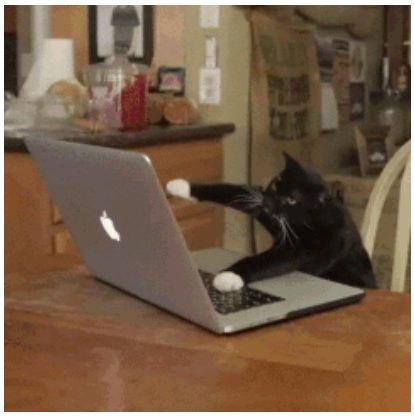
Project 2: CS 61A Autocorrected Typing Software

cats.zip (cats.zip)



Programmers dream of Abstraction, recursion, and Typing really fast.

Introduction

Due Dates:

- Submit Phase 1 by Tuesday, 7/14.
- Submit the whole project by Thursday, 7/23.
- You will receive an early submission bonus point for submitting the entire project by Wednesday, 7/22.

You may work with a partner for the entire project.

In this project, you will write a program that measures typing speed. Additionally, you will implement typing autocorrect, which is a feature that attempts to correct the spelling of a word after a user types it. This project is inspired by typeracer (https://play.typeracer.com/).

Final Product

Our staff solution to the project can be interacted with at cats.cs61a.org (https://cats.cs61a.org) - if you'd like, try it out now! When you finish the project, you'll have implemented a significant part of this game yourself!

Download starter files

You can download all of the project code as a zip archive (cats.zip). This project includes several files, but your changes will be made only to `cats.py`. Here are the files included in the archive:

- `cats.py`: The typing test logic.
- `utils.py`: Utility functions for interacting with files and strings.
- `ucb.py`: Utility functions for CS 61A projects.
- `data/sample_paragraphs.txt`: A file containing text samples to be typed. These are scraped (https://github.com/kavigupta/wikivideos/blob/626de521e04ca643751ed85d549faca6ea528b1d/get_corpus.py) Wikipedia articles about various topics.

- `data/common_words.txt` : A file containing common English words in order of frequency (https://github.com/first20hours/google-10000-english/blob/master/google-10000-english-usa-no-swears.txt).
- `data/words.txt` : A file containing many more English words in order of frequency (https://github.com/first20hours/google-10000-english/blob/master/google-10000-english-usa-no-swears.txt).
- `gui.py` : A web server for the web-based graphical user interface (GUI).
- `gui_files` : A directory of files needed for the graphical user interface (GUI).
- `images` : A directory of images.
- `ok` , `proj02.ok` , `tests` : Testing files.

The CATS GUI is an open source project on Github (https://github.com/Cal-CS-61A-Staff/cats-gui).

Logistics

The project is worth 20 points. 17 points are assigned for correctness of your final code, 1 point for submitting Phase 1 by the checkpoint deadline, and 2 points for composition.

You will turn in the following files:

- `cats.py`

You do not need to modify or turn in any other files to complete the project. To submit the project, run the following command:

```
python3 ok --submit
```

You will be able to view your submissions on the Ok dashboard (http://ok.cs61a.org).

For the functions that we ask you to complete, there may be some initial code that we provide. If you would rather not use that code, feel free to delete it and start from scratch. You may also add new function definitions as you see fit.

However, please do not modify any other functions. Doing so may result in your code failing our autograder tests. Also, please do not change any function signatures (names, argument order, or number of arguments).

Throughout this project, you should be testing the correctness of your code. It is good practice to test often, so that it is easy to isolate any problems. However, you should not be testing *too* often, to allow yourself time to think through problems.

We have provided an autograder called `ok` to help you with testing your code and tracking your progress. The first time you run the autograder, you will be asked to log in with your Ok account using your web browser. Please do so. Each time you run `ok` , it will back up your work and progress on our servers.

The primary purpose of `ok` is to test your implementations.

We recommend that you submit after you finish each problem. Only your last submission will be graded. It is also useful for us to have more backups of your code in case you run into a submission issue.

If you do not want us to record a backup of your work or information about your progress, you can run

```
python3 ok --local
```

With this option, no information will be sent to our course servers. If you want to test your code interactively, you can run

```
python3 ok -q [question number] -i
```

with the appropriate question number (e.g. `01`) inserted. This will run the tests for that question until the first one you failed, then give you a chance to test the functions you wrote interactively.

You can also use the debug printing feature in OK by writing

```
print("DEBUG:", x)
```

which will produce an output in your terminal without causing OK tests to fail with extra output.

Phase 1: Typing

Problem 1 (1 pt)

Implement `choose`, which selects which paragraph the user will type. It takes a list of `paragraphs` (strings), a `select` function that returns `True` for paragraphs that can be selected, and a non-negative index `k`. The `choose` function return's the `k` th paragraph for which `select` returns `True`. If no such paragraph exists (because `k` is too large), then `choose` returns the empty string.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 01 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 01
```

Problem 2 (2 pt)

Implement `about`, which takes a list of `topic` words. It returns a function which takes a paragraph and returns a boolean indicating whether that paragraph contains any of the words in `topic`. The returned function can be passed to `choose` as the `select` argument.

To make this comparison accurately, you will need to ignore case (that is, assume that uppercase and lowercase letters don't change what word it is) and punctuation.

Assume that all words in the `topic` list are already lowercased and do not contain punctuation.

> Hint: You may use the string utility functions in `utils.py`.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 02 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 02
```

Problem 3 (1 pt)

Implement `accuracy`, which takes a `typed` paragraph and a `reference` paragraph. It returns the percentage of words in `typed` that exactly match the corresponding words in `reference`. Case and punctuation must match as well.

A *word* in this context is any sequence of characters separated from other words by whitespace, so treat "dog;" as all one word.

If a typed word has no corresponding word in the reference because `typed` is longer than `reference`, then the extra words in `typed` are all incorrect.

If `typed` is empty, then the accuracy is zero.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 03 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 03
```

Problem 4 (1 pt)

Implement `wpm`, which computes the *words per minute*, a measure of typing speed, given a string `typed` and the amount of `elapsed` time in seconds. Despite its name, *words per minute* is not based on the number of words typed, but instead the number of characters, so that a typing test is not biased by the length of words. The formula for *words per minute* is the ratio of the number of characters (including spaces) typed divided by 5 (a typical word length) to the elapsed time in minutes.

For example, the string `"I am glad!"` contains three words and ten characters (not including the quotation marks). The words per minute calculation uses 2 as the number of words typed (because 10 / 5 = 2). If someone typed this string in 30 seconds (half a minute), their speed would be 4 words per minute.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 04 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 04
```

Time to test your typing speed! You can use the command line to test your typing speed on paragraphs about a particular topic. For example, the command below will load paragraphs about cats or kittens. See the `run_typing_test` function for the implementation if you're curious (but it is defined for you).

```
python3 cats.py -t cats kittens
```

You can try out the web-based graphical user interface (GUI) using the following command.

```
python3 gui.py
```

To submit your Phase 1 checkpoint type:

```
python3 ok --submit
```

You can submit again once you've finished the whole project, and we will score only your latest submission, but please submit at least once before the checkpoint deadline (after finishing at least the Phase 1 questions) to receive credit for the checkpoint.

Phase 2: Autocorrect

In the web-based GUI, there is an autocorrect button, but right now it doesn't do anything. Let's implement automatic correction of typos. Whenever the user presses the space bar, if the last word they typed doesn't match a word in the dictionary but is close to one, then that similar word will be substituted for what they

typed.

Problem 5 (2 pt)

Implement `autocorrect`, which takes a `user_word`, a list of all `valid_words`, a `diff_function`, and a `limit`.

If the `user_word` is contained inside the `valid_words` list, `autocorrect` returns that word. Otherwise, `autocorrect` returns the word from `valid_words` that has the lowest difference from the provided `user_word` based on the `diff_function`. However, if the lowest difference between `user_word` and any of the `valid_words` is greater than `limit`, then `user_word` is returned instead.

A diff function takes in three arguments, which are the two strings to be compared (first the `user_word` and then a word from `valid_words`), as well as the `limit`. The output of the diff function, which is a number, represents the amount of difference between the two strings.

Assume that `user_word` and all elements of `valid_words` are lowercase and have no punctuation.

Important: if multiple strings have the same lowest difference according to the `diff_function`, `autocorrect` should return the string that appears first in `valid_words`.

> Hint: Try using `max` or `min` with the optional `key` argument.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 05 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 05
```

Problem 6 (2 pts)

Implement `shifty_shifts`, which is a diff function that takes two strings. It returns the minimum number of characters that must be changed in the `start` word in order to transform it into the `goal` word. If the strings are not of equal length, the difference in lengths is added to the total.

Here are some examples:

```
>>> big_limit = 10
>>> shifty_shifts("nice", "rice", big_limit)    # Substitute: n -> r
1
>>> shifty_shifts("range", "rungs", big_limit)  # Substitute: a -> u, e -> s
2
>>> shifty_shifts("pill", "pillage", big_limit) # Don't substitute anything, length difference of 3.
3
>>> shifty_shifts("roses", "arose", big_limit)  # Substitute: r -> a, o -> r, s -> o, e -> s, s -> e
5
>>> shifty_shifts("rose", "hello", big_limit)   # Substitue: r->h, o->e, s->l, e->l, length difference of 1.
5
```

If the number of characters that must change is greater than `limit`, then `shifty_shifts` should return any number larger than `limit` and should minimize the amount of computation needed to do so.

These two calls to `shifty_shifts` should take about the same amount of time to evaluate:

```
>>> limit = 4
>>> shifty_shifts("roses", "arose", limit) > limit
True
>>> shifty_shifts("rosesabcdefghijklm", "arosenopqrstuvwxyz", limit) > limit
True
```

Important: You may not use `while` or `for` statements in your implementation. Use recursion.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 06 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 06
```

Try turning on autocorrect in the GUI. Does it help you type faster? Are the corrections accurate? You should notice that inserting a letter or leaving one out near the beginning of a word is not handled well by this diff function. Let's fix that!

Problem 7 (3 pt)

Implement `meowstake_matches`, which is a diff function that returns the minimum number of edit operations needed to transform the `start` word into the `goal` word.

There are three kinds of edit operations:

1. Add a letter to `start`,
2. Remove a letter from `start`,
3. Substitute a letter in `start` for another.

Each edit operation contributes 1 to the difference between two words.

```
>>> big_limit = 10
>>> meowstake_matches("cats", "scat", big_limit)       # cats -> scats -> scat
2
>>> meowstake_matches("purng", "purring", big_limit)   # purng -> purrng -> purring
2
>>> meowstake_matches("ckiteus", "kittens", big_limit) # ckiteus -> kiteus -> kitteus -> kittens
3
```

We have provided a template of an implementation in `cats.py`. This is a recursive function with three recursive calls. One of these recursive calls will be similar to the recursive call in `shifty_shifts`.

You may modify the template however you want or delete it entirely.

If the number of edits required is greater than `limit`, then `meowstake_matches` should return any number larger than `limit` and should minimize the amount of computation needed to do so.

These two calls to `meowstake_matches` should take about the same amount of time to evaluate:

```
>>> limit = 2
>>> meowstake_matches("ckiteus", "kittens", limit) > limit
True
>>> shifty_shifts("ckiteusabcdefghijklm", "kittensnopqrstuvwxyz", limit) > limit
True
```

There are no unlock cases for this problem. Make sure you understand the above test cases!

Test your implementation before proceeding:

```
python3 ok -q 07
```

Try typing again. Are the corrections more accurate?

```
python3 gui.py
```

Extensions: You may optionally design your own diff function called `final_diff`. Here are some ideas for making even more accurate corrections:

- Take into account which additions and deletions are more likely than others. For example, it's much more likely that you'll accidentally leave out a letter if it appears twice in a row.
- Treat two adjacent letters that have swapped positions as one change, not two.
- Try to incorporate common misspellings

Phase 3: Multiplayer

Typing is more fun with friends! You'll now implement multiplayer functionality, so that when you run `gui.py` on your computer, it connects to the course server at cats.cs61a.org (https://cats.cs61a.org) and looks for someone else to race against.

To race against a friend, 5 different programs will be running:

- Your GUI, which is a program that handles all the text coloring and display in your web browser.
- Your `gui.py`, which is a web server that communicates with your GUI using the code you wrote in `cats.py`.
- Your opponent's `gui.py`.
- Your opponent's GUI.
- The CS 61A multiplayer server, which matches players together and passes messages around.

When you type, your GUI sends what you have typed to your `gui.py` server, which computes how much progress you have made and returns a progress update. It also sends a progress update to the multiplayer server, so that your opponent's GUI can display it.

Meanwhile, your GUI display is always trying to keep current by asking for progress updates from `gui.py`, which in turn requests that info from the multiplayer server.

Each player has an `id` number that is used by the server to track typing progress.

Problem 8 (2 pt)

Implement `report_progress`, which is called every time the user finishes typing a word. It takes a list of the words `typed`, a list of the words in the `prompt`, the user `id`, and a `send` function that is used to send a progress report to the multiplayer server. Note that there will never be more words in `typed` than in `prompt`.

Your progress is a ratio of the words in the `prompt` that you have typed correctly, up to the first incorrect word, divided by the number of `prompt` words. For example, this example has a progress of `0.25`:

```
report_progress(["Hello", "ths", "is"], ["Hello", "this", "is", "wrong"], ...)
```

Your `report_progress` function should return this number. Before that, it should send a message to the multiplayer server that is a two-element dictionary containing the keys `'id'` and `'progress'`. The `id` is passed into `report_progress` from the GUI. The progress is the fraction you compute. Call `send` on this dictionary to send it to the multiplayer server.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 08 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 08
```

Problem 9 (1 pt)

Implement `time_per_word`, which takes in `times_per_player`, a list of lists for each player with timestamps indicating when each player finished typing each word. It also takes in a list `words`. It returns a `game` with the given information.

A `game` is a data abstraction that has a list of `words` and `times`. The `times` are stored as a list of lists of how long it took each player to type each word. `times[i][j]` indicates how long it took player `i` to type word `j`.

Timestamps are cumulative and always increasing, while the values in `time` are differences between consecutive timestamps. For example, if `times_per_player = [[1, 3, 5], [2, 5, 6]]`, the corresponding `time` attribute of the `game` would be `[[2, 2], [3, 1]]` (`(3-1)`, `(5-3)` and `(5-2)`, `(6-5)`).

Be sure to use the `game` constructor when returning a `game`, rather than assuming a particular data format.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 09 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 09
```

Problem 10 (2 pt)

Implement `fastest_words`, which returns which words each player typed fastest. This function is called once both players have finished typing. It takes in a `game`.

The `game` argument is a `game` data abstraction, like the one returned in Problem 9. You can access words in the `game` with selectors `word_at`, which takes in a `game` and the `word_index` (an integer). You can access the time it took any player to type any word using `time`.

The `fastest_words` function returns a list of lists of words, one list for each player, and within each list the words they typed the fastest (against all the other players). In the case of a tie, consider the earliest player in the list (the smallest player index) to be the one who typed it the fastest.

Be sure to use the accessor functions for the `game` data abstraction, rather than assuming a particular data format.

Before writing any code, unlock the tests to verify your understanding of the question.

```
python3 ok -q 10 -u
```

Once you are done unlocking, begin implementing your solution. You can check your correctness with:

```
python3 ok -q 10
```

Congratulations! Now you can play against other students in the course. Set `enable_multiplayer` to `True` near the bottom of `cats.py` and type swiftly!

```
python3 gui.py
```

At this point, run the entire autograder to see if there are any tests that don't pass.

```
python3 ok
```

Once you are satisfied, submit to Ok to complete the project.

```
python3 ok --submit
```

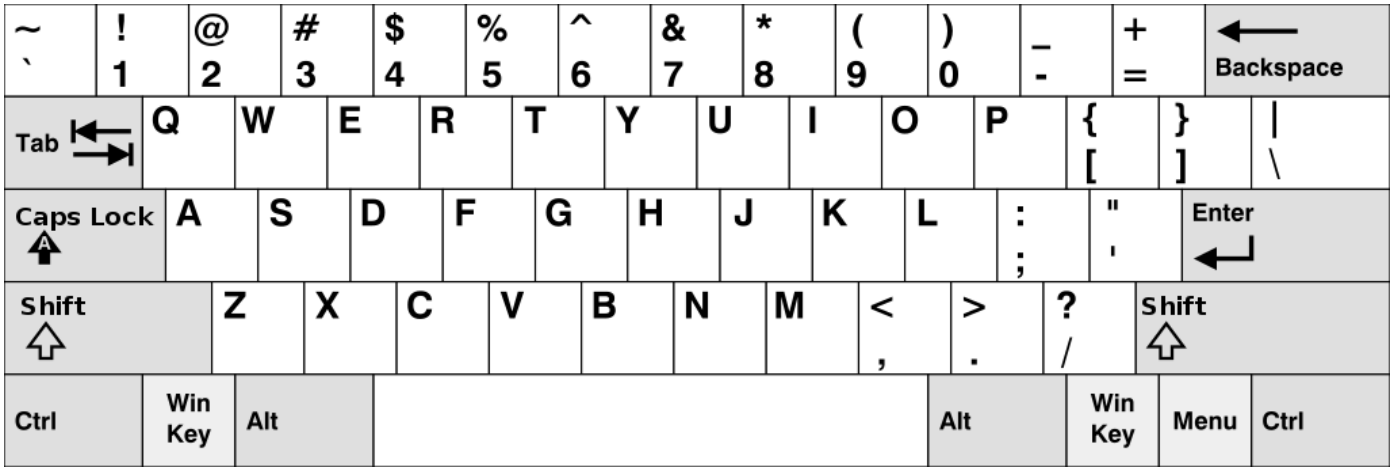
If you have a partner, make sure to add them to the submission on okpy.org.

Check to make sure that you did all the problems by running

```
python3 ok --score
```

Extra Credit

Extra Credit Problem 1: Accuracy (1 pt)



Now, there may be a problem with relying on our existing `meowstake_matches`. Sometimes, for a given `user_input` string, multiple valid words have an equal difference from the `user_input` string. However, given that the user is typing on a QWERTY keyboard (see above), certain words are much more likely to be the intended word compared to others. For example, given the string "wird" as `user_input`, both "bird" and "wire" are valid words that have a difference of 1 from "wird." "wire" is a much more likely to be the word that the user was intending to type, because "d" is much closer to "e" than "w" is to "b" on the keyboard. In other words, "wire" is the better word to autocorrect to.

We've provided you with a dictionary KEY_DISTANCES. You can see how this dictionary was created in utils.py if you're curious, but you will not need to call any function in utils.py in order to create it.

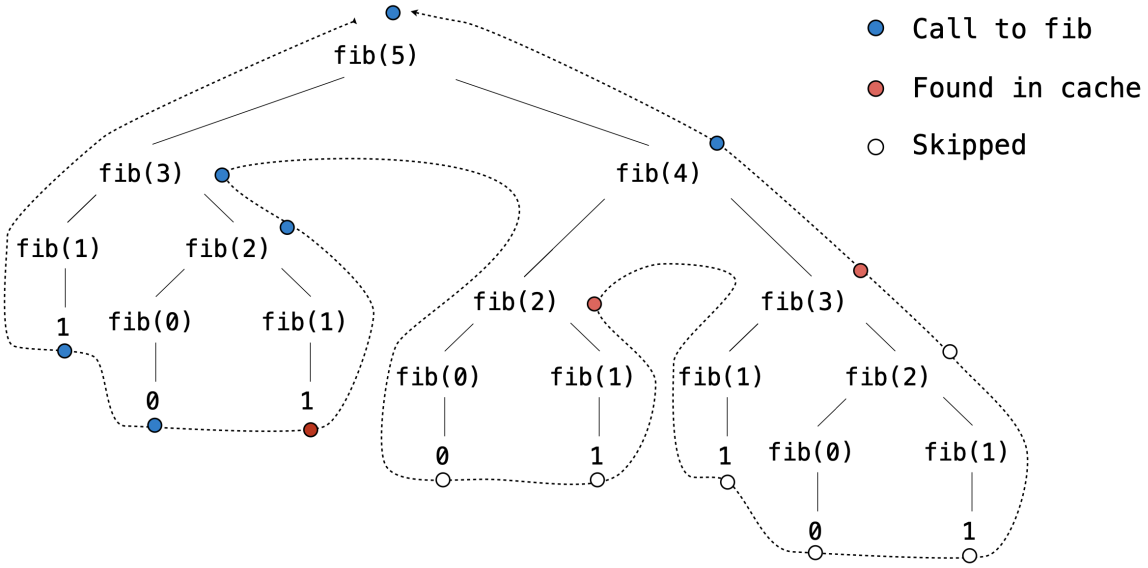
Your task is to implement the function `key_distance_diff`. This is a diff function that, for every substitute operation where an existing letter is substituted with another letter, takes into account how far the letter being substituted in is from the letter being replaced. For example, substituting "q" with "t" increases the difference measurement by twice as much as "q" and "e" because "q" and "t" are 4 keys apart, while "q" and "e" are 2 keys apart. In the event where more than LIMIT number of changes need to be made, return the difference as infinite. Note that in python this can be done with float("inf").

You can assume that all inputs only include letter characters (no numbers, quotation marks ("), commas (,), periods (.), etc.).

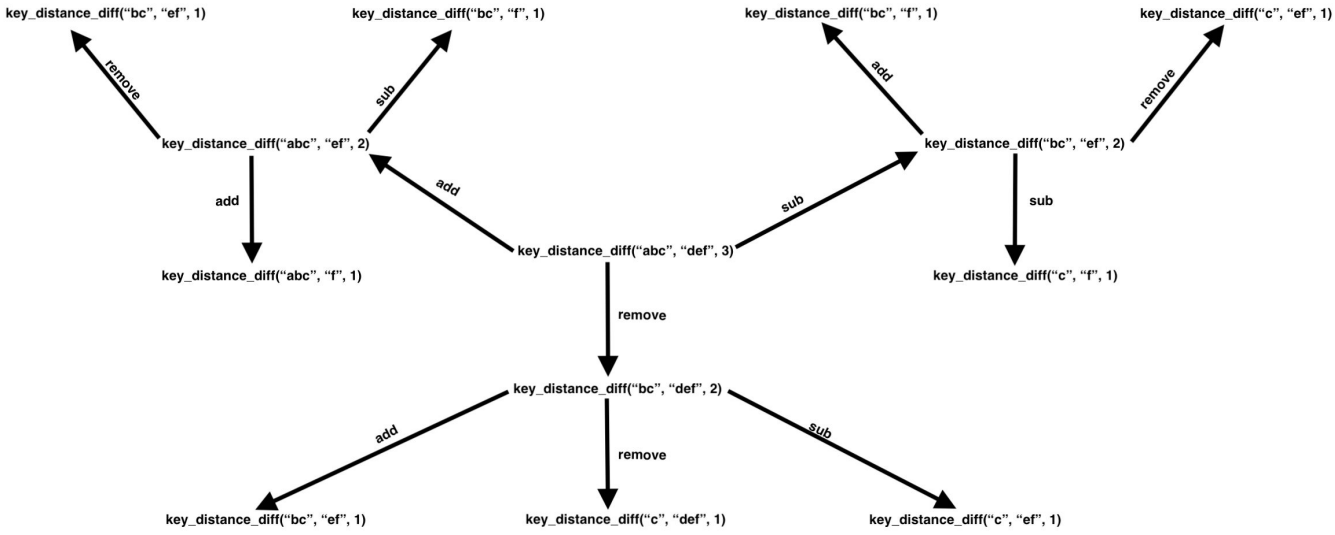
After writing code, test your implementation:

```
python3 ok -q EC1
```

Extra Credit Problem 2: Efficiency (1 pt)



The diff functions we wrote in the previous questions are very inefficient. You will likely find that the computer will make the same recursive call multiple times. For a function with multiple arguments and three recursive calls this can be harder to see. It can be easier to first see this with a function like `fib` that is defined in lecture. Noticed how many redundant recursive calls there are in the above tree diagram. Our goal is to have our program store past results of evaluated recursive calls so that we can reuse them if the same recursive call comes up in the future. For example, the first branch of `fib(5)` calls `fib(3)`, which has not yet been evaluated. So we must go through all of its subsequent recursive calls to find its return value. However when we encounter the call to `fib(3)` that is a branch of `fib(4)`, we have already found its return value before! So if we have a way to store and retrieve that information in something called a cache, we can avoid needless computation. We no longer need to make any subsequent recursive calls to its branches `fib(1)` and `fib(2)`. For your convenience, a simple diagram showing this issue for `key_distance_diff` is shown below, try to identify which calls are redundant.



While evaluating the difference between any two words may not seem that slow, autocorrect tries to run the difference between a typed word and thousands of potential words in the English dictionary. Therefore any inefficiencies in our diff function will be magnified greatly in `autocorrect`. We will need a way to prevent re-computing recursive calls that we have already computed once.

If `key_distance_diff` has already been called on two words with the same arguments, then we should not have to re-calculate their difference. Instead, if `key_distance_diff` is called on two words a second time, the previously calculated difference should be returned without re-calculating the difference. Furthermore, if autocorrect has already been called on a word, no calls to `key_distance_diff` should be necessary. The process of "remembering" when we have made a call on two strings in this way is called memoization. See the lecture (https://www.youtube.com/watch?v=IB8VSP9EZQs&list=PL6BsET-8jgYXsQ35_ZS1e_tX5LZf5PPrS&index=3) including Growth (July 22th) for more details.

Your task is to implement the function `faster_autocorrect`, which should maintain the same accuracy of `autocorrect` but run more quickly using memoization. You should also make sure that every function autocorrect uses is memoized. We will test this by counting the number of recursive calls that are made to see if memoization was correctly implemented. Because of this it is important that you don't have any extraneous calls to `diff_func` in either `faster_autocorrect` or `autocorrect`.

After writing code, test your implementation:

```
python3 ok -q EC2
```

Video hint (https://www.youtube.com/watch?v=ZVvS_w4LNis)

CS 61A (/~cs61a/su20/)

Weekly Schedule (/~cs61a/su20/weekly.html)

Office Hours (/~cs61a/su20/office-hours.html)

Staff (/~cs61a/su20/staff.html)

Resources (/~cs61a/su20/resources.html)

Studying Guide (/~cs61a/su20/articles/studying.html)

Debugging Guide (/~cs61a/su20/articles/debugging.html)

Composition Guide (/~cs61a/su20/articles/composition.html)

Policies (/~cs61a/su20/articles/about.html)

Assignments (/~cs61a/su20/articles/about.html#assignments)

Exams (/~cs61a/su20/articles/about.html#exams)

Grading (/~cs61a/su20/articles/about.html#grading)