Lecture 17: Trees and Merge Sort
10:00 AM, Oct 15, 2018

Contents

1 Tree definitions
2 Analysis of mergesort using a binary tree
3 Analysis of mergesort using induction
4 The importance of $O(n \log n)$
5 Returning to analysis of recurrence relation
6 Lower bound on comparison sorting

1 Tree definitions

Let’s begin by reviewing some tree terminology. Here are some terms:

- node: an entity in a tree that contains a value or data
- edge: branches that connect one node to another
- root: the topmost node in the tree
- leaves: nodes that do not have child nodes
- internal node: any node of a tree that has child nodes (any nodes that are not leaves)
- children: nodes that have parent nodes (all nodes are children except for the root node)
- parents: nodes that have at least one edge to a child node
- depth: depth of a node is the length of the path to its root
- arity: maximum number of children (binary tree has arity two)

## 2 Analysis of mergesort using a binary tree

Sometimes we consider the order of children significant. This is the case of binary trees, which come up all the time in computer science.

A nonempty binary tree has a root, and possibly a left subtree and possibly a right subtree.

In lab you saw one use for binary trees. Here we use binary trees to help us analyze mergesort.

As we went over briefly in last lecture, consider the tree representing the invocations resulting from an initial call to `mergesort` with a list of \( n \) numbers. The root represents the initial call. That initial call makes two recursive calls, one with the first half of the list (represented by the left child) and one with the second half of the list (represented by the right child). The recursions continue similarly, until you get to calls for lists of size one or zero.

At each depth \( d \), for each item in the original input list, there is at most one invocation at that depth that handles that item. Therefore, for any depth \( d \), the sum over the invocations of depth \( d \) of the number of items handled is at most \( n \).

In each invocation, the number of operations (aside from recursive invocations) is proportional to the number of items in the input list. Therefore, for each depth \( d \), the total number of operations done by invocations at depth \( d \) is at most \( cn \) where \( c \) is a constant and \( n \) is the size of the original input.

Therefore the total number of operations is at most \( cn \) times the depth of the tree.

How large is the depth of the tree? Every time you go from parent to child, the number of items in the input list goes down by a factor of two (the number of items that you are dealing with is halved). This means that in order to find our maximum depth, we must ask "How many times can you divide \( n \) by two before it gets down to 1 or less?"

This is the same as asking the question: how many 2’s do you have to multiply together before the product is \( n \) or greater?

That is, what is the smallest integer \( x \) such that \( 2^x \geq n \)?

The inverse of the function that maps \( x \) to \( 2^x \) is the function that maps \( y \) to \( \log_2 y \).

<table>
<thead>
<tr>
<th>( x )</th>
<th>( 2^x )</th>
<th>( y )</th>
<th>( \log_2 y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>20</td>
<td>1048576</td>
<td>20</td>
<td>1048576</td>
</tr>
</tbody>
</table>

These functions also take real numbers, e.g. \( 2^{1.5} = 11.313708... \) and \( \log_2 100.5 = 6.65105... \).

Starting with the equation \( 2^x = n \), we take logs of both sides, and we get \( x = \log_2 n \). Now \( \log_2 n \) is
not necessarily an integer, so the depth of the tree is \( \lceil \log_2 n \rceil \).

This shows that the total number of operations is at most \( cn \cdot \log_2 n \). Therefore we say that mergesort is \( O(n \log_2 n) \).

### 3 Analysis of mergesort using induction

Now that we have an idea of what the running time should be, we can use a recurrence relation and induction to prove it.

Let’s use the following recurrence relation. It’s a little sketchy but we’ll come back to that.

Let \( f(n) \) be the worst-case number of operations done by mergesort on a list of length \( n \).

Aside from the recursive calls, the number of operations done by mergesort is \( O(n) \). That is, there are positive constants \( n_0 \) and \( c \) such that, for \( n \geq n_0 \), the number of operations is at most \( cn \) (again, not counting recursive calls.)

\[
\begin{align*}
f(n) &\leq a \text{ for } n \leq n_0 \\
f(n) &\leq cn + f(n/2) + f(n/2) \text{ for } n \geq n_0
\end{align*}
\]

Note that \( f(n/2) \) occurs twice on the right-hand side because there are two recursive calls. That seems good. Anything wrong with this recurrence relation?

What if \( n \) is an odd number? Then the recurrence relation is not quite right. So for now let’s only state it for even \( n \):

\[
\begin{align*}
f(n) &\leq a \text{ for } n \leq n_0 \\
f(n) &\leq cn + f(n/2) + f(n/2) \text{ for even } n > n_0
\end{align*}
\]

To simplify, set \( b = \max\{a, c\} \). That way, if \( n > 1 \) and \( n \leq n_0 \), \( a \leq bn \), so we can state the recurrence as:

\[
\begin{align*}
f(n) &\leq a \text{ for } n \leq 1 \\
f(n) &\leq cn + f(n/2) + f(n/2) \text{ for even } n > 1
\end{align*}
\]

The fact that we only state the recurrence for even \( n \) limits what we can prove using induction. We need all the list sizes in all recursive calls to be either even or 1. This means we can only prove something using this recurrence for values of \( n \) that are powers of 2, i.e. numbers of the form \( 2^k \) where \( k \) is a nonnegative integer.

**Claim:** For \( n \) a power of two, \( f(n) \leq an + b \cdot n \log_2 n \).
Proof: By induction on \( n \). For \( n \leq 1 \), \( f(n) \leq a \). In this case, the right-hand side of the claimed inequality is \( a \), so the claim holds.

For \( n \geq 2 \),

\[
\begin{align*}
f(n) &\leq bn + 2f(n/2) \text{ by recurrence} \\
&\leq bn + 2(a \cdot (n/2) + b \cdot (n/2) \log_2(n/2)) \text{ by inductive hypothesis} \\
&\leq bn + 2(a \cdot (n/2) + b \cdot (n/2)(\log_2 n - 1)) \text{ because } \log_2(n/2) = \log_2 n - 1 \\
&\leq bn + a \cdot n + b \cdot n(\log_2 n - 1) \text{ by multiplying through} \\
&\leq bn + an + bn \log_2 n - bn \\
&\leq an + bn \log_2 n
\end{align*}
\]

This completes the induction step and the proof. QED.

What about sorting when the number \( n \) of items is not a power of two?

Two approaches to this:

- Can show that
  - the worst-case running time for input size \( n \) is no more than the worst-case running time for larger inputs, and
  - for any positive integer \( n \), there is a power of two \( 2^k \geq n \) such that \( 2^k \leq 2n \).
  - Therefore, the running time for for arbitrary \( n \) is no more than \( a \cdot (2n) + b \cdot (2n) \log_2(2n) \).

- One can get a better recurrence for mergesort, one that does not depend on \( n \) being even. Then can analyze that recurrence. I will show this in a little bit.

4 The importance of \( O(n \log n) \)

For some of the most important algorithmic problems, we don’t have a good way to achieve linear-time algorithms but we do know of \( O(n \log n) \) algorithms. That extra factor of \( \log n \) is not negligible but it is not very big.

When you go from an \( O(n^2) \) algorithm to an \( O(n \log n) \) algorithm, it is tremendous. One example (which we will not do in this class but which you can study in Algorithms class (and sometimes in Coding the Matrix) is the Fast Fourier Transform (FFT). This is an \( O(n \log n) \) algorithm for something that you would think would require \( O(n^2) \). This algorithm is core to the whole field of digital signal processing. It comes up in GPS. It comes up in image analysis. It even comes up in Bignum multiplication: it is used in an \( O(n \log n) \) algorithm for multiplication of \( n \)-digit numbers.

5 Returning to analysis of recurrence relation

Referring back to our code for \texttt{mergesort}, you can give a recurrence relation that works for odd numbers.

The two recursive calls are:
We use the idea of rooted trees. Rooted trees come up all the time in computer science, so you need to learn about them. We will use them in three completely different ways in the coming lectures.

The first call is on a list of length \( \lfloor n/2 \rfloor \) and the second is on a list of length \( n - \lfloor n/2 \rfloor \). Here \( \lfloor x \rfloor \) is the greatest integer less than or equal to \( x \): \( \lfloor 17.5 \rfloor = 17 \) and \( \lfloor 17 \rfloor = 17 \).

Note that \( n - \lfloor n/2 \rfloor = \lfloor n/2 \rfloor \). So we get a recurrence relation

\[
\begin{align*}
\text{Claim:} & \quad \text{For any } n \geq 1, \ f(n) \leq an + b \cdot n \log_{3/2} n. \\
\text{Proof:} & \quad \text{By induction on } n. \ For \ n \leq 1, \ f(n) \leq a. \ In \ this \ case, \ the \ right-hand \ side \ of \ the \ claimed \ inequality \ is \ a, \ so \ the \ claim \ holds.
\end{align*}
\]

For \( n \geq 2 \),

\[
\begin{align*}
f(n) & \leq bn + f(\lfloor n/2 \rfloor) + f(n - \lfloor n/2 \rfloor) \text{ by recurrence} \\
& \leq bn + \left( a \cdot (\lfloor n/2 \rfloor) + b \cdot (\lfloor n/2 \rfloor) \log_{3/2} \lfloor n/2 \rfloor \right) + \\
& \quad \left( a \cdot (n - \lfloor n/2 \rfloor) + b \cdot (n - \lfloor n/2 \rfloor) \log_{3/2} (n - \lfloor n/2 \rfloor) \right) \text{ by inductive hypothesis}
\end{align*}
\]

For \( n = 2 \), \( \lfloor n/2 \rfloor = n/2 \). For \( n \geq 3 \), \( \lfloor n/2 \rfloor \geq (n - 1)/2 \geq \frac{1}{3} n \). Therefore \( n - \lfloor n/2 \rfloor \leq \frac{2}{3} n \). Therefore \( \log_{3/2} (n - \lfloor n/2 \rfloor) \leq \log_{3/2} \frac{2}{3} n \leq \log_{3/2} n - 1 \).

Of course, \( \lfloor n/2 \rfloor \leq n/2 \leq \frac{2}{3} n \) so also \( \log_{3/2} \lfloor n/2 \rfloor \leq \log_{3/2} n - 1 \).

Using these, and continuing with the sequence of inequalities, we get

\[
\begin{align*}
f(n) & \leq bn + \left( a \cdot (\lfloor n/2 \rfloor) + b \cdot (\lfloor n/2 \rfloor) \log_{3/2} \lfloor n/2 \rfloor \right) + \left( a \cdot (n - \lfloor n/2 \rfloor) + b \cdot (n - \lfloor n/2 \rfloor) \log_{3/2} (n - \lfloor n/2 \rfloor) \right) \\
& \leq bn + \left( a \cdot (\lfloor n/2 \rfloor) + b \cdot (\lfloor n/2 \rfloor) \log_{3/2} n - 1 \right) + \left( a \cdot (n - \lfloor n/2 \rfloor) + b \cdot (n - \lfloor n/2 \rfloor) \log_{3/2} n - 1 \right) \\
& \leq bn + a \cdot (\lfloor n/2 \rfloor + n - \lfloor n/2 \rfloor) + b \cdot (\lfloor n/2 \rfloor + n - \lfloor n/2 \rfloor) \log_{3/2} n - 1 \\
& \leq bn + a \cdot n + b \cdot n \log_{3/2} n - bn \\
& \leq an + bn \log_{3/2} n
\end{align*}
\]

This completes the induction step and the proof. QED.

6 Lower bound on comparison sorting

How fast can we expect to sort things? To answer that question, we will think about an algorithm in a different way.

We use the idea of rooted trees. Rooted trees come up all the time in computer science, so you need to learn about them. We will use them in three completely different ways in the coming lectures.
Idea: Consider a given value of $n$. For the purpose of analysis, we represent the algorithm by a decision tree. This is a binary rooted tree (a tree where every node has at most two children). Each internal node represents a comparison between two input items. Each leaf represents a conclusion the algorithm makes about the order of the input items.

I’ll give an example of a decision tree. Suppose a sorting algorithm is given the list with numbers $a, b, c$. At the end, the algorithm has to produce an ordered list. To do so, it must discover the ordering among the input items. It discovers that ordering by comparing pairs of items.

Try out this algorithm. Let’s say $a = 25$, $b = 15$, $c = 7$.

For any given input, the algorithm must start at the root and make its way to a leaf. Every internal node represents a comparison that the algorithm makes. Because each comparison takes at least one operation, the number of operations on a given input is the number of internal nodes on a root-to-leaf path. For any given leaf, the number of comparisons is then the depth of that leaf. The worst-case number of comparisons is the maximum leaf depth.

For every value of $n$, you need a separate decision tree. The number of leaves is the number of orderings of $n$ numbers, which is $n!$ ($n$ factorial).

For any algorithm that sorts items by comparing them, for every value of $n$, there is a decision tree that mimics the algorithm’s behavior on inputs of size $n$.

Here’s a fact about binary trees:

**Lemma** Every binary tree of depth $d$ has at most $2^d$ leaves.

**Proof** By induction on $d$.

Base case: Any binary tree of depth 0 has only one leaf.

Induction Step: Assume it for $d = k - 1$, and prove it for $d = k$.

Induction hypothesis: Assume that every binary tree of depth $k - 1$ has at most $2^{k-1}$ leaves. Let $T$ be any tree of depth $k$ where $k > 0$. It consists of a root and some internal nodes ....
The descendants of the left child form a left subtree and the descendants of the right child form a right subtree.

The left subtree is a rooted tree of depth \( k - 1 \) so it has at most \( 2^{k-1} \) leaves. The right subtree is a rooted tree of depth \( k - 1 \) so it has at most \( 2^{k-1} \) leaves. The total number of leaves is just the number of leaves in the two subtrees, which is at most \( 2^{k-1} + 2^{k-1} \), which is \( 2 \cdot 2^{k-1} \), which is \( 2^k \). This proves the induction step, completing the proof. QED

For example, a binary tree with 6 leaves cannot have depth 2 because \( 2^2 \) is less than 6. It must have depth at least three (and our decision tree for sorting three numbers has depth three).

Suppose we want to sort 4 numbers. There are \( 4 \cdot 3 \cdot 2 \cdot 1 \) different outcomes, so any decision tree for this has this many leaves. It’s 24, so any decision tree with this many leaves has depth at least 5 because \( 2^4 < 24 \).

In general, for any positive integer \( n \), when sorting \( n \) numbers, there are \( n! \) possible results, so a decision tree for sorting \( n \) numbers has \( n! \) leaves. This can tell us how big the depth must be.

Let the depth be \( d \). Then we must have \( 2^d \geq n! \).

Stirling’s Approximation tells us that \( n! \) is at least \( \frac{n}{e} \).

How do we find out how large \( d \) must be?

The function \( \log_2 y \) is monotonically increasing (higher \( y \) means higher \( \log_2 y \)). That means we can start with an true inequality and take logs of both sides, and we get another true inequality.

We have

\[
2^d \geq n! \geq \frac{n}{e}
\]

Take logs of both sides:

\[
\log_2(2^d) \geq \log_2\left(\frac{n}{e}\right)
\]

which is

\[
d \geq n \log_2\left(\frac{n}{e}\right) = n(\log_2 n - \log_2 e) = n \log_2 n - n \log_2 e
\]

This shows that the depth must be at least \( n \log_2 n - n \log_2 e \), which is at least \( \frac{1}{2} n \log_2 n \) for \( n > 7 \).
This shows that any sorting algorithm that makes order decisions based on comparing pairs of numbers has a running time that is $\Omega(n \log_2 n)$.

After lecture, a student in class, David Gao, pointed out that you don’t really need Stirling’s approximation to prove that $\log_2 n!$ is $\Omega(n \log n)$. Here’s an outline: For $n \geq 6$,

$$
\log_2 n! = \log_2(n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot ([n/2]) \cdot ([n/2] - 1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1) \\
= \log_2 n + \log_2(n - 1) + \log_2(n - 2) + \ldots + \log([n/2]) + \log([n/2] - 1) + \ldots + \log 3 + \log 2 + \log 1 \\
\geq \log_2 n + \log_2(n - 1) + \log_2(n - 2) + \ldots + \log([n/2]) \\
\geq \frac{\log([n/2])}{(n - [n/2]) + \text{times}} \\
\geq (n/2) \log \left(\frac{1}{3} n\right) \\
\geq (n/2)(\log n - \log 3) \\
\geq (n/2)(\log n - 1.585) \\
\geq (n/2) \frac{2}{3} \log n \\
\geq \frac{1}{3} n \log n
$$

which shows that $\log n!$ is $\Omega(n \log n)$.

Can this running time be beaten? Is there a linear-time sorting algorithm? Remember, we assumed that the only way the algorithm knows about the numbers is by comparing them in pairs. There is an algorithm that beats this running time by doing things differently.

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.