CSC 108H: Introduction to Computer Programming

Summer 2012

Marek Janicki

Administration

- Midterms grades are posted.
  - They will be returned during the second break/office hours.
  - Mean was 22, Median 23, stdev 10.
- Assignment 2 update.
- Help Centre is in BA2270 2-4 M-R.

Algorithms

- So far we've looked at common components of programming languages.
- And how to get them to implement what we want to computer to do.
- We've mentioned testing as a way to get correct programs.
- How do we decide what code we want to test in the first place?

Designing Code

- When we design code, we don't necessarily want to be writing code.
  - It's a lot of work.
  - We need to worry about syntax and things that aren't core to the design.
- We would like a generic language to talk about code at a high level.

Pseudocode

- Half-code.
- A way of writing 'language-independent' code.
- All languages have variables and types.
- All languages have loops and if statements.
- In general we write at a level that we think could be implemented in any languages.

Pseudocode

- Python code:

```
for i in range(len(my_list)):
    if my_list[i]%2 == 0 :
        my_list[i] = my_list[i]+1
```

- Pseudocode:

```
for every element e in my_list
    add 1 to the even-indexed elements.
```

- Note that pseudocode does use indenting to indicate loops and separate bits of code.

Sorting

- We're going to using sorting as a case study.
- This is a core and thus very well-studied problem in the literature.
- But it's also simple to explain.
- We will be covering basic methods for sorting.
- Our methods will be inferior to pythons `list.sort()` method.

How do we approach the problem?

- Before we start actually solving the problem, we want a formal definition.
  - It is really hard to write code before you know exactly what you're trying to accomplish.
  - This formal definition allows us to start writing testing code.
- We may also want to consider some small examples to see what the result of the definition should be on them.
  - This should help catch poor definitions.

Sorting - Problem Definition

- We assume that we're given a list with *n* elements.
  - Using *n* to denote input size is standard.
- We assume that we want the list sorted in non-decreasing order.
  - non-decreasing to handle case of duplicate elements.
- We assume we can only do pair-wise comparisons.

Testing

- How might we test code that we think successfully sorts a list?
  - Hard coding tests is one way.
- Suppose we want random tests?
  - Is there something we could do to a list to check if it is sort?
  - Recall the definition of a sorted list being one in which the elements are in non-decreasing order.

Testing Criterion

- If we're sorting a list, how do we know when we're done and the list is sorted?
- One way is to check every adjacent pair of elements.
- If (in our case) the larger indexed element is at least as large as the smaller indexed element for every pair, the list is sorted.
  - Why?

Common Approaches to Finding Solutions

- Look at several inputs.
  - Try and decide which would be 'easier' to solve.
  - Then see if there's anything that one can do to make a hard input closer to one that is 'easy to solve'.
- Alternately, try and restrict the inputs in some way, and solve the restricted problem.
  - Then generalise.

Pseudocode and Problem Solving

- Pseudocode is the point at which you want to catch design problems.
- Corner cases are much easier to catch when you actually have working code.

Which of these are 'more sorted'?

- [1,2,3,4,5,6,7,8]
- [1,234,54,22,32423,324,32,234]
- [2,1,4,3,5,6,7,8]

Which of these are more sorted?

- [1, 2, 3, 4, 5, 7, 8, 6]
- [1, 2, 4, 6, 7, 8, 5, 3]
- [2, 1, 4, 3, 6, 5, 8, 7]

Which of these are more sorted?

- [1, 2, 3, 4, 5, 7, 8, 6]
- The smallest 5 elements are sorted.
- [1, 2, 4, 6, 7, 8, 5, 3]

- [2, 1, 4, 3, 6, 5, 8, 7]

Which of these are more sorted?

- [1, 2, 3, 4, 5, 7, 8, 6]
- The smallest 5 elements are sorted.
- [1, 2, 4, 6, 7, 8, 5, 3]
- The first 5 elements are sorted
- [2, 1, 4, 3, 6, 5, 8, 7]

Which of these are more sorted?

- [1, 2, 3, 4, 5, 7, 8, 6]
- The smallest 5 elements are sorted.
- [1, 2, 4, 6, 7, 8, 5, 3]
- The first 5 elements are sorted
- [2, 1, 4, 3, 6, 5, 8, 7]
- Every element is at most 1 space away from it's final destination.

Sorting Distance.

- We just saw several lists which we all 'almost sorted' in different ways.
  - The smallest n-1 elements are sorted.
  - The first n-1 elements are sorted.
  - Each element was at most 1 away from it's final spot.
- We want to generalise this, and then come up with something that can move a 'partially solved solution' to a 'fully solved solution'.

Select

- Suppose we have a list in which the first i elements are sorted and the smallest elements in the list.
- What can we do to make sure the first i+1 elements are sorted and the smallest elements within the list?
- How long does this take?

Select

- Suppose we have a list in which the first i elements are sorted and the smallest elements in the list.
- What can we do to make sure the first i+1 elements are sorted and the smallest elements within the list?
- Find the minimum in the remainder and move it to position i.
- How long does this take?
  - Something like n-i steps.

Select

select(my_lst, i)

max = 0

for j = 0 to n-i-1

if my_lst[j]>my_lst[max] then max = j

swap my_lst[max] and my_lst[n-i-1]

- max contains the index of the biggest value in my_lst[0:j]

Insertion

- Suppose we have a list in which the first i elements are sorted.
- What can we do to make sure the first i+1 elements are sorted?
- How long does this take?

Insertion

- Suppose we have a list in which the first i elements are sorted.
- What can we do to make sure the first i+1 elements are sorted?
  - Take the ith element and sort it into the first i elements.
- How long does this take?
  - Something like i steps.

Insert

```
insert(my_lst, i)
    for j = i-1 to 0
        if my_lst[j]>my_lst[j+1]
            swap my_lst[j] and my_lst[j+1]
        else return
```

- my_lst[0:i] is already sorted, except possibly for the element at position j+1.

Bubble

- Suppose we have a list in which each element is at most i steps away from its final position.
- What can we do to make every element be at most i-1 steps away from it's final position?
- How long does this take?

Bubble

- Suppose we have a list in which each element is at most i steps away from its final position.
- What can we do to make every element be at most i-1 steps away from it's final position?
  - If we swap adjacent elements that are out of order that we will decrease the maximum distance that something is out of place by 1.
- How long does this take?
  - Something like n steps.

Bubble

```
bubble(my_lst)
    for j = 0 to n-i
        if my_lst[j]>my_lst[j+1]
            swap my_lst[j] and my_lst[j+1]
```

- Note that we have no information in this function for how sorted the list is coming in, so it's hard to say anything about the loop.

From single steps to sorting.

- Now we have 3 functions that can take a partially sorted list, and make it into slightly more partially sorted list.
- How can we take these functions and make a general sorted list?

Calling partial sorting function repeatedly.

- We note that for selection and insertion, every time we call them, we can call them again but increase the parameter by i.
- sort(my_lst):
- for i = 1 to n

  partial_sort(my_lst, i)
- Where partial_sort is insert or select.

Calling partial sorting function repeatedly.

- Bubble_sort is also a partial sort?
- How many times do we need to call it before the list is sorted?
- n times.
  - Each time the maximum amount an element is out of place decreases by 1.
- sort(my_lst):
- for i = 1 to n
-       bubble(my_lst)

Break, the first.

How can we optimise Bubble sort?

```
sort(my_lst):
for i = 1 to n
    bubble(my_lst)
bubble(my_lst)
  for j = 0 to n-1
    if my_lst[j]>my_lst[j+1]
        swap my_lst[j] and my_lst[j+1]
```

Optimising Bubblesort

- First Observation:
- Once we bubble a list once, then the maximum elt is at the end.
- If we do it i times, then the last i elements are the largest i elements.
- So if we know how many times we've bubbled, we can stop early.

Optimising Bubblesort

```
bubble(my_lst,i)
    for j = 0 to n-i-1
        if my_lst[j]>my_lst[j+1]
            swap my_lst[j] and my_lst[j+1]
```

- The last my_list[-(i-1):] is sorted and contains the largest i elements.

Optimising Bubblesort

- If bubble doesn't need to swap anything, the list is sorted. So we can rewrite bubble sort to check that, and finish early if it can.
- This means we need to use a while loop to call bubble instead of a for loop.
- For now let's think about this optimisation separate from the first one.

Optimising Bubblesort

```
bubblesort(my_lst)
    while bubble(my_list):
        pass
bubble(my_lst)
    inversion = False
    for j = 0 to n
        if my_lst[j]>my_lst[j+1]
            swap my_lst[j] and my_lst[j+1]
            inversion = True
   return inversion
```

Combining the optimisations.

- How can we combine these optimisations?
- When we combine them, we should be able to improve the first one.
- Note, that this is too large for slides, so you must download the code to see the solution.

Loop Invariants

- Often times loops can be hard to implement, or it can be unclear what a loop is doing.
- A useful tool for analysing loops is a loop invariant.
- A loop invariant is a statement that is true every time to loop begins.
  - So it depends on the loop index.
- They have both informative and imperative functions.

Loop Invariant Example

```
for j = 0 to n-i-1
        if my_lst[j]>my_lst[j+1]
                swap my_lst[j] and my_lst[j+1]
```

- Here we see that the jth element is always the biggest that we've seen. So a loop invariant would be:
  - my_lst[j] is the largest element in my_lst[0:j]
    - This tells us a truth at the beginning of any iteration.
    - It also tells us what we need to do in any iteration.

Pseudocode and Loop Invariants

- Loop invariants are really useful in pseudocode, since they point towards the overall design of the program.
- Also can be useful in finding +/- 1 errors.
  - That is, they are useful in both the actual and pseudocode stages.

Sorting Overview

- We covered three types of sort: Bubble, Insertion, and Selection.
- Selection sort minimises swaps.
- Insertion sort is optimal for small data.
- Bubble sort is optimal for nearly sorted data.

Sorting in practice.

- In practice bubble, selection, and insertion sort are all sort of slow.
- There are better sorting methods out there.
  - The most commonly used ones are merge, heap and quick sort).
  - These all rely on recursion.
- Python uses an adaptive form of merge sort.
- Bubble and insertion sort have specific instances in which they are useful and are used.

Break the second.