CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

• HW2 in already
• HW3 out by 6pm today, due Wednesday, Feb. 19

• HW1 grading update
  – Will be returned in class on Wednesday

• Always, feel free to ask me questions about feedback on HWs!
  – Even if no points are deducted on an exercise, it’s important to understand the feedback given and follow it on future exercises
Sum, More

• Design an algorithm to solve this problem

<table>
<thead>
<tr>
<th>Input: non-negative integer ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: the sum of the integers from 0 to ( n )</td>
</tr>
</tbody>
</table>

• We’ve done an *iterative* (i.e., using loops, not recursion) algorithm

• What would a recursive algorithm for it be?

• How would you argue correctness? How could you show that it always returns \( n(n+1)/2 \)?
  – Testing could show it for a few values of \( n \), but how could we explain that it meets that specification for all values of \( n \)?
Brute Force Algorithms and Other Mathematical Functions

- How many ways are there to arrange all of the elements in a n-element list? (How about when n==0?)

  - $n!$, or $n$ factorial, defined on natural numbers as the product
    $$n! = 1 \times 2 \times \ldots (n-2) \times (n-1) \times n$$
  - $0! = 1$

- Let’s keep building our recursion muscles! How would we write recursive algorithms to compute these functions?

  This can be important for understanding the complexity of a brute force algorithm, which checks all possibilities (e.g., all arrangements of elements in a list, all subsets) to solve a problem.
Brute Force Algorithms and Other Mathematical Functions

• How many ways are there to arrange all of the elements in a \( n \)-element list? (How about when \( n = 0 \)?)
  - \( n! \), or \( n \) factorial, defined on natural numbers as the product
    \[ n! = 1 \times 2 \times \ldots \times (n-2) \times (n-1) \times n \]
    \[ 0! = 1 \]
  - 0!, or \( n \) factorial, defined on natural numbers as the product
    \[ n! = 1 \times 2 \times \ldots \times (n-2) \times (n-1) \times n \]
    \[ 0! = 1 \]

• How many subsets does a set of \( n \) elements have? (How about when \( n = 0 \)?)
  - \( 2^n \), an exponential function; \( 2^0 = 1 \)

This can be important for understanding the complexity of a brute force algorithm, which checks all possibilities (e.g., all arrangements of elements in a list, all subsets) to solve a problem.

• Let’s keep building our recursion muscles! How would we write recursive algorithms to compute these functions? (The exponential function was part of our last HW)
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?

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 i 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."
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 \(first\)
    • and an \(LList\), called \(rest\), representing all the elements after \(first\)

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

  Is this consistent with your understanding of list structures—that is, \link\ed 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?

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
    - $v$ is the element $first$ of $L'$
    - $L$ is the sublist $rest$ of $L'$

Sometimes, the empty list is written as $[]$ We could display all LLLists in [brackets], as is usual for lists

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

For example, how would you write this LLList as a list in [brackets]? $cons(1,(cons(2,cons(3,[]))))$
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 \( [x_0, ..., x_k, x_{k+1}, ..., x_n] \) |
  | 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?

Algorithm: LLRemove(i, T) // 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

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 \( [x_0, ..., x_k, x_{k+1}, ..., x_n] \) |
  | 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?

Algorithm: LLRemove(i, T) // 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