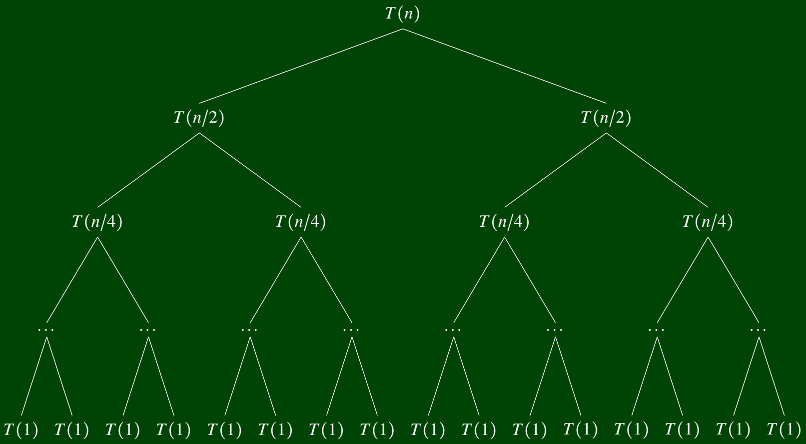
CSE 332

Data Abstractions

Algorithm Analysis 2



Outline

Let x and L be `LinkedList` `Node`s.

Analyzing append

```
append(x, L) {
   Node curr = L;
   while (curr != null && curr.next != null) {
      curr = curr.next;
   }
   curr.next = x;
}
```

What is the...

- best case time complexity of append?
- worst case time complexity of append?

Let x and L be LinkedList Nodes.

Analyzing append

```
append(x, L) {
   Node curr = L;
   while (curr != null && curr.next != null) {
      curr = curr.next;
   }
   curr.next = x;
}
```

What is the...

- best case time complexity of append?
  $\Omega(n)$, because we always **must** do n iterations of the loop.
- worst case time complexity of append?

Warm-Up #1: `append`

Let x and L be `LinkedList` `Node`s.

Analyzing append

```
append(x, L) {
   Node curr = L;
   while (curr != null && curr.next != null) {
      curr = curr.next;
   }
   curr.next = x;
}
```

What is the...

- best case time complexity of `append`?
  $\Omega(n)$, because we always **must** do n iterations of the loop.

- worst case time complexity of `append`?
  $\mathcal{O}(n)$, because we never do **more** than n iterations of the loop.

Warm-Up #1: `append`

Let x and L be `LinkedList Node`s.

Analyzing append

```
append(x, L) {
   Node curr = L;
   while (curr != null && curr.next != null) {
      curr = curr.next;
   }
   curr.next = x;
}
```

What is the...

- best case time complexity of append?
  $\Omega(n)$, because we always **must** do n iterations of the loop.

- worst case time complexity of append?
  $\mathcal{O}(n)$, because we never do **more** than n iterations of the loop.

Since we can **upper** and **lower** bound the time complexity with the same complexity class, we can say append runs in $\Theta(n)$.

Pre-Condition: L_1 and L_2 are sorted.
Post-Condition: Return value is sorted.

Merge

```
merge(L_1, L_2) {
   p1, p2 = 0;
   While both lists have more elements:
      Append the smaller element to L.
      Increment p1 or p2, depending on which had the smaller element
   Append any remaining elements from L_1 or L_2 to L
   return L
}
```

What is the...

- best case # of comparisons of `merge`?
- worst case # of comparisons of `merge`?
- worst case space usage of `merge`?

Pre-Condition: L_1 and L_2 are sorted.
Post-Condition: Return value is sorted.

Merge

```
merge(L1, L2) {
   p1, p2 = 0;
   While both lists have more elements:
      Append the smaller element to L.
      Increment p1 or p2, depending on which had the smaller element
   Append any remaining elements from L1 or L2 to L
   return L
}
```

What is the...

- best case # of comparisons of `merge`?
  $\Omega(1)$. Consider the input: `[0]`, `[1, 2, 3, 4, 5, 6]`.
- worst case # of comparisons of `merge`?
- worst case space usage of `merge`?

Pre-Condition: L_1 and L_2 are sorted.
Post-Condition: Return value is sorted.

Merge

```
merge(L1, L2) {
   p1, p2 = 0;
   While both lists have more elements:
      Append the smaller element to L.
      Increment p1 or p2, depending on which had the smaller element
   Append any remaining elements from L1 or L2 to L
   return L
}
```

What is the...

- best case # of comparisons of `merge`?
  $\Omega(1)$. Consider the input: `[0]`, `[1, 2, 3, 4, 5, 6]`.
- worst case # of comparisons of `merge`?
  $\mathcal{O}(n)$. Consider the input: `[1, 3, 5]`, `[2, 4, 6]`.
- worst case space usage of `merge`?

Pre-Condition: L_1 and L_2 are sorted.
Post-Condition: Return value is sorted.

Merge

```
merge(L1, L2) {
   p1, p2 = 0;
   While both lists have more elements:
      Append the smaller element to L.
      Increment p1 or p2, depending on which had the smaller element
   Append any remaining elements from L1 or L2 to L
   return L
}
```

What is the...

- best case # of comparisons of `merge`?
  $\Omega(1)$. Consider the input: `[0]`, `[1, 2, 3, 4, 5, 6]`.

- worst case # of comparisons of `merge`?
  $\mathcal{O}(n)$. Consider the input: `[1, 3, 5]`, `[2, 4, 6]`.

- worst case space usage of `merge`?
  $\mathcal{O}(n)$, because we allocate a constant amount of space per element.

Well, we did `merge`, what did you think was next?

Consider the following code:

Merge Sort

```
sort(L) {
   if (L.size() < 2) {
      return L;
   }
   else {
      int mid = L.size() / 2;
      return merge(
         sort(L.subList(0, mid)),
         sort(L.subList(mid, L.size()))
      );
   }
}
```

What is the worst case/best case # of comparisons of `sort`?

Yeah, yeah, it's $\mathcal{O}(n \lg n)$, but why?

What is a recurrence?

In CSE 311, you saw a bunch of questions like:

Induction Problem

Let $f_0 = 0, f_1 = 1, f_n = f_{n-1} + f_{n-2}$ for all $n \geq 2$. Prove $f_n < 2^n$ for all $n \in \mathbb{N}$.

(Remember the Fibonacci Numbers? You'd better bet they're going to show up in this course!)

That's a recurrence. That's it.

Definition (Recurrence)

A recurrence is a recursive definition of a function in terms of smaller values.

Merge Sort is hard; so. . .

Let's start with trying to analyze this code:

LinkedList Reversal

```
reverse(L) {
   if (L == null) {
      return null;
   }
   else {
      Node front = L;
      Node rest = L.next;
      L.next = null;

      Node restReversed = reverse(rest);
      append(front, restReversed);
   }
}
```

Notice that `append` is the same function from the beginning of lecture that had runtime $\mathcal{O}(n)$.

So, what is the time complexity of `reverse`?

We split the work into two pieces:

- Non-Recursive Work
- Recursive Work

Non-Recursive Work

LinkedList Reversal

```
reverse(L) {
   if (L == null) {                                //O(1)
      return null;
   }
   else {
      Node front = L;                              //O(1)
      Node rest = L.next;                          //O(1)
      L.next = null;                               //O(1)

      Node restReversed = reverse(rest);
      append(front, restReversed);                 //O(n)
   }
}
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1 n$ for some constants c_0 and c_1.

Non-Recursive Work

LinkedList Reversal

```
reverse(L) {
   if (L == null) {
      return null;
   }
   else {
      Node front = L;
      Node rest = L.next;
      L.next = null;

      Node restReversed = reverse(rest);
      append(front, restReversed);
   }
}
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1 n$ for some constants c_0 and c_1.

Recursive Work: The work it takes to do reverse **on a list one smaller**.

Putting these together almost gives us the recurrence:

$$T(n) = c_0 + c_1 n + T(n-1)$$

We're missing the base case!

LinkedList Reversal

```
reverse(L) {
   if (L == null) {
      return null;
   }
   else {
      Node front = L;
      Node rest = L.next;
      L.next = null;

      Node restReversed = reverse(rest);
      append(front, restReversed);
   }
}
```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ c_0 + c_1 n + T(n-1) & \text{otherwise} \end{cases}$$

Now, we need to **solve** the recurrence.

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ c_0 + c_1 n + T(n-1) & \text{otherwise} \end{cases}$$

$$\begin{aligned}
T(n) &= (c_0 + c_1 n) \quad + \quad T(n-1) \\
&= (c_0 + c_1 n) \quad + \quad (c_0 + c_1(n-1)) \quad + \quad T(n-2) \\
&= (c_0 + c_1 n) \quad + \quad (c_0 + c_1(n-1)) \quad + \quad (c_0 + c_1(n-2)) + \ldots + (c_0 + c_1(1)) + d_0 \\
&= \sum_{i=0}^{n-1} (c_0 + c_1(n-i)) \quad + \quad d_0 \\
&= \sum_{i=0}^{n-1} c_0 + \sum_{i=0}^{n-1} c_1(n-i) \quad + \quad d_0 \\
&= nc_0 + c_1 \sum_{i=1}^{n} i \quad + \quad d_0 \\
&= nc_0 + c_1 \left(\frac{n(n+1)}{2} \right) \quad + \quad d_0 \\
&= \mathcal{O}(n^2)
\end{aligned}$$

A recurrence where we solve some constant piece of the problem (e.g. "-1", "-2", etc.) is called a **Linear Recurrence**.

We solve these like we did above by **Unrolling the Recurrence**.

This is a fancy way of saying "plug the definition into itself until a pattern emerges".

Now, back to `mergesort`.

Analyzing Merge Sort

Merge Sort

```
sort(L) {
   if (L.size() < 2) {
      return L;
   }
   else {
      int mid = L.size() / 2;
      return merge(
         sort(L.subList(0, mid)),
         sort(L.subList(mid, L.size()))
      );
   }
}
```

First, we need to find the recurrence:

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$

This recurrence isn't linear! This is a "divide and conquer" recurrence.

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$

This time, there are multiple possible approaches:

Unrolling the Recurrence

$$\begin{aligned} T(n) &= (c_2 + c_1 n) &+\quad& 2(c_2 + c_1 n + 2T(n/4)) \\ &= (c_2 + c_1 n) &+\quad& 2(c_2 + c_1 n + 2(c_2 + c_1 n + 2T(n/8))) \\ &= c_2 + 2c_2 &+\quad& 4c_2 + \ldots + \text{argh} + \ldots \end{aligned}$$

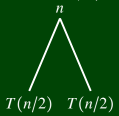
This works, but I'd rarely recommend it.

Insight: We're **branching** in this recurrence. So, represent it as a tree!

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ \underset{T(n)}{c_0 + c_1 n + 2T(n/2)} & \text{otherwise} \end{cases}$$

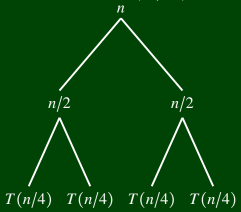
Merge Sort: Solving the Recurrence

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$

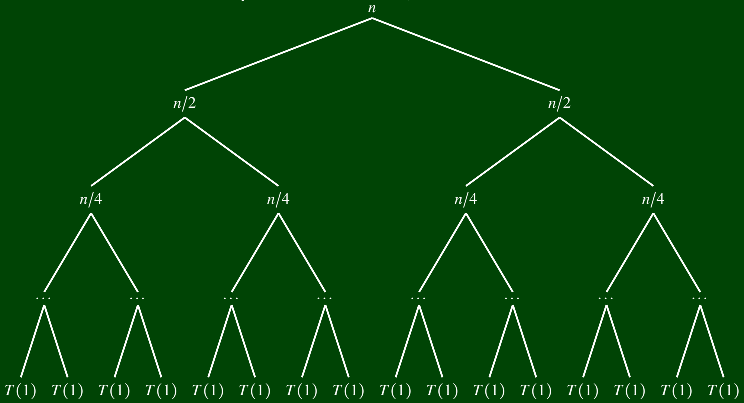


Merge Sort: Solving the Recurrence

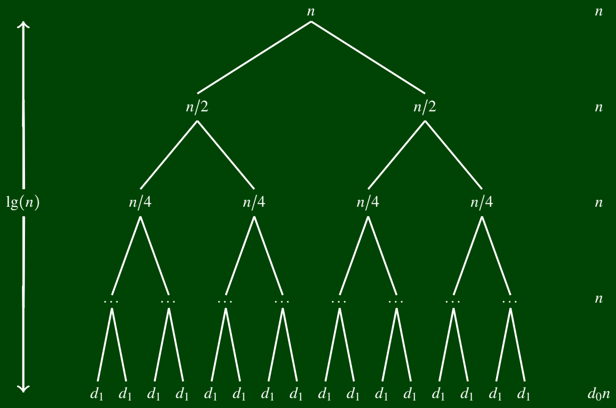
$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$



$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$



$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1 n + 2T(n/2) & \text{otherwise} \end{cases}$$



Since the recursion tree has height $\lg(n)$ and each row does n work, it follows that $T(n) \in \mathcal{O}(n\lg(n))$.

Find A Big-Oh Bound For The Worst Case Runtime

```
sum(n) {
   if (n < 2) {
      return n;
   }
   return 2 + sum(n - 2);
}
```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_0 & \text{if } n = 1 \\ c_0 + T(n-2) & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(n) &= c_0 + c_0 + \cdots + c_0 + d_0 \\ &= c_0\left(\frac{n}{2}\right) + d_0 \\ &= \mathcal{O}(n) \end{aligned}$$

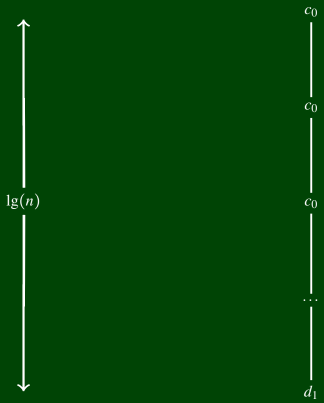
Find A Big-Oh Bound For The Worst Case Runtime

```
binarysearch(L, value) {
   if (L.size() == 0) {
      return false;
   }
   else if (L.size() == 1) {
      return L[0] == value;
   }
   else {
      int mid = L.size() / 2;
      if (L[mid] < value) {
         return binarysearch(L.subList(mid + 1, L.size()), value);
      }
      else {
         return binarysearch(L.subList(0, mid), value);
      }
   }
}
```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + T(n/2) & \text{otherwise} \end{cases}$$

sum Examples #2

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + T(n/2) & \text{otherwise} \end{cases}$$



So, $T(n) = c_0(\lg(n) - 1) + d_1 = \mathcal{O}(\lg n)$.

- Gauss' Sum: $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

- Infinite Geometric Series: $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$, when $|x| < 1$.

- Finite Geometric Series: $\sum_{i=0}^{n} x^i = \frac{1-x^{n+1}}{1-x}$, when $x \neq 1$.

Consider a recurrence of the form:

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

Then,

- If $\log_b(a) < c$, then $T(n) = \Theta(n^c)$.
- If $\log_b(a) = c$, then $T(n) = \Theta(n^c \lg(n))$.
- If $\log_b(a) > c$, then $T(n) = \Theta(n^{\log_b(a)})$.

Sanity Check: For Merge Sort, we have $a = 2, b = 2, c = 1$. Then, $\log_2(2) = 1 = 1$. So, $T(n) = n \lg n$.

Proving the First Case of Master Theorem

$$T(n) = \begin{cases} d & \text{if } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c & \text{otherwise} \end{cases}$$

We assume that $\log_b(a) < c$. Then, unrolling the recurrence, we get:

$$\begin{aligned}
T(n) &= n^c + aT(n/b) \\
&= n^c + a((n/b)^c + aT(n/b^2)) \\
&= n^c + a(n/b)^c + a^2(n/b^2)^c + \cdots + a^{\log_b(n)}(n/b^{\log_b n})^c \\
&= \sum_{i=0}^{\log_b(n)} a^i \left(\frac{n^c}{b^{ic}}\right) \\
&= n^c \sum_{i=0}^{\log_b(n)} \left(\frac{a}{b^c}\right)^i \\
&= n^c \left(\frac{\left(\frac{a}{b^c}\right)^{\log_b(n)+1} - 1}{\left(\frac{a}{b^c}\right) - 1} \right) \approx n^c \left(\left(\frac{a}{b^c}\right)^{\log_b(n)} \right) \approx n^c
\end{aligned}$$

Today's Takeaways!

- Know how to make a recurrence from a recursive program
- Know the different types of recurrences
- Be able to find a closed form for each type of recurrence
- Know the common summations
- Understand why Master Theorem can be helpful