CSC108H Lecture 11

Dan Zingaro
OISE/UT

July 23, 2009

Debugging a Simple Algorithm

```
def find_min_max(L):
  '''Return min and max values in int list L.'''

  min = 0
  max = 0
  for value in L:
    if value > max:
      max = value
    if value < min:
      min = value
  return min, max
```

- If we aren't careful in our analysis or testing, we might think this function is correct
- What is the (possibly incorrect) assumption it makes? How can we resolve this?
- What happens if the list is empty? What can we do?

Indices of a Character

- Given string `text` and single-character string `char`, we want to find all indices of `text` that hold `char`
- We can use `text.find (char)` to find the first occurrence of `char` in `text`
- `find` takes an optional second parameter indicating the index at which to **start** the search
- We can use this to find future occurrences of `char`

Indices of a Character (bad_get_indices.py)

This is bad. What's wrong with it?

```
def get_indices(text, char):
  '''Return a list containing the indices where str char
  appears in str text.'''

  index = 0
  indices = []
  while text.find(char, index) != -1:
    index = text.find(char, index)
    indices.append(index)
  return indices
```

Indices of a Character... (bad_again.py)

This is bad too. What's wrong now? Hint: we can write a correct version with only one call to `find`, not two!

```
def get_indices(text, char):
  '''Return a list containing the indices where str char
  appears in str text.'''

  index = 0
  indices = []
  while index != -1:
    indices.append(text.find(char, index))
    index = text.find(char, index + 1)
  return indices
```

Minimum in a List (min1.py)

- We will explore various techniques for finding the minimum element in a list (without using `min`)
- Approach 1: sort the list and get the first element

```
def find_min1(L):
  '''Return the smallest element from list L.
  Modifies L!'''

  L.sort()
  return L[0]

def find_min2(L):
  '''Return the smallest element from list L.'''

  L = L[:] # L is not modified
  L.sort()
  return L[0]
```

Minimum in a List (min2.py)

- Approach 2: check each value one-by-one (does not modify the list)

```
def find_min(L):
  '''Return the smallest element from list L.'''

  min = L[0]
  for item in L:
    if item < min:
      min = item
  return min
```

Second Smallest in a List (min3.py)

- Approach 1: sort the list and get the second element

```
def find_second_smallest(L):
  '''Return the second smallest element from list L.
  Modifies L!'''

  L.sort()
  return L[1]
```

Second Smallest in a List (min4.py)

- Approach 2: find the smallest, remove it, find the smallest

```
from min2 import find_min

def find_second_smallest(L):
  '''Return the second smallest element from list L.
  Modifies L!'''

  smallest = find_min(L)
  L.remove (smallest)
  return find_min(L)
```

Second Smallest in a List (min5.py)

- The same approach, but putting the smallest back in the list (somewhere!)

```
from min2 import find_min

def find_second_smallest(L):
  '''Return the second smallest element from list L.
  Modifies L!'''

  smallest = find_min(L)
  L.remove (smallest)
  sec_smallest = find_min(L)
  L.append (smallest)
  return sec_smallest
```

Second Smallest in a List (min6.py)

- What if we want the **index** of the second smallest element instead of the element itself?
- We might be off-by-one if we aren't careful!

```
from min2 import find_min

def find_second_smallest(L):
  '''Return index of second smallest element from L.'''

  smallest = find_min(L)
  smallest_loc = L.index (smallest)
  L.remove (smallest)
  sec_smallest = find_min(L)
  sec_smallest_loc = L.index (sec_smallest)
  if sec_smallest_loc >= smallest_loc:
    sec_smallest_loc += 1
  L.insert (smallest_loc, smallest)
  return sec_smallest_loc
```

Second Smallest in a List (min7.py)

- Approach 3: check each element in the list one-by-one

```
def find_second_smallest(L):
  '''Return the second smallest element from list L.'''

  if L[0] < L[1]: # initialize
    smallest = L[0]
    sec_smallest = L[1]
  else:
    smallest = L[1]
    sec_smallest = L[0]

  for item in L[2:]:
    if item < smallest: # new smallest and sec_smallest
      sec_smallest = smallest
      smallest = item
    elif item < sec_smallest: # only a new sec_smallest
      sec_smallest = item
  return sec_smallest
```

Algorithm Analysis

- Algorithm analysis is about determining the computing resources required by an algorithm
- Since there's often more than one way to solve a problem, evaluating the computing resources required by an algorithm allows us to determine its efficiency compared to other algorithms
- Computing resources typically refers to the execution "time" an algorithm requires, but may also refer to the amount of memory it uses
- We'll go through several approaches for solving the same problem, and compare them

Maximum Segment Sum

- The problem we'd like to solve is the **maximum segment sum**
- Input: Python list `s` of `n` integers
- Output: maximum sum of any segment of `s`
- A segment is a slice of the list (i.e. a contiguous portion of the list)
- For example, the segments of the list `[2, -4, 6]` are `[]`, `[2]`, `[-4]`, `[6]`, `[2, -4]`, `[-4, 6]`, `[2, -4, 6]`
- Question: what is the maximum segment sum here?
- Question: is `[2,6]` a segment of `[2, -4, 6]`?

Exercise: Maximum Segment Sum

what is the maximum segment sum in each case?

- A. `[4, -3, 9, -5]`
- B. `[-5, 10, -6, 2]`
- C. `[3, -2, 4, 8, -3, 5, -10, 20]`
- D. `[6, 2, 4, 1, 2]`
- E. `[1, -2, 3, 5, -1]`
- F. `[-3, -2, -1]` (careful!)

Segment Observations

- As we have seen, the maximum segment sum of a list of all positive numbers is the list itself
- When we have some negative numbers in the list, it gets trickier. Which ones do we include?
- If all numbers are negative, the maximum sum is 0, realized by the empty segment
- We can list every segment of a list by pairing each possible starting point with each possible ending point

Summing each Segment

- Our first approach will compute the sum of each possible segment in the list, and compare it to the maximum so far
- For example, here is how this would start operating on `[4, -3, 9, -5]`

```
max: 0
sum of segment [4] = 4
max: 4
sum of segment [4,-3] = 4-3 = 1
max: 4
sum of segment [4,-3,9] = 4-3+9 = 10
max: 10
sum of segment [4,-3,9,-5] = 4-3+9-5 = 5
max: 10
... continue with sum of segment [-3] ...
```

Solution (A)wful (segs1.py)

Note the triply nested for-loops.

```
def max_seg (s):
  max_so_far = 0
  for lower in range (len(s)):
    for upper in range (lower, len(s)):
      sum = 0
      for i in range(lower, upper+1):
        sum = sum + s[i]
      max_so_far = max(max_so_far, sum)
  return max_so_far
```

Improving the Solution

- The above (awful) solution is well-named!
- When we found the sum of segment `[4, -3, 9, -5]`, we computed $4 - 3 + 9 - 5$
- But we **just** computed $4 - 3 + 9$ to sum the previous segment! We're redoing almost all of that work
- In general, when it sums the elements between bounds l and u, it will repeat all of the work it did when previously finding the sum between bounds l and $u - 1$
- Our second approach will compute the sum of a segment by adding **only** the new rightmost element's value to the sum of the segment without that element

Remembering Sums

- For example, here is how we can operate on `[4, -3, 9, -5]`

```
max = 0
sum = 4
sum of segment [4] = 4
max = 4
sum = 4 - 3 = 1
sum of segment [4,-3] = 1
max = 4
sum = 1 + 9 = 10
sum of segment [4,-3,9] = 10
max = 10
sum = 10 - 5 = 5
sum of segment [4,-3,9,-5] = 5
max = 10
sum = -3
... continue with sum of segment [-3] ...
```

Solution (B)ad

Note the doubly nested for-loops.

```
def max_seg (s):
  max_so_far = 0
  for lower in range (len(s)):
    sum = 0
    for upper in range (lower, len(s)):
      sum = sum + s[upper]
      max_so_far = max(max_so_far, sum)
  return max_so_far
```

Improving the Solution, Again

- Solution Bad still has some inefficiency
- Consider finding the maximum segment sum in a long list
- e.g. for simplicity, `[0, 1, 2, 3, 4, 5, 6, 7, 8]`
- Bad will calculate each segment sum starting from 0 by adding each element once (instead of over and over like in Awful)
- But, besides the 0, this is exactly what we do later when calculating the segment sum from 1 to 8
- And besides 1, the rest of that work is repeated **again** when we calculate the segment sum between 2 and 8! And so on ...

Clickers: Awful Review

How many times does Awful compute the sum `1 + 2 + 3` in the following list?

```
[0, 1, 2, 3, 4]
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

Clickers: Bad Review

How many times does Bad compute the sum `1 + 2 + 3` in the following list?

```
[0, 1, 2, 3, 4]
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

Just One Pass

- Our third approach will "look at" each element of the list just once, in a left-to-right scan of the list
  - The Awful and Bad approaches made multiple left-to-right scans
- Assume we've been able to find the maximum sum over all segments in `s[:i]`, and we have it stored in `max`
- Now, we want to extend this to the maximum sum over all segments in `s[:i+1]`
- The key observation is that the only **new segments** we have not considered in `s[:i+1]` are those that **end at** `s[i]`
- The maximum segment sum in `s[:i+1]` will then be the maximum of `max`, and the maximum segment sum of those segments ending at `s[i]`

Just One Pass...

- Consider list `[4, -3, 9, -5]`, and assume we've found that the maximum segment sum of `[4,-3]` is 4
- To extend this to the maximum segment sum in `[4,-3,9]` we compare 4 with the maximum segment sum of those segments ending at 9
- These **new** segments are `[]`, `[9]`, `[-3, 9]`, `[4, -3, 9]`
- 10 is the maximum sum of these new segments, and since $10 > 4$, 10 is our maximum segment sum of `[4, -3, 9]`

Just One Pass...

- Now, let's extend to the whole list `[4, -3, 9, -5]`
- We do not want to go through each segment ending at -5, because that duplicates all of the work we just did for 9!
- But, if we remembered what the maximum segment sum ending at 9 was, we could easily extend this to the maximum segment sum ending at -5 without having to look back in the list
- Exercise: how can we do this? In general, if we have remembered the maximum segment sum ending at `s[i]`, what is the maximum segment sum ending at `s[i+1]`?
- What if the element at `s[i+1]` makes the segment ending at `s[i+1]` negative?
- ...

Solution (C)ool

Just one for-loop now!

```
def max_seg (s):
  max_so_far = 0
  max_ending_here = 0
  for i in range (len(s)):
    max_ending_here = max(max_ending_here+s[i], 0)
    max_so_far = max(max_so_far, max_ending_here)
  return max_so_far
```

Timing the Algorithms (in seconds)

Size	**Awful**	**Bad**	**Cool**
200	0.46	0.03	0.0
300	1.47	0.04	0.0
400	3.49	0.08	0.0
500	6.76	0.13	0.0
600	11.53	0.20	0.0
700	18.06	0.26	0.0
800	27.24	0.35	0.0
900	38.11	0.43	0.0
100000	YAWN	yawn	0.17

Algorithm Analysis

- The above timing tells us something about which algorithm is the fastest
- We can't rely on the wallclock execution time, though, because
  - The time required for a program to execute may vary from computer to computer (a program will probably be a lot slower on a PC from the 90's than a brand-new PC that has a wicked-cool processor)
  - A "fast algorithm" on a slow computer may be slower than a "slow algorithm" on a fast computer on certain inputs
- Instead, we will characterize Awful as a **cubic-time** algorithm, Bad as a **quadratic-time** algorithm, and Cool as a **linear-time** algorithm