CS240
Fall 2014

Mike Lam, Professor

Analysis of Python Sequences

Python Sequence Analysis

Fill in the following tables:

* amortized

Non-mutating behaviors: lists and tuples

Operation	**Running Time**
`len(data)`	$O(1)$
`data[i]`	$O(1)$
`data.count(value)`	
`data.index(value)`	$O(k+1)$
`value in data`	
`data1 == data2`	
`data[i:j]`	
`data1 + data2`	
`c * data`	

Mutating behaviors: lists only

Operation	**Running Time**
`data[i] = value`	
`data.append(value)`	$O(1)$ *
`data.insert(i, value)`	
`data.pop()`	
`data.pop(i)`	
`data.remove(value)`	
`data1.extend(data2)`	
`data.reverse()`	$O(n)$
`data.sort()`	

`data`, `data1`, and `data2` are sequences with lengths of n, n_1, and n_2, respectively

k is the index of the leftmost occurrence; m is the leftmost index of disagreement or $min(n_1,n_2)$

Python Sequence Analysis

- `len(data)` ***O(1)***
  - return the length of data
- `data[i]` ***O(1)***
  - access the element at index i
- `data.count(value)`
  - return the number of times value occurs in data
- `data.index(value)` ***O(k+1)***
  - return the index of the leftmost occurrence of value in data
- `value in data`
  - return True if value is present in data
- `data1 == data2`
  - return True if the arrays contain the same elements
- `data[i:j]`
  - extract sublist of items from index i up to but not including j
- `data1 + data2`
  - create new list with all items from data2 appended to data1
- `c * data`
  - create new list with the items in data duplicated c times
- `data[i] = value`
  - change the element at index i
- `data.append(value)` ***O(1) ****
  - add value to the end of data
- `data.insert(i, value)`
  - add value at index i
- `data.pop()`
  - remove last value from data
- `data.pop(i)`
  - remove item at index i from data
- `data.remove(value)`
  - remove leftmost occurrence of value from data
- `data1.extend(data2)`
  - append all items from data2 to data1
- `data.reverse()` ***O(n)***
  - reverse the ordering of items in data
- `data.sort()`
  - sort the items in data

* amortized

Non-mutating behaviors

- `len(data:)` $O(1)$
  - List: we track the length of the list as it changes
  - Tuple: it is set at initialization and never changed
  - Both are just lookups

Non-mutating behaviors

- `data[i]` *O(1)*
  - Arrays can be indexed in O(1)
    - One multiplication, one addition
    - In Python, also one memory dereference

Non-mutating behaviors

- `data.count(value)` $O(n)$
  - Must examine every element to see if it matches

Non-mutating behaviors

- `data.index(value)` $O(k+1)$
  - k is the index of the leftmost occurrence
    - k = n if value is not in data
  - Must examine elements up to and including the on
    we're looking for
  - O(n) is also true, because n > k

Non-mutating behaviors

- `value in data` $O(k+1)$
  - Same as previous
    - No less work to return a boolean than to return the inde

Non-mutating behaviors

- `data1 == data2` $O(m+1)$
  - m is the leftmost index of disagreement or min(n_1,n_2)
  - Worst case: examine all elements from smallest list/tuple
  - However, if we find a non-matching element, we can short-circuit

Non-mutating behaviors

- `data[i:j]` *O(j-i)*
  - Need to copy j-i elements
  - No need to visit other elements
  - Remember: data[i] provides O(1) access to individual elements

Non-mutating behaviors

- `data1 + data2` $O(n_1 + n_2)$
  - Need to copy all elements of both lists/tuples

Non-mutating behaviors

- `c * data` $O(cn)$
  - Need to copy all elements c times

Python Sequence Analysis

Fill in the following tables:

Non-mutating behaviors: lists and tuples

Operation	Running Time
`len(data)`	$O(1)$
`data[i]`	$O(1)$
`data.count(value)`	$O(n)$
`data.index(value)`	$O(k+1)$
`value in data`	$O(k+1)$
`data1 == data2`	$O(m+1)$
`data[i:j]`	$O(j-i)$
`data1 + data2`	$O(n_1+n_2)$
`c * data`	$O(cn)$

Mutating behaviors: lists only

Operation	Running Time
`data[i] = value`	
`data.append(value)`	$O(1)$ *
`data.insert(i, value)`	
`data.pop()`	
`data.pop(i)`	
`data.remove(value)`	
`data1.extend(data2)`	
`data.reverse()`	$O(n)$
`data.sort()`	

`data`, `data1`, and `data2` are sequences with lengths of n, n_1, and n_2, respectively

k is the index of the leftmost occurrence; m is the leftmost index of disagreement or $min(n_1,n_2)$

Python Sequence Analysis

- `len(data)` ***O(1)***
  - return the length of data
- `data[i]` ***O(1)***
  - access the element at index i
- `data.count(value)`
  - return the number of times value occurs in data
- `data.index(value)` ***O(k+1)***
  - return the index of the leftmost occurrence of value in data
- `value in data`
  - return True if value is present in data
- `data1 == data2`
  - return True if the arrays contain the same elements
- `data[i:j]`
  - extract sublist of items from index i up to but not including j
- `data1 + data2`
  - create new list with all items from data2 appended to data1
- `c * data`
  - create new list with the items in data duplicated c times
- `data[i] = value`
  - change the element at index i
- `data.append(value)` ***O(1) ****
  - add value to the end of data
- `data.insert(i, value)`
  - add value at index i
- `data.pop()`
  - remove last value from data
- `data.pop(i)`
  - remove item at index i from data
- `data.remove(value)`
  - remove leftmost occurrence of value from data
- `data1.extend(data2)`
  - append all items from data2 to data1
- `data.reverse()` ***O(n)***
  - reverse the ordering of items in data
- `data.sort()`
  - sort the items in data

* amortized

Mutating behaviors

- `data[i] = value` *O(1)*
  - Remember, array access/modification is O(1)

Mutating behaviors

- `data.append(value)` *O(1)* *
  - This is the amortized cost!
  - See dynamic array slides for details

Mutating behaviors

- `data.insert(i, value)` $O(n-i+1)$ *
  - Need to shift elements right, starting at index i
  - Then a single copy operation
  - Use amortized argument for expanding arrays
  - Inserting towards the beginning of a list is more expensive than inserting towards the end of a list

Mutating behaviors

- `data.pop()` *O(1)* *
  - No need to shift elements
  - Need amortized analysis because Python lists shrink themselves when the capacity is no longer needed

Mutating behaviors

- `data.pop(i)` *O(n-i)* *
  - Need to shift elements left, starting at index i
  - Removing from the beginning of a list is more expensive than removing from the end of a list
  - As with pop(), need amortized analysis because th list may shrink

Mutating behaviors

- `data.remove(value)`: $O(n)$ *
  - Need a comparison operation for $i \le k$
  - Need a copy/shift operation for $i \ge k$
  - No best/worst/average; it is technically $\Theta(n)$
  - Again, amortized analysis because the list shrinks

Mutating behaviors

- `data1.extend(data2)` $O(n_2)$ *
  - Need to copy every element of data2
  - Need amortized argument because we'll have to expand data1
  - More efficient than repeated appends
    - Not asymptotically, but in terms of actual CPU time
    - We can expand the array once, rather than repeatedly a we append

Mutating behaviors

- `data.reverse()` $O(n)$
  - Need to copy every element
    - In pairs (because it's an in-place reversal)
    - May actually be 1.5n copy operations

Mutating behaviors

- `data.sort()` *O(n log n)*
  - Naive algorithms are $O(n^2)$
    - Compare every element with O(n) other elements
  - Better algorithms use divide-and-conquer
    - Compare every element with O(log n) other elements
  - We'll discuss this more later in the semester

Python Sequence Analysis

Fill in the following tables:

* amortized

Non-mutating behaviors: lists and tuples

Operation	Running Time
`len(data)`	$O(1)$
`data[i]`	$O(1)$
`data.count(value)`	$O(n)$
`data.index(value)`	$O(k+1)$
`value in data`	$O(k+1)$
`data1 == data2`	$O(m+1)$
`data[i:j]`	$O(j-i)$
`data1 + data2`	$O(n_1+n_2)$
`c * data`	$O(cn)$

Mutating behaviors: lists only

Operation	Running Time
`data[i] = value`	$O(1)$
`data.append(value)`	$O(1)$ *
`data.insert(i, value)`	$O(n - i + 1)$ *
`data.pop()`	$O(1)$ *
`data.pop(i)`	$O(n - i)$ *
`data.remove(value)`	$O(n)$ *
`data1.extend(data2)`	$O(n_2)$ *
`data.reverse()`	$O(n)$
`data.sort()`	$O(n \log n)$

`data`, `data1`, and `data2` are sequences with lengths of n, n_1, and n_2, respectively

k is the index of the leftmost occurrence; m is the leftmost index of disagreement or $min(n_1,n_2)$

Python String Analysis

	Complexity class
Derivation `lower(), strip(), center()`	
Testing/comparison `islower(), isnumeric(), ==, <, >`	
Pattern matching `str1 in str2, find(), replace(), split()`	

Repeated concatenation
`for ch in old_str:`
`    new_str += ch`	

String behaviors

- Derivation: $O(n)$
  - `lower(), strip(), center()`
  - Creating a new string of length n inherently require
    O(n) operations
    - Copying n bytes requires O(n) CPU cycles
    - Changing the string will cost even more operations (but
      generally still O(1) per character)

String behaviors

- Testing/comparison: $O(n)$
  - `islower(), isnumeric(), ==, <, >`
  - Worst case: examine all characters
    - Most operations can short-circuit, but the asymptotic behavior is still O(n)

String behaviors

- Pattern matching: $O(mn)$
  - `str1 in str2, find(), replace(), split()`
  - Worst case: compare every character in the string to every element in the pattern
    - m characters in the pattern
    - n characters in the string
  - Usually the pattern is shorter than the string
    - The m could be considered a constant when the pattern is very short (e.g., consider searching for a single character)
  - O(n+m) is possible (see section 13.2)
    - This is O(n) if m is small relative to n

String behaviors

- Repeated concatenation: $O(n^2)$
  - `for ch in old_str:`
  - `        new_str += ch`
  - Strings are immutable in Python (and in Java)
    - new_str += ch creates a new string every time!
    - This requires O(n) copy operations
  - O(n) operations each for the n characters in old_str leads to $O(n^2)$ total
  - Use a temporary list or a comprehension instead

Python String Analysis

	Complexity Class
Derivation `lower(), strip(), center()`	*$O(n)$*
Testing/comparison `islower(), isnumeric(), ==, <, >`	*$O(n)$*
Pattern matching `str1 in str2, find(), replace(), split()`	*$O(mn)$*

Repeated concatenation
`for ch in old_str:`
`    new_str += ch`
	$O(n^2)$

Midterm next week

- Midterm is in-class on Wednesday
  - Topics: anything covered thus far in the class (including today's content)
- Review session on Monday
  - Come with questions!