CSC108H Lecture 31

Dan Zingaro

November 23, 2012

Measuring Runtime of Algorithms

- We want to understand how the execution time for an algorithm increases when the problem size increases
- One approach: measure the execution time on different-sized problems
- We can do this with our Maximum Segment Sum algorithms
- We will see that Awful grows much more quickly than Bad and that Bad grows more quickly than Cool

Measuring Runtime: the Problem

- Precise runtimes may not be useful as a way to describe algorithm performance
- They depend on the specific machine on which the program is being run
- So in addition to runtime, we'd have to include information about OS, processor speed, etc.
- More useful is an abstract characterization of the rate at which an algorithm's runtime grows
- We will characterize an algorithm as linear, quadratic, cubic, etc.

Linear-time Algorithms (linear1.py)

- We have a linear-time algorithm when a linear increase in the problem size leads to a linear increase in the execution time
- The below function is linear-time. What happens when we call it with 1000000? 2000000? 3000000?

```
def linear(n):
  total = 0
  for i in range(n):
    total += 1
  return total
```

Linear-time Algorithms ... (linear2.py)

- This function is **still** linear-time
- If we increase the problem size by d (a linear increase), the number of steps always increases by $2d$

```
def linear(n):
  total = 0
  for i in range(n):
    total += 1
    total += 1
  return total
```

ConcepTest

Is this algorithm linear-time?

```
def mystery(n):
  total = 0
  for i in range(n):
    total += 1
  for i in range(n):
    total += 1
  return total
```

- A. Yes
- B. No

Quadratic-time Algorithms (quadratic.py)

- This function is **not** linear-time
- Increasing the problem size by fixed increments causes nonlinear increases in execution time
- e.g. increasing the input size from 10 to 20 causes less of an execution-time increase than does increasing the input from 20 to 30
- For size n, this function performs a number of steps proportional to n^2

```
def quadratic(n):
  total = 0
  for i in range(n):
    for j in range(n):
      total += 1
  return total
```

Cubic-time Algorithms (cubic.py)

- This function is even worse than quadratic-time
- For size n, this function performs a number of steps proportional to n^3

```
def cubic(n):
  total = 0
  for i in range(n):
    for j in range(n):
      for k in range(n):
        total += 1
  return total
```

ConcepTest

```
def mystery(n):
  total = 0
  for i in range(n):
    for j in range(10000):
      for k in range(50):
        total += 1
  return total
```

This algorithm is:

- A. Linear (n)
- B. Quadratic (n^2)
- C. Cubic (n^3)
- D. Not one of these three

ConcepTest

```
def mystery(n):
  total = 0
  for i in range(n):
    for j in range(n):
      for k in range(50):
        total += 1
  return total
```

This algorithm is:

- A. Linear (n)
- B. Quadratic (n^2)
- C. Cubic (n^3)
- D. Not one of these three

ConcepTest

```
def f(n):
  sum = 0
  while n > 0:
    sum = sum + n ** 2
    n = n // 2
  return sum
```

This algorithm is:

- A. Better than linear
- B. Linear (n)
- C. Quadratic (n^2)
- D. Worse than quadratic

ConcepTest

Your coworker tells you he's found a way to change your algorithm so it does HALF as much work. Knowing that the algorithm is currently $O(n^2)$, you say:

- A. That's great. This reduces our runtime to $O(n)$ so it will take less time to run.
- B. That's great. Our runtime stays the same, but it will finish in around half the time.
- C. That doesn't do anything; our runtime stays the same, so it takes the same amount of time to run.
- D. That doesn't do anything; it does reduce our runtime to $O(n)$ but that doesn't mean it runs faster.

ConcepTest

Your coworker tells you he's found a way to clean up some code; but in doing so, the algorithm will now do TWICE as much work. Knowing that the algorithm is currently $O(n^2)$, you say:

- A. That's bad. This increases our runtime to $O(n^4)$ so it will take longer to run.
- B. That's bad. Our runtime stays the same, but it will finish in around twice the time.
- C. That's great! Our runtime stays the same, so it takes the same amount of time to run and the code is cleaner.
- D. That's great!. This increases our runtime to $O(n^4)$ but that doesn't mean it runs slower and the code is now cleaner.

ConcepTest

(I didn't get to this in lecture, but it's good practice.)

```
def f(s):
  num = 0
  for c1 in s:
    for c2 in s:
      if c1 == c2:
        num = num + 1
  return num
```

The problem size is the length of `s`. This algorithm is:

- A. Better than linear
- B. Linear (n)
- C. Quadratic (n^2)
- D. Worse than quadratic