CSC 108H: Introduction to Computer Programming

Summer 2011

Marek Janicki

Administration

- Assignment updates.
- Codelab updates.

Algorithms.

- So far we've looked at common components of programming languages.
- And we've seen very locally how to think about using them.
- We haven't talked a whole lot about how to approach bigger problems.
- We've mentioned testing as a way to get correct programs.
  - Haven't mentioned analysing the problem.

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.

Pseudocode.

- At this point we're familiar with the basic tools of programming.
- This means we can use simple abstractions and reliably go to and from code with them.
- So when we're designing algorithms we tend to do so using cognitive shortcuts and simplifications.
- This rough code is called Pseudocode.

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 seperate bits of code.

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.
  - This means our methods will be robust enough to handle any class that implements `__cmp__`

Sorting - Stopping 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?
  - Can we use this to get an algorithm?

Towards a sorting algorithm.

- We know that if we compare all the adjacent elements and there are no *inversions* then we're done.
  - An inversion is when a pair of elements are swapped.
- So what if we try and force this to be true by going through the array and swapping any adjacent inversions that we see.

Towards a sorting algorithm.

- That is:

  for every index i but the last in my_lst

  if my_lst[i]>my_lst[i+1]

  swap my_lst[i] and my_lst[i+1]

  - What happens after one pass?
  - Do we need to do more than one?

Bubblesort.

```
bubblesort(my_lst)
    for i = 0 to n-1
        bubble(my_lst, i)

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]
```

Correctness.

- So now we have a sorting algorithm, but how do we know it's true?
- A useful tool for analysing loops is a loop invariant.
- A loop invariant is a statement that is true at every point in the loop.
  - So it depends on the loop index.
- They have both informative and imperative functions.

Loop invariant examples.

```
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.

Loop Invariant Examples.

```
bubblesort(my_lst)
    for i = 0 to n-1
        bubble(my_lst, i)
```

- Here the loop invariant might be:
  - The last i positions of my_lst are sorted in non-decreasing order and contain the i largest elements in my_lst.

Break, the first.

Improving Bubblesort.

```
bubblesort(my_lst)
    for i = 0 to n-1
        bubble the 0th element up to i.
```

- Bubbling elements up is a lot of work. We might potentially have to touch a lot of the list elements and do lots of swaps.
- Can we decrease the number of swaps?

Decreasing the number of swaps.

- Well, if we look at our loop invariant, that might give us a clue:
  - The last i positions of my_lst are sorted in non-decreasing order and contain the i largest elements in my_lst.
- Is there a way we can do with without bubbling elements up?
  - So as to minimise swaps?

Selection sort.

```
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]
```

- What's our loop invariant?

Selection sort.

```
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]

Selection sort.

```
selectionsort(my_lst)
    for i = 0 to n-1
        select(my_lst, i)
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]
```

One last sort.

- Called shell sort or insertion sort.
- This one is effectively how we sort cards when we're dealt them in a card game.
- We're going to use the same general structure:

for every element in list

    call some function.

some function

    loop through some part of list and do something.

Insertion sort.

for every element in list

    call some function.

some function

    loop through some part of list and do something.

- Note, the above is not structured enough to be pseudocode.
- It's closer to gibberish than anything else.
- Can we fix it?

Insertion sort.

- If it were cards we'd have something like:

  for each card you pickup

  do something.
- Can we think of a loop invariant for the for loop?
- Or structure for the something part?

Insertion sort.

```
insertionsort(my_lst)
    for i = 1 to n-1
        insert(my_list, i)
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
```

Loop invariants.

- main loop: The first i elements are already sorted.
- Insert Loop: my_lst[0:i] is already sorted, except possibly for the element at position j+1.

Summary

- We introduce the idea of abstraction (through pseudocode) and algorithmic thinking.
- It is alot easier to catch conceptual errors in the algorithmic stage than in the testing stage.
  - Loop invariants are a useful tool for thinking about algorithms.
- It is much easier to catch index errors and corner case errors while testing.
- Suggests sort of a division of labour.

Summary

- We covered three types of sorting.
  - Bubble, Selection and Insertion.
- There are better sorting methods out there.
  - They generally rely on recursion which we're not covering. (Merge, Heap, Quick).
    - This is covered in later courses.
  - Python uses an adaptive form of merge sort.
    - Adaptive because for small numbers it uses insertion sort.