Lecture #18: Complexity and Orders of Growth

- Certain problems take longer than others to solve, or require more storage space to hold intermediate results.
- We refer to the *time complexity* or *space complexity* of a problem.
- But what does it mean to say that a certain *program* has a particular complexity?
- What does it mean for an *algorithm*?
- What does it mean for a *problem*?

A Direct Approach

- Well, if you want to know how fast something is, you can time it, which Python happens to make easy:

```
>>> def fib(n):
...     if n <= 1: return n
...     else: return fib(n-2) + fib(n-1)
...
>>> import timeit
>>> timeit.repeat('fib(10)', 'from __main__ import fib', number=5)
[0.0004911422729492188, 0.0004868507385253906, 0.0004870891571044922]
>>> timeit.repeat('fib(20)', 'from __main__ import fib', number=5)
[0.06009697914123535, 0.06010794639587402, 0.06009793281555176]
```

- `timeit.repeat(Stmt, Setup, number=N)` says

  Execute Setup (a string containing Python code), then execute Stmt (a string) N times. Repeat this process 3 times and report the time required for each repetition.

A Direct Approach, Continued

- timeit.repeat alone gives a bit too much information: smallest value is probably all that's meaningful; can't trust more that about two significant digits; and would be more useful to get an average time per iteration.
- Fortunately, we can always write programs to support writing programs!

```
>>> def desc_time(expr, setup="", number=1000):
...     time = 1e6 * min(timeit.repeat(expr, setup, number=number)) / number
...     return "{} loops, best of 3: {:.2g} usec per loop"\
...            .format(number, int(time))
>>> print(desc_time('fib(10)', 'from __main__ import fib'))
10000 loops, best of 3: 97 usec per loop"""
```

- You can also get this effect from the command line:

```
...# python3 -m timeit --setup='from fib import fib' 'fib(10)'
10000 loops, best of 3: 97 usec per loop
```

- This command automatically chooses a number of executions of fib to give a total time that is large enough for an accurate average, repeats 3 times, and reports the best time.

Strengths and Problems with Direct Approach

- Good: Gives actual times; answers question completely for given input and machine.
- Bad: Results apply only to tested inputs.
- Bad: Results apply only to particular programs and platforms.
- Bad: Cannot tell us anything about complexity of algorithm or of problem.

But Can't We Extrapolate?

- Why not try a succession of times, and use that to figure out timing in general?

```
...# for t in 5 10 15 20 25 30; do
>     echo -n "$t: "
>     python3 -m timeit --setup='from fib import fib' "fib($t)"
> done
5: 100000 loops, best of 3: 8.16 usec per loop
10: 10000 loops, best of 3: 96.8 usec per loop
15: 1000 loops, best of 3: 1.08 msec per loop
20: 100 loops, best of 3: 12 msec per loop
25: 10 loops, best of 3: 133 msec per loop
30: 10 loops, best of 3: 1.47 sec per loop
```

- This looks to be exponential in t with exponent of ≈ 1.6.
- But. . . what if the program special-cases some inputs?
- . . . and this still only works for a particular program and machine.

Worst Case, Average Case

- To avoid the problem of getting results only for particular inputs, we usually ask a more general question, such as:
  - What is the *worst case* time to compute f(X) as a function of the size of X, or
  - what is the *average case* time to compute f(X) over all values of X (weighted by likelihood).
- Average case is hard, so we'll let other courses deal with it.
- But now we seem to have a harder problem than before: how do we get worst-case times? Doesn't that require testing all cases?
- And when we do, aren't we still sensitive to machine model, compiler, etc.?

Operation Counts and Scaling

- Instead of getting precise answers in units of physical time, we therefore settle for a proxy measure that will remain meaningful over changes in architecture or compiler.
- Choose some operation(s) of interest and count how many times they occur.
- Examples:
  - How many times does fib get called recursively during computation of fib(N)?
  - How many addition operations get performed by fib(N)?
- You can no longer get precise times, but if the operations are well-chosen, results are *proportional* to actual time for different values of N.
- Thus, we look at how computation time *scales* in the worst case.
- Can compare programs/algorithms on the basis of which scale better.

Example: Search

- Here's a simple search function:

```
def find_first(L, p):
    """The index of the first item in list L that satisfies
    predicate function P, or -1 if none does."""
    for i, x in enumerate(L):  # Yields (0, L[0]), (1, L[1]),...
        if p(x): return i
    return -1
```

- It is reasonable to count calls to `p` as a measure.
- Sometimes, this will return immediately (if `p(L[0])`).
- Can't say much about the average case without knowing more.
- Worst case is that no item satisfies `p`,
- . . . in which case, # calls to `p` == `len(L)`.

Example: Intersection

- Now let's look at two lists:

```
def find_common(L0, L1):
    """Returns True iff L0 and L1 have an item in common."""
    for x in L0:
        for y in L1:
            if x == y: return True
    return False
```

- When will this program take longest? ______________________________
- If we count comparisons (==), how long will the worst case take? ______________
- Or, if N = len(L0) = len(L1), then ___.

Example: Intersection

- Now let's look at two lists:

```
def find_common(L0, L1):
    """Returns True iff L0 and L1 have an item in common."""
    for x in L0:
        for y in L1:
            if x == y: return True
    return False
```

- When will this program take longest? When there are no common items.
- If we count comparisons (`==`), how long will the worst case take?

  ____

- Or, if N = len(L0) = len(L1), then ___.

Example: Intersection

- Now let's look at two lists:

```
def find_common(L0, L1):
    """Returns True iff L0 and L1 have an item in common."""
    for x in L0:
        for y in L1:
            if x == y: return True
    return False
```

- When will this program take longest? When there are no common items.
- If we count comparisons (==), how long will the worst case take? len(L0) · len(L1)
- Or, if N = len(L0) = len(L1), then N^2.

Example: Duplicates

- This function looks for repeated items in a sequence:

```
def are_duplicates(L):
    for i, x in enumerate(L):
        for j, y in enumerate(L, i+1): # Starts at i+1
            if x == y: return True
    return False
```

- Again, this returns False in the worst case.
- Formula is more complicated, though. If N is len(L), then it executes the == operation

$$\sum_{1 \le k < N} N - k = (N-1) + (N-2) + \ldots + 0 = \underline{\hspace{3cm}} \text{ times.}$$

- This formula is already getting a bit complicated.
- But it *scales* at the same rate as for find_common when both arguments have the same length, i.e.:
  - Doubling the size of the input quadruples the time.

Example: Duplicates

- This function looks for repeated items in a sequence:

```
def are_duplicates(L):
    for i, x in enumerate(L):
        for j, y in enumerate(L, i+1): # Starts at i+1
            if x == y: return True
    return False
```

- Again, this returns False in the worst case.
- Formula is more complicated, though. If N is len(L), then it executes the == operation

$$\sum_{1\leq k<N} N-k = (N-1)+(N-2)+\ldots+0 = \underline{N(N-1)/2} \quad \text{times.}$$

- This formula is already getting a bit complicated.
- But it *scales* at the same rate as for find_common when both arguments have the same length, i.e.:
  - Doubling the size of the input quadruples the time.

Expressing Approximation

- We are looking for measures of program performance that give us a sense of how computation time scales with size of input.
- Sometimes, results for "small" values are not indicative.
  - E.g., suppose we have a prime-number tester that contains a look-up table of the primes up to 1,000,000,000 (about 50 million primes).
  - Tests for numbers up to 1 billion will be faster than for larger numbers.
- In general, we are interested in ignoring finite sets of special cases that a given program can compute quickly.
- So we tend to ask about *asymptotic* behavior of programs: as size of input goes to infinity.
- Finally, precise worst-case functions can be very complicated, and the precision is generally not terribly important anyway.
- These considerations motivate the use of *order notation* to characterize functions that approximate execution time or space.

The Notation

- We use the notation $O(f)$ to mean "the set of all one-parameter functions whose absolute values are eventually bounded above by some multiple of f's absolute value." Formally:

$$O(f) = \{g \mid \text{there exist } p, M \textit{ such that if } x > M, |g(x)| \leq p|f(x)|\}$$

- Similarly, we have "the set of all one-parameter functions whose absolute values are eventually bounded below by some multiple of f's absolute value:"

$$\Omega(f) = \{g \mid \text{there exist } q > 0, M \textit{ such that if } x > M, q|f(x)| \leq |g(x)|\}$$

- And finally those bounded both above and below:

$$\begin{aligned} \Theta(f) &= \Omega(f) \cap O(f) \\ &= \{g \mid \exists\, q > 0, p, \text{ and } M \text{ such that } q|f(x)| \leq |g(x)| \leq p|f(x)|, \text{ for } x > M\} \end{aligned}$$