Contents

Objectives

By the end of this lecture, you will know:

- how to write the mergesort algorithm
- the run time of the mergesort algorithm

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: (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)];
      }
  };
```

This is made stable by using ≤ instead of <. It is more stable because it nice to not swap elements that don’t need to be swapped. 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.

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

\[
M(0) = 0 \quad M(1) = B \quad M(n) \leq C + M(n - 1) \quad \text{for } n > 1
\]

But is mergesort linear? If the list is empty or one element it is fast but if it is longer then we have to call mergesort recursively on each part then merge them. The code for `merge_sort` is:

```ocaml
(* Input: a list of integers, aloi
 * Output: aloi, sorted in ascending order *)
let rec mergeSort: list(int) => list(int) = fun
  | [] => []
  | [n] => [n]
  | aloi => {
    let (part1, part2) = split(aloi);
    merge(mergeSort(part1), mergeSort(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. It is offensive to compute the length of the list at every computation. Here is another way to do this like dealing a deck of cards into two piles.

```ocaml
(* Input: a list of integers, aloi
 * Output: a tuple of two lists, each with half of the contents*)
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]);
); 

This split is not ideal. It cannot be used for stable sorting; using take and drop is better. 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 $An + B$. Given: $M(1) = 6$ and $M(n) \leq 2M\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n$

We can plug and chug...

\begin{align}
  M(2) &\leq 2 \cdot M(1) + 2 = 2 \cdot 6 + 2 \\
  M(4) &\leq 2 \cdot M(2) + 4 = 2 \cdot (2 \cdot 6 + 2) + 4 \\
  M(8) &\leq 2 \cdot M(4) + 8 = 2 \cdot (2 \cdot (2 \cdot 6 + 2) + 4) + 8 \\
  M(16) &\leq 2 \cdot M(8) + 16 = 2 \cdot (2 \cdot (2 \cdot (2 \cdot 6 + 2) + 4) + 8) + 16
\end{align}

We can combine this to get: $2^4 \cdot 6 + 4(2^4)$. Remember $n = 16 = 2^4$ so, $6 \cdot n + n \log(n)$

$M \in O(n \mapsto n \log n)$

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 $O(n \log n)$. 

3
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 \( \text{hd1} \leq \text{hd2} \)). 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. Permutations of \( n \)-item input list is the number of paths from root to leg in tree.

How many ways are there to shuffle \( n \) items? Well, we can shuffle \( n - 1 \) of the items, and then place the \( n \)th item in one of \( n \) positions between the other \( n - 1 \) items.

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

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 \( n \). So, a binary tree with \( k \) leaves has depth at least \( \log k \).

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 $\log(n!)$:

\begin{align*}
    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) \\
    n! &\geq \left(\frac{n}{2}\right)^{\frac{n}{2}} \\
    \log n! &\geq \frac{n}{2} \log \frac{n}{2} \\
    \log n! &\geq \frac{n}{2} \log(n) - 1 \\
    \log n! &\in \Omega(n \mapsto n \log n)
\end{align*}

From step 1 to step 2, this is because instead of multiplying from $n \cdot (n - 1) \cdots$ 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 $n, n - 1, \text{etc}$ 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 Summary

#### Ideas

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

#### 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 CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)