Lower Complexity Bound for Sorting

Richard Hua

Semester 2, 2021

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

Sorting algorithm summary

We have looked at several different sorting algorithms so far.

Algorithm	Best	Worst	Average
Selection sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Mergesort	$n \log n$	$n \log n$	$n \log n$
Quicksort	$n \log n$	n^2	$n \log n$
Heapsort	$n \log n$	$n \log n$	$n \log n$

Many more sorting algorithms exist.

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

Continued

E.g. Quantum Bogosort;

- Generates a random permutation of the input (using some quantum mechanics);
- Checks if the list is sorted;
- Return if list is sorted;
- Otherwise, destroy the universe.

Assuming that the many-worlds interpretation is true, the use of this algorithm will result in the existence of exactly one surviving universe where the input is successfully sorted.

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

Continued

Algorithm	Best	Worst	Average
Selection sort	n^2	n^2	n^2
Insertion sort	n	n^2	n^2
Mergesort	$n \log n$	$n \log n$	$n \log n$
Quicksort	$n \log n$	n^2	$n \log n$
Heapsort	$n \log n$	$n \log n$	$n \log n$

The best worst/average case time complexity we have seen so far is $n \log n$.

Question

Can we do better? In other words, does there exist a sorting algorithm that can sort an input list of n items with time complexity $\Theta(f(n))$ in the worst/average case such that $\lim_{n \to \infty} \frac{f(n)}{n \log n} = 0$*?*

Continued

Question

Can we do better? In other words, does there exist a comparison-based sorting algorithm that can sort an input list of n items with time complexity $\Theta(f(n))$ in the worst/average case such that $\lim_{n\to\infty} \frac{f(n)}{n \log n} = 0$?

Answer: No such algorithm exist.
In this type of situations, it's often relatively easier to prove that something exists (e.g. by example, constructive proof) rather than the non-existence of such things.

Different types of techniques exist to do this kind of proofs (e.g. polynomial-time reductions, etc.).

Decision trees

Suppose we are given a list of n distinct keys, there are $n!$ possible permutations. A correct sorting algorithm must be able to sort all $n!$ permutations correctly.
Observation: Any correct sorting algorithm must be able to distinguish between all $n!$ permutations (non-explicitly).

If it can't distinguish between two permutations, say s_1 and s_2, it would rearrange s_1 and s_2 in the same way and we get at least one incorrect output.

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

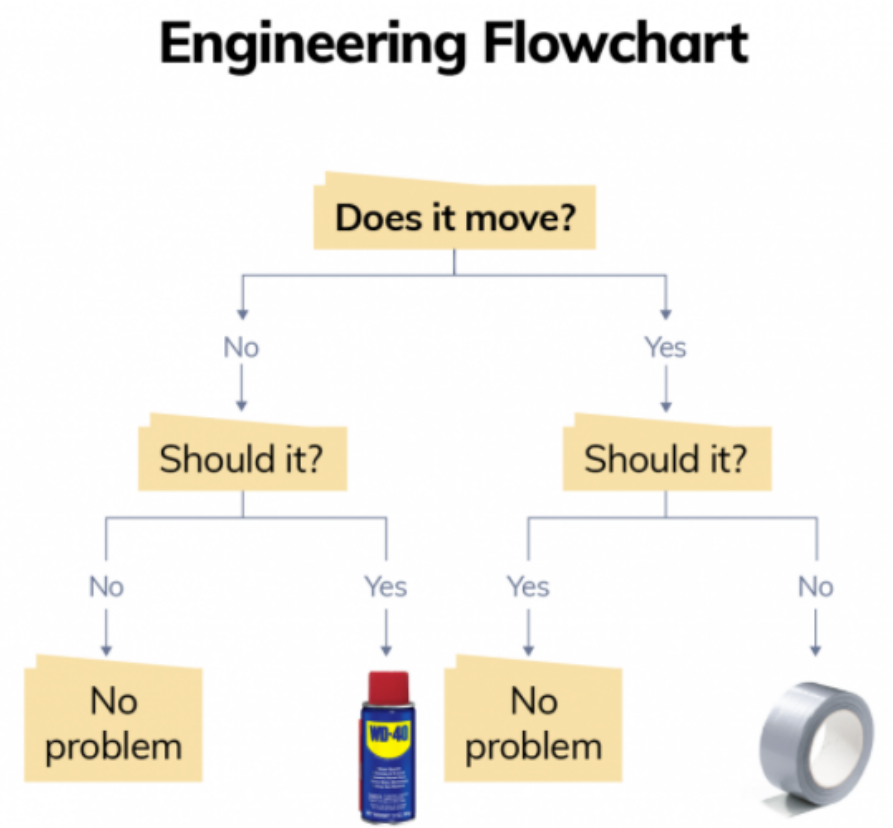
Continued

Conclusion: During the execution of the sorting algorithm, the algorithm must have gained some information about the input permutation s and acted based on these information.
We can model this by using a **decision tree**.

Decision tree

A decision tree looks like a flowchart in which internal nodes represents condition checks and each branch represents the outcome of the condition check.
Very common tool used as visual support.
Depending on the application, The leaves of the tree could be class labels (where the path represents the classification rules) or the path itself could be used to describe certain types of processes.
Used to be very popular in A.I.

Examples

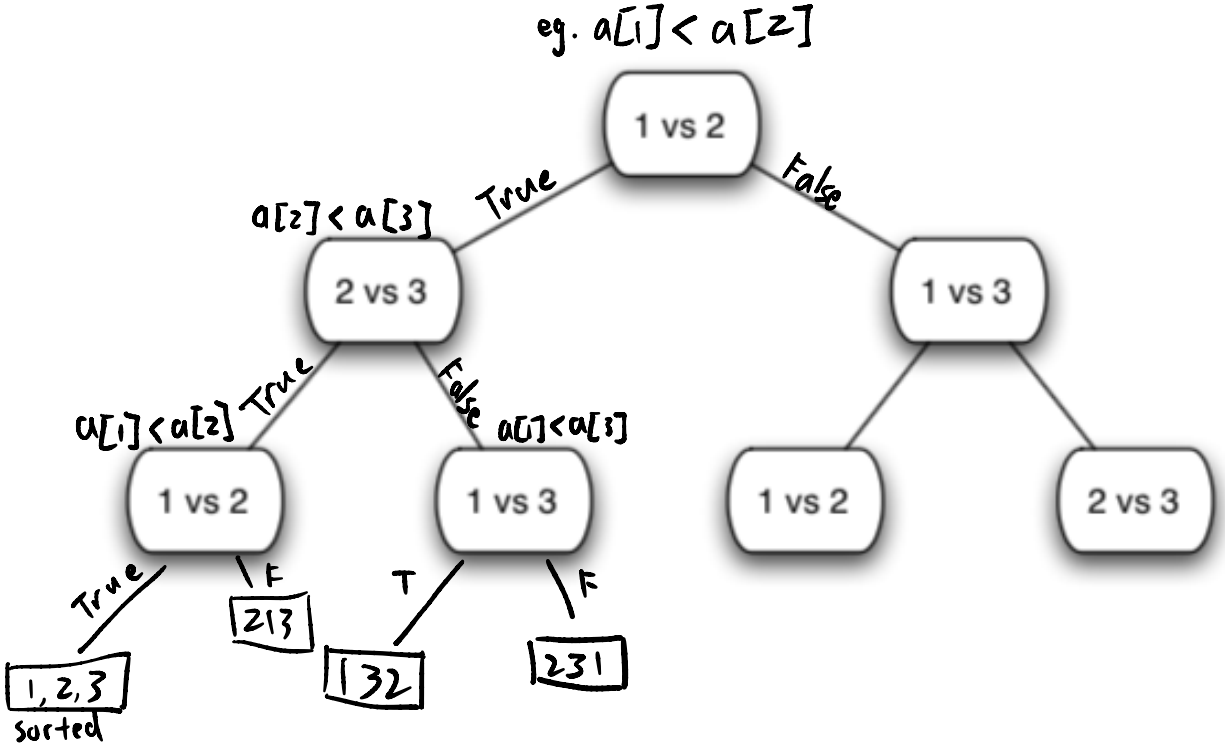

Engineering Flowchart
Does it move?
No
Yes
Should it?
Should it?
No
Yes
Yes
No
No problem
WD-40
No problem

Continued

How does the sorting algorithm gain information?
By data movement?
Only gains information from data comparisons.
The condition check is a True/False check so the decision tree is binary.
The height of the tree is the number of comparisons.

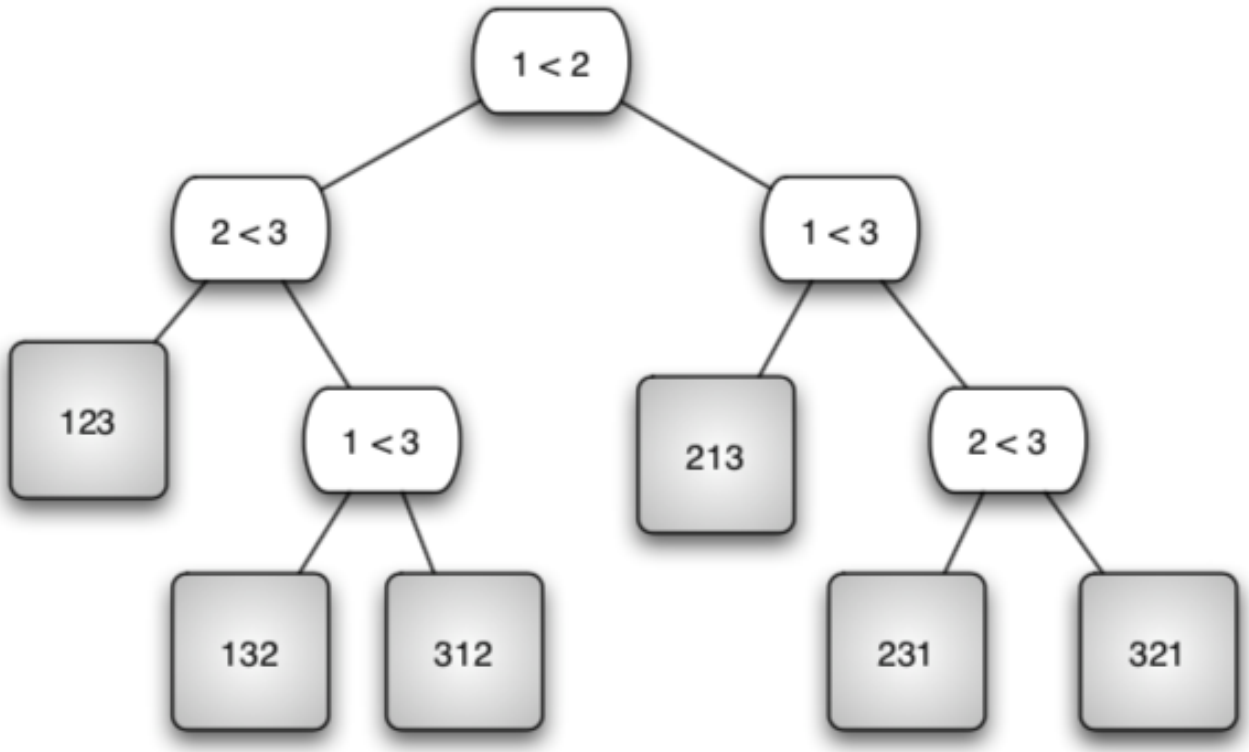
Selection sort decision tree

Selection sort decision tree when run on an input list of size 3.



Insertion sort decision tree

Insertion sort decision tree when run on the same input.



The leaves are the $n!$ permutations of the input list.

Decision tree

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ in the worst case.

Proof.

We first claim that each binary tree of height h has at most 2^h leaves. We use mathematical induction on h. A tree of height 0 obviously has at most $1 = 2^0$ leaves. Now suppose $h \geqslant 1$ and that each tree of height $h-1$ has at most 2^{h-1} (inductive hypothesis). The root of a decision tree of height h is linked to two subtrees, each with height at most $h-1$. By the inductive hypothesis, these two subtrees has at most 2^{h-1} leaves. The number of leaves in the whole decision tree is therefore at most $2 \times 2^{h-1} = 2^h$.

Continued

Proof.

The decision tree has $n!$ leaves so we must have $2^h \geqslant n!$. The least such value h must satisfy $h \geqslant \lg n!$ which is $\Omega(n \log n)$. □

What have we learned here?

Heapsort and Mergesort have asymptotically optimal worst case time complexity.

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

Continued

Question

What about the average case?

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ time in the average case.

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

Continued

Proof.

Let us show that the sum of all heights of leaves of a decision tree with k leaves is at least $k \lg k$. The smallest height is when the tree is balanced so that the number of leaves on the left subtree is equal to the number of leaves on the right subtree. Let $H(k)$ be the sum of all heights of k leaves in such a tree. Then the left and the right subtree attached to the root have $k/2$ leaves each and $H(k) = 2H(\frac{k}{2}) + k$ since we add 1 to the height of each leaf when we join the two subtrees to the root.

Continued

Proof.

$H(k) = k \lg k$. When $k = n!$, $H(n!) = n! \lg n!$. The average leaf height is $\frac{H(n!)}{n!} = \lg n! \in \Omega(n \log n)$. □

The average case time complexity of any comparison-based sorting algorithm is at least $\Omega(n \log n)$.

THE UNIVERSITY OF AUCKLAND NEW ZEALAND

Additional notes

We have shown the number of comparisons of any comparison-based sorting algorithm is bounded by $\Omega(n \log n)$ in the worst/average case.
In practice, it depends on what is exactly being sorted. E.g. very large objects are slower to move in memory than 32-bits integers.
There may exists some sorting algorithm in an **unconventional computing framework** (e.g. quantum computing, biology computing, etc.) which may be able to achieve a better time complexity.

Additional notes

Exercise

Consider the following sorting algorithm (often called counting sort*) applied to an array $a[n]$ all of whose entries are integers in range $[1, 1000]$.*

- *Construct a new array $t[1000]$, all initial values are 0's.*
- *Scan through a and each time an integer i is found, increment the counter $t[i-1]$ by 1.*
- *Loop through $0 \leqslant i \leqslant 999$ and print out $t[i]$ copies of integer $i+1$ at each step.*

What is the time complexity of counting sort? How do we reconcile this with the theorem we have proven?