CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: It’s complicated…

Business, Feb. 24

• For 2/24 business, please see slides in the Divide & Conquer I module

• Note: These notes may have additional slides added as the course module progresses—please check back for the most recent version
Business, Mar. 1

• PS2 due already
• SA1 returned already
  – Recall that graded assignments are placed in your SubmittedWork folder
  – I’ll also email you with Classwide Comments when graded assignments are returned (or are to be returned very soon)
• PS3 out today, due TBD (see assignment sheet)
• Also, Small Assignment 2 (Mergesort) assigned today
  – Due by beginning of class Mar. 3
  – See CS375 Lecture Notes page for the SA2 assignment sheet
    • (Look under “Slides, Readings, Etc.” for this course module to find the assignment)
Business, Mar. 3

- PS1 graded and returned
- PS3 out, due Mar. 15 (see assignment sheet)
- SA3 due already
Business, Mar. 10

• PS_DivConqI graded and returned
  – We’ll go over the “faster” $2^n$ exercise in class today
  – Please ask if there are others you think we should go over in class!
  – (I’m also happy to talk with you about your work outside of class, just let me know!)

• SA2 graded and returned (last Friday)
  – Happy to talk outside of class if you have any questions!
  – I especially welcome talking with anyone who didn’t get a check+ -- please see me!

• PS_DivConqII out, due Mar. 15 (see assignment sheet)

• A note on usage of $(\lg n)$ for $(\log_{10} n)$ or $(\log_2 n)$
  – In CS375, we’ll only use $(\lg n)$ for $(\log_2 n)$, never $(\log_{10} n)$
Business, Mar. 10, pt. 2

• Any questions about LLists?
  – Please recall the restricted LList syntax—using it exactly as specified is part of the problem solving exercise!

• General CS375 Notes:
  – When given a data structure with specified syntax to refer to its components, be sure to use that syntax exactly
    • Not doing so can introduce ambiguity and make it harder to be sure explanations are correct / show command of the relevant concepts
  – In CS375, please treat given deadlines as firm / exact
    • (That way, they’re transparent and the same for everybody)
    • … But if you’d like an extension, let me know, and we’ll see about giving an extension to everybody!

And if there are extenuating circumstances (e.g., illness, personal crisis), please let me know!

Business, Mar. 10, pt. 3

• A general CS375 note about explanations:
  – For full credit on CS375 exercises, unless otherwise specified, please make sure your explanations demonstrate command of the relevant concepts
    • In particular, this includes understand the reasons for a given answer

  – E.g., just saying that a statement is true because it is given as true in the textbook does not do that—it does not demonstrate understanding of reasons why the statement is true, based on the relevant definitions.

  – Please feel free to ask me any questions about this, or about explanations in CS375 more generally!

Or about specific cases as they come up, too!
List Algorithms

- We’ve seen how the definition of a binary tree can guide the design of algorithms on binary trees...
- Many common algorithms are written on lists
  - How does the definition of a list guide the designs of those algorithms?

We’ll also revisit this question soon, in a broader context!
List Algorithms

- We’ve seen how the definition of a binary tree can guide the design of algorithms on binary trees…
- Many common algorithms are written on lists
  - How does the definition of a list guide the designs of those algorithms?
  

```
What are some difference between arrays and lists, in this context?
```

- For example, consider the search problem on lists

```
Input: item / and list L
Output: True if i is an element of L, False otherwise
```

- There are multiple ways to approach designing an algorithm for this… how might you design one?
  - What can you say about the complexity of your algorithm?

List Algorithms and Recursion

- Lists, as opposed to arrays, can have node-based definitions
- As part of that, a List type is commonly defined recursively!

- How would you write a recursive algorithm to solve the search problem on lists?
  - One possibility is shown here:
  - How would we argue its correctness?
  - (Do you believe that it works correctly?)
  - What can we say about its complexity?

```
Algorithm: recListSearch(i, L)
// see specification immediately above
  if L = []
    return False
  else
    if i is L[0]
      return True
    else
      return recListSearch(i,L[1:])

This uses Python-like list slicing syntax to refer to "all but the first element of L"
```

Input: item / and list L
Output: True if i is an element of L, False otherwise
List Algorithms and Recursion

• Lists, as opposed to arrays, can have node-based definitions
• As part of that, a List type is commonly defined recursively!
• How would you write a recursive algorithm to solve the search problem on lists?
  – One possibility is shown here:
  – How would we argue its correctness?
  – (Do you believe that it works correctly?)
  – What can we say about its complexity?
  – List slicing can’t be assumed to be constant time!

```python
Input: item i and list L
Output: True if i is an element of L, False otherwise
Algorithm: recListSearch(i, L)
    // see specification immediately above
    if L = []
        return False
    else
        if i is L[0]
            return True
        else
            return recListSearch(i,L[1:])
```
This uses Python-like list slicing syntax to refer to “all but the first element of L”.

List Algorithms: 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 list \( L = [x_0, \ldots, x_n] \)

**Output:** If \( i = x_k \) and \( k \) is the smallest value for which \( i = x_k \), return \( [x_0, \ldots, x_{k-1}, x_{k+1}, \ldots x_n] \)

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…
  - Iteratively?
  - Recursively?
  - How would the complexity of this be different on a list (i.e., a linked list) than on an array?

Definition of our **LList** Data Structure

- Throughout CS375, we will sometimes refer to an **LList** data structure, representing a list of elements.
- In English, we’d say an LList is:
  - Either empty,
  - Or
    - an element, called \( \text{first} \)
    - and an LList, called \( \text{rest} \), representing all the elements after \( \text{first} \)

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

Is this a good definition? Consider Principle 1: Keep your foundations simple….

Is this consistent with your understanding of list structures—that is, linked list structures (which are typically node-based in implementation)?
Definition of our *LLList* Data Structure

• In English, we’d say an LLList is:
  – Either the empty list,
  – Or
    • an element, called *first*
    • and an LLList, called *rest*, representing all the elements after *first*

• To be unambiguous about how we work with LLLists, these will be the primitive functions defined on LLists:
  – *first(L)*: returns the *first* element of an LLList L
  – *rest(L)*: returns the *rest* sublist of an LLList L
  – *cons(v,L)*: a *constructor* function that takes an item v and an LLList L and returns a new LLList L’ such that…?
    • *(What do you think it might be?)*

*Use these functions as accessors, rather than directly accessing fields—i.e., use first(L) instead of L.first*

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

What do you think the complexities of these functions are?

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

\[
\text{cons}(1, (\text{cons}(2, \text{cons}(3, []))))
\]
Definition of our LLList Data Structure

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

**Important note:** 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 LLList L to 3?

---

Definition of our LLList Data Structure

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

**Important note:** 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 LLList 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?
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

  \[ \text{Input: item } i \text{ and LList } L = [x_0, \ldots, x_n] \]

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

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

  Algorithm: LLRemove(i, T)
  // see specification immediately above
  if \( L = [ ] \)
  return \( L \)
  else
  if \( i = \text{first}(L) \)
  return \( \text{rest}(L) \)
  else:
  return cons(first(L), LLRemove(i, rest(L)))

Break It Down Again

- In general, different ways of breaking down a problem into subproblems can lead to different algorithms e.g., Mergesort vs. … any other sort, basically

- Different data structures, by their definitions, suggest different natural ways to break problems into subproblems
  - How would an IntBinTree suggest breaking a problem into subproblems?
  - How would a list (node-based, e.g., LList) would suggest breaking a problem into subproblems?
  - How about for an array?

This isn’t to say that, for any given data structure, some approach is always applied!
This is just looking for common approaches, and what makes them natural in context.
In general, different ways of breaking down a problem into subproblems can lead to different algorithms. Different data structures, by their definitions, suggest different natural ways to break problems into subproblems:

- How would an IntBinTree suggest breaking a problem into subproblems? (subproblems on sub-trees—kinda one half at a time)
- How would a list (node-based, e.g., LList) suggest breaking a problem into subproblems? (subproblems on lists one element shorter)
- How about for an array? (subproblems involve changing indices and iterating over indexed ranges—index access is central to arrays!)

In all of these cases, the foundations—the definitions of the underlying structure—suggest that approach to breaking into subproblems are different.

Looking back to the search problem...

Input: item / and collection L
Output: True if / is an element of L, False otherwise

We’ve written an algorithm for this on IBT’s

What’s a straightforward way to solve this on arrays?
Break It Down Again

- In general, different ways of breaking down a problem into subproblems can lead to different algorithms
- Looking back to the search problem…

  Input: item \( i \) and collection \( L \)

  Output: True if \( i \) is an element of \( L \), False otherwise

  - We’ve written an algorithm for this on IBT’s
  - What’s a straightforward way to solve this on arrays? \([\text{sequential search}]\)

- What’s a more efficient, perhaps less straightforward solution on arrays?

Binary Search

- \textit{Binary search}: Divide-and-conquer search algorithm on arrays
  - Designed for \textit{sorted} arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?
  - What’s a recursive algorithm for binary search?

\textbf{Important Note:} The divide-and-conquer paradigm is broadly applicable!

Even though arrays don’t naturally suggest recursive strategies the way IntBinTrees and LLLists do, it can still be a useful paradigm on arrays.

Binary search is a classic algorithm!
Binary Search

- **Binary search**: Divide-and-conquer search algorithm on arrays
  - Designed for sorted arrays—uses fact that array is sorted for more efficient algorithm

- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

```python
Algorithm: BinSrch(A,v,low,high)
if low > high
    return False
else
    mid = (low+high)//2 # int division
    if v == A[mid]
        return True
    elif v > A[mid]
        return BinSrch(A,v,mid+1,high)
    else # must be v < A[mid]
        return BinSrch(A,v,low,mid-1)
```

**Problem:**

- **Input**: sorted array A, value v for which to search, integers low and high to specify range of A in which to search

- **Output**: True if v is an element of A[low.. high], False otherwise

Note: It’s the same sequence A each time. Copying or altering A (with, e.g., list slicing) would take extra time.

Important: The recursive cases bring the sub-problems closer to the base case where low > high

What would the initial call to this function be, to find v in all of A?
Binary Search

- **Binary search**: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Are you familiar with this algorithm?
  - How could search be made more efficient on a sorted array?

```
Algorithm: BinSrch(A,v,low,high)
if low > high
    return False
else
    mid = (low+high)/2 # int division
    if v == A[mid]
        return True
    elif v > A[mid]
        return BinSrch(A,v,mid+1,high)
    else # must be v < A[mid]
        return BinSrch(A,v,low,mid-1)
```

What would the initial call to this function be, to find \( v \) in all of \( A \)?

You may have noticed the specification for this is different from the original spec’n for the search problem! We could use a *wrapper* function to make this work with the original specification.
Binary Search

- **Binary search**: Divide-and-conquer search algorithm on arrays
  - Designed for sorted arrays—uses fact that array is sorted for more efficient algorithm
- This is a *classic* algorithm, worth getting to know!
  - Would this work on, say, an LList rather than an array?

```
Algorithm: BinSrch(A,v,low,high)
if low > high
    return False
else
    mid = (low+high)/2 # int division
    if v == A[mid]
        return True
    elif v > A[mid]
        return BinSrch(A,v,mid+1,high)
    else
        return BinSrch(A,v,low,mid-1)
```

We might use *linear* (sequential) search instead of binary search on an LList... do you see why?

Is there a way to make linear search on a node-based list more efficient when the list is sorted?

Binary Search Complexity

- **Binary search**: Divide-and-conquer search algorithm on arrays
  - Designed for sorted arrays—uses fact that array is sorted for more efficient algorithm
- Complexity analysis: In the worst case, how many recursive calls are there? How much work is done each time?

```
Algorithm: BinSrch(A,v,low,high)
if low > high
    return False
else
    mid = (low+high)/2 # int division
    if v == A[mid]
        return True
    elif v > A[mid]
        return BinSrch(A,v,mid+1,high)
    else
        return BinSrch(A,v,low,mid-1)
```

Are the ideas about complexity on this slide new to you?

I'll be happy to talk more about them as the semester goes along!
Binary Search Complexity

- **Binary search**: Divide-and-conquer search algorithm on arrays
  - Designed for *sorted* arrays—uses fact that array is sorted for more efficient algorithm
- Complexity analysis: In the worst case, for input A of size n, there are lg(n) recursive calls and O(1) work each time

Algorithm: `BinSrch(A,v,low,high)`
- if low > high
  - return False
- else
  - mid = (low+high)/2 # int division
  - if v == A[mid]
    - return True
  - elif v > A[mid]
    - return `BinSrch(A,v,mid+1,high)`
  - else # must be v < A[mid]
    - return `BinSrch(A,v,low,mid-1)`

Worst case time complexity: O(lg n)

Are the ideas about complexity on this slide new to you?
I'll be happy to talk more about them as the semester goes along!
<table>
<thead>
<tr>
<th>Design Paradigm</th>
<th>Analysis</th>
<th>Correctness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complexity (Efficiency)</td>
<td></td>
</tr>
<tr>
<td>Iterative</td>
<td>Counting (Exact count of operations / space used)</td>
<td>Loop invariants</td>
</tr>
<tr>
<td>Recursive</td>
<td>Solving <em>recurrences</em></td>
<td>Induction</td>
</tr>
</tbody>
</table>