1 Sorting Algorithms

A sorting algorithm is any algorithm which solves this problem:
Input: A sequence of n numbers $a_1, a_2, \ldots, a_n$
Output: A permutation $a'_1, a'_2, \ldots, a'_n$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

1.1 Comparison Sorts

First, let's consider the insertion sort algorithm. Let A be an array of values we want sorted, and A is indexed from 1 to n.
InsertionSort(A)
for $j = 2$ to A.length
    key = A[j]
    $i = j - 1$
    while $i > 0$ and A[i] > key
        A[$i + 1$] = A[i]
        $i = i - 1$
    A[$i + 1$] = key

First, think about correctness.

To really understand why InsertionSort is correct, we introduce the idea of a *loop invariant* - a property that is true at the beginning of ever iterations of the for loop:

> At the start of each iteration of the for loop, the subarray A[$1 \ldots j-1$] contains all the elements of the original A[$1 \ldots j-1$] in sorted order.

After correctness, we want to think about running time.
With some thought, it is easy to see that InsertionSort does not perform the same number of steps for every input.
Therefore, instead of analyzing the running time of InsertionSort, we will analyze the running time in the best and worst cases for InsertionSort.

Best Case: A is already sorted. Then the while loop never runs, because the A[i] > key condition always fails. Thus, the running time of InsertionSort

for an already sorted list is $\Theta(n)$.

Worst Case: If A comes to us sorted in reverse order, then the while loop will run the maximum number of times in each iteration of the for loop. In this case, the running time will be

$$\sum_{j=2}^{n}\sum_{i=1}^{j-1} 1 = \sum_{j=2}^{n}(j-1) = \sum_{j=2}^{n} j - \sum_{j=2}^{n} 1 = \sum_{j=2}^{n} j - (n-1) = \sum_{j=1}^{n} j - 1 - (n-1)$$
$$= \frac{n(n+1)}{2} - 1 - (n-1)$$

Therefore, the running time of InsertionSort for a list in reverse sorted order is $\Theta(n^2)$.

Next, we consider the merge sort algorithm.

MergeSort(A, p, r)
if $p < r$
    $q = \lfloor \frac{p+r}{2} \rfloor$
    MergeSort(A, p, q)
    MergeSort(A, q+1, r)
    Merge(A, p, q, r)

Merge(A, p, q, r)
$n_1 = q - p + 1$
$n_2 = r - q$
let L, R be new arrays
for $i = 1$ to n_1
    $L[i] = A[p + i - 1]$
for $j = 1$ to n_2
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to r
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else
        $A[k] = R[j]$
        $j = j + 1$

MergeSort is an example of a divide-and-conquer algorithm:
Divide: Divide the n-element sequence into two subsequence of (roughly) $n/2$ elements each.
Conquer: Sort the two subsequences recursively.
Combine: Merge the two sorted subsequence to get the complete sorted sequence.

In this algorithm, the number of steps performed is the same for any list of size n, so we do not need to the best/worst case analysis.

Since there is recursion, we will need to represent the running time through a recurrence relation. We will let

$$T(n) = 2T(n/2) + n$$
$$T(1) = 1$$

represent the runnning time.

Note that this is not exactly right; unless n is a power of 2, we cannot evenly split the list each time.

Therefore, the running time of MergeSort is

$$\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2[2T(n/4) + n/2] + n \\
&= 2[2[2T(n/8) + n/4] + n/2] + n \\
&= 2^{k+1}T(n/2^{k+1}) + (k+1)n && \text{after } k \text{ substitutions} \\
&= 2^{\log_2(n)-1+1}T(n/2^{\log_2(n)-1+1}) + (\log_2(n) - 1 + 1)n && k \approx \log_2(n) - 1 \\
&= 2^{\log_2(n)}T(n/2^{\log_2(n)}) + n\log_2(n) \\
&= nT(1) + n\log_2(n) \\
&= \Theta(n\log(n))
\end{aligned}$$

Insertion sort, merge sort, quicksort, heapsort, and probably any other sorting algorithms you know are called *comparison sorts*, because they rely only on comparing the relative order of pairs of elements to build the sorted output.

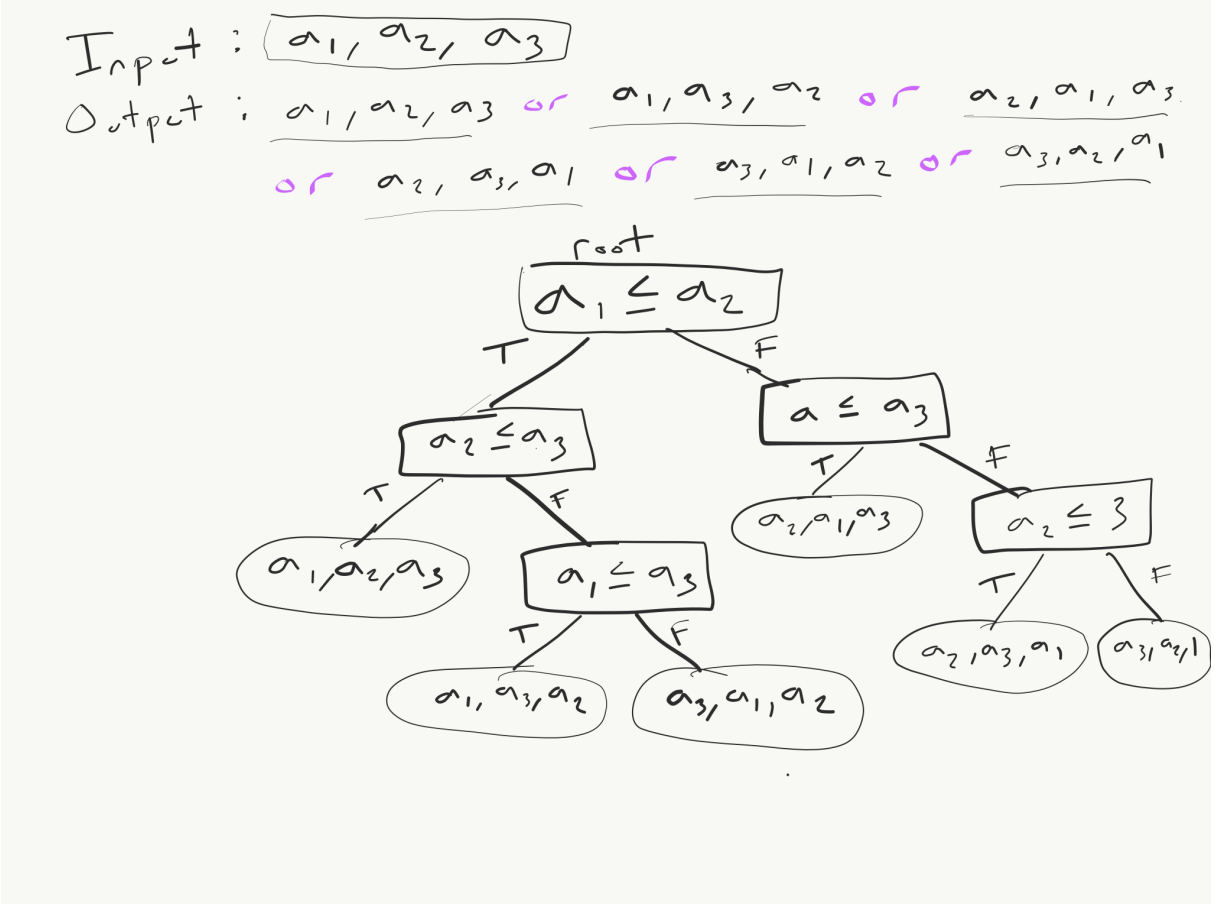
For any comparison sort (even ones no one has thought of yet), we can put a lower bound on the running time:

Theorem 1.1. *Any comparison sort algorithm requires* $\Omega(n\log(n))$ *comparisons in the worst case.*

Proof. Suppose we have some comparison sort algorithm, i.e. for any input sequence $a_1, a_2, \ldots, a_n$ we only use comparisons between elements to determine their relative order.

Without loss of generality, we may assume that all comparisions are of the form $a_i \leq a_j$, since all other comparisions $=, <, >, \geq$ yield the same information about the relative order of a_i and a_j.

For any value of n, we can build a decision tree that encodes the each possible path of execution as a path from the root to a leaf node.



Lets consider the properties this decision tree must have:
(1) The tree must have a unique path for each possible output, therefore there must be at least $n!$ leaf nodes.
(2) If the tree has height h, then since it is a binary tree it has at most 2^h leaf nodes.

Therefore, $2^h \geq n!$ and so $h \geq \log_2(n!) = \Omega(n \log(n))$.

The height of the tree is $\Omega(n \log(n))$, so there exists at least one path from root to leaf node of length $\Omega(n \log(n))$.
So there is some path of execution of our comparison sort that performs $\Omega(n \log(n))$ comparisons. $\square$

1.2 Linear Time Sorting Algorithms

If we make some assumptions about the input, we can create sorting algorithms that run in time $\Theta(n)$.
For example, for our input $a_1, a_2, \ldots, a_n$ lets assume that all a_i are integers between 0 and k.
This leads us to the **Counting Sort** algorithm:

CountingSort(A, B, k)
let $C[0, \ldots, k]$ be a new array, indexed from 0 to k
for $i = 0$ to k
    $C[i] = 0$
for $i = 1$ to A.length
    $C[A[i]] = C[A[i]] + 1$
for $i = 1$ to k
    $C[i] = C[i] + C[i - 1]$
for $i = A$.length down to 1
    $B[C[A[i]]] = A[i]$
    $C[A[i]] = C[A[i]] - 1$

The first two for loops probably make sense, but you are probably wondering why I have the last two for loops.
The last two for loops make this a stable sort - the relative order of the input is preserved when possible.
For example, suppose our input was a_1, a_2, a_3 where $a_1 = a_3 < a_2$. Then a_1, a_3, a_2 or a_3, a_1, a_2 would be valid outputs for a (non-stable) sort. But a stable sort would guarantee to choose the a_1, a_3, a_2 output, preserving the relative order of a_1 and a_3.

Now lets consider the running time.
There are two parameters here that affect the running time, n and k, and we cannot express one in terms of the other.

So our running time needs to be expressed in terms of both.

$$c_1k + c_2n + c_3k + c_4n = (c_1 + c_3)k + (c_2 + c_4)n \leq (c_1 + c_2 + c_3 + c_4)(n + k)$$
$$c_1k + c_2n + c_3k + c_4n = (c_1 + c_3)k + (c_2 + c_4)n \geq (n + k)$$

We can find the running time by adding up the time of each for loop, so the running time is $\Theta(n + k)$.

So as long as $k = O(n)$, the running time of CountingSort is $\Theta(n)$.
Now, lets assume our input values each each have at most d digits, and consider the **Radix Sort** algorithm:

RadixSort(A, d)
for $i = 1$ to d
    use a stable sort to sort A on digit i (starting from least significant digit)

Lets use the counting sort algorithm above as our stable sort.
Then RadixSort runs in time

$$d \cdot (\text{running time of CountingSort})$$

so RadixSort runs in time $\Theta(d(n + k))$.

If we are working in base 10, then each digit takes on values from 0 to 9, and so $k = 9$, in which case RadixSort has running time $\Theta(dn)$.