Max Seg Sum & Merge Sort:

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn + e & n > 1 \end{cases}$$

$$T(n) = dn \log_2 n + (c+e)n - e$$

$$\in \Theta(n \log n)$$

Standard Divide and Conquer Form

$$T(n) = aT(n/b) + \Theta(n^k)$$

- ***a*** is the number of recursive calls
- ***b*** is the rate at which subproblem size decreases
- ***k*** represents the runtime of the non-recursive part of the algorithm
  - like max_crossing in max_seg_sum, k=1
  - merge in MergeSort, k=1

Master Theorem

A >= 1, b > 1, k >= 0

(1) If $k = \log_b a$, then $T(n) = O(n^k \log n)$.

(2) If $k < \log_b a$, then $T(n) = O(n^{\log_b a})$.

(3) If $k > \log_b a$, then $T(n) = O(n^k)$.

WTF Master Theorem Example

$$T(n) = T(n/2) + T(n/3) + n$$

Cannot use the master theorem directly, but can still do some bounding

$$T(n/3) \leq T(n/2)$$

$$T(n) = T(n/2) + T(n/3) + n \leq 2T(n/2) + n \ = O(n \log n)$$

Correctness of Recursive Programs

Each program path is a path from the first line to the first return statement. Show that the precondition holds, and the recursive call runs on smaller data then the original call. Show the postcondition passes after the recursive call is made.

Some terminology

```
while E:
    S
```

```
while i < len(A):
    sum += A[i]
    i += 1
```

- **E** is called the **loop guard** (e.g., `i < len(A)`)
- **S** is called the **loop body** (one or more statements)
- A **loop invariant** gives a relationship between variables
  - it's a **predicate** with the variables being the parameters.
    - e.g., **Inv(i, sum): sum = \sum from A[1] to A[i]**
  - **Requirements** on loop invariant (otherwise it's not an invariant)
    - The invariant must hold prior to the first iteration (i.e., before entering the loop)
    - Assuming that the **invariant** and the **guard** are both true, the invariant must remain true after one **arbitrary** loop iteration

If we pick the right invariant, it will help proving the correctness of the program with the loop.

- **Base case**: Argue that the loop invariant is true when the loop is reached
- **Induction Step**:
  - assume that the invariant and guard are true at the end of an arbitrary iteration (**induction hypothesis P(n-1)**)
  - show that the invariant remains true after one iteration (**P(n)**)
- **Check postcondition**: Argue that the **invariant** and the **negation of the loop guard** together let us conclude the program's postcondition.

A loop variant must be reduced each iteration of the loop. The variant and the loop guard must show that the variant is always >= 0.

$Inv(x, y, total) : 0 \leq x \wedge total + x * y = a * b$

2. Induction Step

Case 1: x_0 is even

$x_1 = x_0/2$

$y_1 = y_0 * 2$

$total_1 = total_0$

$$\begin{aligned} & total_1 + x_1 * y_1 \\ = \ & total_0 + (x_0/2) * (y_0 * 2) \\ = \ & total_0 + x_0 * y_0 \ = a * b \end{aligned}$$

```
def mult(a,b):
    '''
    Pre: a and b are natural numbers
    Post: returns a * b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
            total = total + y
        x = x // 2
        y = y * 2
    return total
```


$Inv(x, y, total) : 0 \leq x \wedge total + x * y = a * b$

2. Induction Step

Case 2: x_0 is odd

$x_1 = (x_0 - 1)/2 \qquad y_1 = y_0 * 2$

$total_1 = total_0 + y_0$

$$\begin{aligned} & total_1 + x_1 * y_1 \\ = \ & (total_0 + y_0) + (x_0 - 1)/2 * y_0 * 2 \\ = \ & (total_0 + y_0) + (x_0 - 1) * y_0 \\ = \ & total_0 + y_0 + x_0 * y_0 - y_0 \\ = \ & total_0 + x_0 * y_0 \ = a * b \end{aligned}$$

```
def mult(a,b):
    '''
    Pre: a and b are natural numbers
    Post: returns a * b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
            total = total + y
        x = x // 2
        y = y * 2
    return total
```

Finding the variant:

- You usually want to identify which variable is decreasing each time.
- Make sure you account for all routes a loop iteration can take.
- Think of addition/subtraction/multiplicationof two terms, or len(list) kind of variants like those done in the lectures/tests/course_notes.
- Make sure by the end of the loop, your variant should be non-negative.
- The decrease amount can be 5,4,3,2,1,0 or like 10,8,6,2,0

Termination of Multiplication Algorithm

```
def mult(a,b):
    '''
    Pre: a and b are natural numbers
    Post: returns a * b
    '''
    x = a
    y = b
    total = 0
    while x > 0:
        if x % 2 == 1:
            total = total + y
        x = x // 2
        y = y * 2
    return total
```

The variant can be **x**:

- x decreases on each iteration
- by the loop guard implies that x > 0

So `mult` terminates.