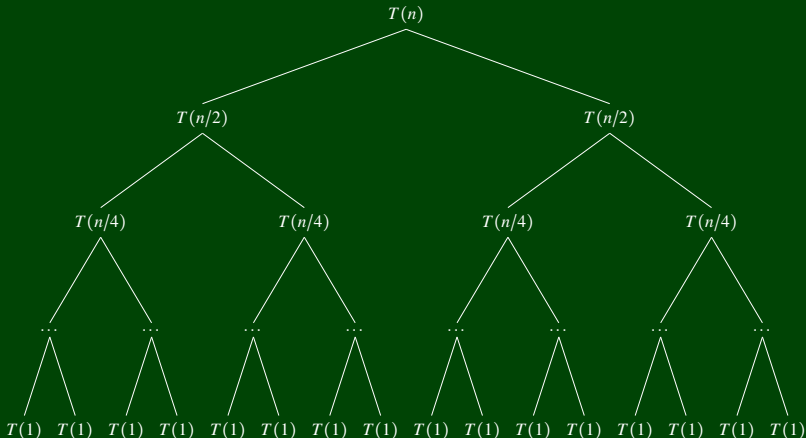


Algorithm Analysis 2



Outline

1 Warm-Ups

2 Analyzing Recursive Code

Let x and L be LinkedList Nodes.

Analyzing append

```
1 append(x, L) {  
2     Node curr = L;  
3     while (curr != null && curr.next != null) {  
4         curr = curr.next;  
5     }  
6     curr.next = x;  
7 }
```

What is ...

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

Let x and L be LinkedList Nodes.

Analyzing append

```
1 append(x, L) {  
2     Node curr = L;  
3     while (curr != null && curr.next != null) {  
4         curr = curr.next;  
5     }  
6     curr.next = x;  
7 }
```

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`?

Let x and L be LinkedList Nodes.

Analyzing append

```
1 append(x, L) {
2   Node curr = L;
3   while (curr != null && curr.next != null) {
4     curr = curr.next;
5   }
6   curr.next = x;
7 }
```

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.

Let x and L be LinkedList Nodes.

Analyzing append

```
1 append(x, L) {  
2     Node curr = L;  
3     while (curr != null && curr.next != null) {  
4         curr = curr.next;  
5     }  
6     curr.next = x;  
7 }
```

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

```
1 merge( $L_1$ ,  $L_2$ ) {  
2     p1, p2 = 0;  
3     While both lists have more elements:  
4         Append the smaller element to L.  
5         Increment p1 or p2, depending on which had the smaller element  
6     Append any remaining elements from  $L_1$  or  $L_2$  to L  
7     return L  
8 }
```

What is the... (remember the lists are Nodes)

- 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

```
1 merge( $L_1$ ,  $L_2$ ) {
2     p1, p2 = 0;
3     While both lists have more elements:
4         Append the smaller element to L.
5         Increment p1 or p2, depending on which had the smaller element
6     Append any remaining elements from  $L_1$  or  $L_2$  to L
7     return L
8 }
```

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.

Merge

```
1 merge( $L_1$ ,  $L_2$ ) {  
2     p1, p2 = 0;  
3     While both lists have more elements:  
4         Append the smaller element to L.  
5         Increment p1 or p2, depending on which had the smaller element  
6     Append any remaining elements from  $L_1$  or  $L_2$  to L  
7     return L  
8 }
```

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?

Pre-Condition: L_1 and L_2 are sorted.

Post-Condition: Return value is sorted.

Merge

```
1 merge( $L_1$ ,  $L_2$ ) {  
2     p1, p2 = 0;  
3     While both lists have more elements:  
4         Append the smaller element to L.  
5         Increment p1 or p2, depending on which had the smaller element  
6     Append any remaining elements from  $L_1$  or  $L_2$  to L  
7     return L  
8 }
```

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.

Let's start with trying to analyze this code:

LinkedList Reversal

```
1 reverse(L) {
2     if (L == null) { return null; }
3     else if (L.next == null) { return L; }
4     else {
5         Node front = L;
6         Node rest = L.next;
7         L.next = null;
8
9         Node restReversed = reverse(rest);
10        append(front, restReversed);
11    }
12 }
```

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

```
1 reverse(L) {
2     if (L == null) { return null; }
3     else if (L.next == null) { return L; }
4     else {
5         Node front = L;
6         Node rest = L.next;
7         L.next = null;
8
9         Node restReversed = reverse(rest);
10        append(front, restReversed);
11    }
12 }
```

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

```
1 reverse(L) {  
2     if (L == null) { return null; }           //O(1)  
3     else if (L.next == null) { return L; }    //O(1)  
4     else {  
5         Node front = L;                       //O(1)  
6         Node rest = L.next;                   //O(1)  
7         L.next = null;                        //O(1)  
8  
9         Node restReversed = reverse(rest);  
10        append(front, restReversed);          //O(n)  
11    }  
12 }
```

Non-Recursive Work:

LinkedList Reversal

```
1 reverse(L) {
2   if (L == null) { return null; }           //O(1)
3   else if (L.next == null) { return L; }    //O(1)
4   else {
5     Node front = L;                         //O(1)
6     Node rest = L.next;                     //O(1)
7     L.next = null;                          //O(1)
8
9     Node restReversed = reverse(rest);
10    append(front, restReversed);            //O(n)
11  }
12 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1n$ for some constants c_0 and c_1 .

LinkedList Reversal

```
1 reverse(L) {  
2     if (L == null) { return null; }  
3     else if (L.next == null) { return L; }  
4     else {  
5         Node front = L;  
6         Node rest = L.next;  
7         L.next = null;  
8  
9         Node restReversed = reverse(rest);  
10        append(front, restReversed);  
11    }  
12 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1n$ for some constants c_0 and c_1 .

Recursive Work:

LinkedList Reversal

```
1 reverse(L) {  
2     if (L == null) { return null; }  
3     else if (L.next == null) { return L; }  
4     else {  
5         Node front = L;  
6         Node rest = L.next;  
7         L.next = null;  
8  
9         Node restReversed = reverse(rest);  
10        append(front, restReversed);  
11    }  
12 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1n$ for some constants c_0 and c_1 .

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

LinkedList Reversal

```
1 reverse(L) {
2     if (L == null) { return null; }
3     else if (L.next == null) { return L; }
4     else {
5         Node front = L;
6         Node rest = L.next;
7         L.next = null;
8
9         Node restReversed = reverse(rest);
10        append(front, restReversed);
11    }
12 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1n$ 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:

$$T(n) = c_0 + c_1n + T(n-1)$$

LinkedList Reversal

```
1 reverse(L) {
2     if (L == null) { return null; }
3     else if (L.next == null) { return L; }
4     else {
5         Node front = L;
6         Node rest = L.next;
7         L.next = null;
8
9         Node restReversed = reverse(rest);
10        append(front, restReversed);
11    }
12 }
```

Non-Recursive Work: $\mathcal{O}(n)$, which means we can write it as $c_0 + c_1n$ 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:

$$T(n) = c_0 + c_1n + T(n-1)$$

We're missing the base case!

LinkedList Reversal

```
1 reverse(L) {  
2     if (L == null) { return null; }  
3     if (L.next == null) { return L; }  
4     else {  
5         Node front = L;  
6         Node rest = L.next;  
7         L.next = null;  
8  
9         Node restReversed = reverse(rest);  
10        append(front, restReversed);  
11    }  
12 }
```

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_0 & \text{if } n = 1 \\ c_0 + c_1n + T(n-1) & \text{otherwise} \end{cases}$$

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

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1n + T(n-1) & \text{otherwise} \end{cases}$$

$$T(n) = (c_0 + c_1n) + T(n-1)$$

$$T(n) = \begin{cases} d_0 & \text{if } n = 0 \\ d_1 & \text{if } n = 1 \\ c_0 + c_1n + T(n-1) & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(n) &= (c_0 + c_1n) + T(n-1) \\ &= (c_0 + c_1n) + (c_0 + c_1(n-1)) + T(n-2) \\ &= (c_0 + c_1n) + (c_0 + c_1(n-1)) + (c_0 + c_1(n-2)) + \dots + (c_0 + c_1(2)) + d_0 + d_0 \\ &= \sum_{i=0}^{n-2} (c_0 + c_1(n-i)) + 2d_0 \\ &= \sum_{i=0}^{n-2} c_0 + \sum_{i=0}^{n-2} c_1(n-i) + 2d_0 \\ &= (n-1)c_0 + c_1 \sum_{i=1}^{n-1} i + 2d_0 \\ &= (n-1)c_0 + c_1 \left(\frac{(n-1)n}{2} \right) + 2d_0 \\ &= \mathcal{O}(n^2) \end{aligned}$$

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”.