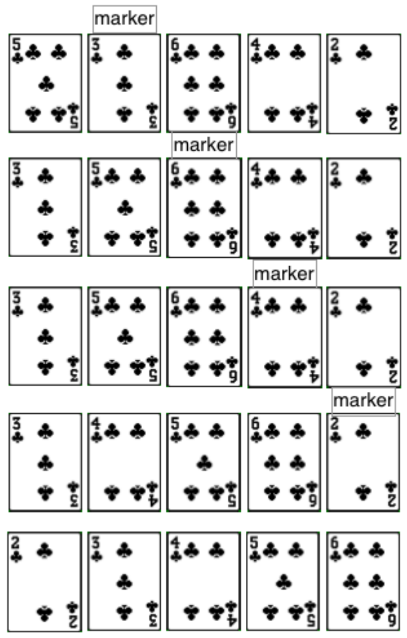
Computer Science I — CSCI-141
Insertion Sorting Lecture — Lecture (1/2)

09/26/2013

1 Problem

Write a program that sorts a deck of cards by their numeric value.



A common approach is to start with the second card in the deck and put it in order with respect to the first, then move to the third card and put it in order with respect to the first two cards, and so on until the entire deck is sorted. This algorithm is called **insertion sort**.

Write a function that performs an insertion sort on a list of numbers. Sort the list in-place, which means the list passed into the function will change and the function will not create a second list.

Something to consider with sorting algorithms is their time complexity. Can we accurately measure how long it takes to sort? Clearly this depends on the organization of the data and the algorithm used. As part of the analysis and development of insertion sort, we will classify its running time with respect to the size of the data.

2 New Python Constructs

We need some additional Python language features and functions to sort lists of data.

2.1 Strings versus Lists

There are a number of similarities between strings and lists of values. The table "Comparison of Python Sequence Capabilities" identifies similarities and differences between strings and lists (and tuples too).

`http://www.cs.rit.edu/~csci141/pub/python-sequences.html`

2.2 Lists, split() and Loops

Suppose we have this data file containing quiz grades:

```
John 80 85 93
Mary 90 93 88
Sue 91 94 87
```

We want to calculate the average for all students.

```
# create a list in which to hold the data
grades = []

# open the input file
dataFile = open( "data.txt" )

# read the file
for inputLine in dataFile:

    # read and discard the name
    elements = inputLine.split()
    name = elements [0]
    grade1 = elements [1]
    grade2 = elements [2]
    grade3 = elements [3]

    print( "Discarding name " + name )
    print( "Adding grades to list " + grade1 + " " + grade2 + \
        " " + grade3 )

    # append the grades to our list
    grades = grades + [int( grade1 )]
    grades = grades + [int( grade2 )]
    # or you can use the += operator
    grades += [int( grade3 )]


print( len( grades ) ) # prints the number of items in list
```

```
# now that we know the number of items,
# use 'range()' to iterate over the list

sum = 0
for loopCount in range( 0, len ( grades ) ):
    sum = sum +  grades[loopCount]

average = sum / len( grades )
print( "Average is " + str( average ) )
```

2.3 For and While Loops Using Indices

The Python function range() produces an *iterator* that returns values in a sequence each time code evaluates it. We often use this to generate index values in loops.

```
numbers = []
for j in range( 0, 100 ): # sets j to values 0, 1, ... 99 in turn
    numbers += [j]

# print all even numbers using a for loop

for index in range( 0, len( numbers ) ):
    if numbers[index] % 2 == 0 :
        print( numbers[index] )

# print all even numbers using a while loop

index = 0;
while index < len( numbers ):
    if numbers[index] % 2 == 0:
        print( numbers[index] )
    index = index + 1
```

3 Analysis and Solution Design

The marker above the card is identifying the boundary of the two sequences: the one that is in order, and the one that is not in order. The initial ordered sequence has just one element, so it is trivially in order. Then we move an element from the unordered sequence to the ordered sequence — making sure to put that element in the right place. When the unordered sequence is empty, the ordered sequence contains all the elements, and the process is complete.

Moving an element from the unordered sequence to the ordered sequence is referred to as *inserting* because the element is inserted into the ordered sequence. There are two aspects of this insertion, or movement: finding the proper place to put the element, and moving all the other elements to make room. Finding the proper place could be accomplished using linear search[1]. Moving all the other elements to make room could be accomplished by repeated swapping. Typically, these two aspects are combined, and repeated swapping is performed until the proper place is discovered.

For the example with cards, the marker was on the first element of the unordered sequence; however, it could have been equivalently on the last element of the ordered sequence. It turns out that this alternative makes coding the algorithm slightly simpler.

3.1 Algorithm and Implementation

```
def swap( lst, i, j ):
    """
    swap: List NatNum NatNum -> None
    swap the contents of the list at pos i and j.
    """
    temp = lst[i]
    lst[i] = lst[j]
    lst[j] = temp

def insert( lst, mark ):
    """
    insert: List(Orderable) NatNum -> None
    Move the value at index mark+1 so that it is in its proper place.
    pre-conditions:
      lst[0:mark+1] is sorted.
    post-conditions:
      lst[0:mark+2] is sorted.
    """
    index = mark
    while index > -1 and lst[index] > lst[index+1]:
        swap( lst, index, index+1 )
        index = index - 1
```

1. Binary search could also be used to find the proper place; however, moving the other elements still requires linear time.

```
def insertion_sort( lst ):
    """
    insertion_sort : List(Orderable) -> None
    Perform an in-place insertion sort on a list of orderable data.
    """
    for mark in range( len( lst ) - 1 ):
        insert( lst, mark )
```

4 Testing (Test Cases, Procedures, etc.)

Let's develop test cases to build confidence that the implementation works:

1. Sorting an empty list:
   ```
   insertion_sort( data = [] )
   expected data = []
   ```
2. Sorting a list with a single element:
   ```
   insertion_sort( data = [10] )
   expected data = [10]
   ```
3. Sorting a larger list of sorted data:
   ```
   insertion_sort( data = [10, 20, 30, 40, 50] )
   expected data = [10, 20, 30, 40, 50]
   ```
4. Sorting a larger list of unsorted data:
   ```
   insertion_sort( data = [30, 50, 20, 10, 40] )
   expected data = [10, 20, 30, 40, 50]
   ```

5 Time Complexity

Now that we are more confident that our algorithm is correct, we would like to have an idea of how efficient it is.

First notice that executing the function `insertion_sort` entails calling the function `insert` $N-1$ times. Hence the time complexity of `insertion_sort` will be $(N-1) \times T_{\texttt{insert}}(N)$, where $T_{\texttt{insert}}(N)$ is the time complexity of `insert`.

Let's consider the best case scenario. What is the best case for `insert`? If `lst[index]` $\leq$ `lst[index+1]`, then the loop is not executed and so the time complexity of `insert` is $O(1)$, or constant time. Can it be that `lst[i]` $\leq$ `lst[i+1]` for every index `i` such that $0 \leq$ `i` $< N$? Yes. That entails that the list is in order. In that case, the time complexity of `insertion_sort` is $O(N)$.

best case data: [10, 20, 30, 40, 50, 60, 70, 80]
complexity: $O(N)$

Conversely, the worst case scenario for `insert` occurs when `lst[index]` $>$ `lst[index+1]` every iteration. For a single call to `insert`, this means that all the elements in the ordered sequence are greater than the element being inserted. Can this worst case for `insert` happen for every iteration in `insertion_sort`? Yes, it happens if the list is in reverse order. The loop in `insert` then takes `mark`+1 iterations, but we'd like to express our answer in terms of N. A crude way to talk about the time complexity of `insert` is to bound `mark` from above. Since `mark` is never more than $N - 2$, the time complexity for `insert` is $O(N)$, and the time complexity for `insertion_sort` is $O(N^2)$. But is that analysis too pessimistic? In fact, the number of iterations performed by `insert` is not always N. The first time it's 1; the second time it's 2; the third time it's 3. The total number of iterations is then $1 + 2 + \cdots + (N - 1) = N(N - 1)/2$ which is again $O(N^2)$.

worst case data: [80, 70, 60, 50, 40, 30, 20, 10]
overall complexity: $O(N^2)$