CS 375 – Analysis of Algorithms

Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

• HW4 in already
• HW3 revisions in already

• HW5 out soon, due Wednesday, Feb. 26

• HW2 back today
Definition of our **LList** Data Structure

- To be unambiguous about how we work with LLists, these will be the primitive functions defined on Llists:
  - `first(L)`: returns the *first* element of an LList `L`
  - `rest(L)`: returns 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'`

**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 LList `L` to 3? [Ans: We could do `L = cons(3,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) 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.
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)

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
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.

...to work with an LLList instead of an array?

We might use linear (sequential) search instead of binary search on an LLList... 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 # must be v < A[mid]
    return BinSrch(A,v,low,mid-1)
```

Worst case time complexity: $O(\log n)$
Two List Structures, pt. 1

• Different programming languages have different fundamental definitions for lists
• LISP is an early programming language (specified in 1958!) with a list structure similar to our LLists
  – Node-based implementation, with references between nodes in memory
  – Functions include: cons, first, rest, append (which takes two lists as input and combines them into one, as append([i0, i1], [j0, j1]) = [i0, i1, j0, j1]
  – Important to note: cons is the function for adding an element to a list, and it adds to the beginning of a list. Why do you think that is?
Two List Structures, pt. 2

- Different programming languages have different fundamental definitions for lists
- LISP is an early programming language (specified in 1958!) with a list structure similar to our LLists
- Java ArrayList and Python lists are newer, similar to each other, but different from LISP. Consider Java ArrayList:
  - Underlying implementation as an array (a contiguous block of memory), not node-based
  - Functions include: add, append / addAll (which takes a list as input and adds it to the end, as [i0, i1].addAll([j0, j1]) = [i0, i1, j0, j1]
  - Functions include sublist (list slicing), accessing by index
  - Important to note: add is the function for adding an element to a list, and it adds to the end of a list. Why do you think that is?

When writing recursive algorithms on these different kinds of lists, how would their different definitions affect the natural ways to break down problems into sub-problems over each list type?

Let’s consider an example that uses recursive functions on lists....
Map-Reduce, pt. 1: map

- Two other useful functions on lists: map and reduce
  - The map-reduce paradigm is one of the keys to Google’s success! (and that of other large-scale platforms)

- map is a function on lists that works by applying a function f to each element of a list $L = [i_0, i_1, \ldots, i_n]$ individually, and returning the list of those results

  $\text{map}(f, L)$
  
  **Input:** function $f$, list $[i_0, i_1, \ldots, i_n]$
  
  **Output:** List$[f(i_0), f(i_1), \ldots, f(i_n)]$

- Example: Use map to…
  - Add 3: Given a list of numbers $L$, add 3 to every element on the list

  $\text{map}(f, L)$
  
  **Input:** function $f$, list $[i_0, i_1, \ldots, i_n]$
  
  **Output:** List$[f(i_0), f(i_1), \ldots, f(i_n)]$
Map-Reduce, pt. 1: map

- Two other useful functions on lists: map and reduce
  - The map-reduce paradigm is one of the keys to Google’s success!
    (and that of other large-scale platforms)

- map is a function on lists that works by applying a function f to each element of a list \( L = [i_0, i_1, \ldots, i_n] \) individually, and returning the list of those results

- Example: Use map to...
  - Add 3: Given a list of numbers \( L \), add 3 to every element on the list

  \[
  \text{map}(f, L)
  \]

  Input: function \( f \), list \([i_0, i_1, \ldots, i_n]\)

  Output: \([f(i_0), f(i_1), \ldots, f(i_n)]\)

  First, create function \( add3 \):
  
  ```python
  def add3(n):
      return n + 3
  ```

  Then use it with map:
  
  ```python
  map(add3, L)
  ```

  Example:
  
  ```python
  map(add3, [42, 375])
  ```
  returns
  
  ```python
  [45, 378]
  ```

  Something similar could be done to return, say, the number of words in each of a list of files, or the number of times a particular pattern occurs, etc. ... Do you see how?
Map-Reduce, pt. 2: reduce

- Two other useful functions on lists: map and reduce
  - The map-reduce paradigm is one of the keys to Google’s success!
    (and that of other large-scale platforms)

- reduce is a function on lists that works by applying a two-argument function \( f \) to elements of a list \( L = [i_0, i_1, \ldots, i_n] \) successively

  For example, imagine you have a function \( \text{add}(x,y) \) that returns \( x + y \) then, \( \text{reduce}(\text{add},[i_0, i_1, \ldots, i_n]) \) would return \( i_0 + i_1 + \ldots + i_n \)

- So, you can think of reduce as reducing a list of values to a single value

- … but this doesn’t say how reduce is implemented
Map-Reduce, pt. 3: *map-reduce*

We'll come back to the question of how reduce is implemented—and, for that matter, how map is implemented, which also wasn’t specified!

First, though, it’s worth seeing how map and reduce work together

- *map* and *reduce* functions work well together!
- Example: Recall list of lists \( L = [L_0, L_1, \ldots, L_n] \) from before
  - How could we use map and reduce to find the length of the longest list?

  Something similar could be done to process files, data streams, etc.

  Note: With \( \text{map}(f, \{L_0, L_1, \ldots, L_n\}) \), the computations \( f(L_0), f(L_1), \ldots \) can be done in parallel. Using *map* to parallelize computation can make map-reduce computations very efficient in multi-processor environments!

Bottleneck: full *map* computation must complete before *reduce* is called

Implementing *map*

- The *map* and *reduce* functions come from the paradigm of *functional programming*
- How might one implement these functions?
  - (Recursive implementations are a good option! Do you see why?)

- For example, how might we efficiently implement \( \text{map}(f, L) \) for our LList data structure?
- How might that differ from an efficient implementation of \( \text{map}(f, L) \) for Java/Python-like lists?

\[
\text{map}(f, L)
\]

**Input:** function \( f \), list \( \{i_0, i_1, \ldots, i_n\} \)

**Output:** list \( \{f(i_0), f(i_1), \ldots, f(i_n)\} \)