let us denote the time taken to Quicksort an array of size N by T_N. If the k'th smallest element is selected as the pivot, then the running time T_N will be given by

$$T_N = N + 1 + T_{k-1} + T_{N-k},$$

where the terms T_{k-1} and T_{N-k} correspond to the time taken to perform the recursive calls after the pivot operation. Given this, if all partitions are equally likely to occur, the average running time over all partitions is given by

$$T_N = N + 1 + \frac{1}{N}\sum_{k=1}^{N}(T_{k-1} + T_{N-k}).$$

Observe that each term T_{k-1} appears a total of twice for each k in the sum, so we have

$$T_N = N + 1 + \frac{2}{N}\sum_{k=1}^{N} T_{k-1}.$$

Multiply this equation by N to obtain

$$NT_N = N^2 + N + 2\sum_{k=1}^{N} T_{k-1}.$$

Now take $N = N - 1$ and subtract the resulting equation from the above to find

$$NT_N - (N-1)T_{N-1} = N^2 + N - (N-1)^2 - (N-1) + 2T_{N-1}.$$

Clean up with some algebra and we will get

$$NT_N = (N+1)T_{N-1} + 2N.$$

Divide by $(N+1)$ and N to find the recurrence

$$\frac{T_N}{N+1} = \frac{T_{N-1}}{N} + \frac{2}{N+1}.$$

Telescoping the right hand side, we obtain

$$\begin{aligned}\frac{T_N}{N+1} &= \frac{T_{N-1}}{N} + \frac{2}{N+1}. \\ &= \frac{T_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1} \\ &= \ \vdots \\ &= \sum_{k=1}^{N} \frac{2}{k+1}\end{aligned}$$

where we have stopped at the base case $T_0 = 0$. Finally, using the fact that

$$\sum_{k=1}^{N} \frac{2}{k+1} = O(\log(N)),$$

(this is the asymptotic behaviour of the harmonic numbers) we obtain that

$$T_N = (N+1)\sum_{k=1}^{N} \frac{2}{k+1} = O(N\log(N))$$

Space complexity

In the worst case, due to the linear number of recursive calls, we would expect that the auxiliary space required by Quicksort would be $O(N)$. This is true, but can actually be improved by making a very simple optimisation. Instead of recursively sorting the left half of the array and then the right half of the array, we can always sort the smallest half first, followed by the largest half. By sorting the smallest half first, we encounter only $O(\log(N))$ levels of recursion on the program stack, while the larger half of the partition is sorted by a tail-recursive call, which adds no overhead to the program stack.

	Best case	**Average Case**	**Worst Case**
Time	$O(N\log(N))$	$O(N\log(N))$	$O(N^2)$
Auxiliary Space	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$

Sorting in Linear Time – Radix Sort

For arbitrary data, we saw before that the best possible time complexity we can achieve for comparison-based sorting is $O(N\log(N))$. However, if we assume that the contents of the array that we are sorting are bounded with lots of duplicate elements, then we can do better.

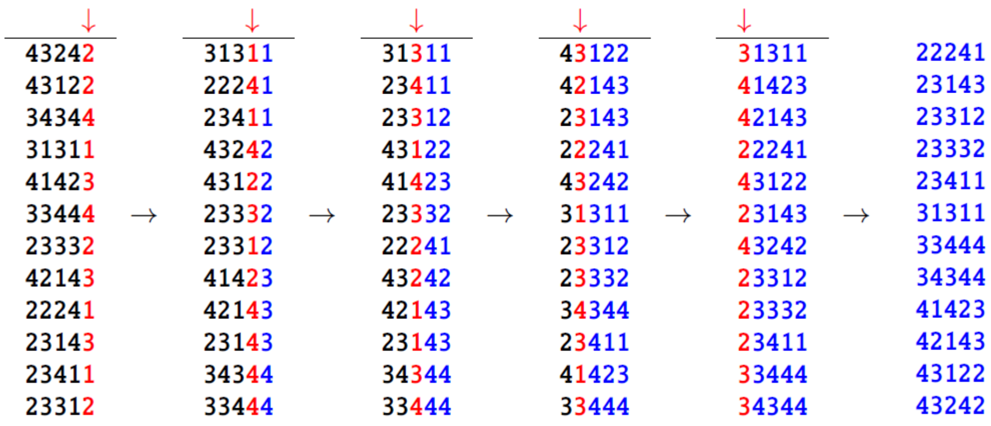
Suppose that we want to sort an array that consists only of 1s, 2s and 3s. How can we do this, and how fast can we do it? Actually we just solved this problem earlier, this is the Dutch National Flag problem, which we saw has a simple $O(N)$ solution. The fact that the elements of the array were assumed to be bounded meant that we did not have to do a comparison based sort, and could exploit the structure of the data to produce a faster algorithm.

Radix sort is a more general, non-comparison-based sort that achieves linear time ($O(N)$) on an array consisting of elements from some small, fixed width alphabet. There are many variants of radix sort, we will look at arguably the simplest one: Least Significant Digit (LSD) Radix Sort. In essence, LSD Radix Sort works by sorting an array of elements one digit at a time, from the least significant to the most significant. Each digit is sorted by grouping the contents of the array by elements that have the same value of the current digit, and then combining all of the groups together in order. Each digit must be sorted in a stable manner in order to maintain the relative ordering of the previously sorted digits.

Key Ideas: LSD Radix Sort

- Sort the array one digit at a time, from least significant (rightmost) to most significant (leftmost)
- For each digit:
  - Divide the contents of the array into groups based on the current digit value of each element
  - Combine the groups together in a stable manner so that the values of the current digit are sorted

A visualisation of an LSD Radix Sort is shown below.



An illustration of LSD Radix Sort. First the numbers are sorted by their least significant digit, then their second, etc. until the most significant digit is sorted and we are finished.

It is important to realise that we do not necessarily have to use their *decimal* digits to sort the elements of the array. We could have instead done Radix Sort with a radix (base) of 100, which would mean sorting consecutive groups of two decimal digits at a time. Using a larger radix size means less passes over the array must be made, but that more work must be done on each pass. It is therefore a balancing act to find the best radix to use for a particular set of data, which will depend on the size and contents of the array on a case-by-case basis.

An implementation of LSD Radix Sort might therefore look like this.

Algorithm: Single Digit Pass of Radix Sort

```
function RADIX_PASS(array[1..N], base, digit)
    Set counter[0..base-1] = [0,0,...],
    for i = 1 to N do
        counter[get_digit(array[i], base, digit)] += 1
    end for
    Set position[0..base-1] = [1,0,...]
    for value = 1 to base - 1 do
        position[value] = position[value-1] + counter[value-1]
    end for
    Set temp[1..N] = [0,0,...]
    for i = 1 to N do
        temp[position[get_digit(array[i],base,digit)]] = array[i]
        position[get_digit(array[i],base,digit)] += 1
    end for
    swap(array, temp)
end function
```

Algorithm: LSD Radix Sort

```
1: function RADIX_SORT(array[1..N], base)
2:     for digit = 1 to NumDigits do
3:         radix_pass(array[1..N], base, digit)
4:     end for
5: end function
```

Where `get_digit(x, b, d)` is a function that computes the value of digit `d` in base `b` of the element `x`.

Note that the method we use to group elements together based on similar digits is to keep a counter of how many times we've seen that digit (this is done by the first `for` loop of the `radix_pass` algorithm). Once we have counted the entire array, we figure out the position of each element by counting how many smaller elements occurred before us. For example, if the digit 1 appeared three times, and the digit 2 appeared twice, then we know that the 3s should occupy positions six and above in the sorted array (this is the purpose of the second `for` loop in the `radix_pass` algorithm). The third and final `for` loop of the `radix_pass` algorithm places each element in its correct position by using the positions computed by the previous `for` loop.

Time and Space Complexity of Radix Sort

If the width (number of digits) of the elements of the array to be sorted is w, then we perform w passes over the array, leading to a time complexity of $O(wN)$ for Radix Sort. If the input is bounded and $w \ll N$, which can be considered a constant, then we can consider the time complexity to be $O(N)$, a linear time sort! Of course, we must consider that if the array were to contain only distinct elements, then the largest element must be at least size N, which has $w = O(\log(N))$ digits. This means that Radix Sort would still have a complexity of $N \log(N)$, or possibly even worse if in fact $w \gg \log(N)$. This shows us that Radix Sort should only perform particularly well on arrays that contain many duplicate elements with a fixed (small) number of digits. In practice, we can not assume that this will always be the case, but when dealing with data for which we know this to be true, Radix Sort can be an excellent choice over traditional comparison-based sorting algorithms.

Since we use a temporary working array to redistribute the groups of elements, and we use an array to count the number of occurrences of each digit, the amount of extra space used is $O(N + w)$. There are variants of Radix Sort that operate in place and require less extra memory, but these suffer from drawbacks such as not being stable.

	Best case	**Average Case**	**Worst Case**
Time	$O(wN)$	$O(wN)$	$O(wN)$
Auxiliary Space	$O(N + w)$	$O(N + w)$	$O(N + w)$

Appendix: Heapsort (REVISION)

You should be familiar with the heap data structure from your previous units. If you've forgotten, here is a quick reminder:

Definition: Binary Heap Data Structure

The binary heap data structure can be described as follows.

- A binary heap is a complete binary tree (all levels except for the last are completely filled).
- Every element in a heap is no smaller than its children (ie. the maximum element is at the top).

Property: Binary Heap Data Structure

The binary heap data structure has the following properties.

- Due to its structure, a binary heap can be represented as a flat array `array[1..N]` where the root node is `a[1]` and for each node `array[i]`, its children (if they exist) are elements `array[2i]` and `array[2i+1]`.
- An existing array can be converted into a heap in-place[a] in $O(N)$ time. (**IMPORTANT**.)
- A new item can be inserted into a binary heap in $O(\log(N))$ time.
- The maximum element can be removed from a binary heap in $O(\log(N))$ time.

[a]in the original array without requiring any extra memory

Making use of the heap data structure, one can easily piece together the Heapsort algorithm, which simply involves converting the given sequence into a heap (*heapifiying* it) and then successively removing the maximum element and placing it at the back of the array.

Algorithm: Heapsort

```
1: function HEAPSORT(array[1..N])
2:     heapify(array[1..N])
3:     for i = N to 1 do
4:         array[i] = extract_max(array[1..i])
5:     end for
6: end function
```

That's it! Of course you'll need to recall how to perform the `heapify` and `extract_max` operations. They are provided in another appendix of these notes for your convenience.

Time and space complexity of Heapsort

Since Heapsort performs one `heapify` which takes $O(N)$ time, and N invocations of `extract_max` taking up to $O(\log(N))$ time each, the total time spent by `heapsort` is $O(N\log(N))$. Note that the behaviour of Heapsort is independent of the structure of the input array, so its best, average and worst-case performances are asymptotically the same. Notably however, since `heapify` is done in-place and each element extracted is placed at the end of the array, `heapsort` requires no extra memory and hence its auxillary space complexity is $O(1)$.

	Best case	**Average Case**	**Worst Case**
Time	$O(N\log(N))$	$O(N\log(N))$	$O(N\log(N))$
Auxiliary Space	$O(1)$	$O(1)$	$O(1)$

Appendix: Heap Operations (REVISION)

Algorithm: Heapify

```
function HEAPIFY(array[1..N])
    for i = N/2 to 1 do
        down_heap(array[1..N], i)
    end for
end function
```

Algorithm: Percolate up

```
function UP_HEAP(array[1..N], i)
    Set parent = floor(i/2)
    while parent ≥ 1 do
        if array[parent] < array[i] then
            swap(array[parent], array[i])
            i = parent
            parent = i/2
        else
            break
        end if
    end while
end function
```

Algorithm: Percolate down

```
function DOWN_HEAP(array[1..N], i)
    Set child = 2*i
    while child ≤ N do
        if child < N and array[child+1] > array[child] then
            child += 1
        end if
        if array[i] < array[child] then
            swap(array[i], array[child])
            i = child
            child = 2*parent
        end if
    end while
end function
```

Algorithm: Heap insertion

```
function INSERT(array[1..N], x)
    array.append(x)
    N += 1
    up_heap(array[1..N], N)
end function
```

Algorithm: Heap removal

```
function EXTRACT_MAX(array[1..N])
    swap(array[1], array[N])
    N = N - 1
    down_heap(array[1..N], 1)
    return array.pop_back()
end function
```

Appendix: Merge Sort (REVISION)

Merge Sort is a divide-and-conquer sorting algorithm that sorts a given sequence by dividing it into two halves, sorting those halves and then merging the sorted halves back together. How does Merge Sort sort the two halves? Using Merge Sort of course! Merge Sort is an example of a recursive sorting algorithm, where each half of the sequence is sorted recursively until we reach a base case (a sequence of only one element). The key part of Merge Sort is the `merge` routine, which takes two sorted sequences and intertwines them together to obtain a single sorted sequence.

Algorithm: Merge

```
function MERGE(array1[i..end1], array2[j..end2])
    Set result = []
    while i ≤ end1 or j ≤ end2 do
        if j > end2 or i ≤ end1 and array1[i] ≤ array2[j] then
            result.append(array1[i])
            i += 1
        else
            result.append(array2[j])
            j += 1
        end if
    end while
    return result
end function
```

Observe that this merge routine is stable (since we write `array1[i] ≤ array2[j]`, rather than <), and hence our Merge Sort implementation will also be stable. Using this merge routine, an implementation of Merge Sort can be expressed like this. We take two array parameters, one which is the actual array that we are sorting, and one for working space.

Algorithm: Split and Merge

```
function MERGE_SORT(array[lo..hi], work[lo..hi])
    if hi > lo then
        Set mid = floor((lo + hi) / 2)
        merge_sort(array[lo..mid], work[1..mid])
        merge_sort(array[mid+1..hi], work[mid+1..hi])
        array[lo..hi] = merge(work[lo..mid], work[mid+1..hi])
    end if
end function
```

For convenience, we usually define an extra function that handles setting up the working array for us.

Algorithm: Merge Sort

```
function MERGE_SORT(array[1..N])
    Set work = copy(input)
    merge_sort(array[1..N], work[1..N])
end function
```

Time and space complexity of Merge Sort

Merge Sort repeatedly splits the input sequence in half until it can no longer do so any more. The number of times that this can occur is $\log_2(N)$. Merging N elements takes $O(N)$ time, so doing this at each level of recursion results in a total of $O(N \log(N))$ time. This is true regardless of the input, hence the best, average and worst-case performances for Merge Sort are equal.

For space, we use an extra working array of size N, and an additional $O(\log(N))$ space to handle the recursion. This yields a total of $O(N)$ auxiliary space.

	Best case	**Average Case**	**Worst Case**
Time	$O(N\log(N))$	$O(N\log(N))$	$O(N\log(N))$
Auxiliary Space	$O(N)$	$O(N)$	$O(N)$

Enhancements of Merge Sort

Merge Sort can also be implemented *bottom-up*, in which we eliminate the recursion all-together and simply iteratively merge together the sub-arrays of size `1, 2, 4, 8, ... N`. This still requires $O(N)$ auxiliary working memory, but eliminates the use of the program stack (for recursion) and hence should be slightly faster.

For the adventurous, there are also in-place implementations of Merge Sort that require only $O(1)$ space, but they are much more difficult to implement correctly! See *Nicholas Pippenger Sorting and Selecting in Rounds, Society for Industrial and Applied Mathematics Journal on Computing, 16, 1032 (1987).*