Computing runtime

- So far, we've been looking a lot at recurrences and how to analyze recursive algorithms.
- Quick refresh of "non-recursive" runtime analysis
- Counting!

```
def example(n):
    j = 0                    # 1 assignment
    j += 1                   # 1 operation
    for i in range(1,n):     # 1 assignment to initialize, one for each increment = n+1
      j = 1                  # n (cost one, inside for loop)
      while j <= n:          # one evaluation per iteration of inner loop
         j += 2              # 1 operation per iteration of inner loop
```

- How many times will inner loop evaluate for arbitrary `i`?
- Can estimate `n/2` by inspection
- Algebra solution:
  - after k iterations, value of `j` is `2k+1`
  - Termination condition: `j <= n`
  - Solve for value of k when loop terminates: `2k+1 <= n -> k = (n-1)/2`
- So inner loop will run $(n-1)/2$ times during *each* iteration of outer loop
- Total count -> $n * c * (n-1)/2$
- What if inner loop instead terminates when $j <= i$??
  - Can simply multiply max runtime by n (over-estimation)
  - To get tighter bound, manually compute sum over i

$$\sum_{i=1}^{n}(i-1)/2$$

$$\rightarrow \frac{1}{2}\sum_{i=1}^{n}(i-1)$$

$$\rightarrow \frac{1}{2}((\sum_{i=1}^{n} i) - n)$$

Tips and Tricks

- Arithmetic series finite sum
  - Example: $1 + 2 + \ldots + n$
  - Example: $2 + 5 + 8 + 11 + 14 = 40$
  - Formula: $\frac{n(a_1+a_n)}{2}$
- Geometric series sum
  - $\sum_{i=0}^{n} ar^i$
  - Case of $r = 1$

  - Otherwise, sum = $a\frac{1-r^{n+1}}{1-r}$
- Logarithm rules (+ exponent rules)
- Don't stress about floor / ceilings
- Big theta: mostly we care about showing that lower and upper bound have same functional form. To justify, need to justify that there is no situation where we will change the functional form of the runtime expression $T(n)$.

Example: Regular Insertion sort

- show tight bound