CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Notes regarding these slides for Nov. 9, 2022

- I was ill and needed to miss class on Nov. 9
  - I am sorry about that inconvenience—it's not a decision I made lightly!
- Dale Skrien very generously agreed to cover class for me—thank you, Dale!!!
- These are some slides I was prepared to use if I had been there
  - I gave them to Dale as an overview of what he could cover
- These notes do not perfectly match what happened in class…
  - … But I think they might still be a useful accompaniment to our Nov. 9 class meeting, so I'm posting them anyway
  - I hope they're useful for you!
- As always, please be in touch with questions!

Time Complexity of Remove (first occurrence of an element)

- How would you analyze the time complexity of this algorithm?

This is something we haven't done before! Let's think it through…
- We analyze complexity as a function of input size, as usual
- Let's let *n* stand for input size, and *T(n)* stand for time complexity on input of size *n*
- We need to figure out what T(n) is… What foundations or definitions can we follow (Zen principles!) to help us?
  - Well, it's recursive… So let's look at the base case and recursive case separately

```
Algorithm: LLRemove(i, L)
// see specification on prev. slide
   if L = [ ]
      return L
   else
      if i = first(L)
         return rest(L)
      else:
         return cons(first(L),LLRemove(i,rest(L)))
```

Functions on LLists:
- first(L): returns *first*
- rest(L): returns *rest*
- cons(v,L): creates new LList with v as *first* and L as *rest*

Assume all of these functions are O(1)—they would be in most implementations

Recurrences for Time Complexity of Recursive Functions

- As an example of analyzing time complexity of recursive functions, let's stay with LLRemove()
  - Complexity of function of input size *n*
- **Definition**: Let *T(n)* stand for runtime of LLRemove() on list of size n
  - Now we figure out… *what is T(n)*?
- Because LLRemove is recursive, let's look at the base case / recursive cases
- In the *base case,* what is *the input size*, and what is *the runtime of the algo*?

```
Algorithm: LLRemove(i, L)
// see specification on prev. slide
  if L = [ ]
    return L
  else
    if i = first(L)
      return rest(L)
    else:
      return cons(first(L),
                  LLRemove(i,rest(L)))
```

Recall that *first(L), rest(L), cons(v,L)* functions are all O(1):

Recurrences for Time Complexity of Recursive Functions

- As an example of analyzing time complexity of recursive functions, let's stay with LLRemove()
  - Complexity of function of input size *n*
- **Definition**: Let *T(n)* stand for runtime of LLRemove() on list of size n
  - Now we figure out… *what is T(n)*?
- Because LLRemove is recursive, let's look at the base case / recursive cases
- In the *base case,* what is *the input size*, and what is *the runtime of the algo*?

```
Algorithm: LLRemove(i, L)
// see specification on prev. slide
  if L = [ ]
    return L
  else
    if i = first(L)
      return rest(L)
    else:
      return cons(first(L),
                  LLRemove(i,rest(L)))
```

Recall that *first(L), rest(L), cons(v,L)* functions are all O(1):

Base case:
- **Input size: empty list, $n = 0$**
- **Runtime: $\theta(1)$ (do you see why?)**

Recurrences for Time Complexity of Recursive Functions

- As an example of analyzing time complexity of recursive functions, let's stay with LLRemove()
  - Complexity of function of input size *n*
- **Definition**: Let $T(n)$ stand for runtime of LLRemove() on list of size n
  - Now we figure out… *what is $T(n)$*?
- Because LLRemove is recursive, let's look at the base case / recursive cases
- In the *base case,* what is *the input size*, and what is *the runtime of the algo*?

```
Algorithm: LLRemove(i, L)
// see specification on prev. slide
  if L = [ ]
    return L
  else
    if i = first(L)
      return rest(L)
    else:
      return cons(first(L),
                  LLRemove(i,rest(L)))
```

Recall that *first(L), rest(L), cons(v,L)* functions are all O(1):

Base case:
- **Input size: empty list, $n = 0$**
- **Runtime: $\theta(1)$ (do you see why?)**

So, we'd say $T(0) = \theta(1)$ to express the base case runtime

Recurrences for Time Complexity of Recursive Functions

- As an example of analyzing time complexity of recursive functions, let's stay with LLRemove()
  - Complexity of function of input size *n*
- **Definition**: Let $T(n)$ stand for runtime of LLRemove() on list of size n
  - Now we figure out… *what is $T(n)$*?
- Because LLRemove is recursive, let's look at the base case / recursive cases
- Base case: $T(0) = \theta(1)$
- How about the recursive case? What is the input size and runtime?

```
Algorithm: LLRemove(i, L)
```

Let's just focus on the recursive case for now…

```
    if i = first(L)
      return rest(L)
    else:
      return cons(first(L),
                  LLRemove(i,rest(L)))
```

Recursive case: We say input is size *n*, as usual—L has n elements. Also…
- **It does some work other than the recursive call—combined, $\theta(1)$** (do you see why?)
- ***All of its other runtime* is in its recursive call. How would we represent the runtime of that particular recursive call?**

Recurrences for Time Complexity of Recursive Functions

- As an example of analyzing time complexity of recursive functions, let's stay with LLRemove()
  - Complexity of function of input size *n*
- **Definition**: Let $T(n)$ stand for runtime of LLRemove() on list of size n
  - Now we figure out… *what is* $T(n)$?
- Because LLRemove is recursive, let's look at the base case / recursive cases
- Base case: $T(0) = \theta(1)$
- How about the recursive case? What is the input size and runtime?

Algorithm: LLRemove(i, L)

Let's just focus on the recursive case for now…

```
if i = first(L)
   return rest(L)
else:
   return cons(first(L),
               LLRemove(i,rest(L)))
```

Recursive case: We say input is size *n*, as usual—L has n elements. Also…
- **It does some work other than the recursive call—combined, $\theta(1)$** (do you see why?)
- ***All of its other runtime* is in its recursive call.**
  - **Input size to recursive call: *n-1* – a list of 1 less element than L** (do you see why?)
  - **How do we express the runtime of that call? *Use our definition of T()***

Recurrences for Time Complexity of Recursive Functions

- As an example of analyzing time complexity of recursive functions, let's stay with LLRemove()
  - Complexity of function of input size *n*
- **Definition**: Let $T(n)$ stand for runtime of LLRemove() on list of size n
  - Now we figure out… *what is* $T(n)$?
- Because LLRemove is recursive, let's look at the base case / recursive cases
- Base case: $T(0) = \theta(1)$
- Recursive case: $T(n) = T(n-1) + \theta(1)$

Algorithm: LLRemove(i, L)

Let's just focus on the recursive case for now…

```
if i = first(L)
   return rest(L)
else:
   return cons(first(L),
               LLRemove(i,rest(L)))
```

This may look unusual—and recursive!—but it follows cleanly from the previous slide. In the recursive case:
- **It does some work other than the recursive call—combined, $\theta(1)$** (do you see why?)
- ***All of its other runtime* is in its recursive call, T(n-1)**

Recurrences for Time Complexity of Recursive Functions

- Putting all the pieces together (so far—there's more coming up!)

- Let's let *n* stand for input size, and *T(n)* stand for time complexity on input of size *n*
- We need to figure out what T(n) is… let's look at the base case and recursive case separately. The base case (prev. slide) is T(0) = $\theta(1)$.

- What's the complexity in the recursive case? T(n) = T(n-1) + $\theta(1)$
- The time taken by everything *but* the recursive call is just $\theta(1)$—do you see why?
- … and the recursive call is on input of size (n-1)
  - So, *by our definition of function T*, complexity of the recursive call is T(n-1)

```
Algorithm: LLRemove(i, L)
// see specification on prev. slide
  if L = [ ]
    return L
  else
    if i = first(L)
      return rest(L)
    else:
      return cons(first(L),LLRemove(i,rest(L)))
```

Do you see how this characterization of T(n) exactly fits our LLRemove(L) algo?

Let's put the pieces together…
- For n = 0, T(0) = $\theta(1)$
- For n > 0,
  T(n) = T(n-1) + $\theta(1)$

That is a *full definition of the complexity of this algorithm…* both the base case and the recursive case!

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…

  - For n = 0, T(0) = $\theta(1)$
  - For n > 0, T(n) = T(n-1) + $\theta(1)$

- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- Note that it follows good design principles for recursive definitions
  - It has a base case
  - Its recursive case is defined in terms of itself *on smaller inputs*
  - Indeed, the two parts together are a complete definition of the runtime
- But we're not done yet! What's the asymptotic complexity of LLRemove()?

```
Algorithm: LLRemove(i, L)
// see specification on prev. slide
  if L = [ ]
    return L
  else
    if i = first(L)
      return rest(L)
    else:
      return cons(first(L),
                  LLRemove(i,rest(L)))
```

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…
  - **For n = 0, T(0) = θ(1)**
  - **For n > 0, T(n) = T(n-1) + θ(1)**
- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- To find the asymptotic complexity represented by this recurrence, we need to *solve* it—come up with a closed, non-recursive form for T(n)
- Let's try *unwinding* the recurrence—plugging in the definition on successively smaller arguments:

We know T(n) = T(n-1) + θ(1)—that's in the definition
But we similarly know from the definition what T(n-1) is… (what is it?)

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…
  - **For n = 0, T(0) = θ(1)**
  - **For n > 0, T(n) = T(n-1) + θ(1)**
- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- To find the asymptotic complexity represented by this recurrence, we need to *solve* it—come up with a closed, non-recursive form for T(n)
- Let's try *unwinding* the recurrence—plugging in the definition on successively smaller arguments:

T(n) = T(n-1) + θ(1)
= …

What does T(n-1) equal, according to the recursive case of the definition of the recurrence, above?

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…
  - For n = 0, T(0) = θ(1)
  - For n > 0, T(n) = T(n-1) + θ(1)
- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- To find the asymptotic complexity represented by this recurrence, we need to *solve* it—come up with a closed, non-recursive form for T(n)
- Let's try *unwinding* the recurrence—plugging in the definition on successively smaller arguments:

$$\begin{aligned} T(n) &= T(n-1) + \theta(1) \\ &= [T(n-2) + \theta(1)] + \theta(1) = T(n-2) + 2\,\theta(1) \\ &= \ldots \end{aligned}$$

- In this step, we replaced T(n-1) with its definition from the recursive case from above—T(n-1) = T(n-2) + θ(1)
- Next step, we'll continue ***unwinding*** by replacing T(n-2)…

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…
  - For n = 0, T(0) = θ(1)
  - For n > 0, T(n) = T(n-1) + θ(1)
- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- To find the asymptotic complexity represented by this recurrence, we need to *solve* it—come up with a closed, non-recursive form for T(n)
- Let's try *unwinding* the recurrence—plugging in the definition on successively smaller arguments:

$$\begin{aligned} T(n) &= T(n-1) + \theta(1) \\ &= [T(n-2) + \theta(1)] + \theta(1) = T(n-2) + 2\,\theta(1) \\ &= [T(n-3) + \theta(1)] + 2\,\theta(1) = T(n-3) + 3\,\theta(1) \\ &= \ldots \end{aligned}$$

Do you see a pattern?

In general, unwinding stops at the base case of the definition of T(n)…
When will this recurrence's unwinding reach its base case?

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…
  - For n = 0, T(0) = $\theta(1)$
  - For n > 0, T(n) = T(n-1) + $\theta(1)$
- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- To find the asymptotic complexity represented by this recurrence, we need to *solve* it—come up with a closed, non-recursive form for T(n)
- Let's try *unwinding* the recurrence—plugging in the definition on successively smaller arguments:

$$
\begin{aligned}
T(n) &= T(n-1) + \theta(1) \\
&= [T(n-2) + \theta(1)] + \theta(1) = T(n-2) + 2\,\theta(1) \\
&= [T(n-3) + \theta(1)] + 2\,\theta(1) = T(n-3) + 3\,\theta(1) \\
&\ldots \\
&= T(0) + n^*\theta(1) = \ldots
\end{aligned}
$$

What does this equal? Use the base case definition above…

Solving a Time Complexity Recurrence

- Let's focus on our definition of runtime function T(n), and how to use it…
  - For n = 0, T(0) = $\theta(1)$
  - For n > 0, T(n) = T(n-1) + $\theta(1)$
- Important vocabulary:
  - We say this definition of T(n) is a *recurrence*—it defines T(n) in terms of itself
- To find the asymptotic complexity represented by this recurrence, we need to *solve* it—come up with a closed, non-recursive form for T(n)
- Let's try *unwinding* the recurrence—plugging in the definition on successively smaller arguments:

$$
\begin{aligned}
T(n) &= T(n-1) + \theta(1) \\
&= [T(n-2) + \theta(1)] + \theta(1) = T(n-2) + 2\,\theta(1) \\
&= [T(n-3) + \theta(1)] + 2\,\theta(1) = T(n-3) + 3\,\theta(1) \\
&\ldots \\
&= T(0) + n^*\theta(1) = \theta(1) + n^*\theta(1) \\
&= \theta(n)
\end{aligned}
$$

So, LLRemove() is a T(n) = θ(n) algorithm—solved!

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Let's start with the base case. What is the input size in the base case for this Mergesort algo, and what is runtime on that input?

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Base case: p ≥ r. When p = r, input size is 1….
T(1) = θ(1)

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```


Recursive case: Let n stand for the input size, (r-p+1)
T(n) = ??
What work is done in the recursive case? (Recall that Merge is *θ(n)*)

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Recursive case: Let n stand for the input size, (r-p+1)
T(n) = 2 recursive calls + [(Merge) *θ(n)* + (other stuff on lines 1, 2) *θ(1)*]
What work is done in the recursive case? (Recall that Merge is *θ(n)*)

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Recursive case: Let n stand for the input size, (r-p+1)
T(n) = 2 recursive calls + [(Merge) *θ(n)* + (other stuff on lines 1, 2) *θ(1)*]
= *2 recursive calls* + *θ(n)*
We're not done yet…. How much work is done in the 2 recursive calls?

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Recursive case: Let n stand for the input size, (r-p+1)
T(n) = 2 recursive calls + [(Merge) *θ(n)* + (other stuff on lines 1, 2) *θ(1)*]
= 2 *recursive calls* + *θ(n)*
- We'll simplify by assuming each recursive call is on input size n/2
- We'll use our definition of T() to express the runtime of each recursive call…

Time Complexity of Mergesort—using Recurrences

- Mergesort is a classic recursive algo, and recurrences are an essential technique for time complexity analysis…
- What would a recurrence be that represents the time complexity of Mergesort?

So, let's define runtime function T(n) as a recurrence for Mergesort!
Recall that recurrences…
- Give a base case for T(n)
- Give a recursive case for T(n)
- Are solved to get asymptotic complexity for the recursive algo

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

Recursive case: Let n stand for the input size, (r-p+1)
T(n) = 2 recursive calls + [(Merge) *θ(n)* + (other stuff on lines 1, 2) *θ(1)*]
= 2 T(n/2) + *θ(n)*
- We'll simplify by assuming each recursive call is on input size n/2
- We'll use our definition of T() to express the runtime of each recursive call…

Time Complexity of Mergesort—using Recurrences

- Let's put all the pieces together

- Let's let *n* stand for input size, and *T(n)* stand for time complexity on input of size *n*
- We need to figure out what T(n) is… let's look at the base case and recursive case separately. The base case (prev. slide) is T(1) = θ(1).

- What's the complexity in the recursive case? T(n) = 2*T(n/2) + θ(n)
- The time taken by everything *but* the 2 recursive calls is θ(n)—do you see why?
- … and we simplify and say each recursive call is on input of size (n/2)
  - So, *by our definition of function T*, complexity of each recursive call is T(n/2)

```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Do you see how this T(n) exactly fits our MergeSort algo?

Let's put the pieces together…
- For n = 1, T(1) = θ(1)
- For n > 1,
  T(n) = 2*T(n/2) + θ(n)

That is a *full definition of the complexity of this algorithm…* both the base case and the recursive case!

Solving Recurrences

- We'll cover three common techniques for solving recurrences—i.e., getting θ or O bounds on the solution:
  - *Unwinding* (or *backward substitution*): "Unroll" the recurrence until it reaches a base case, then count / analyze the cost represented

    We already did an example of unwinding, and we'll do another one soon!
  - *Recursion-tree method*: Represent costs as nodes in a tree and analyze total cost
  - *Master method*: Solve recurrences of the form
    $T(n) = a*T(n/b) + f(n)$

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$

This is a simplified version of the recursive case of our recurrence for Mergesort, but it's close enough to capture the algo's time complexity.

We already know what the answer is… (Do you remember the complexity of Mergesort?)

Let's go through the steps of solving it by unwinding!

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$

What information is missing from this recurrence, which we will need to be able to solve it?

- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n
  - By that same definition, T(n/2) = 2T((n/2)/2) + (n/2) = 2T(n/4) + n/2
  - So, by plugging that in: T(n) = 2[2T(n/4) + n/2] + n
  - What would the next step(s) be in this unwinding process?
  - Where would it stop?

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$

What information is missing from this recurrence, which we will need to be able to solve it?

- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n

```
T(n)   = 2*T(n/2) + n
       = 2[2*T(n/4) + n/2] + n = 4T(n/4) + 2n
       = 4[2*T(n/8) + n/4] + 2n = 8T(n/8) + 3n
       ...
```

Do you see a pattern here? And when does this unwinding end?

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$
- For a base case, let's use T(1) = 1 (or θ(1), if we want)
- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n

```
T(n)  = 2*T(n/2) + n
      = 2[2*T(n/4) + n/2] + n = 4T(n/4) + 2n
      = 4[2*T(n/8) + n/4] + 2n = 8T(n/8) + 3n
      ...
  →   = 2^k[T(n/2^k)] + k*n
      ...
```

The *k'th step* shown here illustrates the pattern that holds for any relevant *k*. It can help with our analysis to show this in our work!

Unwinding

This name may make it sound more relaxing than it actually is, but as methods for solving recurrences go, it's pretty mellow.

- An example: Solve $T(n) = 2*T(n/2) + n$
- For a base case, let's use T(1) = 1 (or θ(1), if we want)
- *Unwind* the recurrence by plugging in the definition on successively smaller arguments:
  - From the definition, T(n) = 2T(n/2) + n

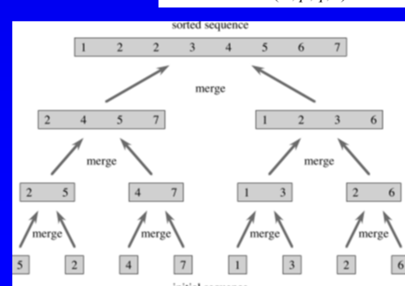

```
T(n)  = 2*T(n/2) + n
      = 2[2*T(n/4) + n/2] + n = 4T(n/4) + 2n
      ...
      = 2^k[T(n/2^k)] + k*n
      ...
      = n*T(1) + (lg n)*n
      = θ(n lg n)
```

The lg n term comes because the recurrence unwinds lg n times before hitting the base case… do you see why?

Recursion Trees: An Overview

- Recursion trees can represent how a recursive algorithm…
  - Breaks input down into recursive calls on sub-problems,
  - Or, *equivalently*, combines recursive calls into a solution on the original problem
- Here's an example from CLRS: Mergesort
  - Each node shows input size at that level of recursive calls
    - Here, original input size 8, breaks into sub-problems of size 4, etc.

```
MERGE-SORT(A, p, r)
  if p < r
      q = ⌊(p + r)/2⌋
      MERGE-SORT(A, p, q)
      MERGE-SORT(A, q + 1, r)
      MERGE(A, p, q, r)
```



- **This example shows the recursion going *up* the tree—combining solutions**
- **Note that the input sizes at each node would be the same for the recursion going *down* the tree, breaking into sub-problems**

Recursion Trees For Solving Time-Complexity Recurrences

- When using recursion trees to solve for time complexity, though, we don't need quite that much information
  - We *do* need the structure, showing how the algo divides and re-combines its inputs
  - We *do* need the input size at each node
  - We *do not* need details about exactly what the input is at each node
- What we need, for each node of the tree:
  - Input size at each node
  - A way to represent the work done (i.e., the runtime) at *that node* of the tree—*not including any other work done above or below it*

Recall: Asymptotic complexity is in terms of input size *n*, not individual inputs of a given size!

Let's do an example!

Recursion-Tree Method

- An example: Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

T(n)

n

What's the recursion tree structure?

What's the cost at each tree-level (i.e., not counting levels below it)?

Recursion-Tree Method

- An example:
  Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$

- Set up a tree to total up the work done by the algorithm

T(n)

What's the recursion tree structure?

n

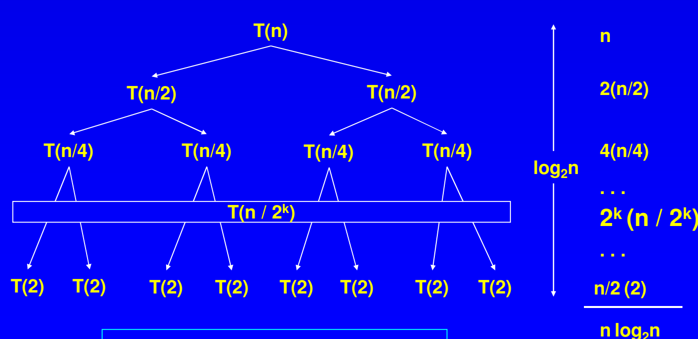
What's the cost at each tree-level (i.e., not counting levels below it)?

- Tree structure for complexity analysis corresponds to tree of recursive calls by the algorithm
- Total work by the algorithm: Sum of work at all levels of the tree

Recursion-Tree Method

- An example:
  Mergesort

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \underbrace{2T(n/2)}_{\text{sorting both halves}} + \underbrace{n}_{\text{merging}} & \text{otherwise} \end{cases}$$



Recursion tree for algorithm

Total work done