Lab Report

Group Project

Steven Contreras

Marwin Gonzales

Ruby Nguyen

CSULB

CECS 174

Professor Susan Tabbaa Nachawati

Source Code

```
import math
import random
import time
import string
import re
import matplotlib.pyplot as plt

# ******************** constants: BEGIN ********************
N_DAYS_IN_YEAR = 365
N_DEFAULT_CLASS_SIZE = 23
N_SIM_SERIES = 6    # note that for formatting purposes, the number of series should be an even number > 0

S_ADD_SINGLETON_LIST_IDIOM = "t += [x] idiom"
S_LIST_APPEND_METHOD = "List.append() method"

LETTER_FREQ_TEMPLATE = "{}:\t{}\t(/{})\t\t{:0.2f}%"   # expect vals in this order: c, n_c, n_c_all, (n_c/n_c_all)*100

TEXT_FILE_SUMMARY_TEMPLATE = """
***** SUMMARY OF TEXT FILE: {} *****
    WORD COUNT: {}
    LETTER FREQUENCY:
        ALL:
{}
{}
{}
"""

QUIT_MESSAGE = "THAT'S ALL FOLKS!  Thanks for playing.  Bye bye."
# ******************** constants: END ********************




# ******************** functions: BEGIN ********************
def is_sorted(l):
    n = len(l)

    if n == 1:
        return True

    # since we compare indices i and i+1, we range from 0 to len(l)-2
    for i in range(n-1):  # recall that n-1 is not inclusive when using range()
        if l[i] > l[i+1]:
            return False
```

```
    return True

def test_is_sorted(l):
    b_result = is_sorted(l)
    print(f"\tTEST is_sorted(l={l}): {b_result}")


def str_to_list(s):
    """
    This function converts a string to a list of chars (in case we want to modify the list somehow)
    """
    return [s[i] for i in range(len(s))] if type(s) is str else s.copy()


def to_lcase(l):
    """
    This function only convert char elements in the list to lower-case.
    Obviously, non-char elements will not be affected.
    """
    l_lcase = str_to_list(l)

    for i in range(len(l_lcase)):
        if type(l_lcase[i]) is str:
            l_lcase[i] = l_lcase[i].lower()

    return l_lcase


def is_anagram(l1, l2, normalize_char_case=True):
    """
    Normally anagrams are based on words only.
    But this function supports numeric lists as anagrams, as well..
    """

    n1 = len(l1)
    n2 = len(l2)

    # we can short-circuit when the lengths are unequal
    if n1 != n2:
        return False

    # we normally disregard case when considering char anagrams
    if normalize_char_case:
        l1 = to_lcase(l1)   # but remember, the to_lcase() leaves non-char elements alone
        l2 = to_lcase(l2)
    else: # in case strings and we don't want to normalize to lcase
        l1 = str_to_list(l1)
        l2 = str_to_list(l2)
```

```
    # here we sort each list
    #   this greatly simplifies the problem compared to not sorting
    if not is_sorted(l1):
        l1 = sorted(l1)
    if not is_sorted(l2):
        l2 = sorted(l2)

    # because the two lists are sorted, we can now short-circuit (exit the loop) when we encounter the first mismatch
    for i in range(n1):
        if l1[i] != l2[i]:
            return False

    # if we made it this far then the two lists are necessarily the same length and have the same elements (unless case
matters, FOR STRINGS, and case differs for some element)
    return True

def test_is_anagram(l1, l2, normalize_char_case=True):
    b_result = is_anagram(l1, l2, normalize_char_case)
    print(f"\tTEST is_anagram(l1={l1}, l2={l2}, normalize_char_case={normalize_char_case}): {b_result}")



def has_duplicates(l, disregard_char_case=False):
    n = len(l)

    # a 0 or single element list is already implicitly sorted, so we can short-circuit
    if n < 2:
        return False

    if not disregard_char_case:
        l = to_lcase(l)
    else: # in case string and we don't want to normalize to lcase
        l = str_to_list(l)

    # here we sort the list
    #   this greatly simplifies the problem compared to not sorting
    if not is_sorted(l):
        l = sorted(l)

    # because the list is sorted, we can now short-circuit (exit the loop) when we encounter the first matching
adjacent pair of elements
    #   range from 0 to len(l)-2
    for i in range(n-1):    # recall that n-1 is not inclusive when using range()
        if l[i] == l[i+1]:
            return True

    return False
```

```
def test_has_duplicates(l, disregard_char_case=False):
    b_result = has_duplicates(l, disregard_char_case=disregard_char_case)
    print(f"\tTEST has_duplicates(l={l}, disregard_char_case={disregard_char_case}): {b_result}")


def run_bd_paradox_sim(n_sims, n_class_size=N_DEFAULT_CLASS_SIZE, is_leap_year=False):
    print(f"Running {n_sims} Birthday Paradox simulations on a class size of {n_class_size} students...")
    p = 0
    n_dups = 0
    x = []
    y = []
    for i_sim in range(n_sims):
        l_birthdays = [random.randint(1, N_DAYS_IN_YEAR + (1 if is_leap_year else 0)) for i in range(n_class_size)]
        n_dups += 1 if has_duplicates(l_birthdays) else 0
        x.append(i_sim)
        y.append(n_dups / (i_sim+1))
    p = n_dups / n_sims
    print(f"\tDONE: The probability that at least 2 students from a class size of {n_class_size} have the same birthday
converged to {p} after {n_sims} simulations.")

    return p, x, y

def run_bd_paradox_sim_series(n_powers_of_ten, n_class_size=N_DEFAULT_CLASS_SIZE, is_leap_year=False, do_plot=True):
    exponents = list(range(1,n_powers_of_ten+1))

    n_cols = 2
    n_rows = len(exponents) // n_cols
    if do_plot:
        fig, axes = plt.subplots(n_rows, n_cols, figsize=(8,4))

    for i, e in enumerate(exponents):
        n_sims = 10**e
        p, x, y = run_bd_paradox_sim(n_sims=n_sims, n_class_size=N_DEFAULT_CLASS_SIZE, is_leap_year=False)
        if do_plot:
            axis = axes[i//n_cols][i%n_cols]
            axis.set_title(f"# sims = {n_sims}, p = {p}")
            axis.plot(x, y)

    if do_plot:
        fig.tight_layout()
        plt.show()


def remove_duplicates(l):
    l = str_to_list(l)  # in case l is a string

    n = len(l)
```

```
    # a 0 or single element list is already implicitly sorted, so we can short-circuit
    if n < 2:
        return l

    # here we sort the list
    #   this greatly simplifies the problem compared to not sorting
    if not is_sorted(l):
        l = sorted(l)

    l_dups_removed = []

    # iterate from index 0 to len(l)-2
    #   only add the last non-repeating element, which we can do since the list has been sorted
    for i in range(n-1): # recall that n-1 is not inclusive when using range()
        if l[i] != l[i+1]:
            l_dups_removed.append(l[i])

    # but we still have the very last index to add
    #   this works since if there was a dup at the n-2 index, it will not have been added to l_dups_removed
    l_dups_removed.append(l[n-1])

    return l_dups_removed

def test_remove_duplicates(l):
    b_result = remove_duplicates(l)
    print(f"\tTEST remove_duplicates(l={l}): {b_result}")



def words_file_to_list(fname, use_list_append=True):
    l_words = []

    try:
        with open(fname, 'r') as f_words:
            for words_line in f_words:
                for word in words_line.split():
                    if use_list_append:
                        l_words.append(word.strip()) # dynamically resizes
                    else:
                        l_words += [word] # adding to separate lists (which exist in two different places in memory)
            f_words.close()
    except Exception as e:
        print(f"words_file_to_list: ***RUNTIME ERROR caught***: {e}")

    return l_words

def benchmark_words_file_to_list(fname, use_list_append, debug=False):
    if debug:
```

```
        s_append_list_mechanic = ("t += [x] idiom" if not use_list_append else S_LIST_APPEND_METHOD)
        print(f"Benchmarking '{fname}' file to words list (using {s_append_list_mechanic})")

    # timestamp for start of the execution of words_file_to_list()
    t0 = time.time()

    # execute words_file_to_list()
    l_words = words_file_to_list(fname)

    # timestamp for end of the execution of words_file_to_list()
    t1 = time.time()

    # the delta is just the elapsed time
    t_delta = t1 - t0

    if debug:
        print(f"\ttime elapsed (using {s_append_list_mechanic}): {t_delta} seconds") # CPU seconds elapsed (floating
point)

    return t_delta, l_words

def run_benchmark_words_file_to_list_series(fname, n_sims, debug=False):
    n_list_append_more_efficient = 0
    n_add_singleton_list_idiom_more_efficient = 0

    print(f"Running {n_sims} words_file_to_list() iterations...")

    for i_sim in range(n_sims):
        # benchmark using List.append()
        tdelta__list_append, l_words = benchmark_words_file_to_list(fname, use_list_append=True, debug=debug)

        if debug:
            print()

        # benchmark using the t += [x] idiom
        tdelta__add_list_idiom, l_words = benchmark_words_file_to_list(fname, use_list_append=False, debug=debug)

        # the rest of this code is just for formatting the summary output when debugging
        s_mech = ""
        eff_factor = 0

        # update summary and count of times the particular mechanic is more efficient
        if tdelta__add_list_idiom < tdelta__list_append:
            s_mech = S_ADD_SINGLETON_LIST_IDIOM
            eff_factor = tdelta__list_append / tdelta__add_list_idiom
            n_add_singleton_list_idiom_more_efficient += 1
        else:
            s_mech = S_LIST_APPEND_METHOD
```

```
            eff_factor = tdelta__add_list_idiom / tdelta__list_append
            n_list_append_more_efficient += 1

        s_efficiency = f"{s_mech} is {eff_factor} more efficient!"

        if debug:
            # print(f"\n{s_efficiency}\n\n{l_words}") # uncomment this to see l_words
            print(f"\n{s_efficiency}")

    # always display summary after all simulations are complete
    eff_ratio__list_append = n_list_append_more_efficient/n_sims
    eff_ratio__add_singleton_list = 1 - eff_ratio__list_append
    print(f"\tDONE: Out of {n_sims} iterations, {S_LIST_APPEND_METHOD} was more efficient
{round(eff_ratio__list_append,2)*100}% of the time, while {S_ADD_SINGLETON_LIST_IDIOM} was more efficient
{round(eff_ratio__add_singleton_list,2)*100}% of the time.")


def bisect(l, i_lb, i_ub, target_value, debug=False):
    """
    This function implements what is also known as 'binary search'.

    Adapted from https://www.geeksforgeeks.org/python-program-for-binary-search/

    Even though the spec we are given idicates we can assume l is already sorted, we will double-check and do the
sorting just in case.

    parameters:
        l:      the list (elementas should all be of the same type and comparable)
        i_lb:   the index lower-bound of l to search
        i_ub:   the index upper-bound of l to search

    From our spec:
        'to check whether a <value> is in the list'
        'returns the index of the value in the list, if it’s there, or None if it’s not'
    """

    # short-circuit for 0-length and singleton lists, this is also the "base case" when recursion is used (but we will
go the iteration route instead of recursion)
    n = len(l)
    if n == 0:
        return None
    if n == 1:
        return 0 if l[0] == target_value else None

    # is_sorted check: avoid performance hit (checking if sorted only at top level) - i.e. only when i_lb==0 AND
i_ub==len(l)-1
    if i_lb==0 and i_ub==len(l)-1:
        if not is_sorted(l):
```

```
            if debug:
                print("\tl is not sorted! sorting...")
            l = sorted(l)
            if debug:
                # print(f"\t\tsorted l: {l}")
                print(f"\t\tDONE")
        print(f"\tbisecting l for target value -->{target_value}<-- ...")

    # if we are here, we are guaranteed that l is sorted... now we can implement proper binary search logic
    #   first step is to validate that i_ub >= i_lb
    if i_ub >= i_lb:

        # since we are here, we can proceed with "bisecting"
        #   so the first thing we need to do is find the midpoint between i_ub and i_lb: this is the basis of "bisection"
        i_midpoint = (i_ub + i_lb) // 2     # integer division

        val_at_midpoint = l[i_midpoint]

        if debug:
            print(f"\t\tmidpoint (index) of l between index {i_lb} and {i_ub} is: {i_midpoint} and l[{i_midpoint}]=={val_at_midpoint}")

        # if target_value is at index i_midpoint, return i_midpoint
        if val_at_midpoint == target_value:
            if debug:
                print(f"\t\ttarget value -->{target_value}<-- found at midpoint index {i_midpoint}")
            return i_midpoint

        else:   # we have already excluded the equality case

            # now we use the fact that elements in l are comparable... this happens recursively
            if target_value < val_at_midpoint:  # then we look in the left half... this is the binary split
                return bisect(l, i_lb, i_midpoint, target_value, debug)

            else:   # otherwise we look in the right half... this happens recursively
                return bisect(l, i_midpoint, i_ub, target_value, debug)

    else: # i_ub < i_lb  (which is illogical, therefore return None)
        return None

def test_bisect(l, i_lb, i_ub, target_value, debug=True):
    result = bisect(l, i_lb, i_ub, target_value, debug=debug)
    print(f"\tTEST bisect(l={l if len(l)<50 else '<l contents SUPRESSED due to length>'}, i_lb={i_lb}, i_ub={i_ub}, target_value={target_value}): {result}")


def process_token_to_word(tkn):
```

```
    """
    This function's sole purpose is to "clean" a token and return a word (or None if the token is not actually a word).

    For instance, we want to strip preceding and trailing whitespace if any exists.

    We also want to remove punctuation characters.
    """

    if tkn is None:
        return None

    tkn = tkn.strip()

    # strip punctuation
    tkn = re.sub(r"[^\w\s]", "", tkn)

    if len(tkn) == 0:
        return None

    return tkn

def tokens_list_to_inverted_index(l_tokens, fn_process_token_to_word=process_token_to_word, debug=False):
    """
    This function converts a list of tokens into two dictionaries:
        1. the first dictionary is keyed by each unqiue word and the corresponding value is the count of that word
        2. the second dictionary is keyed by each unqiue character and the corresponding value is the count of that
character

    The above happens AFTER the token is cleaned by the function specified by the fn_process_token_to_word argument
    """

    d_w_index = {}
    d_c_index = {}

    for tkn in l_tokens:
        w = fn_process_token_to_word(tkn)

        if w is not None:
            w_lower = w.lower()
            d_w_index[w_lower] = d_w_index.get(w_lower, 0) + 1

            for c in w:
                d_c_index[c] = d_c_index.get(c, 0) + 1

    return d_w_index, d_c_index


def summarize_text_file(fname):
```

```
    """
    This function opens a text file and summarizes its word count and letter count.

    Note that case matters!

    arguments:
        fname

    returns:
        1. the summary string, which is formatted, containg the summary statistics:
            1. the frequency count that each (case-sensitive) letter occurs (out of the total number of letters), as 
well as the frequency ratio (as a percentage)
            2. the frequency (count and ratio) of upper-case letters as a group
            3. the frequency (count and ratio) of lower-case letters as a group

        2. a dictionary keyed by words, containing the count of each unique word

        3. a dictionary keyed by letter, containing the count of each unique letter

    """

    l_words = words_file_to_list(fname, use_list_append=False)

    d_w_index, d_c_index = tokens_list_to_inverted_index(l_words)

    n_words = 0
    for k in d_w_index.keys():
        n_words += d_w_index[k]

    # re-arrange d_c_index so that keys are in alphabetical order (based on sorted() order)
    d_c_index = {k:d_c_index[k] for k in sorted(d_c_index.keys())}

    # count all letters (so that we can provide frequency of each letter as a ratio or percentage)
    n_c_all = sum([n_c for _, n_c in d_c_index.items()])

    # now create separate counts of upper and lower case letter (and create formatted individual letter freq summary)
    s_letter_freq__all = ""
    n_c_uc = 0
    n_c_lc = 0
    for c, n_c in d_c_index.items():
        s_letter_freq__all += "\t\t\t" + LETTER_FREQ_TEMPLATE.format(c, n_c, n_c_all, (n_c/n_c_all)*100) + "\n"

        if c.isalpha():
            if c.isupper():
                n_c_uc += n_c
            else:
                n_c_lc += n_c
```

```
    # now create summary strings of upper and lower case freqs
    s_letter_freq__ucase = "\t" + LETTER_FREQ_TEMPLATE.format("UPPER-CASE", n_c_uc, n_c_all, (n_c_uc/n_c_all)*100)
    s_letter_freq__lcase = "\t" + LETTER_FREQ_TEMPLATE.format("LOWER-CASE", n_c_lc, n_c_all, (n_c_lc/n_c_all)*100)

    # return entire formatted summary string as well as the dictionaries (in case we want to use them later)
    return TEXT_FILE_SUMMARY_TEMPLATE.format(
        fname,
        n_words,
        s_letter_freq__all,
        s_letter_freq__ucase,
        s_letter_freq__lcase
    ), d_w_index, d_c_index



def words_file_to_toggle_case(fname_in, fname_out):
    try:
        with open(fname_in, 'r') as f_words_in:
            with open(fname_out, 'w') as f_words_out:
                for words_line_in in f_words_in:
                    words_line_out = ""
                    for c_in in words_line_in:
                        if c_in.isalpha():
                            if c_in.isupper():
                                words_line_out += c_in.lower()
                            else:
                                words_line_out += c_in.upper()
                        else:
                            words_line_out += c_in
                    f_words_out.write(words_line_out)
                f_words_out.close()
                print(f"{fname_out} file written")
            f_words_in.close()
    except Exception as e:
        print(f"words_file_to_toggle_case: ***RUNTIME ERROR caught***: {e}")
# ******************** functions: END ********************





# **************************************** main body (simply calls main() when this py file is exec'ed from bash):
BEGIN ****************************************
if __name__ == '__main__':
    print("Testing is_sorted()...")
    test_is_sorted([1,2,2])
    test_is_sorted(['b','a'])
    test_is_sorted(['b','a','b'])
    print()
```

```
    print("Testing is_anagram()...")
    test_is_anagram("never", "REven")
    test_is_anagram("steve", "STEVEN")
    print()


    print("Testing has_duplicates()...")
    test_has_duplicates([1,2,3])
    test_has_duplicates([1,2,1])
    test_has_duplicates(['a','b','c'])
    test_has_duplicates(['a','b','a'])
    test_has_duplicates(["steven", "steve"])
    test_has_duplicates(["steven", "steven"])
    test_has_duplicates([0])
    test_has_duplicates([])
    print()


    run_bd_paradox_sim_series(N_SIM_SERIES) # note that for formatting purposes, the number of series should be an even
number > 0
    print()


    print("Testing remove_duplicates()...")
    test_remove_duplicates("steven")
    test_remove_duplicates([1,2,3,4,2,1])
    test_remove_duplicates(['s','t','e','v','e','n'])
    print()


    fname = "mobysmall.txt"
    N_ITERATIONS = 100000
    run_benchmark_words_file_to_list_series(fname, N_ITERATIONS, debug=False) # set debug to True will produce A LOT of
output... use it only if you think something is broken
    print()


    print("Testing bisect()...")
    print(f"\tloading word list from {fname}...")
    l_words = words_file_to_list(fname, use_list_append=False)
    print(f"\t\tDONE")
    test_bisect(l_words, 0, len(l_words)-1, "a", debug=True)    # set debug to False for less output
    print()


    s_text_file_summary, _, d_c_index = summarize_text_file(fname)
    print(s_text_file_summary)
```

```
    words_file_to_toggle_case("mobysmall.txt", "mobysmall-case-toggled.txt")


    try:
        with open("mobysmall-summary.txt", "w") as f_summary_out:
            f_summary_out.write(s_text_file_summary)
            f_summary_out.close()
            print(f"mobysmall-summary.txt file written")
    except Exception as e:
        print(f"write textfile summary: ***RUNTIME ERROR caught***: {e}")


    # histogram (bar chart) creation
    fig = plt.figure(figsize=(8,4))
    x = d_c_index.keys()
    y = [d_c_index[c] for c in x]
    plt.bar(x, y)
    plt.title(f"Letter frequency for {fname}")
    plt.show()


    print(f"\n{QUIT_MESSAGE}\n\n")
# **************************************** main body (simply calls main() when this py file is exec'ed from bash): END
****************************************
```

Problems (with associated Source Code)

A

Write a function called **is_sorted** that takes a **list** as a parameter and returns True if the list is **sorted in ascending** order and False otherwise.
You can assume that the elements of the list can be compared with the relational operators <, >, etc. For example, is_sorted([1,2,2]) should return True and is_sorted(['b','a']) should return False.

Answer/Source Code

```
def is_sorted(l):
    n = len(l)


    if n == 1:
        return True
```

```
    # since we compare indices i and i+1, we range from 0 to len(l)-2
    for i in range(n-1):  # recall that n-1 is not inclusive when using range()
        if l[i] > l[i+1]:
            return False

    return True
```

Discussion

If we let sequence (list) L be defined by elements $e_1, \ldots, e_k$, where L has length k, then, by definition, L, is *sorted* if an only if, for ALL all adjacent pairs of elements, we have $e_i \leq e_{i+1}$.

Thus, L is not *sorted*, or fails the condition, when we have $e_i > e_{i+1}$ for ANY pair of adjacent elements. Otherwise, L is *sorted*.

B

Two words are anagrams if you can rearrange the letters from one to spell the other. Write a function called **is_anagram** that takes two strings and returns True if they are anagrams.

Answer/Source Code

Primary Function

```
def is_anagram(l1, l2, normalize_char_case=True):
    """
    Normally anagrams are based on words only.
    But this function supports numeric lists as anagrams, as well..
    """

    n1 = len(l1)
    n2 = len(l2)

    # we can short-circuit when the lengths are unequal
    if n1 != n2:
        return False

    # we normally disregard case when considering char anagrams
    if normalize_char_case:
        l1 = to_lcase(l1)   # but remember, the to_lcase() leaves non-char elements alone
        l2 = to_lcase(l2)
    else: # in case strings and we don't want to normalize to lcase
```

```
        l1 = str_to_list(l1)
        l2 = str_to_list(l2)

    # here we sort each list
    #   this greatly simplifies the problem compared to not sorting
    if not is_sorted(l1):
        l1 = sorted(l1)
    if not is_sorted(l2):
        l2 = sorted(l2)

    # because the two lists are sorted, we can now short-circuit (exit the loop) when we encounter the first mismatch
    for i in range(n1):
        if l1[i] != l2[i]:
            return False

    # if we made it this far then the two lists are necessarily the same length and have the same elements (unless case matters, FOR STRINGS, and case differs for some element)
    return True
```

Helper Functions

```
def str_to_list(s):
    """
    This function converts a string to a list of chars (in case we want to modify the list somehow)
    """
    return [s[i] for i in range(len(s))] if type(s) is str else s.copy()


def to_lcase(l):
    """
    This function only convert char elements in the list to lower-case.
    Obviously, non-char elements will not be affected.
    """
    l_lcase = str_to_list(l)

    for i in range(len(l_lcase)):
        if type(l_lcase[i]) is str:
            l_lcase[i] = l_lcase[i].lower()

    return l_lcase
```

Discussion

Given sequence (list) L_1, defined by elements $e_1, \ldots, e_k$, where L_1 has length k, and L_2, defined by elements $f_1, \ldots, f_k$, where L_2 has length l, then, by definition, L_1 and L_2 are said to be *anagram*s of one another, if an only if:

1. $k = l$, i.e. L_1 and L_2 are the same length
2. upon sorting L_1 and L_2 in non-decreasing order, for EVERY index i, we have $e_i = f_i$, i.e. every corresponding element in each list, after sorting, is identically equal to the other.

Thus, L_1 is not an *anagram* of L_2 (and vice versa) if not both conditions (1) and (2) hold, i.e. if L_1 and L_2 are different lengths, OR there is any index, i, where $e_i \neq f_i$.

C

The Birthday Paradox:
Write a function called **has_duplicates** that takes a list and returns True if there is any element that appears more than once. **It should not modify the original list.**
If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches. Hint: you can **generate random birthdays with the randint function in the random module**.

Answer/Source Code

Primary Function

```
def run_bd_paradox_sim(n_sims, n_class_size=N_DEFAULT_CLASS_SIZE, is_leap_year=False):
    print(f"Running {n_sims} Birthday Paradox simulations on a class size of {n_class_size} students...")
    p = 0
    n_dups = 0
    x = []
    y = []
    for i_sim in range(n_sims):
        l_birthdays = [random.randint(1, N_DAYS_IN_YEAR + (1 if is_leap_year else 0)) for i in range(n_class_size)]
        n_dups += 1 if has_duplicates(l_birthdays) else 0
        x.append(i_sim)
        y.append(n_dups / (i_sim+1))
    p = n_dups / n_sims
    print(f"\tDONE: The probability that at least 2 students from a class size of {n_class_size} have the same birthday converged to {p} after {n_sims} simulations.")

    return p, x, y
```

Helper Functions

```
def run_bd_paradox_sim_series(n_powers_of_ten, n_class_size=N_DEFAULT_CLASS_SIZE, is_leap_year=False, do_plot=True):
    exponents = list(range(1,n_powers_of_ten+1))
```

```
    n_cols = 2
    n_rows = len(exponents) // n_cols
    if do_plot:
        fig, axes = plt.subplots(n_rows, n_cols, figsize=(8,4))

    for i, e in enumerate(exponents):
        n_sims = 10**e
        p, x, y = run_bd_paradox_sim(n_sims=n_sims, n_class_size=N_DEFAULT_CLASS_SIZE, is_leap_year=False)
        if do_plot:
            axis = axes[i//n_cols][i%n_cols]
            axis.set_title(f"# sims = {n_sims}, p = {p}")
            axis.plot(x, y)

    if do_plot:
        fig.tight_layout()
        plt.show()
```

```
def has_duplicates(l, disregard_char_case=False):
    n = len(l)

    # a 0 or single element list is already implicitly sorted, so we can short-circuit
    if n < 2:
        return False

    if not disregard_char_case:
        l = to_lcase(l)
    else: # in case string and we don't want to normalize to lcase
        l = str_to_list(l)

    # here we sort the list
    #   this greatly simplifies the problem compared to not sorting
    if not is_sorted(l):
        l = sorted(l)

    # because the list is sorted, we can now short-circuit (exit the loop) when we encounter the first matching
adjacent pair of elements
    #   range from 0 to len(l)-2
    for i in range(n-1):    # recall that n-1 is not inclusive when using range()
        if l[i] == l[i+1]:
            return True

    return False
```

Discussion

The core logic of the *Birthday Paradox* is housed in the `run_bd_paradox_sim()` function. This function simply implements the logic specified in the reference document https://en.wikipedia.org/wiki/Birthday_problem. As specified in the reference document, the function executes some fixed number of trials (tracked in n_sims) wherein each trial:

1. simulates a list of 23 random birthdays in a given year, i.e. creates a list of 23 random integers in the range $[1, 365]$ (we assume non leap-year)
2. checks if there are any duplicate values via the `has_duplicates()` function and increments n_dups by 1 if true

After all n_sims are executed, `run_bd_paradox_sim()` computes the *convergent probability* (over n_sims trials) that at least 2 out of 23 students will have the same birthday as $\frac{n\_dups}{n\_sims}$.

`run_bd_paradox_sim()` executes a single "batch" of n_sims where $n\_sims = 10^i$. But `run_bd_paradox_sim_series()` wraps `run_bd_paradox_sim()` to execute from 1 to k batches of simulations where, if a batch is indexed by $1 \leq i \leq k$, then each batch executes $n\_sims = 10^i$ simulations (as defined above). For example, if $k = 6$, then batch 1 will execute $10^1 = 10$ simulations, batch 2 will execute $10^2 = 100$, …, (up to) batch 6 will execute $10^6 = 1000000$ (one million) simulations.

With this approach, we show that the probability that at least 2 students out of a class of 23 converges to approximately 0.507311. We note that the reference document has "exact" probability (computed combinatorically) as 0.50730. But since we are dealing with a discrete computational problem, our estimate (after one million simulations) is arguably "pretty darn good".

In order to make it abundantly clear how the number of simulations play out to produce the convergent probability, we end this problem by generating a visual plot of each series of simulations (see output section).

D

Write a function called **remove_duplicates** that takes a list and returns a new list with only the unique elements from the original. Hint: they don't have to be in the same order.

Answer/Source Code

Primary Function

```
def remove_duplicates(l):
    l = str_to_list(l)  # in case l is a string

    n = len(l)

    # a 0 or single element list is already implicitly sorted, so we can short-circuit
    if n < 2:
        return l

    # here we sort the list
    #   this greatly simplifies the problem compared to not sorting
    if not is_sorted(l):
        l = sorted(l)

    l_dups_removed = []

    # iterate from index 0 to len(l)-2
    #   only add the last non-repeating element, which we can do since the list has been sorted
    for i in range(n-1): # recall that n-1 is not inclusive when using range()
        if l[i] != l[i+1]:
            l_dups_removed.append(l[i])

    # but we still have the very last index to add
    #   this works since if there was a dup at the n-2 index, it will not have been added to l_dups_removed
    l_dups_removed.append(l[n-1])

    return l_dups_removed
```

Helper Functions

```
def str_to_list(s):
    """
    This function converts a string to a list of chars (in case we want to modify the list somehow)
    """
    return [s[i] for i in range(len(s))] if type(s) is str else s.copy()
```

```
def is_sorted(l):
    n = len(l)

    if n == 1:
        return True

    # since we compare indices i and i+1, we range from 0 to len(l)-2
    for i in range(n-1):  # recall that n-1 is not inclusive when using range()
        if l[i] > l[i+1]:
            return False

    return True
```

Discussion

The `remove_duplicates()` function depends on the core concept of sorting. So, if the list is not sorted, we sort it (in non-decreasing order). This allows us to exploit the ordering (comparability) of any two adjacent elements e_i and e_{i+1} in list L. L will have duplicates if and only if there exists any index, i, such that $e_i = e_{i+1}$. We have thus described the logic housed within the `has_duplicates()` function. But `remove_duplicates()` uses the complement of the `has_duplicates()` function to identify when a repeating sequence of duplicates terminates and then adds only the terminal element in that (repeating) sequence to the list of distinct/unique elements. Finally, since the very last element in the original list will always be considered unique by following this logic – note that we must index only up to the length minus 2 – we can always safely consider the last element unique.

E

Write a function that reads the file words.txt (You create your own file) and builds a list with one element per word. Write two versions of this function, one using the **append method** and the other using the idiom **t = t + [x]**. Which one takes longer to run? Why?
*use the **time module to measure elapsed time check the following example**.*

```
import time
t0= time.time()
print("Hello")
t1 = time.time() - t0
print("Time elapsed: ", t1) # CPU seconds elapsed (floating point)
```

Answer/Source Code

Primary Function

```
def words_file_to_list(fname, use_list_append=True):
```

```
    l_words = []

    try:
        with open(fname, 'r') as f_words:
            for words_line in f_words:
                for word in words_line.split():
                    if use_list_append:
                        l_words.append(word.strip()) # dynamically resizes
                    else:
                        l_words += [word] # adding to separate lists (which exist in two different places in memory)
            f_words.close()
    except Exception as e:
        print(f"words_file_to_list: ***RUNTIME ERROR caught***: {e}")

    return l_words
```

Helper Functions

```
def benchmark_words_file_to_list(fname, use_list_append, debug=False):
    if debug:
        s_append_list_mechanic = ("t += [x] idiom" if not use_list_append else S_LIST_APPEND_METHOD)
        print(f"Benchmarking '{fname}' file to words list (using {s_append_list_mechanic})")

    # timestamp for start of the execution of words_file_to_list()
    t0 = time.time()

    # execute words_file_to_list()
    l_words = words_file_to_list(fname)

    # timestamp for end of the execution of words_file_to_list()
    t1 = time.time()

    # the delta is just the elapsed time
    t_delta = t1 - t0

    if debug:
        print(f"\ttime elapsed (using {s_append_list_mechanic}): {t_delta} seconds") # CPU seconds elapsed (floating
point)

    return t_delta, l_words
```

```
def run_benchmark_words_file_to_list_series(fname, n_sims, debug=False):
    n_list_append_more_efficient = 0
    n_add_singleton_list_idiom_more_efficient = 0

    print(f"Running {n_sims} words_file_to_list() iterations...")
```

```
    for i_sim in range(n_sims):
        # benchmark using List.append()
        tdelta__list_append, l_words = benchmark_words_file_to_list(fname, use_list_append=True, debug=debug)

        if debug:
            print()

        # benchmark using the t += [x] idiom
        tdelta__add_list_idiom, l_words = benchmark_words_file_to_list(fname, use_list_append=False, debug=debug)

        # the rest of this code is just for formatting the summary output when debugging
        s_mech = ""
        eff_factor = 0

        # update summary and count of times the particular mechanic is more efficient
        if tdelta__add_list_idiom < tdelta__list_append:
            s_mech = S_ADD_SINGLETON_LIST_IDIOM
            eff_factor = tdelta__list_append / tdelta__add_list_idiom
            n_add_singleton_list_idiom_more_efficient += 1
        else:
            s_mech = S_LIST_APPEND_METHOD
            eff_factor = tdelta__add_list_idiom / tdelta__list_append
            n_list_append_more_efficient += 1

        s_efficiency = f"{s_mech} is {eff_factor} more efficient!"

        if debug:
            # print(f"\n{s_efficiency}\n\n{l_words}") # uncomment this to see l_words
            print(f"\n{s_efficiency}")

    # always display summary after all simulations are complete
    eff_ratio__list_append = n_list_append_more_efficient/n_sims
    eff_ratio__add_singleton_list = 1 - eff_ratio__list_append
    print(f"\tDONE: Out of {n_sims} iterations, {S_LIST_APPEND_METHOD} was more efficient
{round(eff_ratio__list_append,2)*100}% of the time, while {S_ADD_SINGLETON_LIST_IDIOM} was more efficient
{round(eff_ratio__add_singleton_list,2)*100}% of the time.")
```

Discussion

Instead of writing two separate functions for this problem, we decided to write a single function, `words_file_to_list()`, and make it behave differently depending on the value of the `use_list_append` (boolean) argument. If `True`, the function will obviously use the `List.append()` Method. Otherwise, the function will use the `l_words += [word]` idiom. But the `words_file_to_list()` function simply opens the file and then inserts each word into a list. It is the `benchmark_words_file_to_list()` function that does the benchmarking, by wrapping timestamps around the call(s) to `words_file_to_list()`, and then taking the difference to get the elapsed time.

The behavior seems to be inconsistent from one single run of `words_file_to_list()` to the next, however. So, we took a *simulation aggregate* approach to this problem, similar to the approach to the Birthday Paradox problem. That is, we run `n_sims` simulations and compute the percentage of times out of `n_sims` times that one of the two mechanics is more efficient (faster) than the other. The `run_benchmark_words_file_to_list_series()` function does this, simply calling `benchmark_words_file_to_list()` and then collating the results into the final efficiency statistics.

F

To check whether a word is in the word list, you could use the **in** operator, but it would be slow because it searches through the words in order.
Because the words are in alphabetical order, we can speed things up with a bisection search (also known as binary search), which is similar to what you do when you look a word up in the dictionary.

You start in the middle and check to see whether the word you are looking for comes before the word in the middle of the list. If so, then you search the first half of the list the same way. Otherwise you search the second half. Either way, you cut the remaining search space in half. If the word list has 113,809 words, it will take about 17 steps to find the word or conclude that it's not there.

Write a function called **bisect** that takes a sorted list and a target value and returns the index of the value in the list, if it's there, or None if it's not.

Answer/Source Code

Primary Function

```
def bisect(l, i_lb, i_ub, target_value, debug=False):
    """
    This function implements what is also known as 'binary search'.
    This implementation is based on RECURSION and is adapted from https://www.geeksforgeeks.org/python-program-for-
binary-search/

    Even though the spec we are given indicates we can assume l is already sorted, we will double-check and do the
sorting just in case.

    parameters:
        l:      the list (elements should all be of the same type and comparable)
        i_lb:   the index lower-bound of l to search
        i_ub:   the index upper-bound of l to search

    From our spec:
        'to check whether a <value> is in the list'
        'returns the index of the value in the list, if it's there, or None if it's not'
    """
```

```
    # short-circuit for 0-length and singleton lists, this is also the "base case" when recursion is used (but we will
go the iteration route instead of recursion)
    n = len(l)
    if n == 0:
        return None
    if n == 1:
        return 0 if l[0] == target_value else None

    # is_sorted check: avoid performance hit (checking if sorted only at top level) - i.e. only when i_lb==0 AND
i_ub==len(l)-1
    if i_lb==0 and i_ub==len(l)-1:
        if not is_sorted(l):
            if debug:
                print("\tl is not sorted! sorting...")
            l = sorted(l)
            if debug:
                # print(f"\t\tsorted l: {l}")
                print(f"\t\tDONE")
        print(f"\tbisecting l for target value -->{target_value}<-- ...")

    # if we are here, we are guaranteed that l is sorted... now we can implement proper binary search logic
    #   first step is to validate that i_ub >= i_lb
    if i_ub >= i_lb:

        # since we are here, we can proceed with "bisecting"
        #   so the first thing we need to do is find the midpoint between i_ub and i_lb: this is the basis of
"bisection"
        i_midpoint = (i_ub + i_lb) // 2     # integer division

        val_at_midpoint = l[i_midpoint]

        if debug:
            print(f"\t\tmidpoint (index) of l between index {i_lb} and {i_ub} is: {i_midpoint} and
l[{i_midpoint}]=={val_at_midpoint}")

        # if target_value is at index i_midpoint, return i_midpoint
        if val_at_midpoint == target_value:
            if debug:
                print(f"\t\ttarget value -->{target_value}<-- found at midpoint index {i_midpoint}")
            return i_midpoint

        else:   # we have already excluded the equality case

            # now we use the fact that elements in l are comparable... this happens recursively
            if target_value < val_at_midpoint:  # then we look in the left half... this is the binary split
                return bisect(l, i_lb, i_midpoint, target_value, debug)

            else:   # otherwise we look in the right half... this happens recursively
```

```
            return bisect(l, i_midpoint, i_ub, target_value, debug)

    else: # i_ub < i_lb  (which is illogical, therefore return None)
        return None
```

Helper Functions

```
def is_sorted(l):
    n = len(l)

    if n == 1:
        return True

    # since we compare indices i and i+1, we range from 0 to len(l)-2
    for i in range(n-1):  # recall that n-1 is not inclusive when using range()
        if l[i] > l[i+1]:
            return False

    return True
```

Discussion

The `bisect()` function is a straightforward ***recursive*** implementation of the classic ***binary search algorithm*** (see https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search). We adapted our specific implementation from https://www.geeksforgeeks.org/python-program-for-binary-search/. Even though the reference documentation provides a description of how the binary search algorithm works, we would like to provide our own summary, as follows:

1. specify the inclusive range of indices (`i_lb` and `i_ub`) to search for `target_value` within list `l`
2. if the list is empty, return `None`
3. if the list has only a single element:
    a. if that element is `target_value` return `0`
    b. otherwise, return `None`
4. the core logic:
    a. if `i_ub` >= `i_lb`
        i. find `i_midpoint = (i_ub + i_lb) // 2     # integer division`
        ii. set `val_at_midpoint = l[i_midpoint]`
        iii. compare `target_value` to `val_at_midpoint`

1. if `val_at_midpoint` equal to `target_value`, then we have found the `target_value` in list `l` at index `i_midpoint` … return the index `i_midpoint`
2. otherwise, we need to check in one of the two "halves" of bisected at index `i_midpoint`
    a. if `target_value` is strictly less than `val_at_midpoint` then we search the **left** "half" by (recursively) calling `bisect()` with arguments:
        i. `i_lb` unchanged
        ii. set `i_ub = i_midpoint`
    b. otherwise, we search the **right** "half" by (recursively) calling `bisect()` with arguments:
        i. set `i_lb = i_midpoint`
        ii. `i_ub` unchanged

b. otherwise `i_ub` < `i_lb` and we have non-sensical range for the upper and lower bounds… return `None`

This logic guarantees that recursively calling `bisect()` will always find `target_value` if it exists in list `l`, or will return `None` if it doesn't. But either way, `bisect()` will not recurse infinitely. Note that this implementation of *binary search* will return the index of the sorted list (assuming `target_value` was in the list), not where it was in the original (unsorted list).

G

Write a function that counts the number of times each unique letter (character) occurs in a sentence entered by the user or in words in a list, and then output the result for each character in the sentence.

H- Read the Moby Deck text file. Find out the:

1. Number of words in the file.
2. The frequency of each letter in the file
3. The frequency of upper case letters file
4. The frequency of lower case letters in the file

5. Convert all uppercase to lower case and vise versa, write in a new file.
6. Towards the end of the file (or in a new file) write the results of 1 - 4 in an output text file
7. Plot the letter frequency.

Answer/Source Code

Primary Function

```
def summarize_text_file(fname):
    """
    This function opens a text file and summarizes its word count and letter count.

    Note that case matters!

    arguments:
        fname

    returns:
        1. the summary string, which is formatted, containing the summary statistics:
            1. the frequency count that each (case-sensitive) letter occurs (out of the total number of letters), as
well as the frequency ratio (as a percentage)
            2. the frequency (count and ratio) of upper-case letters as a group
            3. the frequency (count and ratio) of lower-case letters as a group

        2. a dictionary keyed by words, containing the count of each unique word

        3. a dictionary keyed by letter, containing the count of each unique letter

    """

    l_words = words_file_to_list(fname, use_list_append=False)

    d_w_index, d_c_index = tokens_list_to_inverted_index(l_words)

    n_words = 0
    for k in d_w_index.keys():
        n_words += d_w_index[k]

    # re-arrange d_c_index so that keys are in alphabetical order (based on sorted() order)
    d_c_index = {k:d_c_index[k] for k in sorted(d_c_index.keys())}

    # count all letters (so that we can provide frequency of each letter as a ratio or percentage)
    n_c_all = sum([n_c for _, n_c in d_c_index.items()])

    # now create separate counts of upper and lower case letter (and create formatted individual letter freq summary)
    s_letter_freq__all = ""
    n_c_uc = 0
```

```
    n_c_lc = 0
    for c, n_c in d_c_index.items():
        s_letter_freq__all += "\t\t\t" + LETTER_FREQ_TEMPLATE.format(c, n_c, n_c_all, (n_c/n_c_all)*100) + "\n"

        if c.isalpha():
            if c.isupper():
                n_c_uc += n_c
            else:
                n_c_lc += n_c

    # now create summary strings of upper and lower case freqs
    s_letter_freq__ucase = "\t" + LETTER_FREQ_TEMPLATE.format("UPPER-CASE", n_c_uc, n_c_all, (n_c_uc/n_c_all)*100)
    s_letter_freq__lcase = "\t" + LETTER_FREQ_TEMPLATE.format("LOWER-CASE", n_c_lc, n_c_all, (n_c_lc/n_c_all)*100)

    return TEXT_FILE_SUMMARY_TEMPLATE.format(
        fname,
        n_words,
        s_letter_freq__all,
        s_letter_freq__ucase,
        s_letter_freq__lcase
    ), d_w_index, d_c_index
```

Helper Functions

```
def words_file_to_list(fname, use_list_append=True):
    l_words = []

    try:
        with open(fname, 'r') as f_words:
            for words_line in f_words:
                for word in words_line.split():
                    if use_list_append:
                        l_words.append(word.strip()) # dynamically resizes
                    else:
                        l_words += [word] # adding to separate lists (which exist in two different places in memory)
            f_words.close()
    except Exception as e:
        print(f"words_file_to_list: ***RUNTIME ERROR caught***: {e}")

    return l_words
```

```
def tokens_list_to_inverted_index(l_tokens, fn_process_token_to_word=process_token_to_word, debug=False):
    """
    This function converts a list of tokens into two dictionaries:
        1. the first dictionary is keyed by each unqiue word and the corresponding value is the count of that word
        2. the second dictionary is keyed by each unqiue character and the corresponding value is the count of that
character
```

```
    The above happens AFTER the token is cleaned by the function specified by the fn_process_token_to_word argument
    """

    d_w_index = {}
    d_c_index = {}

    for tkn in l_tokens:
        w = fn_process_token_to_word(tkn)

        if w is not None:
            w_lower = w.lower()
            d_w_index[w_lower] = d_w_index.get(w_lower, 0) + 1

            for c in w:
                d_c_index[c] = d_c_index.get(c, 0) + 1

    return d_w_index, d_c_index
```

```
def process_token_to_word(tkn):
    """
    This function's sole purpose is to "clean" a token and return a word (or None if the token is not actually a word).

    For instance, we want to strip preceding and trailing whitespace if any exists.

    We also want to remove punctuation characters.
    """

    if tkn is None:
        return None

    tkn = tkn.strip()

    # strip punctuation
    tkn = re.sub(r"[^\w\s]", "", tkn)

    if len(tkn) == 0:
        return None

    return tkn
```

Discussion

The logic of the `summarize_text_file()` function is very straightforward. It first loads the text file specified via the `words_file_to_list()` function. Then it calls `tokens_list_to_inverted_index()` to create the required dictionaries: the first dictionary is keyed by each unique word and the corresponding value is the count of that word, and the second dictionary is keyed by each unique character and the corresponding value is the count of that character. But when `tokens_list_to_inverted_index()` creates these dictionaries, it first calls `process_token_to_word()` in order to "clean" a token and return a word (or `None` if the token is not actually a word). Note that this function strips punctuation characters by using ***regular expressions***.

Converting lower to upper-case (and vice versa) characters occurring within “mobysmall.txt” is very straightforward. The `words_file_to_toggle_case()` function that accomplishes this is as follows:

```
def words_file_to_toggle_case(fname_in, fname_out):
    try:
        with open(fname_in, 'r') as f_words_in:
            with open(fname_out, 'w') as f_words_out:
                for words_line_in in f_words_in:
                    words_line_out = ""
                    for c_in in words_line_in:
                        if c_in.isalpha():
                            if c_in.isupper():
                                words_line_out += c_in.lower()
                            else:
                                words_line_out += c_in.upper()
                        else:
                            words_line_out += c_in
                    f_words_out.write(words_line_out)
                f_words_out.close()
                print(f"{fname_out} file written")
            f_words_in.close()
    except Exception as e:
        print(f"words_file_to_toggle_case: ***RUNTIME ERROR caught***: {e}")
```

This function simply opens “mobysmall.txt” for reading, and “mobysmall-case-toggled.txt” for writing. It then reads “mobysmall.txt” line-by-line, followed up by parsing each line read into each corresponding character. If the character is alphabetic, it then simply toggles the case of that character – from upper to lower and vice versa. It then writes that modified character (in the same order) to “mobysmall-case-toggled.txt”. That is all. Very straightforward.

As for problem “H”, we chose to “absorb” it into this problem (G) since problem “H” only involves writing the results of `summarize_text_file()` to a new text file (which we named “mobysmall-summary.txt”). The code that accomplishes the creation of this file uses the results output from the prior call to `summarize_text_file()`, and is toward the end of the "main" driver code and is all but self-explanatory:

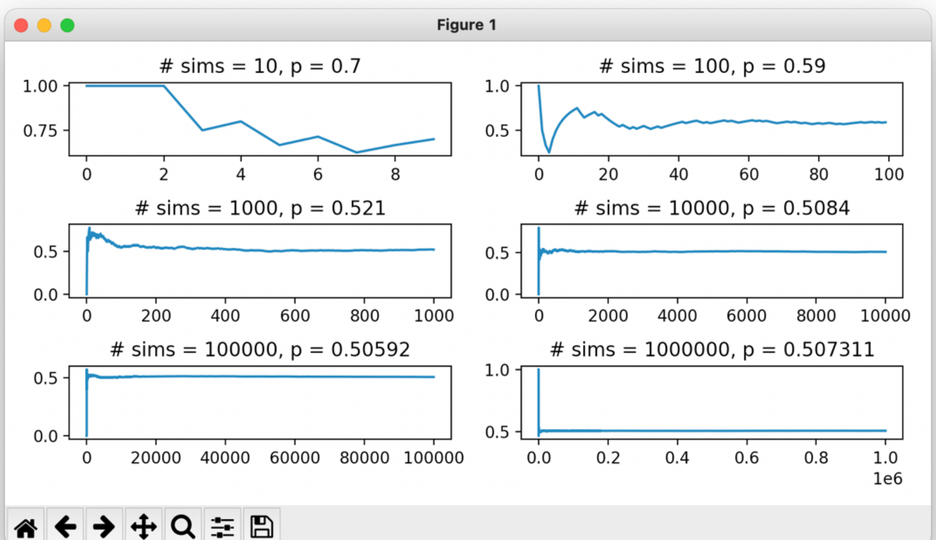
```
try:
        with open("mobysmall-summary.txt", "w") as f_summary_out:
            f_summary_out.write(s_text_file_summary)
            f_summary_out.close()
            print(f"mobysmall-summary.txt file written")
    except Exception as e:
        print(f"write textfile summary: ***RUNTIME ERROR caught***: {e}")
```

Finally, we conclude with the creation of the "histogram" – it's actually a "bar" plot – of the letter frequencies of the characters within the "mobysmall.txt" file. Again, we leverage the results output from the prior call to `summarize_text_file()` – the `d_c_index` dictionary, specifically, which is keyed by each unique character and the corresponding value is the count of that character. The matplotlib library is used to generate the bar plot, with the (alphabetically) sorted sequence of characters as the "x" axis and the counts of each character (provided by the `d_c_index` dictionary) as the "y" axis. The code that accomplishes this is as follows.

```
# histogram (bar chart) creation
    fig = plt.figure(figsize=(8,4))
    x = d_c_index.keys()
    y = [d_c_index[c] for c in x]
    plt.bar(x, y)
    plt.title(f"Letter frequency for {fname}")
    plt.show()
```

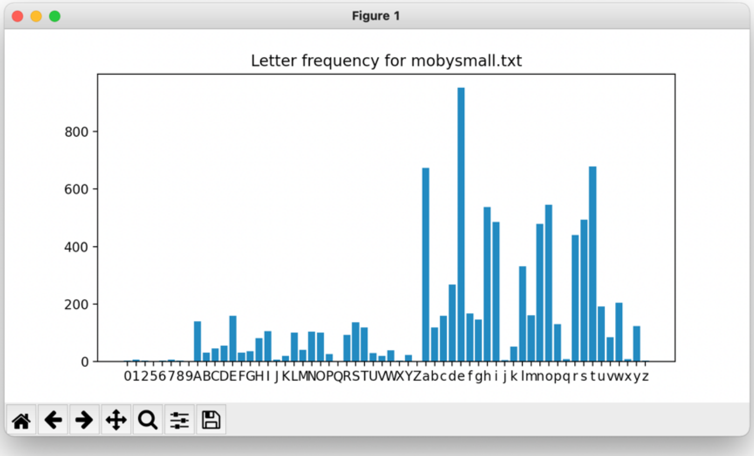
Screenshots of Test Case Executions

```
Stevens-MacBook-Pro:WK14 stevencontreras$ python project6.py
Testing is_sorted()...
        TEST is_sorted(l=[1, 2, 2]): True
        TEST is_sorted(l=['b', 'a']): False
        TEST is_sorted(l=['b', 'a', 'b']): False

Testing is_anagram()...
        TEST is_anagram(l1=never, l2=REven, normalize_char_case=True): True
        TEST is_anagram(l1=steve, l2=STEVEN, normalize_char_case=True): False

Testing has_duplicates()...
        TEST has_duplicates(l=[1, 2, 3], disregard_char_case=False): False
        TEST has_duplicates(l=[1, 2, 1], disregard_char_case=False): True
        TEST has_duplicates(l=['a', 'b', 'c'], disregard_char_case=False): False
        TEST has_duplicates(l=['a', 'b', 'a'], disregard_char_case=False): True
        TEST has_duplicates(l=['steven', 'steve'], disregard_char_case=False): False
        TEST has_duplicates(l=['steven', 'steven'], disregard_char_case=False): True
        TEST has_duplicates(l=[0], disregard_char_case=False): False
        TEST has_duplicates(l=[], disregard_char_case=False): False

Running 10 Birthday Paradox simulations on a class size of 23 students...
        DONE: The probability that at least 2 students from a class size of 23 have the same birthday converged to 0.7 after 10 simulations.
Running 100 Birthday Paradox simulations on a class size of 23 students...
        DONE: The probability that at least 2 students from a class size of 23 have the same birthday converged to 0.59 after 100 simulations.
Running 1000 Birthday Paradox simulations on a class size of 23 students...
        DONE: The probability that at least 2 students from a class size of 23 have the same birthday converged to 0.521 after 1000 simulations.
Running 10000 Birthday Paradox simulations on a class size of 23 students...
        DONE: The probability that at least 2 students from a class size of 23 have the same birthday converged to 0.5084 after 10000 simulations.
Running 100000 Birthday Paradox simulations on a class size of 23 students...
        DONE: The probability that at least 2 students from a class size of 23 have the same birthday converged to 0.50592 after 100000 simulations.
Running 1000000 Birthday Paradox simulations on a class size of 23 students...
        DONE: The probability that at least 2 students from a class size of 23 have the same birthday converged to 0.507311 after 1000000 simulations.
```




```
Testing remove_duplicates()...
        TEST remove_duplicates(l=steven): ['e', 'n', 's', 't', 'v']
        TEST remove_duplicates(l=[1, 2, 3, 4, 2, 1]): [1, 2, 3, 4]
        TEST remove_duplicates(l=['s', 't', 'e', 'v', 'e', 'n']): ['e', 'n', 's', 't', 'v']

Running 100000 words_file_to_list() iterations...
        DONE: Out of 100000 iterations, List.append() method was more efficient 30.0% of the time, while t += [x] idiom was more efficient 70.0% of the time.

Testing bisect()...
        loading word list from mobysmall.txt...
                DONE
        l is not sorted! sorting...
                DONE
        bisecting l for target value -->a<-- ...
                midpoint (index) of l between index 0 and 2001 is: 1000 and l[1000]==if
                midpoint (index) of l between index 0 and 1000 is: 500 and l[500]==againe
                midpoint (index) of l between index 0 and 500 is: 250 and l[250]==IN
                midpoint (index) of l between index 250 and 500 is: 375 and l[375]==TEN
                midpoint (index) of l between index 375 and 500 is: 437 and l[437]==WHALE.
                midpoint (index) of l between index 437 and 500 is: 468 and l[468]==a
                target value -->a<-- found at midpoint index 468
        TEST bisect(l=<l contents SUPRESSED due to length>, i_lb=0, i_ub=2001, target_value=a): 468
```

```
***** SUMMARY OF TEXT FILE: mobysmall.txt *****
    WORD COUNT: 1992
    LETTER FREQUENCY:
        ALL:
                        0:      3       (/9015)         0.03%
                        1:      7       (/9015)         0.08%
                        2:      3       (/9015)         0.03%
                        5:      2       (/9015)         0.02%
                        6:      4       (/9015)         0.04%
                        7:      6       (/9015)         0.07%
                        8:      3       (/9015)         0.03%
                        9:      2       (/9015)         0.02%
                        A:      140     (/9015)         1.55%
                        B:      31      (/9015)         0.34%
                        C:      45      (/9015)         0.50%
                        D:      55      (/9015)         0.61%
                        E:      159     (/9015)         1.76%
                        F:      31      (/9015)         0.34%
                        G:      35      (/9015)         0.39%
                        H:      81      (/9015)         0.90%
                        I:      105     (/9015)         1.16%
                        J:      7       (/9015)         0.08%
                        K:      19      (/9015)         0.21%
                        L:      100     (/9015)         1.11%
                        M:      40      (/9015)         0.44%
                        N:      104     (/9015)         1.15%
                        O:      101     (/9015)         1.12%
                        P:      26      (/9015)         0.29%
                        Q:      1       (/9015)         0.01%
                        R:      93      (/9015)         1.03%
                        S:      137     (/9015)         1.52%
                        T:      119     (/9015)         1.32%
                        U:      30      (/9015)         0.33%
                        V:      20      (/9015)         0.22%
                        W:      39      (/9015)         0.43%
                        X:      4       (/9015)         0.04%
                        Y:      22      (/9015)         0.24%
                        Z:      1       (/9015)         0.01%
                        a:      673     (/9015)         7.47%
                        b:      119     (/9015)         1.32%
                        c:      159     (/9015)         1.76%
                        d:      267     (/9015)         2.96%
                        e:      952     (/9015)         10.56%
                        f:      167     (/9015)         1.85%
                        g:      146     (/9015)         1.62%
                        h:      537     (/9015)         5.96%
                        i:      486     (/9015)         5.39%
                        j:      5       (/9015)         0.06%
                        k:      52      (/9015)         0.58%
                        l:      331     (/9015)         3.67%
                        m:      160     (/9015)         1.77%
                        n:      478     (/9015)         5.30%
                        o:      546     (/9015)         6.06%
                        p:      130     (/9015)         1.44%
                        q:      8       (/9015)         0.09%
                        r:      439     (/9015)         4.87%
                        s:      493     (/9015)         5.47%
                        t:      678     (/9015)         7.52%
                        u:      191     (/9015)         2.12%
                        v:      84      (/9015)         0.93%
                        w:      204     (/9015)         2.26%
                        x:      8       (/9015)         0.09%
                        y:      123     (/9015)         1.36%
                        z:      4       (/9015)         0.04%

        UPPER-CASE:     1545    (/9015)         17.14%
        LOWER-CASE:     7440    (/9015)         82.53%

mobysmall-case-toggled.txt file written
mobysmall-summary.txt file written
```




```
THAT'S ALL FOLKS!  Thanks for playing.  Bye bye.
```

Video

You can view our video at https://youtu.be/3X-L12RSRwA.

References

Birthday problem - Wikipedia. (2021). Retrieved from https://en.wikipedia.org/wiki/Birthday_problem

Cormen, T., Balkcom, D., & Khan Academy. (2021). Binary search. Retrieved from https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

Python Program for Binary Search (Recursive and Iterative) - GeeksforGeeks. (2021). Retrieved from https://www.geeksforgeeks.org/python-program-for-binary-search