CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Definition of our *LList* Data Structure

NOTE: This definition may show up on HW, too!

- In English, we'd say an LList is:
  - Either the empty list,
  - Or
    - an element, called *first*
    - and an LList, called *rest,* representing all the elements after *first*
- To be unambiguous about how we work with LLists, these will be the primitive functions defined on Llists:
  - first(L): returns value of the *first* element of an LList L
  - rest(L): returns value of the *rest* sublist of an LList L
  - cons(v,L): a ***constructor*** function that takes an item v and an LList L and returns a new LList L' such that…?
    - (What do you think it might be?)

What do you think the complexities of these functions are?

Business

- Grading update:
  - PS2 in progress
- Expect PS3 out Real Soon Now
  - Due no sooner than 1 week after it's assigned
- Project 2
  - First part due already
  - Other parts due Nov. 3
  - Please note some restrictions on my schedule:
    - I expect to be traveling on Nov. 3 and probably won't be on email or able to answer questions after noon on that day, so please plan accordingly

Definition of our *LList* Data Structure

NOTE: This definition may show up on HW, too!

- In English, we'd say an LList is:
  - Either the empty list,

Sometimes, the empty list is written as [ ]

  - Or
    - an element, called *first*
    - and an LList, called *rest,* representing all the elements after *first*

We could display all LLists in [brackets], as is usual for lists

- To be unambiguous about how we work with LLists, these will be the primitive functions defined on Llists:
  - first(L): returns value of the *first* element of an LList L
  - rest(L): returns value of the *rest* sublist of an LList L
  - cons(v,L): a ***constructor*** function that takes an item v and an LList L and returns a new LList L' such that
    - v is the element *first* of L'
    - L is the sublist *rest* of L'

We'll treat all three of these as constant-time fns

For example, how would you write this LList as a list in [brackets]?

cons(1,(cons(2,cons(3,[ ]))))

Definition of our *LList* Data Structure

NOTE: This definition may show up on HW, too!

- To be unambiguous about how we work with LLists, these will be the primitive functions defined on Llists:
  - first(L): returns value of the *first* element of an LList L
  - rest(L): returns value of the *rest* sublist of an LList L
  - cons(v,L): a ***constructor*** function that takes an item v and an LList L and returns a new LList L' such that
    - v is the element *first* of L'
    - L is the sublist *rest* of L'

This convention (below) is common in *functional* programming languages

Reminder: *first(L), rest(L),* and *cons(v,L)* are *functions* that return values; they are not fields of an object. Because of this, we cannot assign values to them—e.g., first(L) = 3 or rest(L) = [3] is not permitted.

What could be done instead, with this syntax, to change the first element of some LList *L* to 3?

Definition of our *LList* Data Structure

NOTE: This definition may show up on HW, too!

- To be unambiguous about how we work with LLists, these will be the primitive functions defined on Llists:
  - first(L): returns value of the *first* element of an LList L
  - rest(L): returns value of the *rest* sublist of an LList L
  - cons(v,L): a ***constructor*** function that takes an item v and an LList L and returns a new LList L' such that
    - v is the element *first* of L'
    - L is the sublist *rest* of L'

This convention (below) is common in *functional* programming languages

Reminder: *first(L), rest(L),* and *cons(v,L)* are *functions* that return values; they are not fields of an object. Because of this, we cannot assign values to them—e.g., first(L) = 3 or rest(L) = [3] is not permitted.

What could be done instead, with this syntax, to change the first element of some LList *L* to 3? **[Ans: We could do L = cons(3,rest(L))]**

LList Example:
Remove (first occurrence of an element)

- Consider the problem of removing the first occurrence of an element from a sequence, specified here for a list

> **Input: item *i* and LList L = [x_0, …, x_n]**
>
> **Output: If *i* = x_k and k is the smallest value for which *i* = x_k,**
> **return LList [x_0, …, x_{k-1}, x_{k+1}, … x_n]**
> **Otherwise—i.e., when there is no k such that *i* = x_k—return L**

- How would you design an algorithm to solve this problem?

It's going to be recursive, because the definition of LList is recursive. Follow the definition!

How would we break the problem down into 1 or more smaller subproblems, and then use the result(s) in a solution for the original problem?

LList: either empty [ ] or
- element *first*
- LList *rest*

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*

Correctness:
Remove (first occurrence of an element)

- How would you argue the *correctness* of this algorithm?
  - Be sure to refer to these specifications, as well as lines of pseudocode….

> **Input: item *i* and LList L = [x_0, …, x_n]**
>
> **Output: If *i* = x_k and k is the smallest value for which *i* = x_k,**
> **return LList [x_0, …, x_{k-1}, x_{k+1}, … x_n]**
> **Otherwise—i.e., when there is no k such that *i* = x_k—return L**

  - Hint: It will be an inductive argument, because the algo is recursive…

```
Algorithm: LLRemove(i, L)
// see specification immediately above
   if L = [ ]
      return L
   else
      if i = first(L)
         return rest(L)
      else:
         return cons(first(L),LLRemove(i,rest(L)))
```

LList: either empty [ ] or
- element *first*
- LList *rest*

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*

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: *θ(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: *θ(1)* (do you see why?)**

So, we'd say T(0) = *θ(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) = θ(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, *θ*(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) = θ(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, *θ*(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) = θ(1)***
- Recursive case: ***T(n) = T(n-1) + θ(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, *θ*(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) = *θ*(1).**

- **What's the complexity in the recursive case? T(n) = T(n-1) + *θ*(1)**
- **The time taken by everything *but* the recursive call is just *θ*(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) = *θ*(1)**
- **For n > 0, T(n) = T(n-1) + *θ*(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)))
```