hw 2

page 67:
4. consider algorithm:

```
def mystery(n): # n = [non negative number input]
 S = 0
 for i in range(1, n + 1)
  S = S + i * i
 return S
```

n = 3: 0 + 1 + 4 + 9 = 14

a. what does this algorithm compute? this algorithm computes the sum of all squared integers from 0 to n.
b. What is its basic operation? S = S + i * i: This basic operation sets the current sum equal to the current sum + the square of the iterator.
c. How many times is the basic operation executed? n times.
d. What is the efficiency clas of this algorithm? $\Theta(n)$
e. Suggest an improvement to the algorithm, or prove it cannot be improved.
1 + 2 + ... + n = n(n+1) / 2. 1^2 + 2^2 + ... + n^2 = (n(n+1)/2)((2n+1)/3) = n(n+1)(2n+1)/6. Use algorithm defined below:

```
def mystery_2(n):
  return int(n*(n+1)*(2*n+1)/6)
```

efficiency class of mystery_2: $\Theta(1)$

page 76:

1. solve the following recurrence relations:
    1. x(n) = x(n-1) + 5 for n > 1, x(1) = 0 0. x(n-1) = x(n-2) + 5
        1. x(n) = (x(n-2) + 5) + 5
        2. x(n) = x(n-2) + 10
        3. x(n-2) = x(n-3) + 5
            1. x(n) = (x(n-3) + 5) + 10
            2. x(n) = x(n-3) + 15
        4. x(n) = x(n-k) + 5*k
        5. n-k=1
            1. k=n-1
        6. x(n) = x(n-(n-1)) + 5*(n-1)
            1. x(n) = x(1) + 5*(n-1)
            2. x(n) = 0 + 5*(n-1)
            3. x(n) = 5*n - 5
    2. x(n) = 3*x(n − 1) for n > 1, x(1) = 4 0. x(n-1) = 3*x(n - 2)

        1. x(n) = 3*3*x(n-2)
        2. x(n) = 9*x(n-2)
        3. x(n-2) = 3*x(n-3)
            1. x(n) = 9*(3*x(n-3))
            2. x(n) = 27*x(n-3)
        4. x(n) = 3^k*x(n-k)
        5. n-k=1
            1. k=n-1
        6. x(n) = 3^(n-1)*x(n-(n-1))
            1. x(n) = 3^(n-1)*x(1)
            2. x(n) = 4*3^(n-1)
    3. x(n) = x(n − 1) + n for n > 0, x(0) = 0 0. x(n-1) = x(n-2) + (n-1)
        1. x(n) = (x(n-2) + n - 1) + n
        2. x(n) = x(n-2) + 2*n - 1
        3. x(n-2) = x(n-3) + (n-2)
            1. x(n) = (x(n-3) + (n-2)) + 2*n - 1
            2. x(n) = x(n-3) + 2*n + n - 1 - 2
            3. x(n) = x(n-3) + 3*n - 3
        4. 1 3 6 10 15... = k*(k+1)/2
            1. x(n) = x(n-k) + k*n - (k*(k+1)/2)
        5. n-k=0
            1. n=k
        6. x(n) = x(n-n) + n*n - (n*(n+1)/2)
            1. x(n) = x(0) + n^2 - n(n+1)/2
            2. x(n) = n^2 - n(n+1)/2
    4. x(n) = x(n/2) + n for n > 1, x(1) = 1 (solve for n = 2k) 0. x(n/2) = x(n/4) + n/2
        1. x(n) = (x(n/4) + n/2) + n
        2. x(n) = x(n/4) + 3n/2
        3. x(n/4) = x(n/8) + n/4
            1. x(n) = (x(n/8) + n/4) + 3n/2
            2. x(n) = x(n/8) + 7n/4
        4. x(n/(2*k)) = x(n/(2^k)) +(2k+1)/(2*k)
    5. x(n) = x(n/3) + 1 for n > 1, x(1) = 1 (solve for n = 3k)
    6. give up for now

page 76-77

```
def S(n):
 if n is 1 return 1
 return S(n-1) + n * n * n
```

3. consider the following recursive algorithm for computing sum on *n* cubes: S(n) = 1^3 + 2^3 + ... + n^3
    1. set up and solve a recurrence relation for the number of times the algorithm's basic operation is executed 0. x(n) = x(n-1) + 1, x(1) = 1
        1. x(n-1) = x(n-2) + 1
            1. x(n) = (x(n-2) + 1) + 1

        2. x(n) = x(n-2) + 2
    2. x(n-2) = x(n-3) + 1
        1. x(n) = (x(n-3) + 2) + 1
        2. x(n) = x(n-3) + 3
    3. x(n) = x(n-k) + k
    4. n-k = 1
        1. k=n-1
    5. x(n) = x(n-(n-1)) + (n-1)
        1. x(n) = x(1) + (n-1)
        2. x(n) = 1 + (n-1)
        3. x(n) = n
2. The non-recursive, straightforward algorithm for computing the sum is also O(n). This is because there is a for loop, and you have a sum variable that you keep adding the cube of the iterator to. i.e.:

```
def iterative(n):
  if n is 1 return 1
  S = 0
  for i in range(1, n + 1):
    S += i ** 3
  return S
```