Runtime Analysis

- Measuring elapsed runtime is one way to get a sense of an algorithm's efficiency
- However, runtime depends on the computer on which the program is run, the programming language being used, and lots of other factors
- Instead, we will characterize time efficiency in a way that
  - Is independent of the particular computer, and
  - Ignores small differences between algorithms

Runtime Analysis...

- A **step** is a basic unit of computation that can be carried out in a fixed amount of time
- We want to determine the number of steps that an algorithm takes as a function of its input size
- How we define input size depends a lot on the problem
  - e.g. for the anagram problem, the input size involves word length and number of words in the dictionary
- Typically the input size is the number of elements in the input
- For a given algorithm, we'll use the function $T(n)$ to denote the number of steps that the algorithm takes on input size n

Segment-Sum

```
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
```

Analyzing the Segment-Sum Algorithm

- Question: how many times is `sum = sum + L[i]` executed? (This is the same as asking for the number of iterations of the inner loop)
- The outer loop executes n times
- The middle loop is executed at most n times for each iteration of the outer loop
- So, the middle loop executes at most n^2 times
- The inner loop is executed at most n times for each iteration of the middle loop
- So, the inner loop executes at most n^3 times
- Similarly, `sum = 0` and `max_so_far = ...` are executed at most n^2 times
- Conclusion: an upper bound on the number of steps we execute is $n^3 + 2n^2$

Analyzing Segment-Sum Algorithm...

Observation: as n increases, the n^3 term in $n^3 + 2n^2$ comes to dominate, and $2n^2$ doesn't contribute much to $T(n)$

n	n^3	$2n^2$
10	1000	200
20	8000	800
30	27000	1800
40	64000	3200
50	125000	5000
100	1000000	20000
200	8000000	80000
300	27000000	180000
400	64000000	320000
500	125000000	500000
1000	1000000000	2000000

Big Oh

- Even if we had $T(n) = n^3 + 4n^2$, or $T(n) = n^3 + 50n^2$, when n becomes large, the n^2 term will be much smaller than the n^3 term
- To measure the efficiency of an algorithm, we focus only on the approximate number of steps it takes
- We are not concerned with deriving an exact value for $T(n)$, so we ignore its constant factors and nondominant terms
- Big Oh notation makes this idea precise

Big Oh...

- Say we have an algorithm whose running time on input of size n is $f(n)$
- Three requirements:
  - We want to bound $f(n)$ from above by $g(n)$
  - We want $g(n)$ to be a reasonable estimate; that is, it only "overestimates" by a constant c
  - We only require $g(n)$ to be such an estimate for sufficiently large values of n (since we don't care about small instances)
- This all amounts to requiring that $f(n) \leq cg(n)$ for all $n >= n_0$ and positive constant c
- We then say that $f(n) = O(g(n))$

Properties of Big Oh

- Constant factors disappear
  - e.g. $6n$ and $n/2$ are both $O(n)$
- Lower-order terms disappear
  - e.g. $n^5 + n^3 + 6n^2$ is $O(n^5)$
- We can often just look at the loop structure of a program to determine its growth rate

Big Oh Proof

Prove that the triply-nested segment-sum code is $O(n^3)$.

- You have to show that $n^3 + 2n^2 <= cn^3$ for all $n >= n_0$, where n_0 and c are positive constants

Big Oh Proof

Prove that the triply-nested segment-sum code is $O(n^3)$.

- You have to show that $n^3 + 2n^2 <= cn^3$ for all $n >= n_0$, where n_0 and c are positive constants
- Dividing by n^3, we get $1 + \frac{2}{n} \leq c$
- We can make this true (and complete the proof) if we set $c = 3$ and $n_0 = 1$

Big Oh Approximations

- Big oh gives us an upper bound on the time that our algorithm takes to execute
- It gives no guarantee that the bound is "close" to what actually happens
- It's equally valid to say that the segment-sum code is $O(n^3)$, $O(n^6)$, $O(2^n)$, etc.
- However, saying that it is $O(n^3)$ gives us the most useful information
- $O(n^3)$ is a "tight bound" (i.e. most accurate bound): there is no smaller function q for which it is still $O(q)$

What is the Time Efficiency? (1)

```
def bigoh1(n):
  sum = 0
  for i in range(100, n):
    sum = sum+1

  print(sum)
```

What is the Time Efficiency? (2)

```
def bigoh2(n):
  sum = 0
  for i in range(1, n // 2):
    sum = sum + 1
  for j in range(1, n * n):
    sum = sum + 1

  print(sum)
```

What is the Time Efficiency? (3)

```
def bigoh3(n):
  sum = 0
  if n % 2 == 0:
    for j in range(1, n * n):
      sum = sum + 1
  else:
    for k in range(5, n + 1):
      sum = sum + k

  print(sum)
```

What is the Time Efficiency? (4)

```
def bigoh4(m, n):
  sum = 0
  for i in range(1, n + 1):
    for j in range(1, m + 1):
      sum = sum + 1

  print(sum)
```