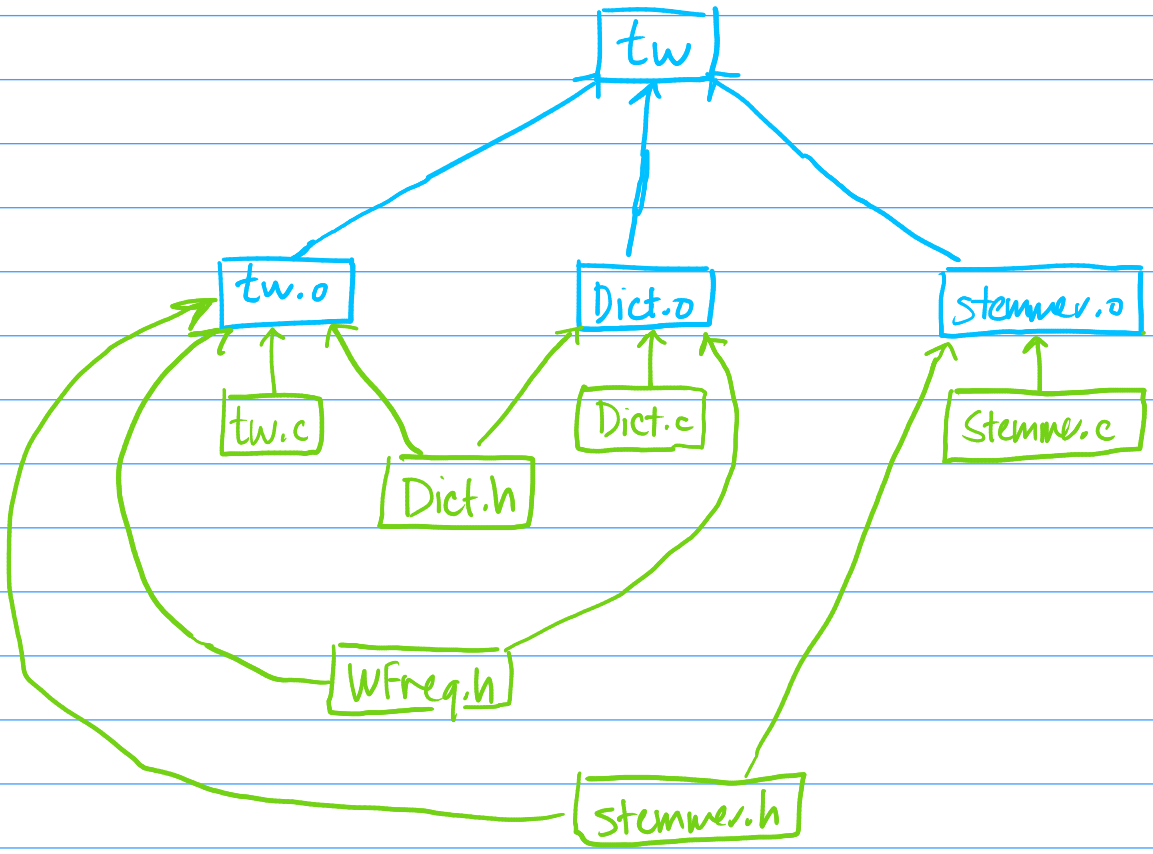
GCC and Makefiles Q2

Makefile:

```
tw: tw.o Dict.o stemmer.o
tw.o: tw.c Dict.h WFreq.h stemmer.h
Dict.o: Dict.c Dict.h WFreq.h
stemmer.o: stemmer.c stemmer.h
```

Dependency tree:



How do we know when to recompile things?

- check top-level rule (in this case, tw)
- If the target exists (eg. tw), then check if its dependencies have been changed / don't exist (recursively!)
- If no changes and everything exists, do nothing
- otherwise, recompile the current target

Complexity – Q4

```
PrintPermutation(A, P):
    Input array of items A and array of positions P of length n

    for i = 1 up to n do // for each position i
        for j = 0 up to n - 1 do // find the item that belongs at position i
            if P[j] = i then
                print A[j]
            end if
        end for
    end for
```

Time complexity: $O(n^2)$

Space complexity: $O(1)$, as we only have 2 extra variables

An improved algorithm:

How can we use more space to use less time?

Precompute into an array the permutation!
Then we can split the nested loop up into two single loops in series.

(Details left to you to try on your own.)

Lesson: we can sometimes save time by using more space if there is some form of repetitive computation or searching involved.

Complexity – Q5

```
BinaryConversion:
    Input  positive integer n
    Output binary representation of n on a stack

    create empty stack S          ⟶ O(1)
    while n > 0 do
        push (n mod 2) onto S     → O(1)
        n = floor(n / 2)          → O(1)
    end while
    return S                      ⟶ O(1)
```

Q: How many loop iterations?

[Note: floor(n) rounds n down to the nearest integer. eg. floor(3.2) = 3, floor(3.5)=3 and floor(3.9)=3.]

Time complexity: $O(\log n)$

To count the number of loop iterations, we need to figure out how many times we can halve n before $n=1$, because when we halve again, $\text{floor}(1/2) = 0 \neq 0$.

Suppose it takes k times. Then

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad n = 2^k$$
$$\Rightarrow \quad k = \log_2(n)$$

which is $O(\log n)$. So there are $O(\log n)$ iterations.

[General rule: halving each step usually (but not always) means you get logarithmic complexity!]