Background

- First year python course – fairly late in the term
  - Variables, how python works
  - standard data types, control flow, functions
  - pot_hole_case and camelCase both used
  - encourage code reuse
  - Have just covered basics of time complexity and went through some simple examples
    - e.g. x in List is O(len(List)) and x in Set is O(len(Set))
  - Full slides for this lecture have been posted shortly before lecture, but the example we will consider has been provided beforehand for students to think about, along with some questions. (See next slide)

Background Material

- In this lecture, we'll discuss how to write code with time complexity in mind.
- Given a problem to solve, what can we do to make sure our implementation is good enough?
- "Scrabble" game example:
  - **Given a hand of 7 letters and a list of English words, find a word with the highest score.**
  - Think about what the code would look like to do this
  - What is the time complexity of your algorithm?
- See Problem 6A: Computer Word Choose in Problem Set 3 at:
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00sc-introduction-to-computer-science-and-programming-spring-2011/unit-1/lecture-7-debugging/
- Or look at the 2008 version of the assignment:
  https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/assignments/pset6.pdf

Thinking about Time Complexity when you program

- Usually there are multiple ways to implement the same code specification.
- Code needs to be correct and sufficiently fast for the application
  - Video game frame rate ~ 60 fps
  - UI needs to be responsive
  - Keep in mind what hardware code will run on
- Poor implementation can lead to code that may be surprisingly slow
  - Be aware of the the time complexity of any functions you are calling (check **documentation**)

Domain Knowledge

- Understanding the problem you are solving can help you determine the best implementation
  - What ranges of values do the inputs take?
  - Are certain inputs more likely to occur? Or do some *never* occur?
  - What parts of the code are going to potentially make the program slow? And just *how* slow?
    - e.g. Short-circuiting: if B is more expensive to evaluate than A, which code is cheaper to run?

      B and A

      OR

      A and B

Example: “Scrabble” assignment from MIT's CSC6.00.1x course

- The full assignment implements a “Scrabble” game, where the player (human or computer) tries to form words given a set of letters, such that they get the highest score.
- e.g. Hand: r p o t r s a => ______________
- Program depends on a **list** of valid English words that are read in from a file (words.txt)
- words.txt contains **83667 words**
- See Problem 6A: Computer Word Choose in Problem Set 3 at:

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00sc-introduction-to-computer-science-and-programming-spring-2011/unit-1/lecture-7-debugging/MIT6_00SCS11_ps3.pdf

Scrabble Game Example

- We’ll focus on the code for the computer player:

  comp_choose_word(hand,word_list)

  - Given a hand of letters, pick the highest scoring word
  - hand = frequency **dictionary**
  - e.g. hand = {‘a’: 1, ‘u’ : 1, ‘l’ : 1, ‘c’: 1, ‘t’: 2, ‘f’: 1}
  - word_list = list of English words
  - Word score = len(word)*sum(letter values)

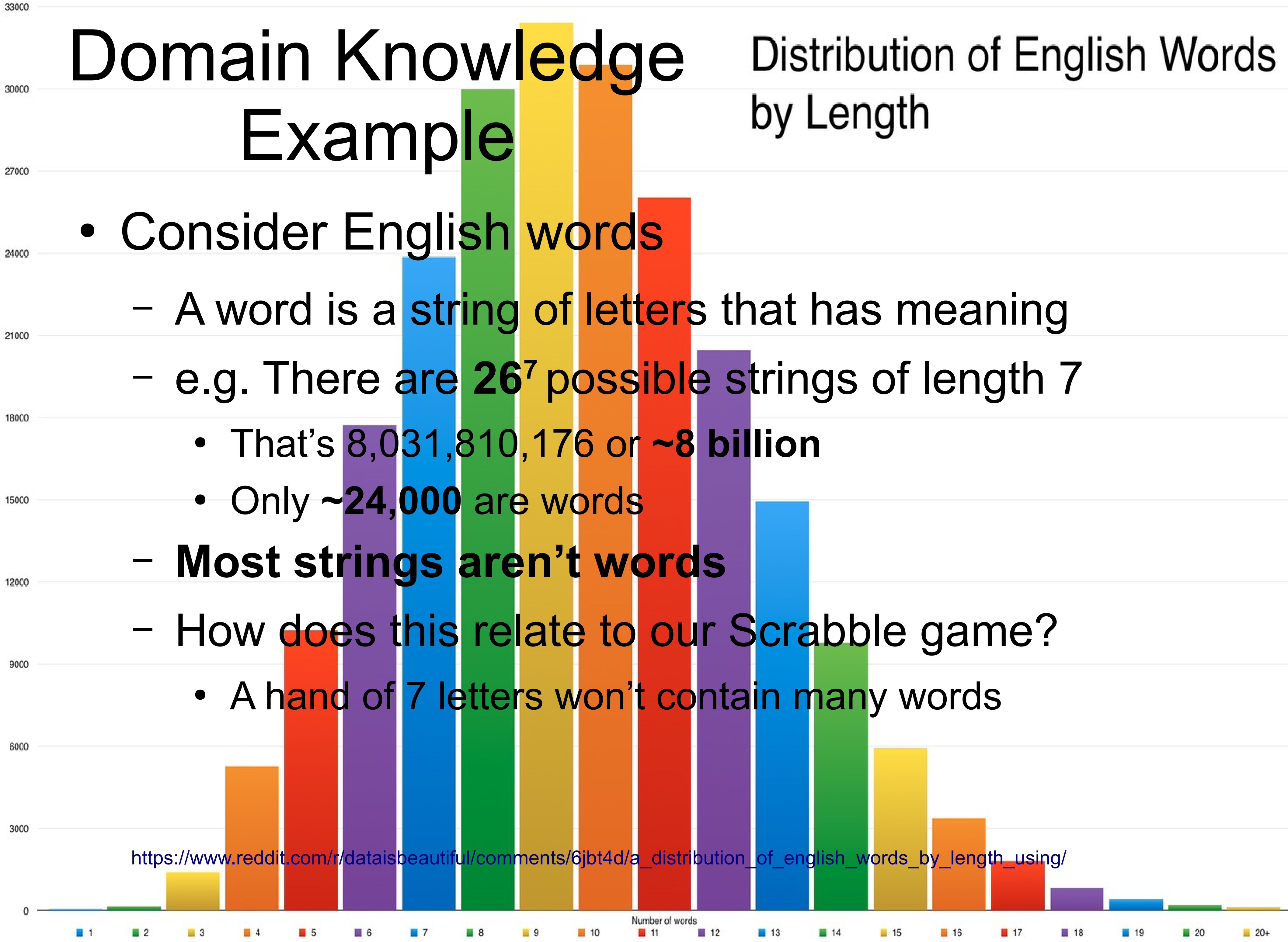
    + bonus 50 points if use all letters

  – Best word?

  – _________ (score?)

  – ______________________________

Domain Knowledge Example



Distribution of English Words by Length

- Consider English words
  - A word is a string of letters that has meaning
  - e.g. There are **26^7** possible strings of length 7
    - That’s 8,031,810,176 or **~8 billion**
    - Only **~24,000** are words
  - **Most strings aren’t words**
  - How does this relate to our Scrabble game?
    - A hand of 7 letters won’t contain many words

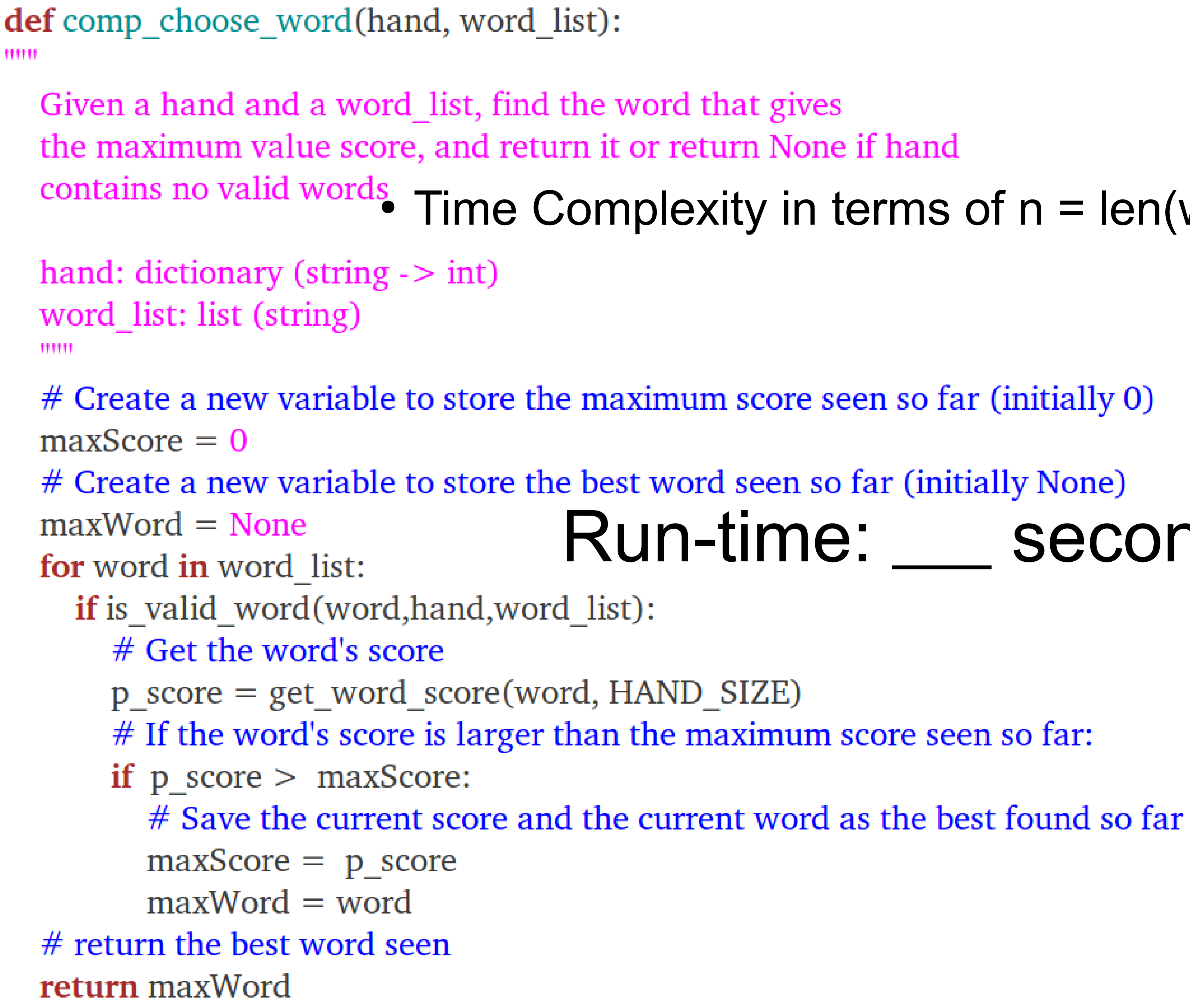
https://www.reddit.com/r/dataisbeautiful/comments/6jbt4d/a_distribution_of_english_words_by_length_using/

comp_choose_word(hand,word_list)

- We are first going to look at two implementations and try to identify how they can be improved:
  - a student's code
  - a solution posted in an offering of the course
- Time permitting, we'll consider two more solutions based on the original assignment:

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-00-introduction-to-computer-science-and-programming-fall-2008/assignments/pset6.pdf

A Student Solution

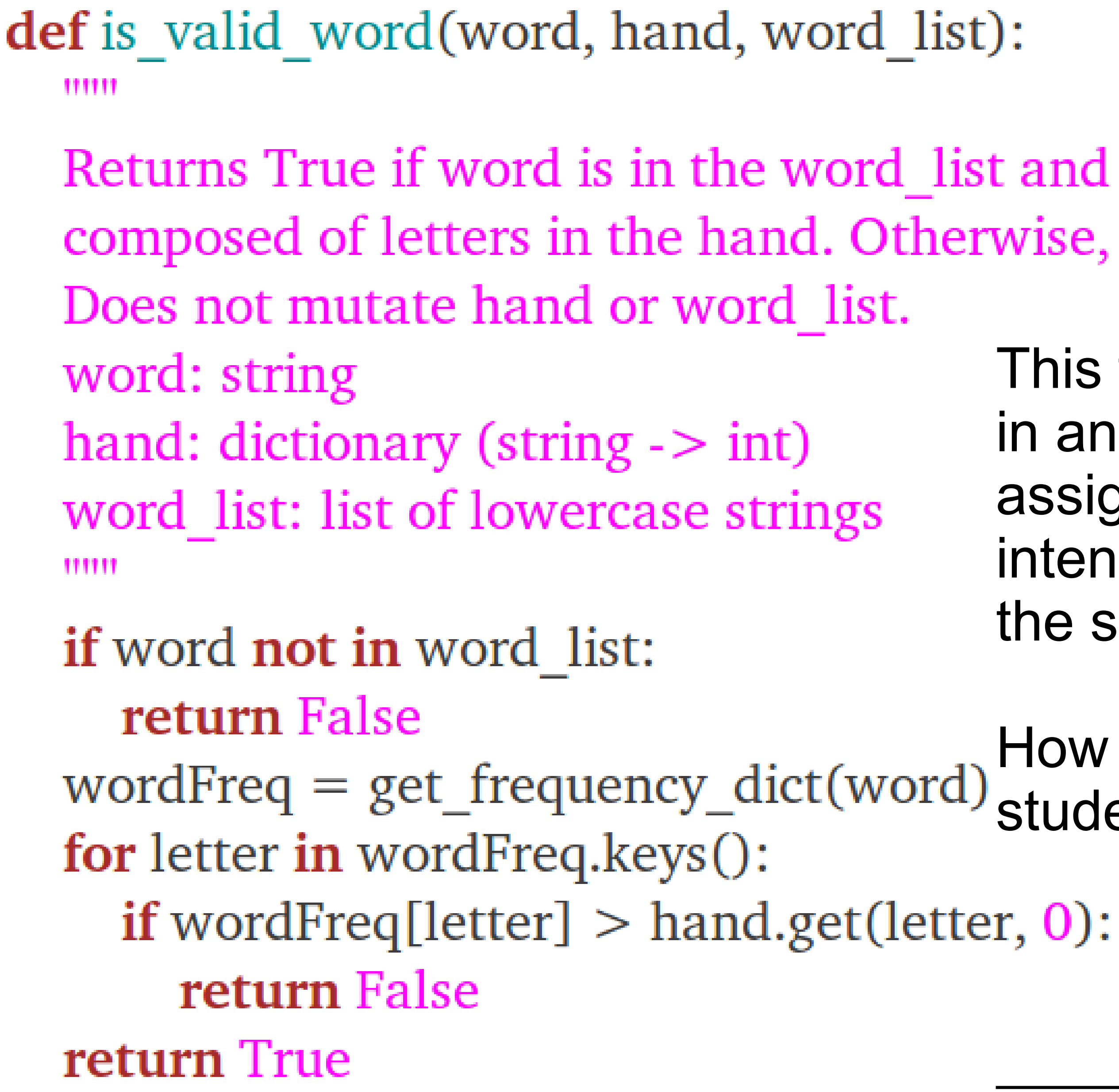


```
def comp_choose_word(hand, word_list):
"""
    Given a hand and a word_list, find the word that gives
    the maximum value score, and return it or return None if hand
    contains no valid words

    hand: dictionary (string -> int)
    word_list: list (string)
    """
    # Create a new variable to store the maximum score seen so far (initially 0)
    maxScore = 0
    # Create a new variable to store the best word seen so far (initially None)
    maxWord = None
    for word in word_list:
        if is_valid_word(word,hand,word_list):
            # Get the word's score
            p_score = get_word_score(word, HAND_SIZE)
            # If the word's score is larger than the maximum score seen so far:
            if  p_score >  maxScore:
                # Save the current score and the current word as the best found so far
                maxScore =  p_score
                maxWord = word
    # return the best word seen
    return maxWord
```

- Time Complexity in terms of n = len(word_list)?

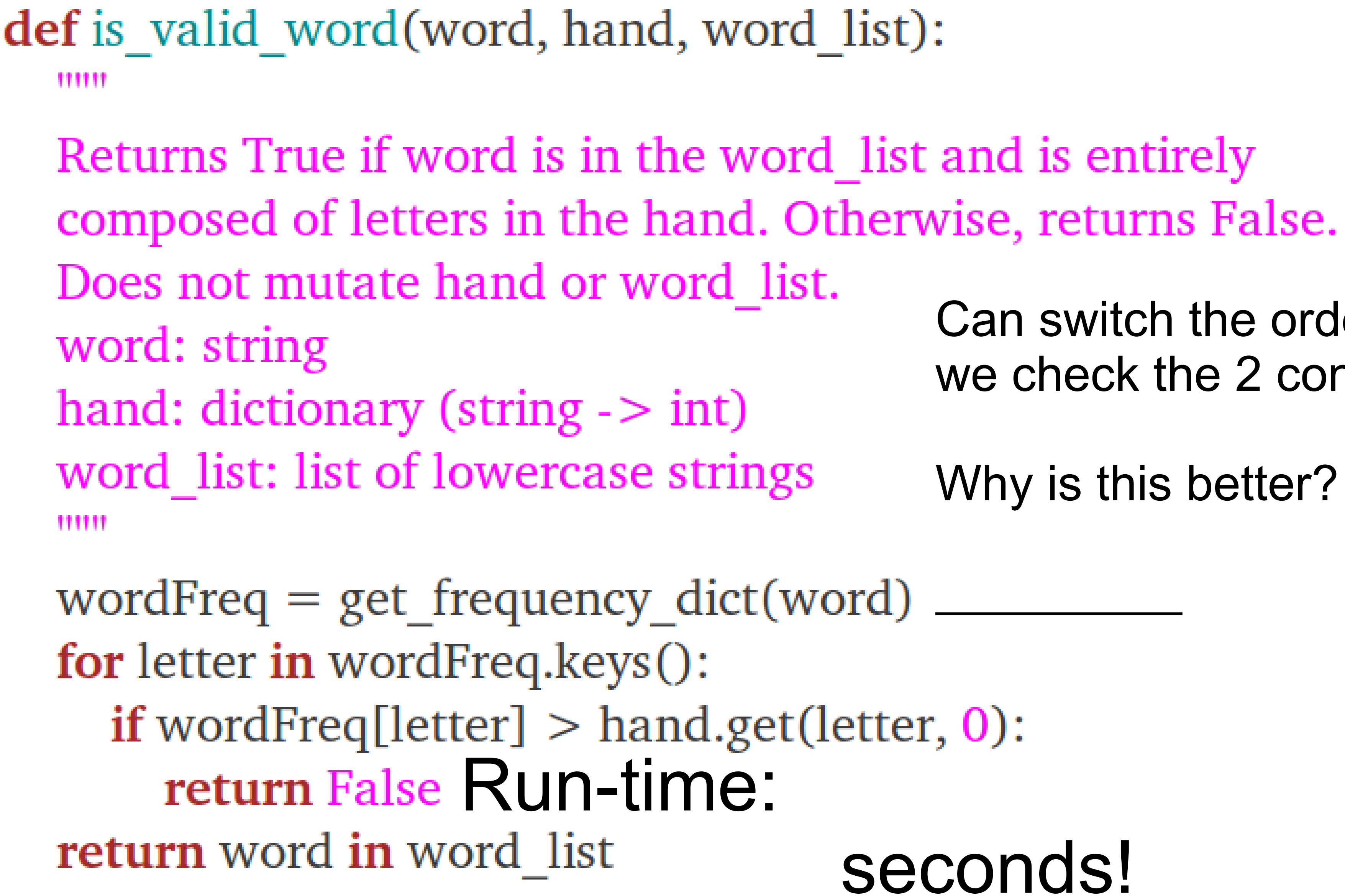
Run-time: ___ seconds!



```
def is_valid_word(word, hand, word_list):
    """
    Returns True if word is in the word_list and is entirely
    composed of letters in the hand. Otherwise, returns False.
    Does not mutate hand or word_list.
    word: string
    hand: dictionary (string -> int)
    word_list: list of lowercase strings
    """
    if word not in word_list:
        return False
    wordFreq = get_frequency_dict(word)
    for letter in wordFreq.keys():
        if wordFreq[letter] > hand.get(letter, 0):
            return False
    return True
```

This function was implemented in an earlier part of the assignment and was **not** intended to be used as it was by the student.

How can we change the student's code?



```
def is_valid_word(word, hand, word_list):
    """
    Returns True if word is in the word_list and is entirely
    composed of letters in the hand. Otherwise, returns False.
    Does not mutate hand or word_list.
    word: string
    hand: dictionary (string -> int)
    word_list: list of lowercase strings
    """
    wordFreq = get_frequency_dict(word)
    for letter in wordFreq.keys():
        if wordFreq[letter] > hand.get(letter, 0):
            return False
    return word in word_list
```

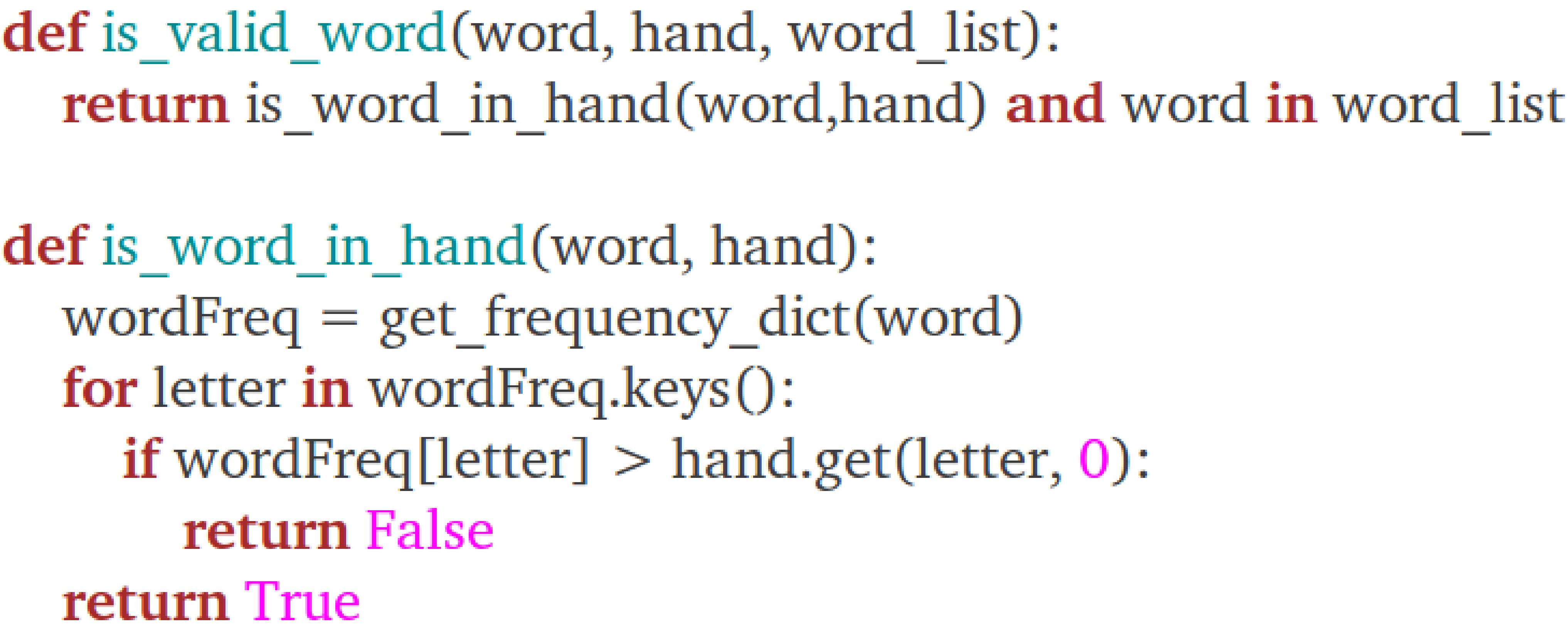
____

Can switch the order that we check the 2 conditions.

Why is this better?

Run-time:

____ seconds!



```
def is_valid_word(word, hand, word_list):
    return is_word_in_hand(word,hand) and word in word_list

def is_word_in_hand(word, hand):
    wordFreq = get_frequency_dict(word)
    for letter in wordFreq.keys():
        if wordFreq[letter] > hand.get(letter, 0):
            return False
    return True
```

- Define a new function is_word_in_hand
- student’s code still takes (0.12s)
- student’s code can call is_word_in_hand instead (0.11s)
- Why is the run-time so similar?
  - How often does “word in word_list” get evaluated?

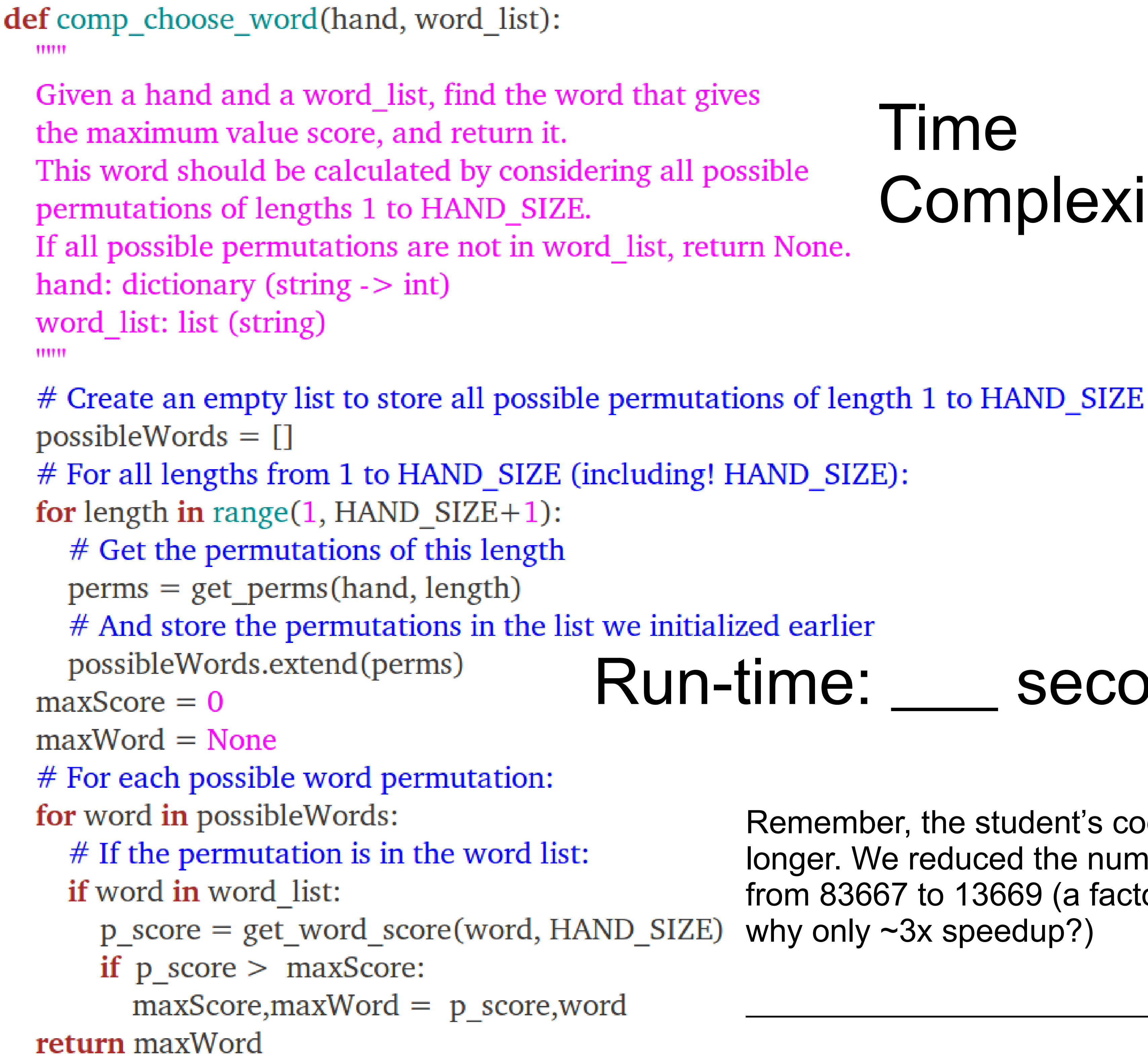
comp_choose_word (the posted solution)

- The assignment provided a utility function:

  get_perms(hand,length)

  - returns a list of all **permutations** of the given length using the letters in hand
- e.g. hand = { ‘c’ : 1, ‘a’ : 1, ‘t’ : 1 }
  - get_perms(hand,1) -> [__________]
  - get_perms(hand,2) -> [______________________]
- Multiple calls to get_perms gives us all ***potential*** words in hand
- With a 7 letter hand, this gives **13669** potential words to check
- Recall, word_list contains **83667** words

Time Complexity?



```python
def comp_choose_word(hand, word_list):
    """
    Given a hand and a word_list, find the word that gives
    the maximum value score, and return it.
    This word should be calculated by considering all possible
    permutations of lengths 1 to HAND_SIZE.
    If all possible permutations are not in word_list, return None.
    hand: dictionary (string -> int)
    word_list: list (string)
    """
    # Create an empty list to store all possible permutations of length 1 to HAND_SIZE
    possibleWords = []
    # For all lengths from 1 to HAND_SIZE (including! HAND_SIZE):
    for length in range(1, HAND_SIZE+1):
        # Get the permutations of this length
        perms = get_perms(hand, length)
        # And store the permutations in the list we initialized earlier
        possibleWords.extend(perms)
    maxScore = 0
    maxWord = None
    # For each possible word permutation:
    for word in possibleWords:
        # If the permutation is in the word list:
        if word in word_list:
            p_score = get_word_score(word, HAND_SIZE)
            if  p_score >  maxScore:
                maxScore,maxWord =  p_score,word
    return maxWord
```

Run-time: ____ seconds!

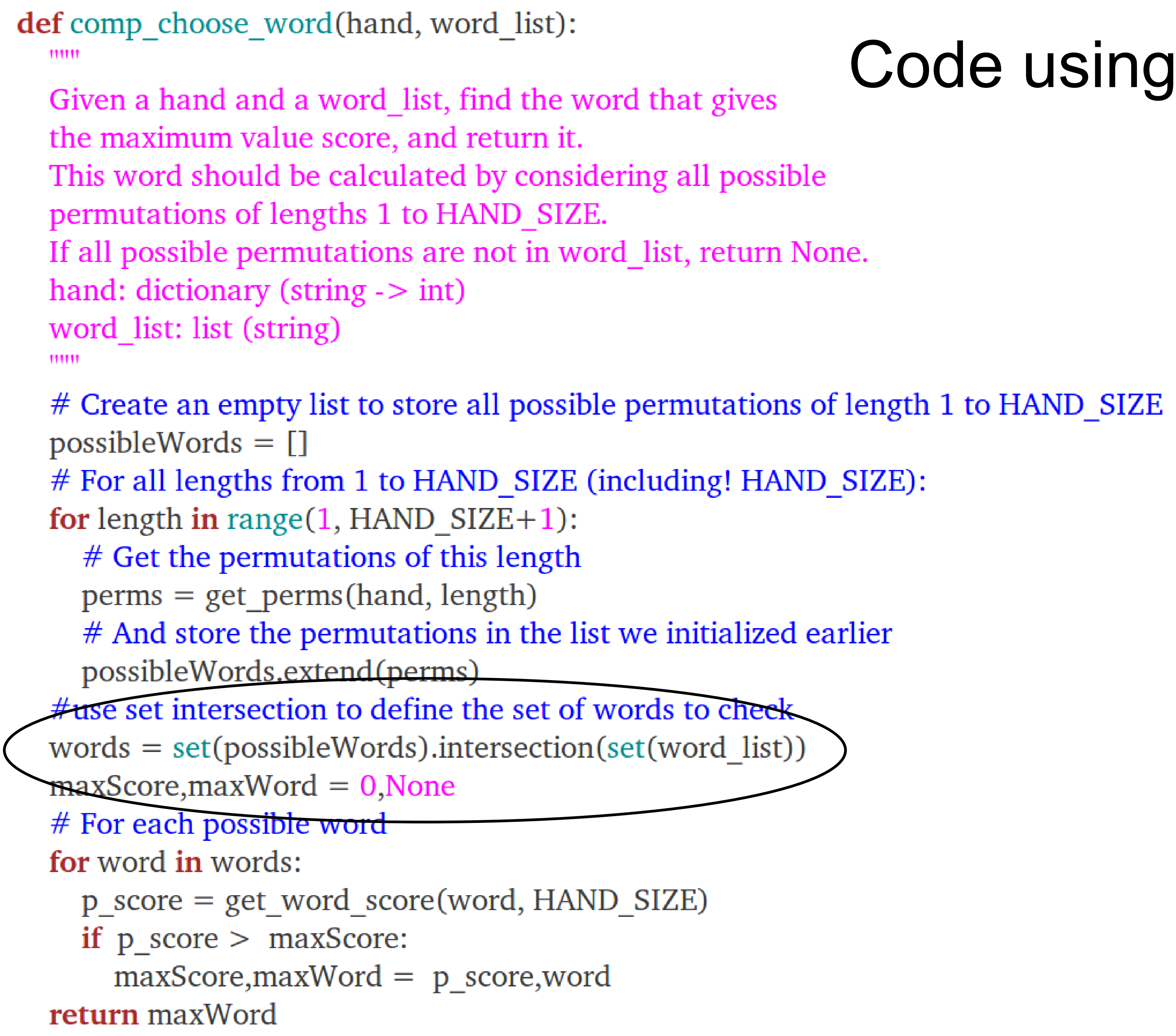
Remember, the student's code took ~3x longer. We reduced the number of loops from 83667 to 13669 (a factor of 6, so why only ~3x speedup?)

____

But wait, this is the “solution”?

- The modified student’s code is much faster than the posted solution code (and arguably simpler)
- How can we fix this?
  - Could use **sets**
  - We want the **intersection** of possibleWords and word_list
  - Set intersection time complexity?
  - Run-time: **0.059** seconds! (student code was 0.12s)

Code using sets



```
def comp_choose_word(hand, word_list):
    """
    Given a hand and a word_list, find the word that gives
    the maximum value score, and return it.
    This word should be calculated by considering all possible
    permutations of lengths 1 to HAND_SIZE.
    If all possible permutations are not in word_list, return None.
    hand: dictionary (string -> int)
    word_list: list (string)
    """
    # Create an empty list to store all possible permutations of length 1 to HAND_SIZE
    possibleWords = []
    # For all lengths from 1 to HAND_SIZE (including! HAND_SIZE):
    for length in range(1, HAND_SIZE+1):
        # Get the permutations of this length
        perms = get_perms(hand, length)
        # And store the permutations in the list we initialized earlier
        possibleWords.extend(perms)
    #use set intersection to define the set of words to check
    words = set(possibleWords).intersection(set(word_list))
    maxScore,maxWord = 0,None
    # For each possible word
    for word in words:
        p_score = get_word_score(word, HAND_SIZE)
        if p_score > maxScore:
            maxScore,maxWord = p_score,word
    return maxWord
```

Student Solution: can we do better?

- What if we need the code to be faster than this?
- Can we avoid calls to is_valid_word?
  - Use one (or more) cheaper checks combined with short circuiting
  - if ________________ and is_valid_word(...)
  - if ____________ and (_________ or ___________) and is_valid_word(...)

# of extra checks	0	1	2	3	4
Run-time	0.12s	0.031s	0.014s	0.0082s	0.0073s

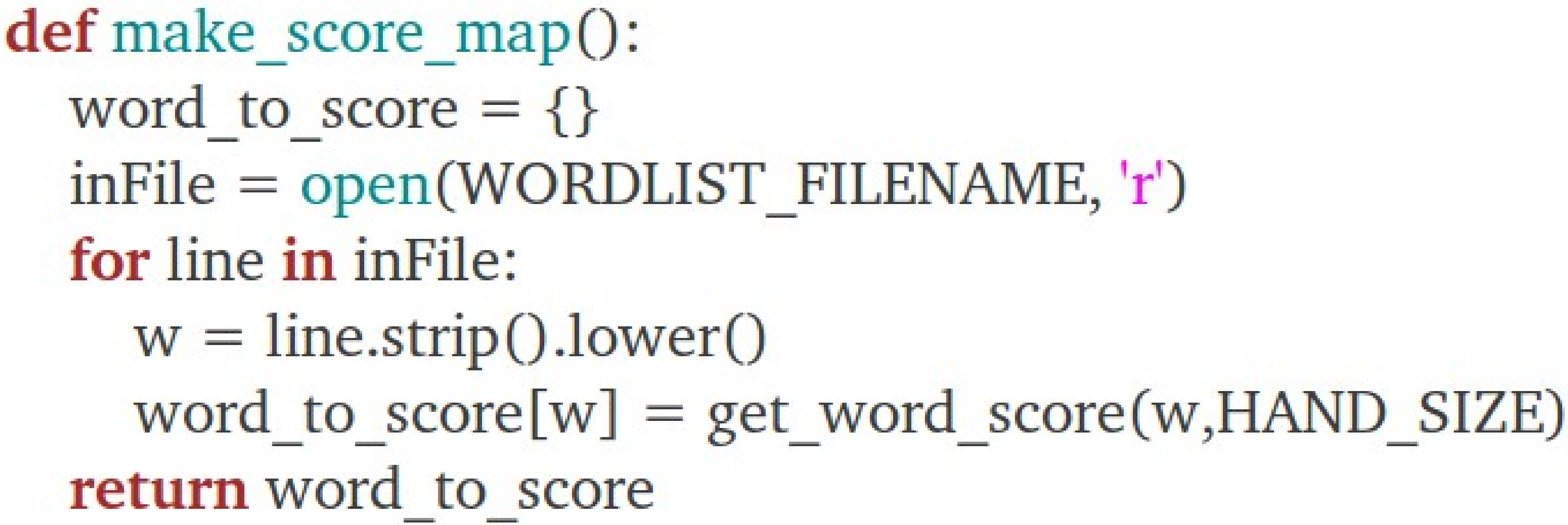
What if word_list doesn’t have to be a **list**?

comp_choose_word (based on 2008 course offering)

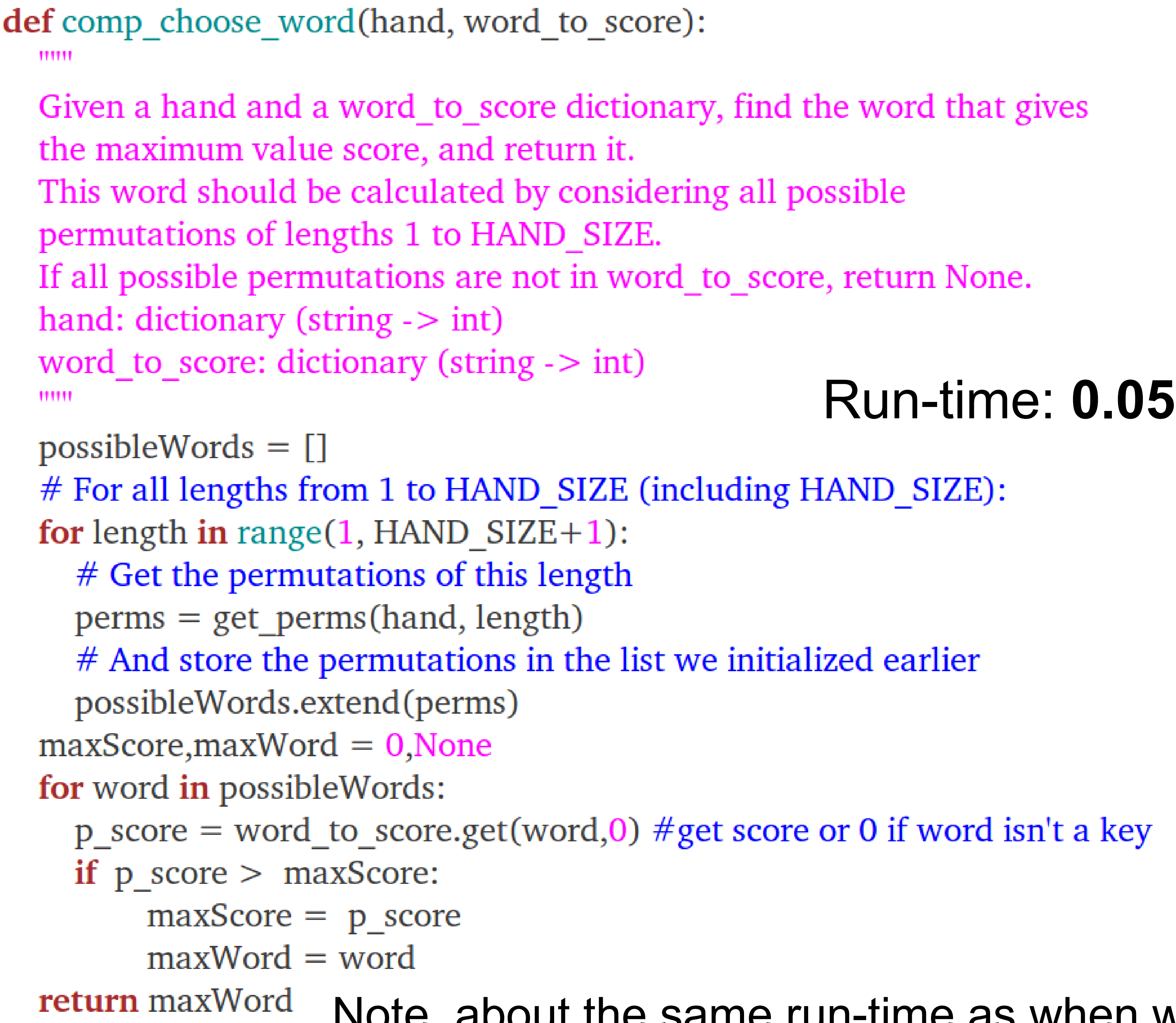
comp_choose_word(hand,**word_map**)

- In the original version of the assignment, you are asked to implement 2 improvements – both based on making **dictionaries that map strings to scores**.
- Why might using a dictionary be better here?
  - __________________

Word to Score dictionary



```
def make_score_map():
    word_to_score = {}
    inFile = open(WORDLIST_FILENAME, 'r')
    for line in inFile:
        w = line.strip().lower()
        word_to_score[w] = get_word_score(w,HAND_SIZE)
    return word_to_score
```



```
def comp_choose_word(hand, word_to_score):
    """
    Given a hand and a word_to_score dictionary, find the word that gives
    the maximum value score, and return it.
    This word should be calculated by considering all possible
    permutations of lengths 1 to HAND_SIZE.
    If all possible permutations are not in word_to_score, return None.
    hand: dictionary (string -> int)
    word_to_score: dictionary (string -> int)
    """
    possibleWords = []
    # For all lengths from 1 to HAND_SIZE (including HAND_SIZE):
    for length in range(1, HAND_SIZE+1):
        # Get the permutations of this length
        perms = get_perms(hand, length)
        # And store the permutations in the list we initialized earlier
        possibleWords.extend(perms)
    maxScore,maxWord = 0,None
    for word in possibleWords:
        p_score = word_to_score.get(word,0) #get score or 0 if word isn't a key
        if  p_score >  maxScore:
              maxScore =  p_score
              maxWord = word
    return maxWord
```

Run-time: **0.055** seconds!

Note, about the same run-time as when we used sets.

Why doesn’t storing the word scores in the dictionary help much?

- ____________________

Permutations and Combinations

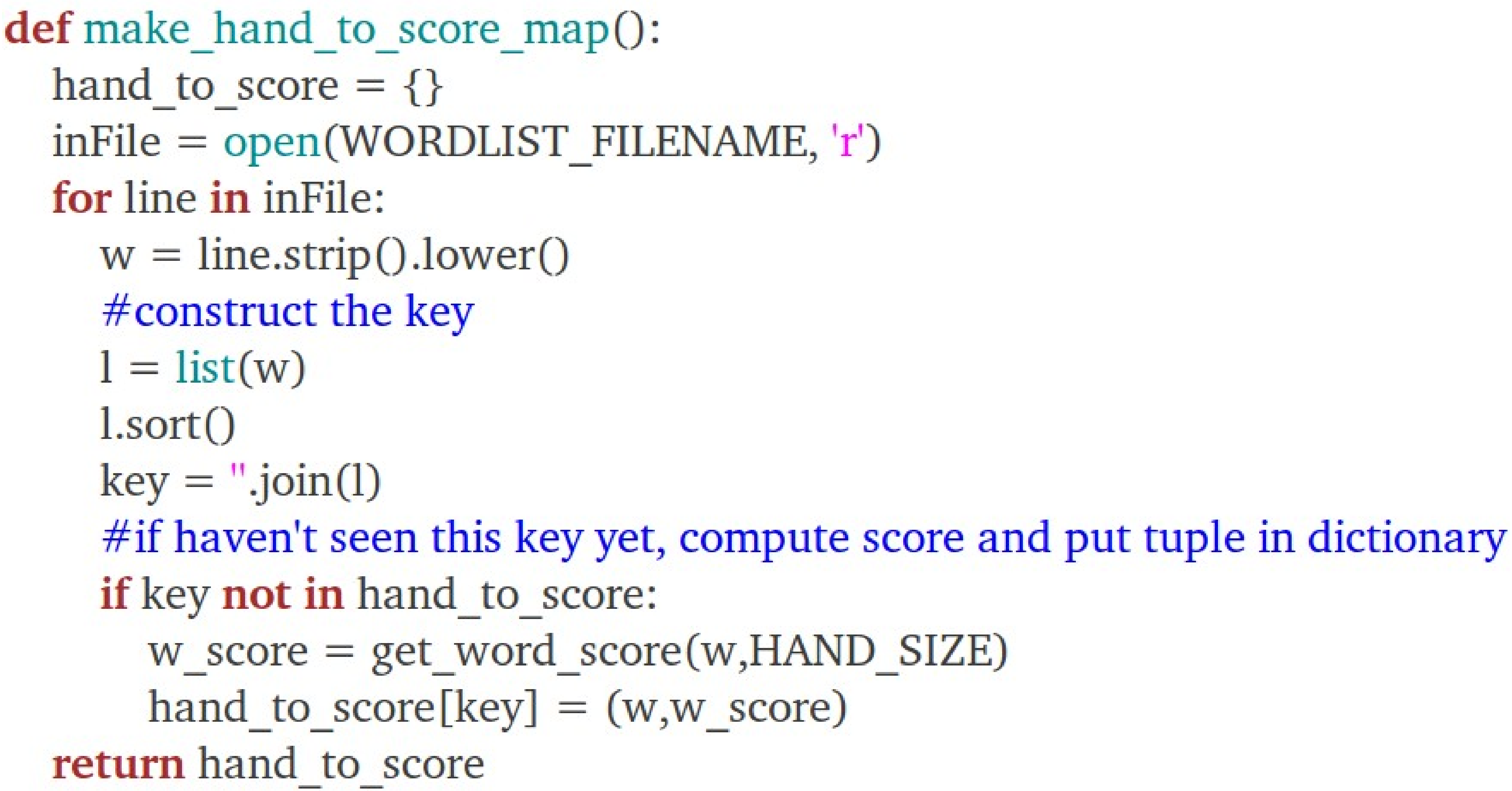
- Can view a hand as a **combination** rather than a **permutation** if we **sort** the hand
- e.g. ‘car’ and ‘arc’ are **permutations** of ‘acr’
  - A **hand** can be turned into a **key** by sorting the letters in alphabetical order
- The dictionary can map combinations of letters rather than permutations (words).
- This reduces the dictionary to **69091 keys** (from the original 83667 words)

Permutations and Combinations

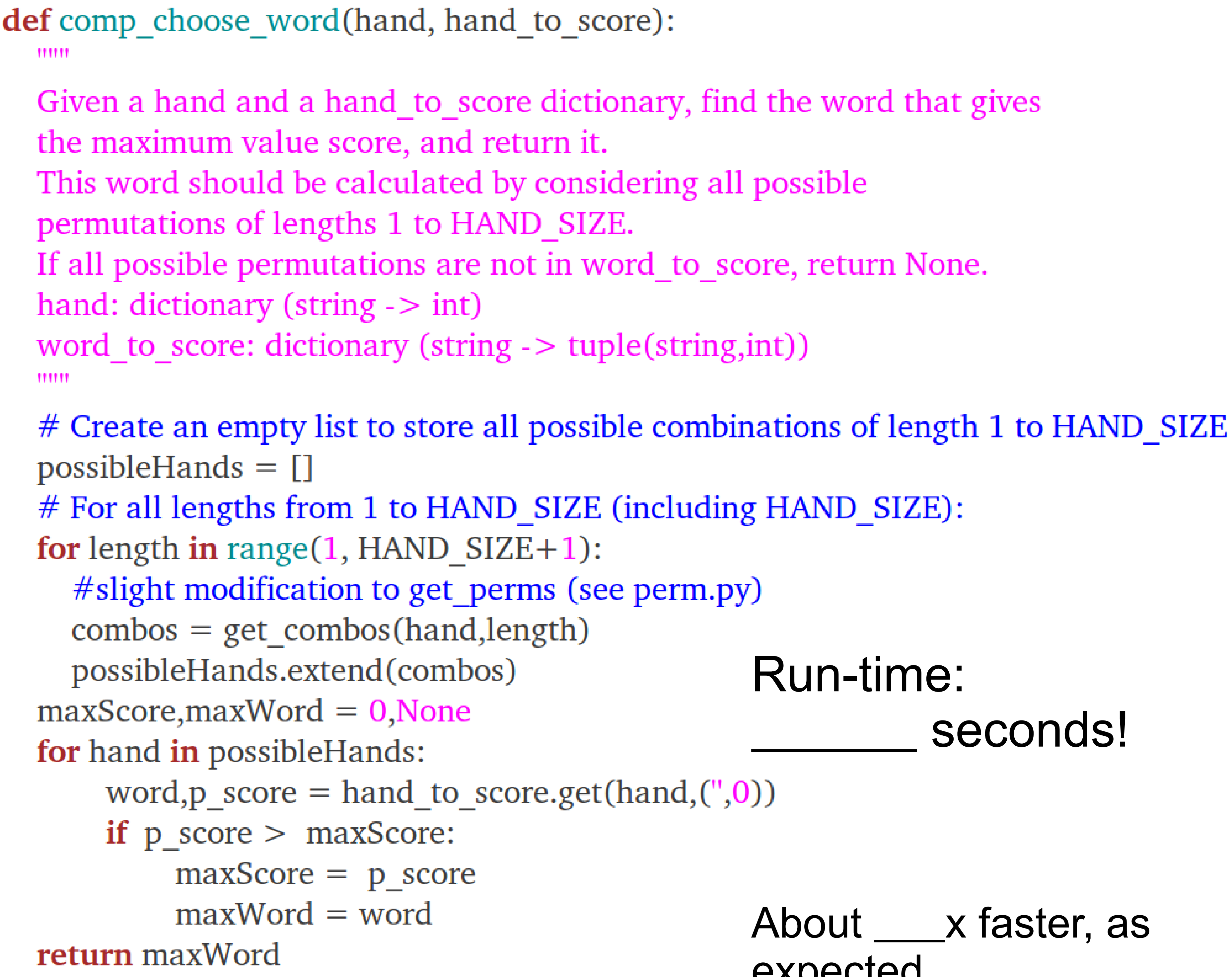
- ${}_nC_k = n! / k!(n-k)!$
- ${}_nP_k = k!\ {}_nC_k = n! / (n-k)!$
- Considering combinations instead of permutations drops a factor of k!
- Recall, a hand of 7 letters contains **13699 permutations**
  - Only **127** combinations!
  - 13699 / 127 => expect about 100x faster code!

Constructing the combination based dictionary

- The keys are no longer words, so need to store (one of) the words too (dict{str : tuple(str,int)})



```
def make_hand_to_score_map():
    hand_to_score = {}
    inFile = open(WORDLIST_FILENAME, 'r')
    for line in inFile:
        w = line.strip().lower()
        #construct the key
        l = list(w)
        l.sort()
        key = ''.join(l)
        #if haven't seen this key yet, compute score and put tuple in dictionary
        if key not in hand_to_score:
            w_score = get_word_score(w,HAND_SIZE)
            hand_to_score[key] = (w,w_score)
    return hand_to_score
```



```
def comp_choose_word(hand, hand_to_score):
    """
    Given a hand and a hand_to_score dictionary, find the word that gives
    the maximum value score, and return it.
    This word should be calculated by considering all possible
    permutations of lengths 1 to HAND_SIZE.
    If all possible permutations are not in word_to_score, return None.
    hand: dictionary (string -> int)
    word_to_score: dictionary (string -> tuple(string,int))
    """
    # Create an empty list to store all possible combinations of length 1 to HAND_SIZE
    possibleHands = []
    # For all lengths from 1 to HAND_SIZE (including HAND_SIZE):
    for length in range(1, HAND_SIZE+1):
        #slight modification to get_perms (see perm.py)
        combos = get_combos(hand,length)
        possibleHands.extend(combos)
    maxScore,maxWord = 0,None
    for hand in possibleHands:
        word,p_score = hand_to_score.get(hand,('',0))
        if p_score >  maxScore:
            maxScore =  p_score
            maxWord = word
    return maxWord
```

Run-time:
________ seconds!

About ____x faster, as expected

Summary of Scrabble Example

- Student code **46s**
  - With short circuiting **0.0073s**
- Posted solution **14s**
  - Using sets **0.059s**
- word_to_score dictionary **0.055s**
- hand_to_score dictionary **0.00041s**
  - Switching from permutations to combinations helped

We didn’t talk about it, but constructing the word_to_score and hand_to_score dictionaries isn’t without cost. Depending on the context, this **fixed startup cost** may outweigh the benefit of **comp_choose_word** being **faster per call**.

What about if we increase the hand size? How does each approach scale with hand size?

Summary

- Understand the problem you are solving
  - Try to use domain knowledge to make the problem simpler
- Consider time complexity of all operations
  - Given the expected inputs, will the run-time be fast enough?
- If necessary, optimize the code to achieve the required level of performance
  - Identify bottlenecks in the code and try to find a more efficient algorithm
  - Avoid unnecessary computations
    - (e.g. use short circuited and)

Additional Resources

- There are *many* websites with coding problems:
  - https://www.hackerrank.com/
  - https://codingcompetitions.withgoogle.com/codejam
  - https://projecteuler.net/
    - Mostly math / combinatorics / number theory problems

Wrap-Up

- A more *practical* lecture, which is to follow a more formal discussion of time complexity and code run-time.
- A **case study** of a simple task that can be solved in several similar ways – with *drastically* different performance.
- Designed so it *could* be a straight lecture, but with opportunities for students to volunteer ideas / think about what is going on
  - Attempt to give some hints as to what ideas we would be seeing later in the case study

Thanks for Listening!

Questions or comments?