Sorting

Sorting

Given a set of records (or objects), we can sort them by many different criteria. For example, a set of student/mark records that contain marks for different students in different modules could be sorted:

- in decreasing order by mark
- in alphabetic order of their surname first, then by their firstname, if there are students with the same surname
- in increasing order of module name, then by decreasing order of mark

Sort algorithms mostly work on the basis of a *comparison* function that is supplied to them that defines the order required between any two objects or records.

In some special cases, the nature of the data means that we can sort without using a comparison function.

Comparing objects in Java

Java provides two interfaces to implement comparison functions:

`Comparable`: A `Comparable` object can compare itself with another object using its `compareTo(...)` method. There can only be one such method for any class, so this should implement the default ordering of objects of this type. `x.compareTo(y)` should return a negative int, 0, or a positive int if `x` is less than, equal to, or greater than `y` respectively.

`Comparator`: A `Comparator` object can be used to compare two objects of some type using the `compare(...)` method. This does not work on the current object but rather both objects to be compared are passed as arguments. You can have many different comparison functions implemented this way. `compare(x, y)` should return a negative int, 0, or a positive int if `x` is less than, equal to, or greater than `y` respectively.

Comparison-based Sorting Strategies

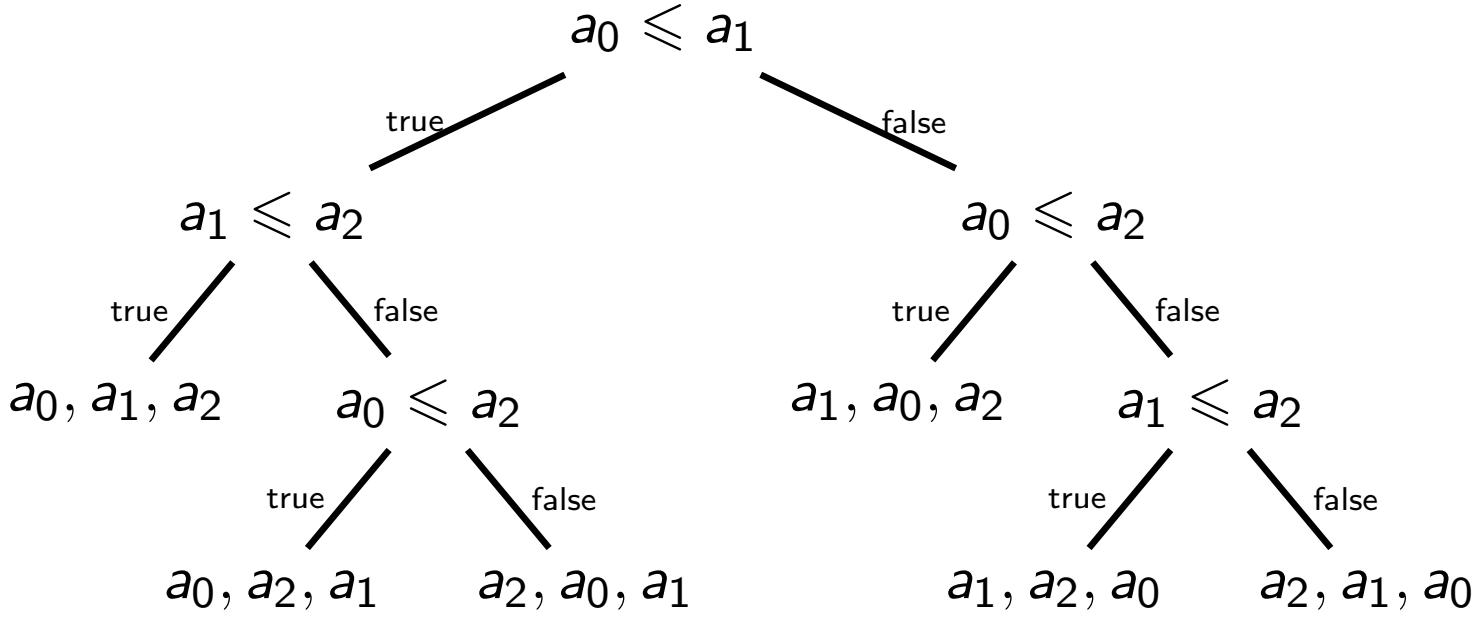
There are a number of basic strategies for comparison-based sorting, and different sorting algorithms based on each strategy:

- **Enumeration:** For each item, count the number of items less than it, say N, then put the current item at position $N + 1$.
- **Exchange:** If two items are found to be out of order, exchange them. Repeat until all items are in order.
- **Selection:** Find the smallest item, put it in position 1, find the smallest remaining item, put it in position 2, ...
- **Insertion:** Take the items one at a time and insert into an initially empty data structure such that the data structure continues to be sorted at each stage.
- **Divide and conquer:** Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted 'parts' back together in a way that preserves the sorting.

Minimum number of Comparisons

For comparison-based sorting, the minimum number of comparisons necessary to sort n items gives us a lower bound on the complexity of any comparison based sorting algorithm.

Consider an array a of 3 elements: a_0, a_1, a_2. We can make a decision tree to figure out which order the items should be in (note: no comparisons are repeated on any path from the root to a leaf):



Minimum number of Comparisons

- This decision tree is a binary tree where there is one leaf for every possible ordering of the items
- The **average** number of comparisons that are necessary to sort the items will be the average path length from the root to a leaf of the decision tree.
- The **worst case** number of comparisons that are necessary to sort the items will be the height of the decision tree.
- Given n items, there are n ways to choose the first item, $n-1$ ways to choose the second, $n-2$ ways to choose the third, etc. so there are $n(n-1)(n-2)\ldots 3 \cdot 2 \cdot 1 = n!$ different possible orderings of n items
- Thus the minimum number of comparisons necessary to sort n items is the height of a binary tree with $n!$ leaves

Minimum number of Comparisons

A binary tree of height h has the most number of leaves if all the leaves are on the bottom-most level, thus it has at most 2^h leaves.

Hence we need to find h such that

$$\begin{aligned} & 2^h \geqslant n! \\ \implies \quad & \log_2 2^h \geqslant \log_2 n! \\ \implies \quad & h \geqslant \log_2 n! \end{aligned}$$

But $\log_2 n! = \Theta(n \log n)$, thus we need at least $n \log n$ comparisons to complete a comparison based sort in general.[1]

[1] There are many ways to prove this but the easiest involves showing that $(\frac{n}{2})^{\frac{n}{2}} \leqslant n! \leqslant n^n$ and taking the log of all terms

Stability in Sorting

A *stable* sorting algorithm does not change the order of items in the input if they have the same sort key.

Thus if we have a collection of student records which is already in order by the students' first names, and we use a stable sorting algorithm to sort it by students' surnames, then all students with the same surname will still be sorted by their firstnames.

Using stable sorting algorithms in this way, we can "*pipeline*" sorting steps to construct a particular order in stages.

In particular, a stable sorting algorithm is often faster when applied to an already sorted, or nearly sorted list of items. If your input is usually nearly sorted, then you may be able to get higher performance by using a stable sorting algorithms. However, many stable sorting algorithms have higher complexity than unstable ones, so the compexities involved should be carefully checked.