- Optimization & Algorithm Analysis (Ch 1.2, 10.4)
  - Concerned with:
    - space utilization (eg memory)
    - time efficiency
  - Usually optimize for just one of the two
    - usually a tradeoff between space & time
    - fast alg uses more mem for example
  - Not as critical for smaller programs, but many tasks where still important
    - real-time devices
    - games
    - large-scale data crunching
  - Time efficiency
    - how long it takes to do a task
    - affected by several of the following:
      - amount of data
      - nature of data
      - algorithm
    - use approximate measurement instead of seconds
      - seconds can be affected by hardware, OS, system load, etc
    - need a standardized measure that is machine independent
      - approximate count of instructions in the algorithm
      - can't use court of machine instructions (compiled code), not machine independent
    - Counting algorithm instructions
      - must look at each line of code and see how many steps it takes
        - basic code (eg assignment, math) is seen as taking 1 step
        - loops more complex
          - have to figure how many times the loop executes
            - Ex: for(i=0; i<n; i++)
              - goes from 0 to n-1, that's n times
              - plus 1 more step to check the stopping condition
          - then have to consider how often each stmt in the loop body executes
            - Ex:for(i=0; i<n; i++) { sum += i; }
              - from above there are n loops
                - so loop body executes n times
              - how long for each loop body?
                - look at each loop stmt
                - have one basic code stmt: add i to sum
                - since basic, takes 1 step
                - so 1 step for each loop body
          - now multiply steps in a single loop body by number of loops
            - Ex: 1 step for each loop body, n loops total

n * 1 = n
so n steps for loop body plus 1 step to check stopping condition
this loop has n+1 steps
Ex: Two methods to find sum in Ch1.2
Algorithm 1
1. Have user input valve for n
2. Set sum = 0
3. For each i in the range 1 to n
a. add i to sum
4. Return sum
Counting steps
1. Basic task so 1 step
2. Basic task, 1 step
3. Loop, same as above example, n+1 steps
4. Basic task, 1 step
Total: n+4 steps
Algorithm 2
1. Have user input value for n
2. Return (n * (n + 1)) / 2
Counting steps
1. Basic task, 1 step
2. Basic task, 1 step
Total: 2 steps
recursive functions are counted similarly to loops
find out how many times it calls itself and how long recursive case body takes
add in number of steps for stopping case
time can also be affected by structure of data
this is one way data structures differ
for example, fast to print a sorted list if data is stored sorted
have to consider following for each alg:
best case
worst case
average case
often the worst case is what most look at
sets an upper bound on the performance
can't get any worse

Big O notation
just a formal way to express counted steps
takes the order of magnitude
ignores constants
3n is order of magnitude n
3 is order of magnitude 1 (constant)
takes the largest factor
eg in n+1, n is greater factor than 1, so considered order of magnitude n
Above examples
Alg1 is n +4 steps, so that's order of magnitude n
Alg2 is 2 steps, so that's order of magnitude 1
Alg1 is O(n)
Alg2 is O(1)

- Alg2 is O(1)
- Common notations
  - O(1) is constant
  - O(n) is linear
  - O(n^2) is quadratic
  - O(n^3) is cubic
  - O(2^n) is exponential
  - Figure 10.6 on page 558 shows how these graph out for various values of n
- Can be used to express space utilization too

Space utilization
- how much memory data is stored in
- again, must have a standardized method for comparison
  - can't just see how much mem the program used
  - also use a count instead of size (ie n units)
  - count says what it is a count of
    - Ex: you are storing student records
      - you'd say the storage is O(n)
        - where n is the number of student records
- computing space utilization for an entire program can be tedious
  - course will focus on memory utilization for each data structure

Know the Problem
- no one best alg or data structure
- have to analyze the problem
  - come up with the best solution for the problem at hand
- don't forget about the context of the problem
  - what sort of system is it geared for?
    - a PDA will have different ideas of "optimal" than a gaming PC
  - what are the target users tolerances
    - some people want speed
    - some people want low "footprint"
- remember that big-O is approximate
  - ignores constants & smaller factors that may be a big part of performance
    - particularly w/ small data sets
  - many compilers can optimize code
    - differences between similar algs may be minimized