28: More Sorting

Mergesort review
analysis

Lower bound on comparison-based sorting
Mergesort: A different approach to sorting

- Divide and conquer
  - So far we’ve divided into an item and a shorter list
  - Alternative: *divide into two equal-sized lists*
  - Huge performance improvement
    - Used across computer science!
Merge sort idea

• Divide input into two lists somehow
• Sort each one
• “merge” together the results
• Can be made "stable" so that sorting (1,A) (2, Q) (1, C), based on the "int" part, gives (1,A) (1, C) (2, Q) instead of (1, C) (1, A) (2, Q).
Revised recursive diagram

• Input: two sorted lists
• Output: sorted list containing all items of both lists
• Original input: [1; 3; 6] [2; 7; 8]
  • Recursive input: [3; 6] [2; 7; 8]
  • Recursive output: [2; 3; 6; 7; 8]
  Cons smaller of two heads onto result of merging everything else

• Overall output: [1; 2; 6; 7; 8]
let rec merge: (list(int), list(int)) => list(int) = (sl1, sl2) =>
  switch (sl1, sl2) {
    | ([], _) => sl2
    | (_, []) => sl1
    | ([hd1, ...tl1], [hd2, ...tl2]) =>
      if (hd1 < hd2) {
        [hd1, ...merge(tl1, sl2)];
      } else {
        [hd2, ...merge(sl1, tl2)];
      }
  }

NB: The use of <= instead of < helps make the sort "stable"

Let M(n) be operation count for merge on two lists whose total number of items is no more than n. Recurrence for M(n)

\[ M(0) = A \]
\[ M(1) = B \]
\[ M(n) \leq C + M(n - 1) \]

Big-O class of M?

O(n \rightarrow n)

Why not use "M(n) = op count for merge on two lists, each of no more than n items"?
/* split into approx. equal-sized parts */
let rec split = (aloi: list(int)): (list(int), list(int)) =>
    switch (aloi) {
    | [] => ([], [])
    | [n] => ([n], [])
    | [a, b, ...rest] =>
        let (s1, s2) = split(rest);
        ([a, ...s1], [b, ...s2]);
    }

let rec mergeSort: list(int) => list(int) = fun
| [] => []
| [n] => [n]
| aloi => {
    let (part1, part2) = split(aloi);
    merge(mergeSort(part1), mergeSort(part2))
};

• Obs: runtime of “split” is in O(n → n)
• **This** split is not ideal – it can't be used for stable sorting
Analysis

let rec mergeSort: list(int) => list(int) = fun
 | [] => []
 | [n] => [n]
 | aloi => {
 let (part1, part2) = split(aloi);
 merge(mergeSort(part1), mergeSort(part2))
 };

• Let $M(n)$ be worst runtime for mergesort on n-item list; $Q(n)$ be worst-case for merge on total of n items. Then...

\[
M(0) = C \\
M(1) = D \\
M(n) \leq 2M\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + Q(n) \\
M(n) \leq 2M\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + An + B
\]
\[ M(n) \leq 2 \ M \left( \left\lfloor \frac{n}{2} \right\rfloor \right) + An + B \]

• This week's homework shows \( M \in O(n \mapsto n \log n) \)
Total work diagrams for analysis

Green: Work at each level to do split.
Red: work at each level to do merge.
To split a list, it takes $n$ time, where $n$ is the length of the list. So at each level, it takes 8 amount of work to do split. Additionally, in order to merge a list, it takes $n$ time, where $n$ is the length of the list. So at each level, it takes 8 amount of work to do merge. There are $\log n$ levels, because the starting list is length 8, and $\log 8 = 3$. So, it takes $16 \log 8$ amount of work for mergesort, or $2n \log n$. So merge sort is in $O(n \to n \log n)$. 

1*8 = 8

4+4 = 8
2+2+2+ 2 = 8
1*8 = 8

[2; 8]  [3; 1]  [4; 6]  [2; 5]
"Paths through the mergesort code"

• Think back to the "length" procedure.
• In each recursive call there was a "cond" that made one of two choices.
• We could draw a picture to indicate possible ways that "length" might work
Paths through length
Same deal for sorting using <

• Every possible input consisting of the numbers 1...n in some order corresponds to a path through a tree of choices

• One "choice" for each comparison (e.g., if hd1 < hd2)

• If we imagine taking that same "path" through the code, ignoring the actual data, the input data ends up shuffled by the time we get to the output

• For every possible shuffling of the input data, there must be a different path!

• If there are k possible shuffles, the tree must have at least k leaves.
Paths through sorting algorithm
How many ways to shuffle $n$ items?

- Let $S(n)$ denote the number of ways to shuffle $n$ items.
- Shuffle $n-1$ of them; then place the $n$th one in one of $n$ positions.

$S(1) = 1$

$S(n) = n \cdot S(n-1)$ for $n > 1$

Conclusion: $S(n) = n!$
Recall earlier result about trees

- If a (binary) tree has depth $n$, it has at most $2^n - 1$ nodes
- Useful fact: a binary tree with $n$ nodes, that terminates with leaves, always has $n + 1$ leaves. (well-ordering proof works nicely here)
- A binary tree of depth $n$ has at most $2^n$ leaves.
- A binary tree with $2^n$ leaves has depth at least $n$.
- A binary tree with $K$ leaves has depth at least $\log K$. 
Facts

• Our "execution tree" for sorting a shuffling of 1…n has leaves
• A tree with leaves has depth at least \( \log \)
• Our execution tree has depth at least
• How big is that?
Estimating $\log n!$

\[ n! = n(n - 1)(n - 2) \cdots 2 \cdot 1 \]

\[ n! \geq n(n - 1)(n - 2) \cdots \left(\frac{n}{2}\right) \]

\[ n! \geq \left(\frac{n}{2}\right)^{n/2} \]

\[ \log n! \geq \frac{n}{2} \log \frac{n}{2} \]

\[ \log n! \geq \frac{1}{2} (\log(n) - 1) \]

\[ \log n! \in \Omega(n \mapsto n \log n) \]
• $\log n! \in \Omega(n \mapsto n \log n)$
• So the depth of our "execution" tree is at least proportional to $n \log n$
• Each node represents a comparison, i.e., an operation that takes time $O(1)$.
• So sorting $n$ items, using comparisons, takes time at least $n \log n$. (!)