CSC108H Lecture 30

Dan Zingaro

November 21, 2012

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 `L` of `n` integers
- Output: maximum sum of any segment of `L`
- A segment is a slice of the list (i.e. a contiguous portion of the list)
- For example, the segments of the list `[-4, 2, 6]` are `[]`, `[-4]`, `[2]`, `[6]`, `[-4, 2]`, `[2, 6]`, `[-4, 2, 6]`
- ... and the maximum segment sum of this list is 8 (from `[2, 6]`

ConcepTest

`[2, -5, 8, -6, 10]`

What is the maximum segment sum in this list?

- A. 8
- B. 9
- C. 10
- D. 12
- E. 20

ConcepTest

`[2, -5, 8, -15, 10]`

What is the maximum segment sum in this list?

- A. 3
- B. 8
- C. 10
- D. 12
- E. 15

Segment Observations

- The maximum segment sum of a list of all positive numbers is the sum of all elements in the list
- 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 (from the empty segment)
- We can check every segment of a list by pairing each possible starting point with each possible ending point

Approach A

- Our first approach will compute the sum of each 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 changes to: 4
  sum of segment [4, -3] = 4-3 = 1
max remains: 4
  sum of segment [4, -3, 9] = 4-3+9 = 10
max changes to: 10
  sum of segment [4, -3, 9, -5] = 4-3+9-5 = 5
max remains: 10
  sum of segment [-3] = -3
max remains: 10
  ...
```

Approach (A)wful (segs1.py)

Note the triply nested for-loops.

```
def max_segment_sum(L):
  '''(list of int) -> int
  Return maximum segment sum of L.
  '''
  max_so_far = 0
  for lower in range(len(L)):
    for upper in range(lower, len(L)):
      sum = 0
      for i in range(lower, upper+1):
        sum = sum + L[i]
      max_so_far = max(max_so_far, sum)
  return max_so_far
```

ConcepTest

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

How many times does Approach A compute the sum $1 + 2 + 3$ in the above list?

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

ConcepTest

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

How many times does Approach A compute the sum $0 + 1 + 2$ in the above list?

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

Improving the Approach

- The above (awful) approach is well-named!
- To find the sum of segment `[4, -3, 9]`, we computed $4 - 3 + 9$
- But then, to find the sum of `[4, -3, 9, -5]`, we redo the $4 - 3 + 9$ again!
- 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

Approach B

- Our second approach, operating on `[4, -3, 9, -5]`:

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

Approach (B)ad (segs2.py)

Note the doubly nested for-loops.

```
def max_segment_sum(L):
  '''(list of int) -> int
  Return maximum segment sum of L.
  '''
  max_so_far = 0
  for lower in range(len(L)):
    sum = 0
    for upper in range(lower, len(L)):
      sum = sum + L[upper]
      max_so_far = max(max_so_far, sum)
  return max_so_far
```

Improving the Approach, Again

- Approach Bad still has some inefficiency
- Consider list `[0, 1, 2, 3, 4, 5, 6, 7, 8]`
- Bad will calculate all segment sums 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 sums starting from 1
- And besides 1, the rest of that work is repeated **again** when we calculate the segment sums starting from 2

Just One Pass

- There is a third approach that examines each element just once
- It makes a single left-to-right pass over the list
  - The Awful and Bad approaches made multiple left-to-right passes
- Assume `m` is the maximum sum over all segments in `L[:i]`
- Now, we want to extend this to the maximum sum over all segments in `L[:i+1]`
- The key observation is that the only **new segments** we have not considered in `L[:i+1]` are those that **end with** `L[i]`
- The maximum segment sum in `L[:i+1]` will then be the maximum of `m`, and the maximum segment sum of those segments ending at `L[i]`

Approach (C)ool (segs3.py)

Just one for-loop now!

```
def max_segment_sum(L):
  '''(list of int) -> int
  Return maximum segment sum of L.
  '''
  max_so_far = 0
  max_ending_here = 0
  for i in range(len(L)):
    max_ending_here = max(max_ending_here + L[i], 0)
    max_so_far = max(max_so_far, max_ending_here)
  return max_so_far
```

Timing the Algorithms (in seconds)

List Length	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