COM1009
Introduction to Algorithms and Data Structures

Topic 06: Lower Bounds and Sorting in Linear Time

Essential Reading:
Chapter 8 up to page 198.

Aims of this topic

- To discuss the class of **comparison sorts**: sorting algorithms that sort by comparing elements.
- To show a general **lower bound** for the running time of this class of sorting algorithms. They can't do better than this.
- Other approaches can sometimes do better. We'll see how to sort numbers in a **bounded range** in **linear time**.

Comparison Sorts

- InsertionSort
- SelectionSort
- MergeSort
- HeapSort
- QuickSort (see Exercise Sheet 4)

- All of these by comparing elements – we call these **comparison sorts**.
- We'll see today that there's a limit to how fast they can run, but…
  - Sometimes we can go faster by using extra information

▶ Performance of Comparison Sorts

- The best comparison sorts we have seen so far take time $\Omega(n \log n)$ in the worst case.
  - InsertionSort: worst case = $\Theta(n^2)$
  - MergeSort, HeapSort: worst case = $\Theta(n \log n)$
- Can we do better?
  - Having a better algorithm could be useful
- Can we prove it's impossible to do better?
  - Stops us wasting time looking for something that doesn't exist

Complexity Theory

(very briefly - more in COM2109 Automata, Computation and Complexity)

- Deals with the difficulty of problems.
- Investigates limits to the efficiency of algorithms
  - Results like: “no algorithm can solve problem X faster than worst case time T”.
  - Stops us from wasting time trying to achieve the impossible
  - Informs the design of efficient algorithms.
- Two sides of the same coin:

Complexity theory $\leftrightarrow$ Efficient algorithms

Appetiser: P and NP

(not relevant for the assessment in COM1009, but relevant for Computer Science)

- P and NP (much more in COM2109)
  - P = problems that can be solved quickly (i.e. in polynomial worst-case time)
  - NP = problems whose solutions can be verified quickly
- Sometimes we can **verify** solutions quickly, but we can't (currently) **find** solutions quickly.
  - Example: Given that $n > 0$ is a product of two primes, find the two primes p and q that satisfy $pq = n$
    - *Finding p and q can be hard, but if you tell me p and q, I can easily multiply them together and check whether the answer equals n*
    - *Try it! See https://en.wikipedia.org/wiki/RSA_numbers*

Appetiser: NP-Completeness

(not relevant for the assessment in COM1009, but relevant for Computer Science)

- NP-complete problems (much more in COM2109)
  - >3000 important but seemingly different problems: satisfiability, scheduling, selecting, cutting, routing, packing, colouring, …
  - If you can solve any one of these problems quickly, then every other problem in NP can also be solved quickly (i.e. P = NP)
  - But can any of them be solved quickly? No one (currently) knows.
- The P vs NP Problem
  - Is every easily-verified problem also easy to solve in the first place?
  - $1m prize for a definite solution: https://www.claymath.org/millennium-problems/millennium-prize-problems

How (Not) to Show Lower Bounds

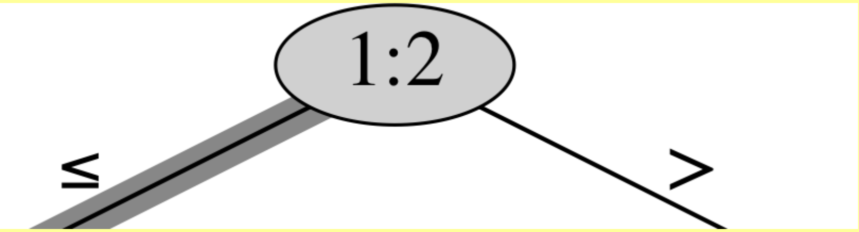
- How can we show that time $\Theta(\ldots)$ is best possible?
- *"We didn't manage to find a better algorithm."*
  - Perhaps you didn't look hard enough
- *"No one has ever found a better algorithm."*
  - How do you know this? What if tomorrow someone does?
  - We have to find arguments that apply to **all algorithms that can ever be invented**.
- *"Surely, every efficient algorithm must do things this way."*
  - Intuition is often wrong. For example, efficient algorithms for multiplying matrices start by *subtracting* elements!

Comparison Sorts as Decision Trees

- There is one thing that all comparison sorts have to do: **compare elements**
- Let's strip away all the overhead, data movement, looping, recursing, etc. and take the number of comparisons as a lower time bound.
- We assume that elements $a_1, \dots, a_n$ are distinct – then we can assume that all comparisons have the form $a_i \leq a_j$.
- A **decision tree** reflects all comparisons **a particular comparison sort** makes, and how the outcome of one comparison determines future comparisons.
  - Like a skeleton of a sorting algorithm.

Decision tree for a comparison sort

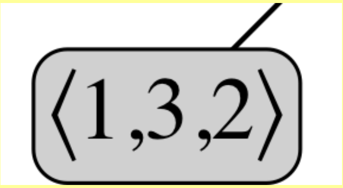
- Inner node *i:j* means comparing a_i and a_j.



- Leaves: ordering $\pi_1, \pi_2, \ldots, \pi_n$ established by the algorithm:

$$a_{\pi_1} \leq a_{\pi_2} \leq \cdots \leq a_{\pi_n}$$

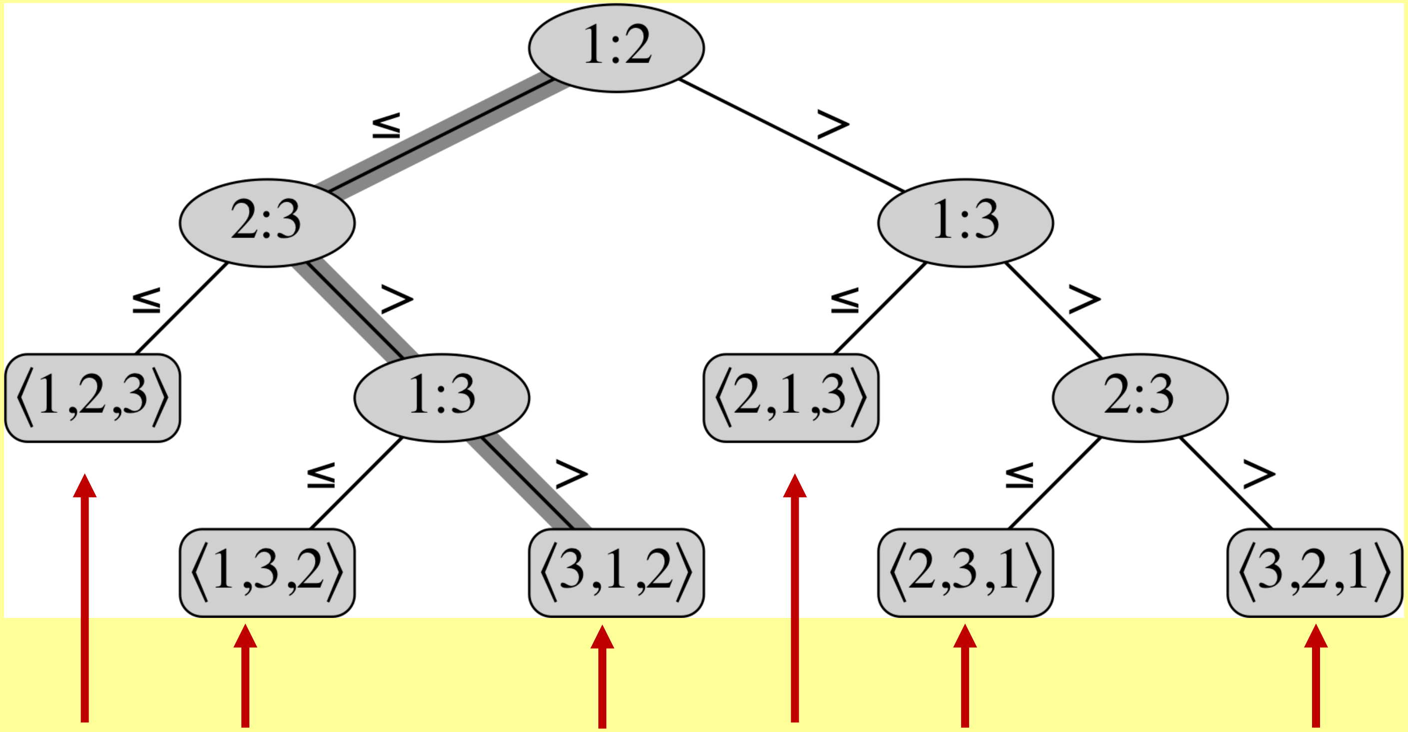
A leaf contains a sorted output for a particular input.



$$a_1 \leq a_3 \leq a_2$$

- **The execution of a sorting algorithm corresponds to tracing a simple path from the root down to a leaf.**

Example of a decision tree



The leaves include all the possible orderings of the n values, so there must be n! of them (in this example, 3! = 6)

Lower bound for comparison sorts

Theorem: Every comparison sort requires $\Omega(n \log n)$ comparisons in the worst case.

- This includes all comparison sorts that will ever be invented.
- Proof follows (see Theorem 8.1 in the book).
- The theorem can be extended towards an $\Omega(n \log n)$ bound for the **average-case time** (not done here).
- The theorem implies that HeapSort and MergeSort are asymptotically **optimal comparison sorts**. They achieve the best possible $\Omega(n \log n)$ worstcase runtime.

Proof of the lower bound

- The **worst-case number of comparisons** equals the **length of the longest simple path** from the root to any reachable leaf: we call this the **height h** of the tree (as in HeapSort).
- Every correct algorithm must be able to produce a sorted output for each of the $n!$ possible orderings of the input.
- A binary tree of height h has no more than 2^h leaves.
  - We'll prove this formally in a bit; let's take this for granted for now.
- To accommodate $n!$ leaves we need $2^h \geq n!$ And so $h \geq \log(n!)$.
- So the worst-case number of comparisons is at least log(n!)

How big is log(n!)?

- $\log(n!) = \log(n) + \log(n-1) + \ \ldots + \log(1)$
  - So $\log(n!) \leq n \log(n)$

n terms, biggest is log(*n*)

- $\log(n!) = \log(n) + \cdots + \log(n/2) + \ \ldots + \log(1)$
  - So $\log(n!) \geq \log(n) + \cdots + \log\left(\frac{n}{2}\right)$
  - So $\log(\mathrm{n}!) \geq \frac{n}{2} . \log\left(\frac{n}{2}\right) = \frac{n}{2}\log(n) - \frac{n}{2}\log(2)$

n/2 terms, smallest is log(*n*/2)

Lower order terms can be ignored

$$\boldsymbol{\log(n!) = \Theta(n \log(n))}$$

So worst case comparison sorting must take $\Omega(n \log n)$

▶ Can we do better?

- The lower bound of $\Omega(n \log n)$ is bad news for applications where comparisons are the only source of information.
- However, it suggests a way out: where possible, **use more information** than mere comparisons!
- Elements to be sorted are often **numbers or strings**, which reveal more information.
  - e.g. representing a value as a decimal (rather than a pile of pebbles) requires work; we have to represent the ones, the tens, the hundreds, ..., separately. This generates information.
  - We can use this information: is a 2-digit number bigger than a 1 digit number? We can answer this without even knowing what the numbers are.

CountingSort: Idea

- Assume that the input elements are integers in $\{0, \dots, k\}$. This is extra information – we know up-front that the inputs are no bigger than k.
- For each element x, CountingSort **counts the number of elements less than x**.
  - For instance, if 17 elements are smaller than x, then x belongs in output position 18.
  - Beware: we need to make sure that equal elements are put in **different** output positions.
- CountingSort uses an array $C[0 \dots k]$ for counting and an array $B[1 \dots n]$ for writing the output.

CountingSort

- Initialise counter array
- Count elements
- Running sum: #elements $\leq i$
- Write elements to output

COUNTINGSORT(A, B, k)	Time
let $C[0 \dots k]$ be a new array	
for $i = 0$ to k **do**	$\Theta(k)$
$\quad C[i] = 0$	
for $j = 1$ to A.length **do**	$\Theta(n)$
$\quad C[A[j]] = C[A[j]] + 1$	
for $i = 1$ to k **do**	$\Theta(k)$
$\quad C[i] = C[i] + C[i-1]$	
for $j = A$.length downto 1 **do**	
$\quad B[C[A[j]]] = A[j]$	$\Theta(n)$
$\quad C[A[j]] = C[A[j]] - 1$	

- Runtime is $\Theta(n + k)$
  - Depends on two input parameters instead of just the problem size n.
  - This is $O(n)$ if $k = O(n)$.

Stability

- CountingSort is **stable**: numbers with the same value appear in the output in **the same order as** they do **in the input** array.
  - The order of equal elements is preserved.
  - This property is relevant when **satellite data** (e.g. Java objects) is attached to keys being sorted.
  - We may think of the original order being used to break ties between elements with equal keys.
  - Works well for sorting emails according to (1) read/unread and (2) date.

Radix Sort

- Stability helps for sorting numbers digit by digit (or English words letter by letter).
- Assume that each array element has d digits (from lowest significance to highest significance)

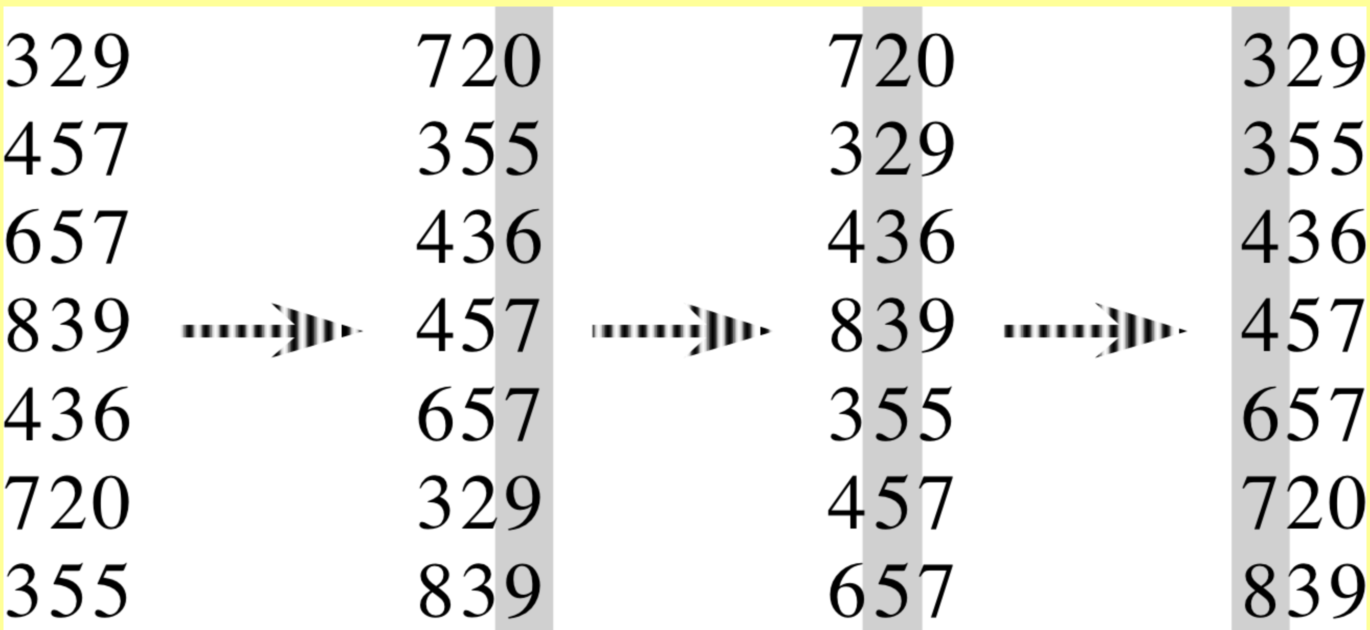
RADIXSORT(A, d)

for $i = 1$ to d **do**
    use a stable sort to sort array A on digit i

Radix Sort: Example

RadixSort(A, d)

> **for** $i = 1$ to d **do**
>
> use a stable sort to sort array A on digit i



Correctness follows from stability and induction on columns.

Radix Sort: Runtime

- Given n d-digit numbers in which each digit can take up to k possible values, RadixSort using CountingSort sorts these numbers in time $\Theta\big(d(n+k)\big)$.
  - This is just the runtime of running CountingSort d times.

Summary

- **Complexity Theory** gives limits to the efficiency of algorithms.
  - How (not) to prove lower bounds for all algorithms.
- All comparison sorts need time $\Omega(n \log n)$ in the worst case.
  - Decision trees capture the behaviour of every comparison sort.
- CountingSort sorts *n* numbers in a bounded range $\{0, \ldots, k\}$ in time $\Theta(n + k)$.
- RadixSort uses a **stable sorting algorithm** to sort digit by digit.
  - **Stability** preserves the order of equal elements.
  - The time for sorting d-digit numbers is $\Theta\big(d(n + k)\big)$.
  - This is $\Theta(n)$ when $d = O(1)$ and $k = O(n)$.