Lecture #18: Complexity, Memoization
How Fast Is This (I)?

• For this program:

```python
for x in range(N):
    if L[x] < 0:
        c += 1
```

• What is the worst-case time, measured in number of comparisons?

• What is the worst-case time, measured in number of additions (+=)?

• How about here?

```python
for x in range(N):
    if L[x] < 0:
        c += 1
        break
```
How Fast Is This (I)?

• For this program:

```python
for x in range(N):
    if L[x] < 0:
        c += 1
```

# Answer: $\Theta(N)$ comparisons

• What is the worst-case time, measured in number of comparisons?

• What is the worst-case time, measured in number of additions ($+=1$)?

• How about here?

```python
for x in range(N):
    if L[x] < 0:
        c += 1
    break
```
How Fast Is This (I)?

• For this program:
  
  ```python
  for x in range(N):
      if L[x] < 0:
          c += 1
  # Answer: \( \Theta(N) \) additions
  ```

• What is the worst-case time, measured in number of comparisons?

• What is the worst-case time, measured in number of additions (+=)?

• How about here?
  
  ```python
  for x in range(N):
      if L[x] < 0:
          c += 1
      break
  ```
How Fast Is This (I)?

• For this program:

```python
for x in range(N):
    if L[x] < 0:
        c += 1
```

# Answer: $\Theta(N)$ comparisons

# Answer: $\Theta(N)$ additions

• What is the worst-case time, measured in number of comparisons?

• What is the worst-case time, measured in number of additions (+=)?

• How about here?

```python
for x in range(N):
    if L[x] < 0:
        c += 1
        break
```

# Answer: $\Theta(N)$ comparisons
How Fast Is This (I)?

• For this program:

```python
for x in range(N):
    if L[x] < 0:
        c += 1
```

# Answer: $\Theta(N)$ comparisons
# Answer: $\Theta(N)$ additions

• What is the worst-case time, measured in number of comparisons?
• What is the worst-case time, measured in number of additions (+=)?

• How about here?

```python
for x in range(N):
    if L[x] < 0:
        c += 1
        break
```

# Answer: $\Theta(N)$ comparisons
# Answer: $\Theta(1)$ additions
How Fast Is This (II)?

- Assume that execution of $f$ takes constant time.

- What is the complexity of this program, measured by number of calls to $f$? (Simplest answer)

```python
for x in range(2*N):
    f(x, x, x)
    for y in range(3*N):
        f(x, y, y)
        for z in range(4*N):
            f(x, y, z)
```
How Fast Is This (II)?

- Assume that execution of $f$ takes constant time.
- What is the complexity of this program, measured by number of calls to $f$? (Simplest answer)

```python
for x in range(2*N):
    f(x, x, x)
    for y in range(3*N):
        f(x, y, y)
        for z in range(4*N):
            f(x, y, z)
```

# Answer: $\Theta(N^3)$
How Fast Is This (II)?

• Assume that execution of $f$ takes constant time.

• What is the complexity of this program, measured by number of calls to $f$? (Simplest answer)

  ```python
  for x in range(2*N):
    f(x, x, x)
    for y in range(3*N):
      f(x, y, y)
      for z in range(4*N):
        f(x, y, z)
  # Answer: $\Theta(N^3)$
  ```

• Why not $\Theta(24N^3 + 6N^2 + 2N)$?
How Fast Is This (II)?

• Assume that execution of $f$ takes constant time.

• What is the complexity of this program, measured by number of calls to $f$? (Simplest answer)

```python
for x in range(2*N):
    f(x, x, x)
for y in range(3*N):
    f(x, y, y)
for z in range(4*N):
    f(x, y, z)
```

# Answer: $\Theta(N^3)$

• Why not $\Theta(24N^3 + 6N^2 + 2N)$? That’s correct, but equivalent to the simpler answer of $\Theta(N^3)$. 
How Fast Is This (III)?

- What is the complexity of this program, measured by number of calls to $f$?

```python
for x in range(N):
    for y in range(x):
        f(x, y)
```
How Fast Is This (III)?

- What is the complexity of this program, measured by number of calls to $f$?

```python
for x in range(N):
    # Answer $\Theta(N^2)$
    for y in range(x):
        f(x, y)
```

- This is an arithmetic series $0 + 1 + 2 + \cdots + N - 1 = \frac{N(N-1)}{2} \in \Theta(N^2)$. 
How Fast Is This (IV)?

- What about this one, measured by number of calls to $f$?
- How about measured by number of comparisons ($<$)?

```python
z = 0
for x in range(N):
    for y in range(N):
        while z < N:
            f(x, y, z)
            z += 1
```
How Fast Is This (IV)?

- What about this one, measured by number of calls to \( f \)?
- How about measured by number of comparisons (\(<\)?)

\[
\begin{align*}
z &= 0 \\
\text{for } x \text{ in range}(N): & \quad & \# \text{ Answer } \Theta(N) \text{ calls to } f. \\
\text{for } y \text{ in range}(N): & \\
\text{while } z < N: & \\
& f(x, y, z) \\
z &= 1
\end{align*}
\]
How Fast Is This (IV)?

• What about this one, measured by number of calls to $f$?

• How about measured by number of comparisons ($<$)?

$$z = 0$$

```python
for x in range(N):    # Answer $\Theta(N)$ calls to $f$.
    for y in range(N): # Answer $\Theta(N^2)$ comparisons.
        while z < N:
            f(x, y, z)
            z += 1
```

• In practice, which measure (calls to $f$ or comparisons) would matter?
How Fast Is This (IV)?

• What about this one, measured by number of calls to $f$?
• How about measured by number of comparisons ($<$)?

$$z = 0$$

```python
for x in range(N):
    for y in range(N):
        while z < N:
            f(x, y, z)
            z += 1
```

• In practice, which measure (calls to $f$ or comparisons) would matter?
• Depends on size of $N$, actual cost of $f$. For large enough $N$, comparisons will matter more.
Avoiding Redundant Computation

• Consider again the classic Fibonacci recursion:

```python
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

• This is tree recursion with a serious speed problem.

• Computation of, say \( \text{fib}(5) \) computes \( \text{fib}(2) \) several times, because both \( \text{fib}(4) \) and \( \text{fib}(3) \) compute it, and both \( \text{fib}(5) \) and \( \text{fib}(4) \) compute \( \text{fib}(3) \). Computing time grows exponentially.

• The usual iterative version does not have this problem because it saves the results of the recursive calls (in effect) and reuses them.

```python
def fib(n):
    if n <= 1: return n
    a, b = 0, 1
    for k in range(2, n+1): a, b = b, a+b
    return b
```
Change Counting

- Consider the problem of determining the number of ways to give change for some amount of money:

```python
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    """Return the number of ways to make change for AMOUNT, where the coin denominations are given by COINS."
    ""
    if amount == 0:
        return 1
    elif len(coins) == 0 or amount < 0:
        return 0
    else:
        # = Ways with largest coin + Ways without largest coin
        return count_change(amount-coins[0], coins) + \
            count_change(amount, coins[1:])
```

- Here, we often revisit the same subproblem:
  - E.g., Consider making change for 87 cents.
  - When we choose to use one half-dollar piece, we have the same subproblem as when we choose to use no half-dollars and two quarters.
Memoizing

• Extending the iterative Fibonacci idea, let’s keep around a table (“memo table”) of previously computed values.

• Consult the table before using the full computation.

• Example: `count_change`:

```python
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = {}
    def count_change(amount, coins):
        if (amount, coins) not in memo_table:
            memo_table[amount, coins] = full_count_change(amount, coins)
        return memo_table[amount, coins]
    def full_count_change(amount, coins):
        # original recursive solution goes here verbatim
        # when it calls count_change, calls memoized version.
        return count_change(amount, coins)
```

• Question: how could we test for infinite recursion?
Optimizing Memoization

- Used a dictionary to memoize `count_change`, which is highly general, but can be relatively slow.

- More often, we use arrays indexed by integers (lists in Python), but the idea is the same.

- For example, in the `count_change` program, we can index by amount and by the starting index of the original value of `coins` that we use.

```python
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    # memo_table[amt][k] contains the value computed for
    # count_change(amt, coins[k:])
    memo_table = [ [ -1 ] * (len(coins)+1) for i in range(amount+1) ]

def count_change(amount, coins):
    if amount < 0: return 0
    elif memo_table[amount][len(coins)] == -1:
        memo_table[amount][len(coins)] = full_count_change(amount, coins)
    return memo_table[amount][len(coins)]
...
Order of Calls

• Going one step further, we can analyze the order in which our program ends up filling in the table.

• So consider adding some tracing to our memoized `count_change` program:

```python
memo_table = {}
def count_change(amount, coins):
    # ... full_count_change(amount, coins) ...
    return memo_table[amount,coins]
@trace
def full_count_change(amount, coins):
    if amount == 0:
        return 1
    elif len(coins) == 0 or amount < 0:
        return 0
    else:
        return count_change(amount, coins[1:]) + count_change(amount-coins[0], coins)
return count_change(amount,coins)
```
Result of Tracing

- Consider `count_change(57)` (returns only):

  ```
  full_count_change(57, ()) -> 0  # Need shorter 'coins' arguments
  full_count_change(56, ()) -> 0  # first.
  ...
  full_count_change(1, ()) -> 0  # For same coins, need smaller
  full_count_change(0, (1,)) -> 1  # amounts first.
  full_count_change(1, (1,)) -> 1
  ...
  full_count_change(57, (1,)) -> 1
  full_count_change(2, (5, 1)) -> 1
  full_count_change(7, (5, 1)) -> 2
  ...
  full_count_change(57, (5, 1)) -> 12
  full_count_change(7, (10, 5, 1)) -> 2
  full_count_change(17, (10, 5, 1)) -> 6
  ...
  full_count_change(32, (10, 5, 1)) -> 16
  full_count_change(7, (25, 10, 5, 1)) -> 2
  full_count_change(32, (25, 10, 5, 1)) -> 18
  full_count_change(57, (25, 10, 5, 1)) -> 60
  full_count_change(7, (50, 25, 10, 5, 1)) -> 2
  full_count_change(57, (50, 25, 10, 5, 1)) -> 62
  ```
Dynamic Programming

• Now rewrite count_change to make the order of calls explicit, so that we needn’t check to see if a value is memoized.

• Technique is called *dynamic programming* (for some reason).

• We start with the base cases (0 coins) and work backwards.

```python
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = [ [-1] * (len(coins)+1) for i in range(amount+1) ]
    def count_change(amount, coins):
        if amount < 0: return 0
        else: return memo_table[amount][len(coins)]
    def full_count_change(amount, coins):
        # How often called?
        ... # (calls count_change for recursive results)

    for a in range(0, amount+1):
        memo_table[a][0] = full_count_change(a, ())
    for k in range(1, len(coins) + 1):
        for a in range(1, amount+1):
            memo_table[a][k] = full_count_change(a, coins[-k:])
    return count_change(amount, coins)
```