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
  ```

• 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
  ```

• 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
  ```

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

# Answer
\[ \Theta(N^2) \]

• 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 ($<$)?

```
z = 0
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.

Change Counting

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

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    # = Ways with largest coin + Ways without largest coin
    return count_change(amount-coins[0], coins) + count_change(amount, coins[1:]).
```

In practice, which measure (calls to $f$ or comparisons) would matter?

- How do we measure the number of comparisons?
- What about this one measured by number of calls to $f$?
- How fast is this (IV)?

Avoiding Redundant Computation

- Consider again the classic Fibonacci recursion:

```
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 $\text{fib}(3)$ several times, because it
  computes $\text{fib}(2)$ and $\text{fib}(1)$.

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

```
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:

```
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-dollar pieces.

- E.g, consider making change for 87 cents.
We start with the base cases (0 coins) and work backwards.

The idea is the same.

For example, in the coin change program, we can index by amount in our table:

\[
\text{table}[a][k] = \text{full}\]

We consult the table before using the full computation.

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

The memoizing technique is called dynamic programming (or some reason).

• Now rewrite our function to make the order of calls explicit, so that we can track the order of calls.

Dynamic Programming

Memoizing

Optimizing Memoization

Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[amount][coins] = \text{full}\]

Or we use some approach:\n
\[
\text{count change (returns only)}:\n\]

That we use.

Coins ends up filling in the table.

• Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[a][k] = \text{full}\]

We consult the table before using the full computation.

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

The memoizing technique is called dynamic programming (or some reason).

• Now rewrite our function to make the order of calls explicit, so that we can track the order of calls.

Dynamic Programming

Memoizing

Optimizing Memoization

Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[amount][coins] = \text{full}\]

Or we use some approach:

\[
\text{count change (returns only)}:\n\]

That we use.

Coins ends up filling in the table.

• Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[a][k] = \text{full}\]

We consult the table before using the full computation.

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

The memoizing technique is called dynamic programming (or some reason).

• Now rewrite our function to make the order of calls explicit, so that we can track the order of calls.

Dynamic Programming

Memoizing

Optimizing Memoization

Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[amount][coins] = \text{full}\]

Or we use some approach:\n
\[
\text{count change (returns only)}:\n\]

That we use.

Coins ends up filling in the table.

• Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[a][k] = \text{full}\]

We consult the table before using the full computation.

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

The memoizing technique is called dynamic programming (or some reason).

• Now rewrite our function to make the order of calls explicit, so that we can track the order of calls.

Dynamic Programming

Memoizing

Optimizing Memoization

Consider adding some tracing to our memoized function. The program, we can index by amount in our table:

\[
\text{table}[amount][coins] = \text{full}\]

Or we use some approach:\n
\[
\text{count change (returns only)}:\n\]

That we use.

Coins ends up filling in the table.