Lecture #21: Complexity, Memoization

How Fast Is This (I)?

- For this program (L is a list and N <= len(L)):

```
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?

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


How Fast Is This (I)?

- For this program (L is a list and N <= len(L)):

```
for x in range(N):          # Answer: Θ(N) comparisons
    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?

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


How Fast Is This (I)?

- For this program (L is a list and N <= len(L)):

```
for x in range(N):          # Answer: Θ(N) comparisons
    if L[x] < 0:            # Answer: Θ(N) additions
        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?

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

How Fast Is This (I)?

- For this program (L is a list and N <= len(L)):

```
for x in range(N):        # Answer: Θ(N) comparisons
    if L[x] < 0:          # Answer: Θ(N) additions
        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?

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


How Fast Is This (I)?

- For this program (L is a list and N <= len(L)):

```
for x in range(N):        # Answer: Θ(N) comparisons
    if L[x] < 0:          # Answer: Θ(N) additions
        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?

```
for x in range(N):        # Answer: Θ(N) comparisons
    if L[x] < 0:          # Answer: Θ(1) additions
        c += 1
        break
```


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)

```
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)

```
for x in range(2*N):              # Answer: Θ(N^3)
    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)

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

- 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)

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

- 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?

```
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?

```
for x in range(N):          # Answer Θ(N^2)
    for y in range(x):
        f(x, y)
```

- The complexity is given by an arithmetic series:

$$0 + 1 + 2 + \cdots + N - 1 = N(N-1)/2 \in \Theta(N^2).$$

- Again, constant factors ($1/2$) and linear terms ($N/2$) are ignorable.

How Fast Is This (IV)?

- What about this one, measured by number of calls to `f`? (Careful! This is tricky.)
- 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
```


How Fast Is This (IV)?

- What about this one, measured by number of calls to `f`? (Careful! This is tricky.)
- How about measured by number of comparisons (<)?

```
z = 0
for x in range(N):        # Answer Θ(N) calls to f.
    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`? (Careful! This is tricky.)
- How about measured by number of comparisons (<)?

```
z = 0
for x in range(N):        # Answer Θ(N) calls to f.
    for y in range(N):    # Answer Θ(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`? (Careful! This is tricky.)
- How about measured by number of comparisons (<)?

```
z = 0
for x in range(N):        # Answer Θ(N) calls to f.
    for y in range(N):    # Answer Θ(N^2) comparisons.
        while z < N:
           f(x, y, z)
           z += 1
```

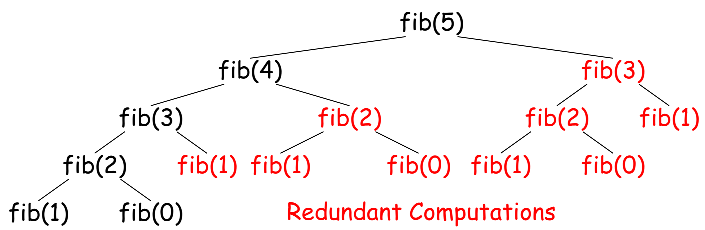
- In practice, which measure (calls to `f` or comparisons) would matter?
- Depends on size of N, actual cost of `f`. For large enough N, comparisons will matter more.

New Subject: 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 a tree recursion with a serious speed problem.



- Redundant computations and therefore computing time grow exponentially.

Avoiding Redundant Computation (II)

- The usual iterative version of `fib` does not have this problem because it saves the results of the recursive calls (in effect) and reuses them.
- Each computation of a number in the sequence happens exactly once, so the computation is linear in n (if we count additions as constant-time operations).

```
def fib(n):
   if n <= 1:
       return n
   a = 0
   b = 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 (change for 37 cents) 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`:

```
def count_change(amount, coins = (50, 25, 10, 5, 1)):
    memo_table = {}
    def count_change(amount, coins):
        key = (amount, coins)
        if key not in memo_table:
              memo_table[key] = full_count_change(amount, coins)
        return memo_table[key]
    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 *number of coins* remaining in `coins`.

```
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)]
    def full_count_change(amount, coins):
        # Full recursive version.
    return count_change(amount,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 (using an extension of the `@trace1` decorator from Lecture #9.)

```
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)` (-> N means "returns N"):

```
full_count_change(57, ()) -> 0   # Need shorter 'coins' arguments
full_count_change(56, ()) -> 0   # first.
...
full_count_change(1, ()) -> 0
full_count_change(0, (1,)) -> 1  # For same coins, need smaller
full_count_change(1, (1,)) -> 1  # amounts first.
...
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
```


Order of Calls (II)

- (New slide; not in lecture)
- We can see from the code that to compute the value of `full_count_change(N, C)`, it is sufficient to have
  - The values of `full_count_change(N, C[k:])` for $1 \le k \le \text{len}(C)$, and
  - The values of `full_count_change(k, C)` for $k < N$.
- And that tells us that, for example, we can compute all the values for `full_count_change(k, C)` for `C == ()`, then `C == (1,)`, then `C == (5, 1)`,
- And for each of these values of C, we can compute `full_count_change(k, C)` for all values of k in order,
- ...and at each point, we will already have computed all the recursive call values we need.

Filling in the Memo Table

Amount	Coins Left 0	1	2	3	4	5
0	1	1	1	1	1	1
1	0	1	1	1	1	1
2	0	1	1	1	1	1
3	0	1	1	1	1	1
4	0	1	1	1	1	1
5	0	1	2	2	2	2
◦◦◦						
23	0	1	5	9	9	9
24	0	1	5	9	9	9
25	0	1	6	12	13	13
26	0	1	6	12	13	13
◦◦◦						
54	0	1	11	36	49	50
55	0	1	12	42	60	62
56	0	1	12	42	60	62
57	0	1	12	42	60	62

Arrows show order of filling

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.

```
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)
```