1 Computational Model of Sorting

The world is full of sorting algorithms.

- **Quicksort**: worst-case $\Theta(n^2)$, average $\Theta(n \log n)$
- **Mergesort**: worst-case $\Theta(n \log n)$
- **Heapsort**: worst-case $\Theta(n \log n)$
- **Insertionsort**: worst-case $\Theta(n^2)$

What do all these sorting algorithms have in common?

- They are all **comparison sorts**
- all based on comparing array elts to each other
- can be implemented given only this operation:
  \[
  \text{Greater}(A, i, j) \{ \text{return } A[i] > A[j] \}\]
- (in particular, don’t care about actual value in each array cell!)

2 How Fast Can Comparison Sort Run?

- worst-case time of every sort I’ve mentioned is $\Omega(n \log n)$
- want to show that no asymptotically faster comparison sort exists
- argument must cover *all* such sorts, including ones I’ve never imagined
- only info allowed:
  1. input array can be in arbitrary order
  2. output array must be sorted
  3. only permitted operations are moving elements and testing with \text{Greater}
3 Decision Trees

I don’t know many lower bound techniques, but all the ones I do know are closely tied to the idea of a decision tree. (not a recursion tree)

- **decision tree**: graphically represents all possible computations by some algorithm A on inputs of size n
- example for some algorithm A sorting three elements:

- **leaves** of tree = possible outputs
- **internal nodes** of tree = constant-time compute operations
- **edges** of tree = outcomes for each operation

Every pair \((A, n)\) has its own specific tree.

4 A Lower Bound Argument

_Idea_: lower bound related to size of decision tree

**Intuition:**

- A certain problem can produce many possible outputs.
- Our operations are of limited power – can only do a little work to choose among possible outputs at each node.
• May need to do many operations to pick an output in worst case.

How does this map onto the tree?
• tree starts with single root node
• every internal node of tree can split \( w \) ways (e.g. two for \( > \))
• suppose tree has at least \( t \) leaves (possible outputs)

**How tall must the tree be?**
• Every level increases the number of nodes by at most a factor of \( w \)
• To get minimum tree height \( h \), must solve
  \[
  w^h \geq t.
  \]
• Conclude that \( h \geq \log_w t \)
• Therefore, to produce output corresponding to deepest leaf, need \( \Omega(\log_w t) \) computations (one per node on path from root down to deepest leaf).

5 Example Argument for Sorting

**Step 1**: how many leaves in any sorting algo’s decision tree?
• output is one particular arrangement of elements
• input is *any* possible arrangement
• hence, must be able to compute every possible permutation of input (else we could not correctly sort some input)
• Given an array of size \( n \), how many ways can we permute its elements?
  • (From 240): \( n! \) (\( n \) factorial)
  • Conclude: tree has at least \( n! \) leaves

**Step 2**: how much can tree grow at each level?
• only allowed operations are comparisons
• each comparison has two outcomes: greater / not greater
• hence, each node can split in two at each level
• conclude # nodes can at most double at each level

**Step 3**: Conclude that tree height is at least \( \log(n!) \)
• How big is \( \log(n!) \) anyway?
• Want lower bound (not obvious)

• From text: use Stirling’s approximation:

\[ n! > \left(\frac{n}{e}\right)^n \]

• Therefore, we have

\[
\log n! > \log \left(\frac{n}{e}\right)^n \\
= n(\log n - \log e) \\
= n \log n - cn \\
\geq c'n \log n.
\]

• Conclude that tree height, and hence worst-case # of comparisons, is \( \Omega(n \log n) \)

• Time to sort is at least number of comparisons (could do other work, e.g. swaps)

• Hence, any sort that does only comparisons takes worst-case time \( \Omega(n \log n) \) QED

6 Implications of Lower Bound

• \( n \log n \) comparison sorts like MERGE SORT are asymptotically optimal in the worst case

• any asymptotically faster sort must use operations other than comparison (we’ll see some in a little while)

• not covered: how much can we improve constant factor?

• not covered: algorithms that can fail to sort some input arrangements (e.g. Monte Carlo methods)

7 More Lower Bounds

What about searching a sorted array? (Note: we can’t say “What about BSEARCH?” because lower bounds apply to problems, not particular algorithms!)

• General problem: given a sorted array of length \( n \), find the location of a query value \( x \) in the array (or fail)

• suppose algorithm can only compare any two array elements, or any element with \( x \) (using \( = \) or \( > \))

• how many possible outputs?

• \( x \) can be anywhere in array or nowhere: \( n + 1 \) outputs!

• allow each operation to have two outcomes: \( \leq, > \) or \( =, \neq \)

• So, we have \( w = 2, t = n + 1 \). Tree must have height at least \( \log_w t = \log_2(n + 1) \).
• Conclude that worst-case time to find an element in an array is
\[ \Omega(\log(n + 1)) = \Omega(\log n) \]

What about closest pair?

• General problem: given a collection of \( n \) points in the plane, find the closest pair

• suppose algorithm can only compute and compare any two interpoint distances (i.e.
  can test \( d(p_i, p_j) < d(p_k, p_l) \)?)

• how many possible outputs?

• any pair could be closest: \( n(n-1)/2 \) possible pairs

• each comparison can have two outcomes

• So, we have \( w = 2 \), \( t = n(n-1)/2 \). Tree must have height at least \( \log_w t = \log_2 n(n-1)/2 \).

• Conclude that worst-case time to find closest pair is \( \Omega(\log n(n-1)/2) = \Omega(\log n) \)

Whoa. Anyone know of any \( \log n \)-time closest pair algorithms? How about linear time?

• we have a gap between fastest known c.p. algorithm \( O(n \log n) \) and lower bound
  \( \Omega(\log n) \)

• true fastest algorithm might be anywhere in between

• (if we work harder, can actually show that \( n \log n \) is lower bound too)