Recitation 10 (12/2)

Reminders:

- MQ10 on Monday (12/5)
- PSet 5 due Wednesday (12/7) at 9pm

Lecture Recap: Writing Efficient Programs & Complexity

Writing Efficient Programs

- Until now, we've mostly talked about correctness, but we also need think about efficiency when writing programs.
- When we talk about improving efficiency, we often mean writing a program in a different way so that it is executed faster.

Some ways to evaluate the efficiency of programs

1. **With a timer and using the time module**

   Example:
   ```
   import time  # import time module
   tstart = time.time()  # "start" timer
   count = 0
   for i in range(1000):
      count += 1
   tend = time.time()  # "end" timer
   dt = tend - tstart
   print(dt)  # print time to run program
   ```

2. **Counting number of operations**

   The following steps take constant time:
   a. Mathematical operations
   b. Comparisons
   c. Assignments
   d. Accessing objects in memory

3. **Abstraction of order of growth (see next section).**

Complexity/Order of Growth

- "Big-O" notation.
- Gives us an idea of how long an algorithm will take to run with respect to the size of its inputs (arguments), regardless of what machine it's running on.
- Gives the worst case scenario.
- We don't care about lower-order terms or constants. We are interested in trends as input grows very large, so highest order terms dominate.
    - Adding:
      $O(n^2) + O(n) + O(1) \rightarrow O(n^2 + n + 1) \rightarrow O(n^2)$
    - Multiplicative or additive constants don't matter
      $O(10*n) \rightarrow O(n)$
      $O(n) * O(n) \rightarrow O(n^2)$
      $O(\log_2(n)) \rightarrow O(\log(n)/\log(2)) \rightarrow O(\log(n))$
      $O(n + 1) \rightarrow O(n)$
- We want the tightest bound possible. Technically, an algorithm that is $O(n)$ is also $O(n^2)$, $O(2^n)$, etc, since O is just <=, but we want the closest upper bound possible.
- Big Θ bound is a lower and upper bound on the growth of some function – tighter bound:
  $f(x) = \Theta(x^2) \neq \Theta(n^3) \neq \Theta(2^x)$
  $n^2 + 2n + 6 \rightarrow \Theta(n^2)$

Common orders of growth ordered by increasing complexity. Ideally, we want to design algorithms as close to the top of the table as possible.

Complexity	**Time**	**Examples**
O(1)	Constant	Adding two numbers together, appending to a list – Independent of input size!
O(log(n))	Logarithmic	Binary search
O(n)	Linear	list.copy(), or scanning through an entire list to look for a value
O(nlog(n))	Log-linear	Merge-sort
$O(n^2)$, $O(n^3)$ etc…	Polynomial	Nested for loops
$O(2^n)$, $O(3^n)$ etc…	exponential	Trying to guess an n-character password

Complexity of Python methods/objects https://wiki.python.org/moin/TimeComplexity

- Constant-time operations:
    - Assigning a variable (x = 1)
    - Performing basic operations (+, -, /, **, <, >, ==, etc.)
    - Some built-in methods for data structures in Python are also constant-time, but many are not. Although we don't look at the underlying machinery of many built-in methods, the complexity of their implementations affects the complexity analysis of our own methods.
- Dictionaries (will depend on hash function)
    - O(1): lookup, checking if key in dictionary, length, insert, delete
    - O(n): d.keys() or d.values() - list of length n must be generated
- Lists
    - O(1): append, length
    - O(n): insert, delete (move around elements), copy, check if item in (unsorted) list
    - O(nlog(n)): sort

Strategies for Order-of-Growth Analysis

- Loops: # of iterations times cost of each iteration
- Recursive functions
    - How many recursive calls are being made?
    - How much work does each recursive call take?
    - Draw a tree connecting subproblems

Helpful Big-O Complexity Chart