The main problem, stated carefully

For now, assume we have \( n \) comparable elements in an array and we want to rearrange them to be in increasing order.

Input:
- An array \( A \) of data records
- A key value in each data record
- A comparison function (consistent and total)

Effect:
- Reorganize the elements of \( A \) such that for any \( i \) and \( j \), if \( i < j \) then \( A[i] \leq A[j] \)
- (Also, \( A \) must have exactly the same data it started with)
- Could also sort in reverse order, of course

An algorithm doing this is a comparison sort.
Surprising amount of neat stuff to say about sorting:

- **Simple algorithms:** $O(n^2)$
  - Insertion sort
  - Selection sort
  - Shell sort
  - ...

- **Fancier algorithms:** $O(n \log n)$
  - Heap sort
  - Merge sort
  - Quick sort (avg)
  - ...

- **Comparison lower bound:** $\Omega(n \log n)$

- **Specialized algorithms:** $O(n)$
  - Bucket sort
  - Radix sort

- **Handling huge datasets**
  - External sorting
Mergesort Analysis

Having defined an algorithm and argued it is correct, we should analyze its running time and space:

To sort \( n \) elements, we:
- Return immediately if \( n=1 \)
- Else do 2 subproblems of size \( n/2 \) and then an \( O(n) \) merge

Recurrence relation:
\[
T(1) = c_1 \\
T(n) = 2T(n/2) + c_2n
\]
One of the recurrence classics…

For simplicity let constants be 1 – no effect on asymptotic answer

\[
T(1) = 1
\]
\[
T(n) = 2T(n/2) + n
\]
\[
= 2(2T(n/4) + n/2) + n
\]
\[
= 4T(n/4) + 2n
\]
\[
= 4(2T(n/8) + n/4) + 2n
\]
\[
= 8T(n/8) + 3n
\]
\[
\ldots
\]
\[
= 2^kT(n/2^k) + kn
\]

So total is \(2^kT(n/2^k) + kn\) where \(n/2^k = 1\), i.e., \(\log n = k\)

That is, \(2^{\log n} T(1) + n \log n\)
\[
= n + n \log n
\]
\[
= O(n \log n)
\]
Or more intuitively…

This recurrence is common you just “know” it’s $O(n \log n)$

Merge sort is relatively easy to intuit (best, worst, and average):
- The recursion “tree” will have $\log n$ height
- At each level we do a total amount of merging equal to $n$
Quicksort

- Also uses divide-and-conquer
  - Recursively chop into two pieces
  - Instead of doing all the work as we merge together, we will do all the work as we recursively split into halves
  - Unlike merge sort, does not need auxiliary space

- \( O(n \log n) \) on average 😊, but \( O(n^2) \) worst-case 😞

- Faster than merge sort in practice?
  - Often believed so
  - Does fewer copies and more comparisons, so it depends on the relative cost of these two operations!
Quicksort Overview

1. Pick a pivot element

2. Partition all the data into:
   A. The elements less than the pivot
   B. The pivot
   C. The elements greater than the pivot

3. Recursively sort A and C

4. The answer is, “as simple as A, B, C”

(Alas, there are some details lurking in this algorithm)
Think in Terms of Sets

select pivot value

partition S

Quicksort(S₁) and Quicksort(S₂)

Presto! S is sorted

[Weiss]
Example, Showing Recursion

Divide

Divide

Divide

1 Element

Conquer

Conquer

Conquer
Details

Have not yet explained:

• How to pick the pivot element
  – Any choice is correct: data will end up sorted
  – But as analysis will show, want the two partitions to be about equal in size

• How to implement partitioning
  – In linear time
  – In place
Pivots

• Best pivot?
  – Median
  – Halve each time

• Worst pivot?
  – Greatest/least element
  – Problem of size n - 1
  – $O(n^2)$
Potential pivot rules

While sorting \texttt{arr} from \texttt{lo} (inclusive) to \texttt{hi} (exclusive)...

- Pick \texttt{arr[lo]} or \texttt{arr[hi-1]}
  - Fast, but worst-case occurs with mostly sorted input

- Pick random element in the range
  - Does as well as any technique, but (pseudo)random number generation can be slow
  - Still probably the most elegant approach

- Median of 3, e.g., \texttt{arr[lo]}, \texttt{arr[hi-1]}, \texttt{arr[(hi+lo)/2]}
  - Common heuristic that tends to work well
Partitioning

• Conceptually simple, but hardest part to code up correctly
  – After picking pivot, need to partition in linear time in place

• One approach (there are slightly fancier ones):
  1. Swap pivot with arr[lo]
  2. Use two fingers i and j, starting at lo+1 and hi-1
  3. while (i < j)
     if (arr[j] > pivot) j--
     else if (arr[i] < pivot) i++
     else swap arr[i] with arr[j]
  4. Swap pivot with arr[i] *

*skip step 4 if pivot ends up being least element
Example

- Step one: pick pivot as median of 3
  - \( l_0 = 0, h_i = 10 \)

```
0 1 2 3 4 5 6 7 8 9
8 1 4 9 0 3 5 2 7 6
```

- Step two: move pivot to the \( l_0 \) position

```
0 1 2 3 4 5 6 7 8 9
6 1 4 9 0 3 5 2 7 8
```
Example

Now partition in place

Move fingers

Swap

Move fingers

Move pivot

Often have more than one swap during partition – this is a short example
Analysis

• Best-case: Pivot is always the median
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = 2T(n/2) + n \] -- linear-time partition
  Same recurrence as mergesort: \( O(n \log n) \)

• Worst-case: Pivot is always smallest or largest element
  \[ T(0) = T(1) = 1 \]
  \[ T(n) = T(n-1) + n \]
  Basically same recurrence as selection sort: \( O(n^2) \)

• Average-case (e.g., with random pivot)
  – \( O(n \log n) \), not responsible for proof (in text)
Cutoffs

• For small $n$, all that recursion tends to cost more than doing a quadratic sort
  – Remember asymptotic complexity is for large $n$

• Common engineering technique: switch algorithm below a cutoff
  – Reasonable rule of thumb: use insertion sort for $n < 10$

• Notes:
  – Could also use a cutoff for merge sort
  – Cutoffs are also the norm with parallel algorithms
    • Switch to sequential algorithm
  – None of this affects asymptotic complexity
Cutoff skeleton

```java
void quicksort(int[] arr, int lo, int hi) {
    if (hi - lo < CUTOFF)
        insertionSort(arr, lo, hi);
    else
        ...
}
```

Notice how this cuts out the vast majority of the recursive calls
– Think of the recursive calls to quicksort as a tree
– Trims out the bottom layers of the tree
Visualizations

How Fast Can We Sort?

- Heapsort & mergesort have $O(n \log n)$ worst-case running time
- Quicksort has $O(n \log n)$ average-case running time
- These bounds are all tight, actually $\Theta(n \log n)$
- So maybe we need to dream up another algorithm with a lower asymptotic complexity, such as $O(n)$ or $O(n \log \log n)$
  - Instead: we know that this is impossible
    - Assuming our comparison model: The only operation an algorithm can perform on data items is a 2-element comparison
A General View of Sorting

• Assume we have \( n \) elements to sort
  – For simplicity, assume none are equal (no duplicates)

• How many permutations of the elements (possible orderings)?

• Example, \( n=3 \)
  
  \[
  \begin{align*}
  \end{align*}
  \]

• In general, \( n \) choices for least element, \( n-1 \) for next, \( n-2 \) for next, …
  – \( n(n-1)(n-2)\ldots(2)(1) = n! \) possible orderings
Counting Comparisons

• So every sorting algorithm has to “find” the right answer among the $n!$ possible answers
  – Starts “knowing nothing”, “anything is possible”
  – Gains information with each comparison
  – Intuition: Each comparison can at best eliminate half the remaining possibilities
  – Must narrow answer down to a single possibility

• What we can show:
  Any sorting algorithm must do at least $(1/2)n \log n - (1/2)n$ (which is $\Omega(n \log n)$) comparisons
  – Otherwise there are at least two permutations among the $n!$ possible that cannot yet be distinguished, so the algorithm would have to guess and could be wrong [incorrect algorithm]
Optional: Counting Comparisons

- Don’t know what the algorithm is, but it cannot make progress without doing comparisons
  - Eventually does a first comparison “is $a < b$ ?"
  - Can use the result to decide what second comparison to do
  - Etc.: comparison $k$ can be chosen based on first $k-1$ results

- Can represent this process as a decision tree
  - Nodes contain “set of remaining possibilities”
    - Root: None of the $n!$ options yet eliminated
  - Edges are “answers from a comparison”
  - The algorithm does not actually build the tree; it’s what our proof uses to represent “the most the algorithm could know so far” as the algorithm progresses
Optional: One Decision Tree for $n=3$

- The leaves contain all the possible orderings of $a$, $b$, $c$
- A different algorithm would lead to a different tree
Optional: Example if $a < c < b$

Possible orders:
- $a < b < c$, $b < c < a$
- $a < c < b$, $c < a < b$
- $b < a < c$, $c < b < a$

Actual order:
- $b < c < a$
- $b < a < c$
**Optional: What the Decision Tree Tells Us**

- A binary tree because each comparison has 2 outcomes
  - (We assume no duplicate elements)
  - (Would have 1 outcome if algorithm asks redundant questions)

- Because any data is possible, any algorithm needs to ask enough questions to produce all $n!$ answers
  - Each answer is a different leaf
  - So the tree must be big enough to have $n!$ leaves
  - Running *any* algorithm on *any* input will *at best* correspond to a root-to-leaf path in *some* decision tree with $n!$ leaves
  - So no algorithm can have worst-case running time better than the height of a tree with $n!$ leaves

- Worst-case number-of-comparisons for an algorithm is an input leading to a longest path in algorithm’s decision tree
Optional: Where are we

• Proven: No comparison sort can have worst-case running time better than the height of a binary tree with $n!$ leaves
  – A comparison sort could be worse than this height, but it cannot be better

• Now: a binary tree with $n!$ leaves has height $\Omega(n \log n)$
  – Height could be more, but cannot be less
  – Factorial function grows very quickly

• Conclusion: Comparison sorting is $\Omega(n \log n)$
  – An amazing computer-science result: proves all the clever programming in the world cannot comparison-sort in linear time
Optional: Height lower bound

The height of a binary tree with \( L \) leaves is at least \( \log_2 L \)

So the height of our decision tree, \( h \):

\[
\begin{align*}
h & \geq \log_2 (n!) \\
& = \log_2 (n*(n-1)*(n-2)...(2)(1)) \\
& = \log_2 n + \log_2 (n-1) + \ldots + \log_2 1 \\
& \geq \log_2 n + \log_2 (n-1) + \ldots + \log_2 (n/2) \quad \text{drop smaller terms (\( \geq 0 \))} \\
& \geq \log_2 (n/2) + \log_2 (n/2) + \ldots + \log_2 (n/2) \quad \text{shrink terms to} \ \log_2 (n/2) \\
& = (n/2)\log_2 (n/2) \\
& = (n/2)(\log_2 n - \log_2 2) \\
& = (1/2)n\log_2 n - (1/2)n \\
& "=\ " \Omega (n \log n)
\end{align*}
\]