Comparison Sorting

A *comparison sort* is a sorting algorithm that sorts an array of elements based only on comparison of its elements: e.g. on whether `b[i]` < `b[j]`, or whether `b[i]` ≤ `b[j]`, or whether `b[i]` *comes before* `b[j]` (whatever that means) in the desired ordering.

Insertion sort, selection sort, quick sort, merge sort, and heap sort are all comparison sorts. Radix sort, bucket sort, and counting sort are *not* comparison sorts —they use other operations besides comparison of array elements in sorting.

Worst-case time for a comparison sort is at least O(n log n)

We now prove that any comparison sort algorithm requires worst-case time O(`n` log `n`) to sort an array of size `n`. This means that it is impossible to improve on merge sort, or heap sort, which both require worst-case time O(`n` log `n`). Most of the reasoning in the proof is relatively simple, but some math is needed to obtain the final result.

An array with n distinct elements has n! permutations

First, suppose the values in array `b` of size `n` are all different. How many different permutations of `b` are there?

> For the first value of a permutation, there are `n` possibilities.
> Once the first is chosen, there are `n-1` possibilities for the second value.
> Once the first and second are chosen, there are `n-2` possibilities for the third value.
> …

Continuing in this fashion, we see that there are `n(n-1)(n-2)…1 = n!` different permutations of an array of `n` different elements. For example, here are the 3! = 6 permutations of the array {1, 2, 3}: (1, 2, 3), (1, 3, 2), (2, 3, 1), (2, 1, 3), (3, 1, 2), (3, 2, 1).

Minimum number of comparisons needed to sort an array of n different elements

A comparison like b < c has two possible outcomes: t (true) or f (false). Two comparisons have four possible outcomes (t, t), (t, f), (f, t), (f, f). And in general, a sequence of `k` comparisons has 2^k possible outcomes.

Therefore, using `k` comparisons, one could possibly identify an element of a set of size 2^k but not an element of a larger set. For example, using one comparison, one could identify an element of a set of size 2, but not an element of a set of size ≥ 3.

Comparison sorting an array can be considered to be identifying what the input array is so that its values can be swapped into sorted order. Each different input array requires different swapping.

Consider comparison sorting an array of size `n`. We want the algorithm to make at most `f(n)` comparisons, and we want to determine a lower bound on `f(n)`. From above, we have:

- The number of permutations of a set of size `n` is `n!`.
- `f(n)` comparisons cannot identify an element of a set of size more than $2^{f(n)}$.

Therefore,

$2^{f(n)} \geq n!$, or, equivalently, $f(n) \geq \log_2(n!)$.

Now, using some math (Sterling's approximation) that is beyond the scope of CS2110 we know that

$\log_2(n!)$ is $n \log_2 n - (\log_2 3) + O(\log_2 n)$.

From this, we conclude that `f(n)`, the number of comparisons needed, is at least `n` log `n`.