1 Mergesort

1.1 Mergesort Review

Mergesort uses the algorithmic technique of divide-and-conquer. So far, in writing procedures that operate on lists, we have generally followed our template for list recursion, performing some operation on the head of the list and recurring on the tail of the list. This recursive decomposition of the list naturally follows the structure of our data: a list is either empty or the cons of an element onto a list.

Now, a list of length \( n - 1 \) is certainly smaller than a list of length \( n \), as a list of length \( n - 2 \) is smaller than a list of length \( n - 1 \). However, it’s not much smaller. In order to write a quicker procedure, we want to decompose our list in such a way that it gets smaller quickly. How can we do that? Well, let’s try breaking it in half.

This alternative approach to divide the list into two equal-sized lists gives a huge performance improvement. This idea of dividing a problem into two equally sized sub-problems, solving each of them, and combining them, is used across computer science.
The general merge_sort idea is to first divide the input into two lists. Then, we sort each one, and “merge” together the results. Mergesort can also be made “stable” so that sorting a list of tuples \([(1, “A”); (2, “Q”); (1, “C”)]\) based on the int part gives \([(1, “A”); (1, “C”); (2, “Q”)]\) instead of \([(1, “C”); (1, “A”); (2, “Q”)]\).

Our revised recursive diagram for the procedure merge is:

Input: two sorted lists
Output: sorted list containing all items of both lists
Original Input: \([1; 3; 6]\) \([2; 7; 8]\)
Recursive Input: \([3; 6]\) \([2; 7; 8]\)

Idea: cons smaller of two heads onto result of merging everything else
Overall Output: \([1; 2; 6; 7; 8]\)

The recursive input is the same two original lists but one of them, the one with the smaller head, has its head removed.

The code for merge looks like this:

```ocaml
let rec merge (sl1 : int list) (sl2 : int list) : int list =
  match (sl1, sl2) with
  | [], _ -> sl2
  | _, [] -> sl1
  | hd1 :: tl1, hd2 :: tl2 -> if (hd1 <= hd2)
    then
      hd1 :: (merge tl1 sl2)
    else
      hd2 :: (merge sl1 tl2);;
```

This is made stable by using \(\leq\) instead of <.

What is the runtime of merge? If \(n\) is the length of both of the lists combined, the procedure runs in linear time, because you are recurring on a total combined list of length \(n - 1\) each time, and cons-ing on a head.

The code for merge_sort is:

```ocaml
(* Input: a list of integers, aloi
 * Output: aloi, sorted in ascending order *)
let rec merge_sort (aloi: int list) =
  match aloi with
  | [] -> []
  | [n] -> [n]
  | _ -> let
    (part1, part2) = split(aloi)
    in
    merge (merge_sort part1) (merge_sort part2);;
```

For split, we could find the length of the list, and take the first \(\frac{n}{2}\) items to get the first list, and drop the first \(\frac{n}{2}\) items to get the second list. Here is another way to do this.
This split is not ideal. It cannot be used for stable sorting; you’ll replace it during this week’s lab.
What is the runtime of split? The runtime of split of linear with respect to the length of the list because we are taking 2 elements off the list, and then recurring on the rest of the rest of the list.

2 Mergesort Analysis

Let $M(n)$ be the worst runtime for merge_sort on an $n$-item list.

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

This is because we have to apply merge_sort to two lists of size $\frac{n}{2}$, so we have two recursive calls. We have to apply merge on the two lists which runs in linear time with respect to $n$, so we can replace $Q(n)$ with $An + B$. This week’s homework shows that $M \in O(n \log n)$.

2.1 Total Work Diagrams

In class, Spike drew a total work diagram to analyze the runtime of merge_sort. Look at the lecture slides for this diagram! This diagram showed that in order to split a list at each level, it takes $n$ time. So at each level, it takes an amount of work proportional to $n$ to split all the lists. Then, to merge two lists, it also takes an amount of work proportional to $n$ at each level to merge the lists. This showed that merge_sort takes $2n \log n$ to sort a list of size $n$, because each level takes an amount of work proportional to $n$, and there are $\log n$ levels. Therefore, merge_sort is in $O(n \log n)$.

2.2 Paths through mergesort Code

Can we sort $n$ items in linear time?

In class, we learned about drawing diagrams to show the number of paths through the length procedure. Thinking back to the length procedure, in each recursive call, there was a cond expression that made one of two choices: empty or cons. We could draw a picture to indicate the possible ways that length might work. Look in the lecture slides for this diagram! You can think of applying length to some list as giving you a path through this diagram.

Similarly for merge_sort, we drew a diagram to show the number of paths through the merge_sort code. Look at the lecture slides for this diagram! In merge_sort, every possible input consisting of
the numbers 1, 2, ... , n in any order corresponds to a path through a tree of choices. This is because in the process of `merge_sort`, in `merge` procedure, there is one “choice” for each comparison (`if hd1 <= hd2`). So in the tree, you can think of each node as a decision point for a comparison we are making. Therefore, each different permutation of the input data requires a different shuffling in order to get to the sorted list. Each path represents a different shuffling because if we imagine taking each specific “path” through the code with the same input data, the input data ends up shuffled in a different way for each path.

Because of this, for every possible shuffling of the input data, there must be a different path. So, if there are k possible shuffles, the tree must have at least k leaves.

How many ways are there to shuffle n items? Well, we can shuffle n−1 of the items, and then place the nth item in one of n positions between the other n−1 items.

\[
S(1) = 1
\]

\[
S(n) = n \times S(n-1) \text{ for } n > 1
\]

So, \( S(n) = n! \). The number of ways to shuffle n items is n!. Therefore, the tree must have at least n! leaves.

Recall our earlier result about trees. If a binary tree has depth n, it has at most \( 2^n - 1 \) nodes. Additionally, a binary tree with n nodes, that terminates with leaves, always has n + 1 leaves. So a binary tree of depth n has at most \( 2^n \) leaves. Or, put another way, a binary tree with \( 2^n \) leaves has depth at least \( \log n \).

So our “execution tree” for mergesort on a shuffling of 1...n has n! leaves. A tree of k leaves has as depth at least \( \log k \). Because of this, our execution tree has depth at least \( \log(n!) \).

In order to estimate \( \log(n!) \):

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

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

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

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

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

\[
\log n! \in \Omega(n \mapsto n \log n)
\]

From step 1 to step 2, this is because instead of multiplying from \( n \cdot (n-1) \) ... all the way down to 1, we are multiplying down to \( \left(\frac{n}{2}\right) \), which will be less than the result of multiplying down to 1. From step 2 to step 3, this is because \( n, n-1, \) etc are all greater than \( \left(\frac{n}{2}\right) \). So step 3 is less than the result from step 2. From step 3 to step 4, we simplify the expression using exponents. From step 4 to step 5, we take the log of both sides, and use the log exponent rule. From step 5 to step 6, we use the log division rule. From step 6 to step 7, since \( \log n! \) is greater than the right side, it is 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 1. This means that in the course of sorting n items, you took at least \( n \log n \) time. So sorting n items, using comparisons, takes time at least \( n \log n \).
This result shows that for all comparison based sorting, it takes at least time $n \log n$.

3 Evaluation Review

In class, we reviewed examples of how we evaluate \texttt{let} expressions and \texttt{lambda} expressions. Look at the lecture slides for these examples and explanations of how they are evaluated!

The important explanation here is that for a \texttt{lambda} expression, it evaluates to a closure with 3 parts, the arg list, the body, and the local environment. The local environment contains all the bindings of the current environment not in the TLE.

Additionally, for a procedure-application-expression, if the procedure evaluates to a user defined proc, we evaluate the actual arguments. Then, we start from the TLE, and create new temporary bindings from the closure’s environment. Then, we add the new bindings from the formal arguments in the closure’s arg list to the values of the actual arguments. Then, we evaluate the body of the closure in this new extended environment. Finally, we remove all the temporary bindings.

4 Summary

Ideas

- We analyzed the runtime of mergesort to be in $O(n \rightarrow n\log n)$.

Skills

- We implemented mergesort. Mergesort works by dividing the input into two lists, sorting them, and merging them back together.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)