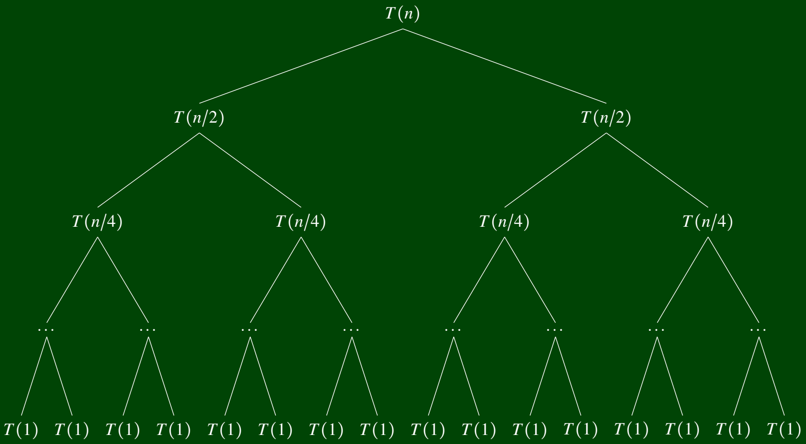
CSE 332

Data Structures and Parallelism

Algorithm Analysis 2



Outline

1 Summations

2 Warm-Ups

3 Analyzing Recursive Code

4 Generating and Solving Recurrences

- 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 $x \neq 1$.

Warm-Up #1: 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 . . .

- a lower bound on the time complexity of append?
- an upper bound on the 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 . . .

- a lower bound on the time complexity of append?
  $\Omega(n)$, because we always **must** do n iterations of the loop.
- an upper bound on the 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 . . .

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

- an upper bound on the 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 Nodes.

Analyzing append

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

What is . . .

- a lower bound on the time complexity of append?
  $\Omega(n)$, because we always **must** do n iterations of the loop.
- an upper bound on the 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(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... (remember the lists are `Nodes`) $|L_1| + |L_2| = n$

- best case # of comparisons of `merge`?

  $\Omega(1)$

- 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. . . (remember the lists are `Nodes`)

- 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.

[0], [1, 2, ..., n]

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... (remember the lists are `Nodes`)

- 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`?

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... (remember the lists are `Nodes`)

- 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 if (L.next == null) { return L; }
   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`?

Let's start with trying to analyze this code:

LinkedList Reversal

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

LinkedList Reversal

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

Non-Recursive Work

LinkedList Reversal

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

LinkedList Reversal

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

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

n − 1

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: $|L| = n$

$T(n-1)$

$T(n) = ?R + ?NR$

$= T(n-1) + n$

Non-Recursive Work

LinkedList Reversal

```
reverse(L) {
   if (L == null) { return null; }
   else if (L.next == null) { return L; }
   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**.

Non-Recursive Work

LinkedList Reversal

```
reverse(L) {
   if (L == null) { return null; }
   else if (L.next == null) { return L; }
   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:

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

Non-Recursive Work

LinkedList Reversal

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

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

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

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

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

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

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

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

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".

Today's Takeaways!

- Understand that Big-Oh is just an "upper bound" and Big-Omega is just a "lower bound"
- Know how to make a recurrence from a recursive program
- Understand what a linear recurrence is
- Be able to find a closed form linear recurrences
- Know the common summations