CS 260: Foundations Of Computer Science

Class 11 – September 28, 2012

1

Thought for the Day

Beware the man who won't be bothered with details.

2

Today's Agenda

- Chapter 5 – Searching & Sorting

3

Fun with computers...

4

The 90-90 rule of project schedules

The first 90% of a task takes 90% of the time

and the last 10% takes the other 90% of the time.

5

Where do I start?

- **Start with your class diagram. Write the Python (or …) class for that Abstract Data Type.**
  - Write some code to test your ADT. Be sure to include some way to print it out so you can see it.
- **Build other classes in your diagrams.**
- **Write code that uses your classes… (your "main" program).**
- **Test everything!!**

6

Searching and Sorting

Chapter 5

Searching

- The process of selecting particular information from a collection of data based on specific criteria.
  - Can be performed on different data structures.
  - **sequence search** – search within a sequence.
  - **search key** (or key) – identifies a specific item.
  - **compound key** – consists of multiple parts.

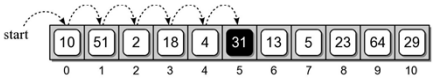
Linear Search

- Iterates over the sequence, item by item, until the specific item is found or the list is exhausted.
  - The simplest solution to sequence search problem.
  - Python's **in** operator: find a specific item.

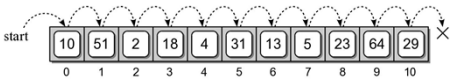
```
if key in theArray :
  print( 'The key is in the array.' )
else :
  print( 'The key is not in the array.' )
```


Linear Search Examples

- Searching for 31



- Searching for 8

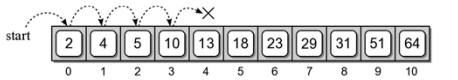


Linear Search Code

```
def linearSearch( theValues, target ) :
  n = len( theValues )
  for i in range( n ) :
    if theValues[i] == target:
      return True
```


Linear Search: Sorted Sequence

- A linear search can be performed on a sorted sequence.
- Example: searching for 8.



Linear Search: Sorted Sequence

- Similar to the unsorted sequence, with one major difference.

```
def sortedLinearSearch( theValues, target ) :
  n = len( theValues )
  for i in range( n ) :
    if theValues[i] == target :
      return True
    elif theValues[i] > target :
      return False
  return False
```

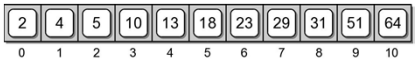

Linear Search: Smallest Value

- We can search for an item based on certain criteria.
- Example: Find the smallest value.

```
def findSmallest( theValues ):
  n = len( theValues )
  smallest = theValues[0]
  for i in range( 1, n ) :
    if theList[i] < smallest :
      smallest = theValues[i]

  return smallest
```

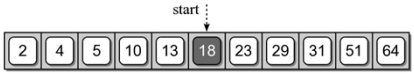

Binary Search

- The linear search has a linear time-complexity.
  - We can improve the search time if we modify the search technique itself.
  - Use a **divide and conquer** strategy.
  - Requires a sorted sequence.



Binary Search Algorithm

- Examine the middle item (searching for 10):



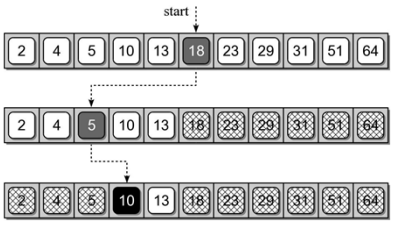
- One of three possible conditions:
  - target is found in the middle item.
  - target is less than the middle item.
  - target is greater than the middle item.

Binary Search Algorithm

- Since the sequence is sorted, we can eliminate half the values from further consideration.

Binary Search Algorithm

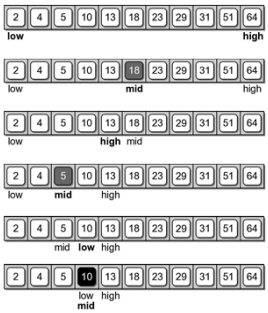
- Repeat the process until either the target is found or all items have been eliminated.



Binary Search Implementation

```
def binarySearch( theValues, target ) :
  low = 0
  high = len(theValues) - 1

  while low <= high :
    mid = (high + low) // 2
    if theValues[mid] == target :
      return True
    elif target < theValues[mid] :
      high = mid - 1
    else :
      low = mid + 1

  return False
```


Binary Search Implementation



Sorting

- The process of arranging a collection of items such that each item and its successor satisfy a prescribed relationship.
  - **sequence sort** – sorting within a sequence.
  - **sort key** – values on which items are ordered.
  - items arranged in ascending or descending order.
  - sorted in place – within the same structure.

Bubble Sort

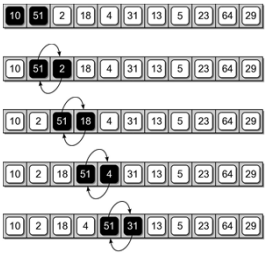
- A simple solution to the sorting problem.
- Arranges the items by
  - iterating over the sequence multiple times.
  - larger values bubble to the top (or end).

Bubble Sort Code

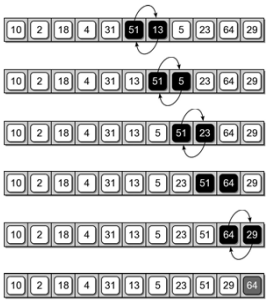
```
def bubbleSort( theSeq ):
    n = len( theSeq )
    for i in range( n - 1 ) :
        # for j in range( i + n - 1 ) :  # ERROR!
        for j in range( n - i - 1 ) :  # CORRECTION!
            if theSeq[j] > theSeq[j + 1] :
                # swap the j and j+1 items.
                tmp = theSeq[j]
                theSeq[j] = theSeq[j + 1]
                theSeq[j + 1] = tmp
```


Bubble Sort Example

- First complete iteration of the inner loop.



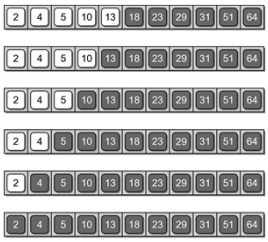
Bubble Sort Example



Bubble Sort Example

- Results after each iteration of the outer loop.

Bubble Sort Example



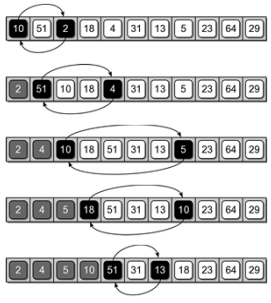
Selection Sort

- Improves on the bubble sort.
- Works in a fashion similar to what a human may use to sort a sequence.
- Instead of swapping many items,
  - repeatedly selects the next largest item from among the unsorted items.
  - requires a search to select the smallest item.

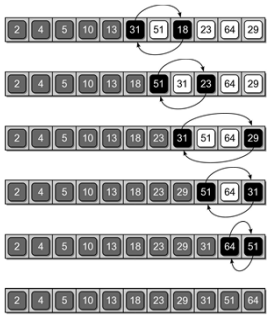
Selection Sort Code

```
def selectionSort( theSeq ):
  n = len( theSeq )
  for i in range( n - 1 ):
    smallNdx = i
    for j in range( i + 1, n ):
      if theSeq[j] < theSeq[smallNdx] :
        smallNdx = j

    if smallNdx != i :
       tmp = theSeq[i]
       theSeq[i] = theSeq[smallNdx]
       theSeq[smallNdx] = tmp
```


Selection Sort Example



Selection Sort Example



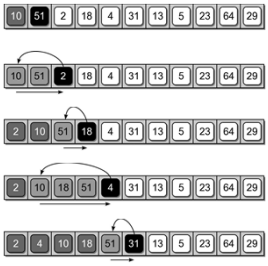
Insertion Sort

- Another commonly studied algorithm.
- Arranges the items by
  - iterating over the sequence one complete time.
  - inserts each unsorted item into its proper place.

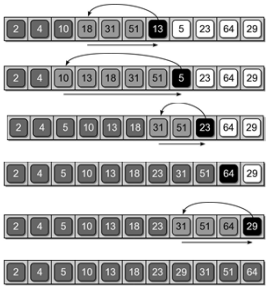
Insertion Sort Code

```
def insertionSort( theSeq ):
  n = len( theSeq )
  for i in range( 1, n ) :
    value = theSeq[i]

    pos = i
    while pos > 0 and value < theSeq[pos - 1] :
      theSeq[pos] = theSeq[pos - 1]
      pos -= 1

    theSeq[pos] = value
```


Insertion Sort Example



Insertion Sort Example

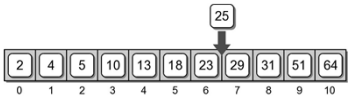


Working With Sorted Lists

- The efficiency of some algorithms can be improved when working with sorted sequences.
  - For non-static collections, it would be inefficient to re-sort a sequence for each add/remove.
  - Better to maintain a sorted sequence.

Maintaining a Sorted List

- To maintain a sorted list, new items must be inserted into their proper position.
  - Can not simply be appended at the end.
  - Must locate the proper position and use `insert()`.



Modified Binary Search

- A modified version of the binary search can be used to find the proper location of an item.

```
def findSortedPosition( theList, target ):
  low = 0
  high = len(theList) - 1
  while low <= high :
    mid = (high + low) // 2
    if theList[mid] == target :
      return mid              # Index of the target.
    elif target < theList[mid] :
      high = mid - 1
    else :
      low = mid + 1

  # Index where the target value should be.
  return low
```


Merging Sorted Lists

- Sometimes it may be necessary to merge two sorted lists into a new list.
  - For example

```
listA = [ 2, 8, 15, 23, 37 ]
listB = [ 4, 6, 15, 20 ]
newList = mergeSortedLists( listA, listB )
print( newList )
```

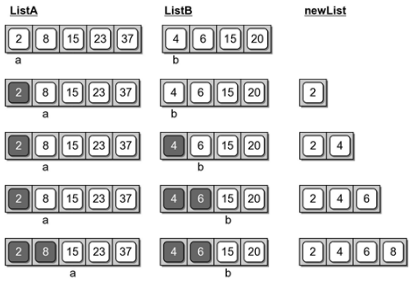
creates a new merged list

```
[2, 4, 6, 8, 15, 15, 20, 23, 37]
```

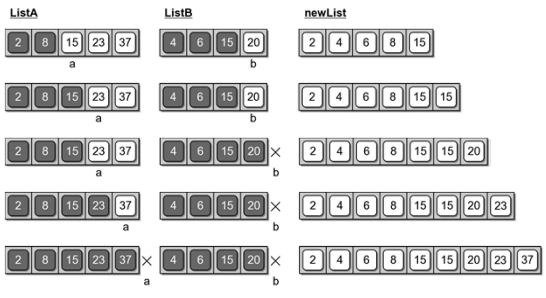

Merging Algorithm

- We can use an approach similar to what you might do, if merging two stacks of cards by hand.
  - Examine the value of the top card.
  - Select the smallest of the two.
  - Flip it over and place it on top of a third stack.
  - Repeat until one of the two original stacks is empty.
  - Finally, flip all of the cards in the remaining stack onto the top of the new stack.

Merging Example



Merging Example



Merging Implementation

```
def mergeSortedLists( listA, listB ) :
  newList = list()
  a = 0
  b = 0
  while a < len( listA ) and b < len( listB ) :
    if listA[a] < listB[b] :
      newList.append( listA[a] )
      a += 1
    else :
      newList.append( listB[b] )
      b += 1

  while a < len( listA ) :
    newList.append( listA[a] )
    a += 1

  while b < len( listB ) :
    newList.append( listB[b] )
    b += 1

  return newList
```


Merging Analysis

- Assume `listA` and `listB` each contain n items.
- The worst case depends on the number of iterations performed by all three loops combined.
  - Max iterations of the 1st loop:
    - occurs when the selection alternates between the two lists.
    - *2n -1* iterations + *1* iteration of either of the next two loops.
  - Min iterations of the 1st loop:
    - occurs when all values of one list are selected in order.
    - *n* iterations + *n* iterations of either of the next two loops.

Set ADT Revisited

- We examined the worst come run-times in the previous chapter.
  - The *contains* operation required linear time due to the linear search.
  - The quadratic times of some operations were due to the use of linear *contains* operation.

Set ADT: Run-Times

List Operation	Worst Case
s = Set()	O(1)
len(s)	O(1)
x in s	O(n)
s.add(x)	O(n)
s.isSubsetOf(t)	$O(n^2)$
s == t	$O(n^2)$
s.union(t)	$O(n^2)$
traversal	O(n)

If we could improve the speed of **these** operations…

Set ADT: Sorted List

- Can the efficiency of the set operations be improved if we used a sorted list?
- What changes would be necessary?

Set Class: Sorted List

binaryset.py

```
class Set :
  def __init__( self ):
    self._theElements = list()

  def __len__( self ):
    return len( self._theElements )

  def __contains__( self, element ):
    ndx = self._findPosition( element )
    return ndx < len( self ) and\
           self._theElements[ndx] == element
# ...
```

Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def add( self, element ):
    if element not in self :
      ndx = self._findPosition( element )
      self._theElements.insert( ndx, element )


  def remove( self, element ):
    assert element in self,
           "The element must be in the set."
    ndx = self._findPosition( element )
    return self._theElements.pop( ndx )
```


Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def isSubsetOf( self, setB ):
    for element in self :
      if element not in setB :
        return False
    return True
```


Comparing Implementations

Operation	Linear Set	Binary Set
s = Set()	O(1)	O(1)
len(s)	O(1)	O(1)
x in s	O(n)	O(log n)
s.add(x)	O(n)	O(n)
s.isSubsetOf(t)	$O(n^2)$	O(n log n)
s == t	$O(n^2)$	
s.union(t)	$O(n^2)$	
traversal	O(n)	O(n)

New Set Equals

- If we use the original `isSubsetOf()`, the result is a worst case time of O(n log n).

```
class Set :
# ...
  def __eq__( self, setB ):
    if len( self ) != len( setB ) :
      return False
    else :
      return self.isSubsetOf( setB )
```


New Set Equals

- A more efficient implementation is possible.

```
class Set :
# ...
  def __eq__( self, setB ):
    if len( self ) != len( setB ) :
      return False
    else :
      for i in range( len(self) ) :
        if self._theElements[i] != setB._theElements[i] :
          return False
      return True
```


New Set Union

- The efficiency of the set union operation can also be improved.
- Set union using two sorted lists is very similar to the problem of merging two sorted lists.
  - The new list only contains unique elements.
  - If both lists contains a given element, only one instance is placed in the new list.

Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def union( self, setB ):
    newSet = Set()
    a = 0
    b = 0
    while a < len( self ) and b < len( setB ) :
      valueA = self._theElements[a]
      valueB = setB._theElements[b]
      if valueA < valueB :
        newSet._theElements.append( valueA )
        a += 1
      elif valueA > valueB :
        newSet._theElements.append( valueB )
        b += 1
      else :
        newSet._theElements.append( valueA )
        a += 1
        b += 1
```


Set Class: Sorted List

binaryset.py

```
class Set :
# ...
  def union( self, setB ):
  # ...
    while a < len( self ) :
      newSet._theElements.append(self._theElements[a])
      a += 1

    while b < len( setB ) :
      newSet._theElements.append(setB._theElements[b])
      b += 1

    return newSet
```


Comparing Implementations

Operation	Linear Set	Binary Set
s = Set()	O(1)	O(1)
len(s)	O(1)	O(1)
x in s	O(n)	O(log n)
s.add(x)	O(n)	O(n)
s.isSubsetOf(t)	$O(n^2)$	O(n)
s == t	$O(n^2)$	**O(n)**
s.union(t)	$O(n^2)$	**O(n)**
traversal	O(n)	O(n)

On a scale of 1-5, how comfortable are you with this material so far?

1. Comfortable as LSU trying to cross the 50-yard line!
2. …
3. …
4. …
5. Comfortable as Alabama watching LSU try…!