CS110 Final Project - Plagiarism Detector

In [1]:

```
NAME = 'William Nguyen'
COLLABORATORS = 'Ha, Catherine'
```

Question 1: Rolling Hashing

My approach:

1. I have a function to clean all weird or disturbing symbols in the inputs so that the string comparison can be dealt with more easily
2. I have two functions to calculate hash value for each k-long slice of the two strings. The first one calculates the initial window. And then for the rest, we only have to roll another next character and calculate based on the previous window.
3. My rolling hashing implementation takes in 4 arguments: 2 strings, the base in calculation and the hash table size (or q). I chose 26 as the base in the hash value calculation because there can be only 26 distinct characters after cleaning the string. The higher the base, the less likelihood windows get the same position. Therefore, to balance between space the hash table consumes and few collision in the tables, 26 seems fine to work on. Furthermore, I also chose the hash table size to be q because this ensures all hash values will be in the hash table. In the division method, we need to take the remainder to simplify our hash value, yet still try to assign windows to distinct positions. Therefore, the divisor should be a prime number to minimize collisions. The bigger the hash table size, the more chance for all windows to fall into different positions. However, we also need to beware of space consumption. In my first attempt, I will choose the hash table to be 97 because it is the largest 2-digit prime number. This initial choice helped me debug better and verify my results. Later on, I will conduct experiments to visualize the bahavior of changes in collisions with different prime numbers and choose the hash table size thanks to this.
4. For data structure, I intend to use a dictionary serving as the hash table. Dictionary really works because it is easy and only takes constant time to insert, search and extract things. To specify, we map value (the window of inputted string) to the corresponding key (the hash value - position in the hash table) so that we can save time in string comparison. Unlike the naive approach which compares all possible combinations in the inputs, the hash table allows us to only compare substrings with the same hash value. Therefore, after putting all k-length substrings of x into the hash table with their hash values, we can calculate hash values of y substrings and only have to examine similarities with x substrings with the same hash values. This is more efficient in runtime because we are down to the comparison of hash value before string matching. Therefore, few collisions can even make the algorithm better. In this problem, choosing a heap or a kind of tree will no longer be appropriate because we have no demands of sorting substrings according to their values or something. Furthermore, for heaps, it takes O(log n) to insert new substrings into the heap because of maintaining heap properties and O(n) to search for substring in the heap in the worst case scenario with n as the number of inputs. Likewise, extracting information out of the heap or a Binary Search Tree is also computationally expensive with O(h) with h being the height of the tree or the heap. O(h) tends to be O(log n) in average case, but can also reach O(n) in the worst case . All search and insert operations in a dictionary only takes O(1) time.
5. When collisions happen, I will use chaining to address this problem because I want to take full advantage of dictionary or list data structure. In the dictionary or the hash table, with the key (the hash value), I will include a list of values (substrings of the input). As a result, when collisions happen, I just append this substring to this list at that hash value. This is actually easy to handle because when I want to examine the y string to detect plagiarism, I can easily use dictionary and list to search for information in constant time.
6. For my algorithmic strategy, instead of comparing the string directly, I will compute hash values for each substring and only compare those with the same hash values. This is more computationally efficient because it reduces redundant comparisons. In other words, with a filtering step before the actual comparison, I can only focus on potential candidates for string matching.

    - First, I will preprocess inputs to ensure clear strings without disturbing aspects
    - Then, I calculate hash value for the first substring by the division method which uses the modulus

- **After that, I recursively roll to the next character, calculate the next hash value based on the previous substring's hash value**
- **After putting all substrings of x into their positions in the hash table, I turn to the y substring**
- **This time, I compute the hash value of y substring and immediately check if at that position, there is any x substring there**
- **If yes, I will compare the y-substring and if this is a match, I add their x - y positions into the result list**
- **This process happens recursively until I scan through the entire y-list to ensure no string match is missed**

To evaluate, this method will definitely work better than the brute force approach because it saves time in making reasonable comparisons.

In [2]:

```
def cleaning(text):
    """
    Preprocess the inputted string for plagiarism detection, remove all unusual or distra
cting characters

    Input
    ------
    text: the string itself

    Output
    ------
    str
        the cleaned version of the text
    """
    # lowercase all characters
    text = text.lower()
    ans = ''
    for char in text:
        # only accept normal letters
        if char.isalpha():
            ans += char
    return ans

assert cleaning('Hello everyone! I am William!!!!!') == 'helloeveryoneiamwilliam'
```

In [3]:

```
def first_hashcode(text, base = 26, size = 97):
    """
    Compute the hash value of the initial window

    Inputs
    ------
    text: str
        the substring
    base: int
        the base in our calculation
    size: int
        the hash table size or the modulus - division method

    Output
    ------
    int
        the hash value of the initial window

    """
    # initialize
    value = 0
    for i in range(len(text)):
        value += ord(text[-i-1])*(base**i)
    # get the remainder
    position = value % size
    return position
```

```
william_test = ord('w')*26**6 + ord('i')*26**5 + ord('l')*26**4 + ord('l')*26**3 + ord('
i')*26**2 + ord('a')*26**1 + ord('m')*26**0

assert william_test % 97 == first_hashcode('william', 26, 97)
```

In [4]:

```
def rolling_hashcode(text, new_symbol, base = 26, size = 97):
    """
    Rolling hash to compute all hash values

    Inputs
    ------
    text: str
        the substring
    new_symbol: str
        the rolled charactere
    base: int
        the base in our calculation
    size: int
        the hash table size or the modulus - division method

    Output
    ------
    hash value: int
        hash value of the next windows after rolling a new character
    """
    # get the hash value of the previous window
    value = first_hashcode(text, base, size)
    # first letter to be eliminated
    subtraction = ord(text[0])*base**(len(text))
    # move the whole hash value to the next order and add the new symbol at the lowest or
der
    cumulative_value = (value*base + ord(new_symbol)) % size
    # minus the beginning letter
    final_value = (cumulative_value - subtraction % size) %size

    return final_value % size

assert rolling_hashcode('william', 's', 26, 97) == first_hashcode('illiams', 26, 97)
```

In [5]:

```
def rh_get_match(x, y, k, size = 97):
    """
    Finds all common length-k substrings of x and y using rolling hashing on both strings
.

    Inputs
    ------
        - x, y: strings
        - k: int, length of substring
        - size: int, the hash table size

    Output
    ------
        - A list of tuples (i, j) where x[i:i+k] = y[j:j+k]
        - dict: The hash table
    """
    # an empty list storing tuples
    results = []
    base = 26
    # cleaning strings
    x, y = cleaning(x), cleaning(y)

    # get an empty hash table
    hashtable = {}

    for i in range(size):
        hashtable[i] = []
```

```
    # edge case
    if len(x) < k or len(y) < k:
        return None

    # compute hash value for the first substring
    x_string = x[:k]
    x_pos = first_hashcode(x_string, base, size)

    # loop over the entire x-string
    for i in range(len(x) - k + 1):

        # put it to its position in the hash table
        if not hashtable[x_pos]:
            hashtable[x_pos] = [(x_string, i)]

        # if there is collision, use chaining
        else:
            hashtable[x_pos].append((x_string, i))

        # move to the next substring and still stay inside the loop
        if i < len(x) - k:
            x_pos = rolling_hashcode(x_string, x[i+k], base, size)
            x_string = x_string[1:] + x[i+k]

    # compute hash value for the first y-substring
    y_string = y[:k]
    y_pos = first_hashcode(y_string, base, size)

    # loop over the entire y-string
    for j in range(len(y) - k + 1):

        # if at that position, there are x-substrings
        if hashtable[y_pos]:
            # compare y-substring with the same hash value with x-substrings there
            for candidate in hashtable[y_pos]:
                if candidate[0] == y_string:
                    # add to the result list
                    results.append((candidate[1], j))
        # move to the next y-substring
        if j < len(y) - k:
            y_pos = rolling_hashcode(y_string, y[j+k], base, size)
            y_string = y_string[1:] + y[j+k]

    # return the hash table for future examination and the list of tuples
    return hashtable, results

# Test case 1: simple and short
x = 'Today is Monday'
y = 'day'

# with small string, k should be small to engage more closely with inputs
# i also reduce the hash table size because we do not need much space for small strings
assert rh_get_match(x, y, 3, 15)[1] == [(2, 0), (10, 0)]
```

In [6]:

```
# Test case 2: there is little effort in paraphrasing

x = 'This is an amazing HC to contrast the performance of multiple algorithms by experime
ntal analysis, specifically #ComputationalCritique. \
  With the randoization of the inputs at different lengths, we can reduce the likelihood
of worst case, thus increasing the validity of our comparison \
  between the runtime of distinct algorithms. We need the best way to store results of ru
ntime with insanely large inputs and portray the scaling behaviour \
  in running time of algorithms. A data visualization would be most suitable to communica
te this message to the audience. This not only simplifies nuances \
  in the scaling behaviour of the algorithms with larger inputs but also captures an intu
itive understanding for readers so that audience can easily understand \
  which algorithm works faster in real life. '

y = 'This is a very helpful HC in contrasting the performance between different algorithm
```

```
s by experimental results, specifically #ComputationalCritique. \
  With randomized inputs of different lengths, we can eliminate weird low likelihood of w
orst case or best case scenarios, thus increasing the validity of \
  our comparison between the runtime of different algorithms. We need the best way to acc
ommodate results of the runtime with insanely large inputs and portray the \
  scaling behaviour in the runtime, a data visualization would be best to communicate thi
s idea to the audience. This not only simplifies different nuances \
  in the scaling behaviour of the algorithm when inputs get bigger but also captures an i
ntuitive understanding for readers so that anyone looking at can \
  understand which algorithm is faster or in real life, when these algorithms deal with t
housands of data points, how the runtime growth can become.'

# with longer strings like paragraphs, I set k to be 15 which involves around 2-3 words
# this makes more sense because in two paragraphs, we can easily find similar 3 contiguou
s characters, yet not
# correctly identify plagiarism. instead, 2-3 words can imply the same meaning.
rh_get_match(x, y, 15, 97)[1]
```

Out[6]:

```
[(51, 63),
 (52, 64),
 (53, 65),
 (54, 66),
 (55, 67),
 (56, 68),
 (57, 69),
 (58, 70),
 (59, 71),
 (60, 72),
 (82, 93),
 (83, 94),
 (84, 95),
 (85, 96),
 (86, 97),
 (87, 98),
 (88, 99),
 (89, 100),
 (90, 101),
 (91, 102),
 (92, 103),
 (93, 104),
 (94, 105),
 (95, 106),
 (96, 107),
 (97, 108),
 (98, 109),
 (99, 110),
 (100, 111),
 (101, 112),
 (102, 113),
 (103, 114),
 (104, 115),
 (105, 116),
 (148, 149),
 (149, 150),
 (150, 151),
 (151, 152),
 (152, 153),
 (153, 154),
 (154, 155),
 (178, 187),
 (179, 188),
 (180, 189),
 (181, 190),
 (182, 191),
 (183, 192),
 (184, 193),
 (199, 227),
 (200, 228),
 (201, 229),
 (202, 230),
 (203, 231)
```

(203, 231),
(204, 232),
(205, 233),
(206, 234),
(207, 235),
(208, 236),
(209, 237),
(210, 238),
(211, 239),
(212, 240),
(213, 241),
(214, 242),
(215, 243),
(216, 244),
(217, 245),
(218, 246),
(219, 247),
(220, 248),
(221, 249),
(222, 250),
(223, 251),
(224, 252),
(225, 253),
(226, 254),
(227, 255),
(228, 256),
(229, 257),
(230, 258),
(231, 259),
(232, 260),
(233, 261),
(234, 262),
(235, 263),
(236, 264),
(237, 265),
(238, 266),
(239, 267),
(240, 268),
(241, 269),
(242, 270),
(243, 271),
(244, 272),
(245, 273),
(265, 294),
(266, 295),
(267, 296),
(268, 297),
(269, 298),
(270, 299),
(271, 300),
(272, 301),
(273, 302),
(274, 303),
(275, 304),
(276, 305),
(277, 306),
(278, 307),
(279, 308),
(308, 346),
(309, 347),
(310, 348),
(311, 349),
(312, 350),
(313, 351),
(314, 352),
(315, 353),
(316, 354),
(317, 355),
(318, 356),
(319, 357),
(320, 358),
(321, 359),
(322, 360)

(322, 360),
(323, 361),
(324, 362),
(325, 363),
(326, 364),
(327, 365),
(328, 366),
(329, 367),
(330, 368),
(331, 369),
(332, 370),
(333, 371),
(334, 372),
(335, 373),
(336, 374),
(337, 375),
(338, 376),
(339, 377),
(340, 378),
(341, 379),
(342, 380),
(343, 381),
(344, 382),
(345, 383),
(346, 384),
(347, 385),
(348, 386),
(496, 386),
(349, 387),
(497, 387),
(350, 388),
(498, 388),
(351, 389),
(499, 389),
(352, 390),
(500, 390),
(353, 391),
(354, 392),
(392, 417),
(393, 418),
(394, 419),
(395, 420),
(396, 421),
(397, 422),
(398, 423),
(399, 424),
(400, 425),
(401, 426),
(402, 427),
(429, 446),
(430, 447),
(431, 448),
(453, 467),
(454, 468),
(455, 469),
(456, 470),
(457, 471),
(458, 472),
(459, 473),
(460, 474),
(461, 475),
(462, 476),
(463, 477),
(464, 478),
(465, 479),
(466, 480),
(467, 481),
(468, 482),
(469, 483),
(470, 484),
(471, 485),
(472, 486),
(487, 510),

(487, 510),
(488, 511),
(489, 512),
(490, 513),
(491, 514),
(492, 515),
(493, 516),
(494, 517),
(495, 518),
(348, 519),
(496, 519),
(349, 520),
(497, 520),
(350, 521),
(498, 521),
(351, 522),
(499, 522),
(352, 523),
(500, 523),
(501, 524),
(502, 525),
(503, 526),
(504, 527),
(505, 528),
(506, 529),
(507, 530),
(508, 531),
(509, 532),
(510, 533),
(511, 534),
(512, 535),
(513, 536),
(514, 537),
(546, 571),
(547, 572),
(548, 573),
(549, 574),
(550, 575),
(551, 576),
(552, 577),
(553, 578),
(554, 579),
(555, 580),
(556, 581),
(557, 582),
(558, 583),
(559, 584),
(560, 585),
(561, 586),
(562, 587),
(563, 588),
(564, 589),
(565, 590),
(566, 591),
(567, 592),
(568, 593),
(569, 594),
(570, 595),
(571, 596),
(572, 597),
(573, 598),
(574, 599),
(575, 600),
(576, 601),
(577, 602),
(578, 603),
(579, 604),
(580, 605),
(581, 606),
(582, 607),
(583, 608),
(584, 609),
(585, 610),

```
 (586, 611),
 (587, 612),
 (618, 644),
 (619, 645),
 (620, 646),
 (621, 647),
 (622, 648),
 (623, 649),
 (624, 650),
 (625, 651),
 (626, 652),
 (627, 653)]
```

In [7]:

```
# Test case 3: The second string learns content from the first one and rewrites everythin
g
x = 'Market failure refers to inefficiency in the production and distribution of goods an
d services. \
    To specify, the allocation of products fails to meet the demand and affordability of
consumers \
    nor cause consumers to be better off, which means the problem of availability, satisf
action, \
    and cost-effectiveness. Due to asymmetric information between physicians and patients
, externalities \
    from producing drugs or taking communicable medicines, and market power derived from
patents, \
    the pharmaceutical industry, with sharp differences from neoclassical theory, is pron
e to inequity \
    and inability to guarantee public health and invites crucial solutions, namely govern
ment regulations \
    and subsidies'

y = 'When it comes to market failure, we mean sluggish production and allocation of resou
rces to the society. \
To elaborate, consumers either fail to pay for the goods they need or cannot reach their
desired products \
due to scarcity or geographical constraints. As a result, the pleasure as well as welfare
of the public is \
limited. Reasons for this phenomenon include informational assymetry or externalities tha
t influence the price \
determination. The sector bearing typical qualities of room for market failure is pharmac
eutical industry where \
imbalance in knowledge between patients and physicians happen. These implications are del
eterious due to \
social inequality and health risks, thus calling for appropriate governmental interventio
n'


rh_get_match(x, y, 15, 97)[1]
```

Out[7]:

```
[(405, 424),
 (406, 425),
 (407, 426),
 (408, 427),
 (409, 428),
 (410, 429),
 (411, 430),
 (412, 431),
 (413, 432)]
```

Evaluation:

The code works pretty well. For the first test case, it prints out the correct result as expected in the prompt. For the second and third case, we can see that plagiarism appears more in the second test case while in the third one, the writer tries to get the core idea and develop with his own writing style. As a result, results in two test cases differ considerably.

Experiments to try out different prime numbers for our hash table size:

As presented above, the divisor should be prime number and the hash table size because this will:

1. **Ensure all substrings' hash values fall into positions within the hash table size**
2. **Minimize collisions**

Therefore, I need to have more functions to print out a list of prime numbers within a given range and to calculate metrics showcasing the collision phenomenon in our algorithm.

In [8]:

```
def isPrime(number):
    """
    Check if a number if a prime number

    Input
    -----
    number: the number

    Output
    ------
    bool: return True if the number is a prime number, False otherwise
    """
    # we only have to examine from 2 to the square root of the number
    for i in range(2, int(number**0.5) + 1):
        # if it is divisible by a number within that range, its not a prime
        if number % i == 0:
            return False
    return True


def prime_numbers(lower_bound, upper_bound):
    """
    Compute a list of prime numbers within a range

    Inputs
    ------
    - lower_bound, upper_bound: int
        The two endpoints of a range

    Output
    ------
    list: a list of prime numbers

    """
    lst = []
    for num in range(lower_bound, upper_bound+1):
        # use the above function
        if isPrime(num):
            lst.append(num)
    return lst

assert prime_numbers(2, 100) == [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,7
1,73,79,83,89,97]
```

In [9]:

```
def collisions(hashtable):
    """
    Compute the collision rate and average capacity in one slot of the hash table

    Input:
        - dict: The hash table computed above when comparing two strings for plagiarism d
etection

    Outputs:
        - tuple: (a, b)
        a, b: int - the average capacity and collision rate
```

```
    """
    # initialize
    filled_slot = 0
    num_occupied = 0
    overfilled_slot = 0
    # loop through the hash table
    for i in hashtable:
        # nonempty slots
        if len(hashtable[i]) >= 1:
            # sum the total substrings
            num_occupied += len(hashtable[i])
            # calculate the number of nonempty slots
            filled_slot += 1

        # if there is collision
        if len(hashtable[i]) > 1:
            # calculate the number of overfilled slots
            overfilled_slot += 1



    average_capacity = num_occupied / filled_slot
    collision_rate = overfilled_slot / filled_slot

    return (round(average_capacity, 2), round(collision_rate, 2))

x = 'Today is Monday'
y = 'day'
# try out printing the collision metrics
collisions(rh_get_match(x, y, 3, 15)[0])
```

Out[9]:

```
(1.57, 0.57)
```

In [10]:

```
x = 'This is an amazing HC to contrast the performance of multiple algorithms by experime
ntal analysis, specifically #ComputationalCritique. \
  With the randoization of the inputs at different lengths, we can reduce the likelihood
of worst case, thus increasing the validity of our comparison \
  between the runtime of distinct algorithms. We need the best way to store results of ru
ntime with insanely large inputs and portray the scaling behaviour \
  in running time of algorithms. A data visualization would be most suitable to communica
te this message to the audience. This not only simplifies nuances \
  in the scaling behaviour of the algorithms with larger inputs but also captures an intu
itive understanding for readers so that audience can easily understand \
  which algorithm works faster in real life. '

y = 'This is a very helpful HC in contrasting the performance between different algorithm
s by experimental results, specifically #ComputationalCritique. \
  With randomized inputs of different lengths, we can eliminate weird low likelihood of w
orst case or best case scenarios, thus increasing the validity of \
  our comparison between the runtime of different algorithms. We need the best way to acc
ommodate results of the runtime with insanely large inputs and portray the \
  scaling behaviour in the runtime, a data visualization would be best to communicate thi
s idea to the audience. This not only simplifies different nuances \
  in the scaling behaviour of the algorithm when inputs get bigger but also captures an i
ntuitive understanding for readers so that anyone looking at can \
  understand which algorithm is faster or in real life, when these algorithms deal with t
housands of data points, how the runtime growth can become.'


collisions(rh_get_match(x, y, 15, 97)[0])
```

Out[10]:

```
(6.69, 0.99)
```

In [11]:

```
x = 'Market failure refers to inefficiency in the production and distribution of goods an
```

```
d services. \
    To specify, the allocation of products fails to meet the demand and affordability of
consumers \
    nor cause consumers to be better off, which means the problem of availability, satisf
action, \
    and cost-effectiveness. Due to asymmetric information between physicians and patients
, externalities \
    from producing drugs or taking communicable medicines, and market power derived from
patents, \
    the pharmaceutical industry, with sharp differences from neoclassical theory, is pron
e to inequity \
    and inability to guarantee public health and invites crucial solutions, namely govern
ment regulations \
    and subsidies'

y = 'When it comes to market failure, we mean sluggish production and allocation of resou
rces to the society. \
To elaborate, consumers either fail to pay for the goods they need or cannot reach their
desired products \
due to scarcity or geographical constraints. As a result, the pleasure as well as welfare
of the public is \
limited. Reasons for this phenomenon include informational assymetry or externalities tha
t influence the price \
determination. The sector bearing typical qualities of room for market failure is pharmac
eutical industry where \
imbalance in knowledge between patients and physicians happen. These implications are del
eterious due to \
social inequality and health risks, thus calling for appropriate governmental interventio
n'

collisions(rh_get_match(x, y, 15, 97)[0])
```

Out[11]:

```
(5.9, 0.96)
```

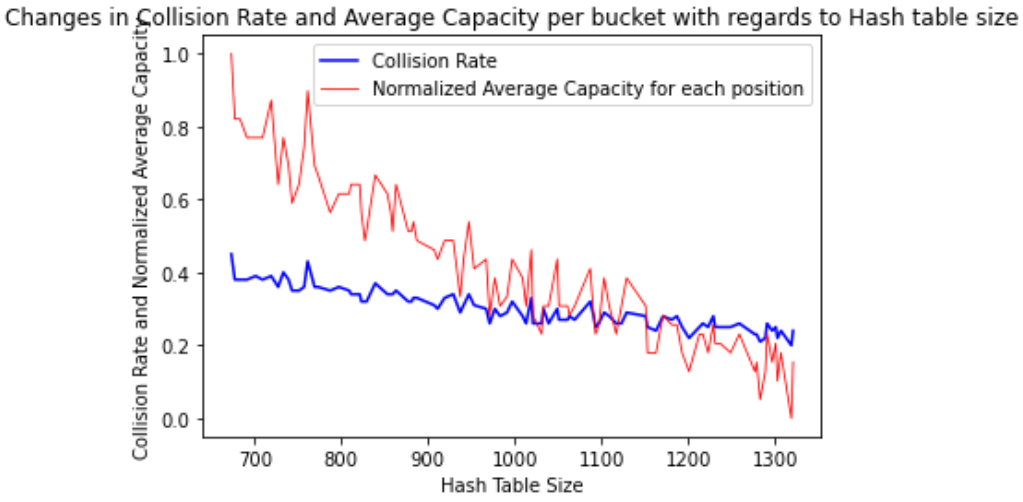
As can be observed from the collision rate and average capacity of each slot, the apropriate hash table size seems dependent on the length of inputted strings. In the first test case, with a larger hash table size than string lengths, the collision rate is pretty small. In contrast, in the other two test cases, the collision rate nearly reaches 1, which means almost all buckets in the hash table contain more than one substring. And with more plagiarism in the second test case, each bucket stores more substrings than in the third test case

With these insights from the analysis, I would love to conduct more experiments with different prime numbers - hash table sizes - on the second test case because with greater plagiarism, more collisions are likely.

In [12]:

```
import matplotlib.pyplot as plt
import numpy as np


x = 'This is an amazing HC to contrast the performance of multiple algorithms by experime
ntal analysis, specifically #ComputationalCritique. \
  With the randoization of the inputs at different lengths, we can reduce the likelihood
of worst case, thus increasing the validity of our comparison \
  between the runtime of distinct algorithms. We need the best way to store results of ru
ntime with insanely large inputs and portray the scaling behaviour \
  in running time of algorithms. A data visualization would be most suitable to communica
te this message to the audience. This not only simplifies nuances \
  in the scaling behaviour of the algorithms with larger inputs but also captures an intu
itive understanding for readers so that audience can easily understand \
  which algorithm works faster in real life. '

y = 'This is a very helpful HC in contrasting the performance between different algorithm
s by experimental results, specifically #ComputationalCritique. \
  With randomized inputs of different lengths, we can eliminate weird low likelihood of w
orst case or best case scenarios, thus increasing the validity of \
  our comparison between the runtime of different algorithms. We need the best way to acc
ommodate results of the runtime with insanely large inputs and portray the \
  scaling behaviour in the runtime, a data visualization would be best to communicate thi
```

```
s idea to the audience. This not only simplifies different nuances \
  in the scaling behaviour of the algorithm when inputs get bigger but also captures an i
ntuitive understanding for readers so that anyone looking at can \
  understand which algorithm is faster or in real life, when these algorithms deal with t
housands of data points, how the runtime growth can become.'


# get all candidates of prime numbers to test out
# lower bound is the length of the string to ensure low collision rate
# to avoid taking up too much space, i put the upper bound to be double the string length
potential_size = prime_numbers(len(cleaning(x)), len(cleaning(x)) * 2)

collision = []
avg_capacity = []
optimal_size = []
for size in potential_size:
    # store all collision rates and average capacity of buckets into two lists

    # get the first thing in the tuple of rh_get_match - the hash table
    avg_capacity.append(collisions(rh_get_match(x, y, 15, size)[0])[0])
    collision.append(collisions(rh_get_match(x, y, 15, size)[0])[1])

    # because we examine string matching within the substring's position, we care about l
ow average capacity in a slot
    if collisions(rh_get_match(x, y, 15, size)[0])[0] <= 1.25:
        optimal_size.append(size)

# normalize the average capacity in each bucket to be within the range from 0 to 1
avg_capacity = np.array(avg_capacity)
normalized_avg_capacity = (avg_capacity - min(avg_capacity)) / (max(avg_capacity) - min(
avg_capacity))

plt.plot(potential_size, collision, label = 'Collision Rate', color = 'blue')
plt.plot(potential_size, normalized_avg_capacity, label = 'Normalized Average Capacity fo
r each position', color = 'red', linewidth = 0.8)
plt.xlabel('Hash Table Size')
plt.ylabel('Collision Rate and Normalized Average Capacity')
plt.title('Changes in Collision Rate and Average Capacity per bucket with regards to Hash
table size')
plt.legend()
plt.show()
print("Length of cleaned inputted string", len(cleaning(x)))
print("Potential sizes:", optimal_size)
```




```
Length of cleaned inputted string 663
Potential sizes: [1283, 1319]
```

From the experiment above, we can clearly see that both metrics reduce as the hash table size grows. This is very good for string matching. Furthermore, we can see that there are two potential numbers of q we can choose - 1283 and 1319. I will choose to take 1319 to optimize the chance of avoiding collisions in my hash table

In [13]:

```
x = 'This is an amazing HC to contrast the performance of multiple algorithms by experime
```

```
x = 'This is an amazing HC to conclude the performance of multiple algorithms by experime
ntal analysis, specifically #ComputationalCritique. \
  With the randoization of the inputs at different lengths, we can reduce the likelihood
of worst case, thus increasing the validity of our comparison \
  between the runtime of distinct algorithms. We need the best way to store results of ru
ntime with insanely large inputs and portray the scaling behaviour \
  in running time of algorithms. A data visualization would be most suitable to communica
te this message to the audience. This not only simplifies nuances \
  in the scaling behaviour of the algorithms with larger inputs but also captures an intu
itive understanding for readers so that audience can easily understand \
  which algorithm works faster in real life. '

y = 'This is a very helpful HC in contrasting the performance between different algorithm
s by experimental results, specifically #ComputationalCritique. \
  With randomized inputs of different lengths, we can eliminate weird low likelihood of w
orst case or best case scenarios, thus increasing the validity of \
  our comparison between the runtime of different algorithms. We need the best way to acc
ommodate results of the runtime with insanely large inputs and portray the \
  scaling behaviour in the runtime, a data visualization would be best to communicate thi
s idea to the audience. This not only simplifies different nuances \
  in the scaling behaviour of the algorithm when inputs get bigger but also captures an i
ntuitive understanding for readers so that anyone looking at can \
  understand which algorithm is faster or in real life, when these algorithms deal with t
housands of data points, how the runtime growth can become.'


collisions(rh_get_match(x, y, 15, 1319)[0])
```

Out[13]:

```
(1.22, 0.2)
```

In [14]:

```
x = 'Market failure refers to inefficiency in the production and distribution of goods an
d services. \
    To specify, the allocation of products fails to meet the demand and affordability of
consumers \
    nor cause consumers to be better off, which means the problem of availability, satisf
action, \
    and cost-effectiveness. Due to asymmetric information between physicians and patients
, externalities \
    from producing drugs or taking communicable medicines, and market power derived from
patents, \
    the pharmaceutical industry, with sharp differences from neoclassical theory, is pron
e to inequity \
    and inability to guarantee public health and invites crucial solutions, namely govern
ment regulations \
    and subsidies'

y = 'When it comes to market failure, we mean sluggish production and allocation of resou
rces to the society. \
To elaborate, consumers either fail to pay for the goods they need or cannot reach their
desired products \
due to scarcity or geographical constraints. As a result, the pleasure as well as welfare
of the public is \
limited. Reasons for this phenomenon include informational assymetry or externalities tha
t influence the price \
determination. The sector bearing typical qualities of room for market failure is pharmac
eutical industry where \
imbalance in knowledge between patients and physicians happen. These implications are del
eterious due to \
social inequality and health risks, thus calling for appropriate governmental interventio
n'

collisions(rh_get_match(x, y, 15, 1319)[0])
```

Out[14]:

```
(1.26, 0.2)
```

After trying out with the same test cases, we can see that the collision rate drops dramatically. Likewise, a bucket in the table will now averagely contain fewer substrings

bucket in the table will now averagely contain fewer substrings.

Question 2: Regular Hashing

My approach:

Before, we applied rolling hashing with the division method which is simple to implement by only taking a remainder in one calculation. We also explored that if we increased the divisor, aka the hash table size, to a bigger prime number than the inputted string length, the collision rate will reduce. However, a really big hash table size can consume too much unnecessary space and we are now interested in complicating our hash function to minimize collisions right in our calculation. Therefore, we are facing a trade off here. We want to save space consumption, yet will have to sacrifice a simplified hash function and waste more time computing hash vaues for substrings.

Reflection on what makes a good hash function:

With close engagement and multiple experiments on rolling hashing, I derive that a good hash function should first effectively compute the values for substrings and then distribute substrings evenly or uniformly to different positions in the hash table. With uniform distribution, collisions will rarely happen, which speeds up our process of searching for the position and examine string matching.

Therefore, in this second attempt, I intend to make only changes in the hash functions compared to the rolling hashing. First, for the hash value calculation, I would love to apply multiplication method thanks to external research (Cormen et al. Section 11.3, 1989). From our implementation in the first question, I derive that the more random our calculation of hash values becomes, the fewer collision will happen because same hash values will now only happen to similar strings. Therefore, I will not simply take a modulus now but will multiply with a real number ranging from 0 to 1. I consulted in the textbook and realized that 0.618 is an effective number. I made a few modifications to the first_hashcode function by removing the modulus calculation. Furthermore, I time this with 0.618 to get an interesting decimal number. Then, I will get only the decimal part to maximize differences, time this with the hash table size which will be discussed later and round this to an integer. This complicated calculation is supposed to reduce collisions and improve on the runtime of the algorithm. For the rest, I will keep the same with rolling hashing approach because the mechanism of two algorithms differs only in the calculation of hash values to put substrings into positions more effectively.

I will still apply dictionary and list data structures because of their advantages over inserting, searching and extracting information. This does not change much from the rolling hashing implementation.

In [15]:

```
# In the division method, with a simple test case of "Today is Monday" and "day", there c
an still be
# spurious hits even when substrings are not similar. For this reason, we hope to minimiz
e these regrettable
# collisions with the multiplication method.

assert first_hashcode('ayi') == first_hashcode('mon')
```

In [16]:

```
import math
def hash_function(text, base = 26, size = 1319):
    """
    Compute the hash value of all windows with base 26 and no division method

    Input
    -----
    str
        the substring

    Output
    ------
    int
        the hash value
```

```
    """

    value = 0
    for i in range(len(text)):
        value += ord(text[-i-1])*(base**i)
    # Cormen et al. Section 11.3
    a = (math.sqrt(5) - 1)/2

    position = math.floor(size * ((value * a) % 1 ))
    return position

a = (math.sqrt(5) - 1)/2

old_hash_value = ord('w')*26**6 + ord('i')*26**5 + ord('l')*26**4 + ord('l')*26**3 + ord
('i')*26**2 + ord('a')*26**1 + ord('m')*26**0

william = 1319 * ((old_hash_value * a) % 1)

assert hash_function('william') == math.floor(william)
```

In [17]:

```
def regular_get_match(x, y, k, size = 1319):
    """
    Finds all common length-k substrings of x and y
    NOT using rolling hashing on both strings.

    Inputs
    ------
    str
        x, y: strings to detect plagiarism

    int
        k: the length of substrings

    Output
    ------
    a tuple of
        - the hash table
        - a list of tuples (i, j) where x[i:i+j] ==  y[j:j+k]
    """
    # initialize
    base = 26
    results = []
    x, y = cleaning(x), cleaning(y)

    # get an empty hash table
    hashmap = {}

    for i in range(size):
        hashmap[i] = []

    # edge case
    if len(x) < k or len(y) < k:
        return None
    # because we dont use rolling hashing, we put things into a loop
    for i in range(len(x) - k + 1):
        # get a substring and compute its position - hash value
        x_string = x[i:i+k]
        x_pos = hash_function(x_string, base, size)

        # put it into the right position
        if hashmap[x_pos] == None:
            hashmap[x_pos] = [(x_string, i)]

        # use chaining to fix collisions
        else:
            hashmap[x_pos].append((x_string, i))

    # loop over the y-string
    for j in range(len(y) - k + 1):
```

```
        # get a substring and compute its position - hash value
        y_string = y[j:j+k]
        y_pos = hash_function(y_string, base, size)

        # at the y-substring's position, if there are x-substrings, compare!
        if hashmap[y_pos]:
            for candidate in hashmap[y_pos]:
                # if there is a match, add it and its index to the result
                if candidate[0] == y_string:
                    results.append((candidate[1], j))
    # return the table and the list of tuples
    return hashmap, results

# test case 1: simple and short
x = 'Today is Monday'
y = 'day'

# with small string, k should be small to engage more closely with inputs
# i also reduce the hash table size because we do not need much space for small strings
assert regular_get_match(x, y, 3)[1] == [(2, 0), (10, 0)]
```

In [18]:

```
# Test case 2: there is little effort in paraphrasing
x = 'This is an amazing HC to contrast the performance of multiple algorithms by experime
ntal analysis, specifically #ComputationalCritique. \
  With the randoization of the inputs at different lengths, we can reduce the likelihood
of worst case, thus increasing the validity of our comparison \
  between the runtime of distinct algorithms. We need the best way to store results of ru
ntime with insanely large inputs and portray the scaling behaviour \
  in running time of algorithms. A data visualization would be most suitable to communica
te this message to the audience. This not only simplifies nuances \
  in the scaling behaviour of the algorithms with larger inputs but also captures an intu
itive understanding for readers so that audience can easily understand \
  which algorithm works faster in real life. '

y = 'This is a very helpful HC in contrasting the performance between different algorithm
s by experimental results, specifically #ComputationalCritique. \
  With randomized inputs of different lengths, we can eliminate weird low likelihood of w
orst case or best case scenarios, thus increasing the validity of \
  our comparison between the runtime of different algorithms. We need the best way to acc
ommodate results of the runtime with insanely large inputs and portray the \
  scaling behaviour in the runtime, a data visualization would be best to communicate thi
s idea to the audience. This not only simplifies different nuances \
  in the scaling behaviour of the algorithm when inputs get bigger but also captures an i
ntuitive understanding for readers so that anyone looking at can \
  understand which algorithm is faster or in real life, when these algorithms deal with t
housands of data points, how the runtime growth can become.'

# compare the result with the rolling hashing approach
assert regular_get_match(x, y, 15)[1] == rh_get_match(x, y, 15)[1]
```

In [19]:

```
# Test case 3: The second string learns content from the first one and rewrites everythin
g
x = 'Market failure refers to inefficiency in the production and distribution of goods an
d services. \
    To specify, the allocation of products fails to meet the demand and affordability of
consumers \
    nor cause consumers to be better off, which means the problem of availability, satisf
action, \
    and cost-effectiveness. Due to asymmetric information between physicians and patients
, externalities \
    from producing drugs or taking communicable medicines, and market power derived from
patents, \
    the pharmaceutical industry, with sharp differences from neoclassical theory, is pron
e to inequity \
    and inability to guarantee public health and invites crucial solutions, namely govern
ment regulations \
```

```
    and subsidies'

y = 'When it comes to market failure, we mean sluggish production and allocation of resou
rces to the society. \
To elaborate, consumers either fail to pay for the goods they need or cannot reach their
desired products \
due to scarcity or geographical constraints. As a result, the pleasure as well as welfare
of the public is \
limited. Reasons for this phenomenon include informational assymetry or externalities tha
t influence the price \
determination. The sector bearing typical qualities of room for market failure is pharmac
eutical industry where \
imbalance in knowledge between patients and physicians happen. These implications are del
eterious due to \
social inequality and health risks, thus calling for appropriate governmental interventio
n'

# compare the result with the rolling hashing approach
assert rh_get_match(x, y, 15)[1] == regular_get_match(x, y, 15)[1]
```

I will use the same three test cases above that check my rolling hashing algorithm to compare results with regular hashing algorithm. It seems that both algorithms return the same result. Therefore, I can at least say that the regular hashing algorithm up to now works effectively.

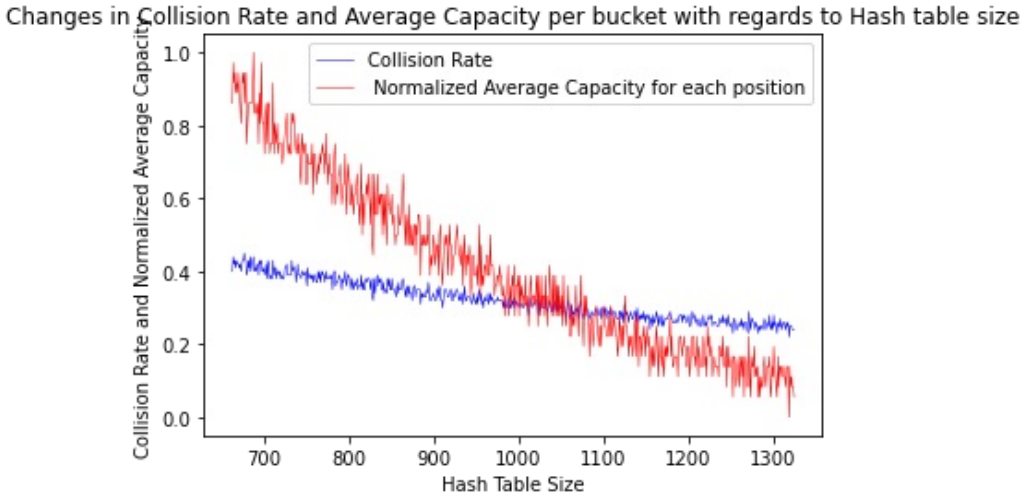
Run experiments to choose hash table size and support the choice for hash function

As presented above, the hash table size will mainly depend on the length of the inputted strings. Therefore, I will now try to experiment on one specific case and one general, randomly generated case to visualize the desirable relationship between the text length and the hash table size

In [20]:

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

letters = string.ascii_lowercase


collision_regular = []
avg_capacity_regular = []
optimal_size = []
x = 'This is an amazing HC to contrast the performance of multiple algorithms by experime
ntal analysis, specifically #ComputationalCritique. \
  With the randoization of the inputs at different lengths, we can reduce the likelihood
of worst case, thus increasing the validity of our comparison \
  between the runtime of distinct algorithms. We need the best way to store results of ru
ntime with insanely large inputs and portray the scaling behaviour \
  in running time of algorithms. A data visualization would be most suitable to communica
te this message to the audience. This not only simplifies nuances \
  in the scaling behaviour of the algorithms with larger inputs but also captures an intu
itive understanding for readers so that audience can easily understand \
  which algorithm works faster in real life. '

y = 'This is a very helpful HC in contrasting the performance between different algorithm
s by experimental results, specifically #ComputationalCritique. \
  With randomized inputs of different lengths, we can eliminate weird low likelihood of w
orst case or best case scenarios, thus increasing the validity of \
  our comparison between the runtime of different algorithms. We need the best way to acc
ommodate results of the runtime with insanely large inputs and portray the \
  scaling behaviour in the runtime, a data visualization would be best to communicate thi
s idea to the audience. This not only simplifies different nuances \
  in the scaling behaviour of the algorithm when inputs get bigger but also captures an i
ntuitive understanding for readers so that anyone looking at can \
  understand which algorithm is faster or in real life, when these algorithms deal with t
housands of data points, how the runtime growth can become.'
```

```
hashtable_sizes = [j for j in range(len(cleaning(x)), len(cleaning(x)) * 2)]

for sizes in hashtable_sizes:
    # store all collision rates and average capacity of buckets into two lists
    avg_capacity_regular.append(collisions(regular_get_match(x, y, 7, size = sizes)[0])[
0])
    collision_regular.append(collisions(regular_get_match(x, y, 7, size = sizes)[0])[1])

    # because we examine string matching within the substring's position, we care about l
ow average capacity in a slot
    if collisions(regular_get_match(x, y, 7, size = sizes)[0])[0] <= 1.3:
        optimal_size.append(sizes)

# normalize the range of 0 to 1
avg_capacity_regular = np.array(avg_capacity_regular)
normalized_avg_capacity_regular = (avg_capacity_regular - min(avg_capacity_regular)) / (
max(avg_capacity_regular) - min(avg_capacity_regular))

plt.plot(hashtable_sizes, collision_regular, label = 'Collision Rate', color = 'blue', l
inewidth = 0.5)
plt.plot(hashtable_sizes, normalized_avg_capacity_regular, label = ' Normalized Average C
apacity for each position', color = 'red', linewidth = 0.5)
plt.xlabel('Hash Table Size')
plt.ylabel('Collision Rate and Normalized Average Capacity')
plt.title('Changes in Collision Rate and Average Capacity per bucket with regards to Hash
table size')
plt.legend()
plt.show()
print("Length of cleaned inputted string", len(cleaning(x)))
print("Potential sizes:", optimal_size)
```

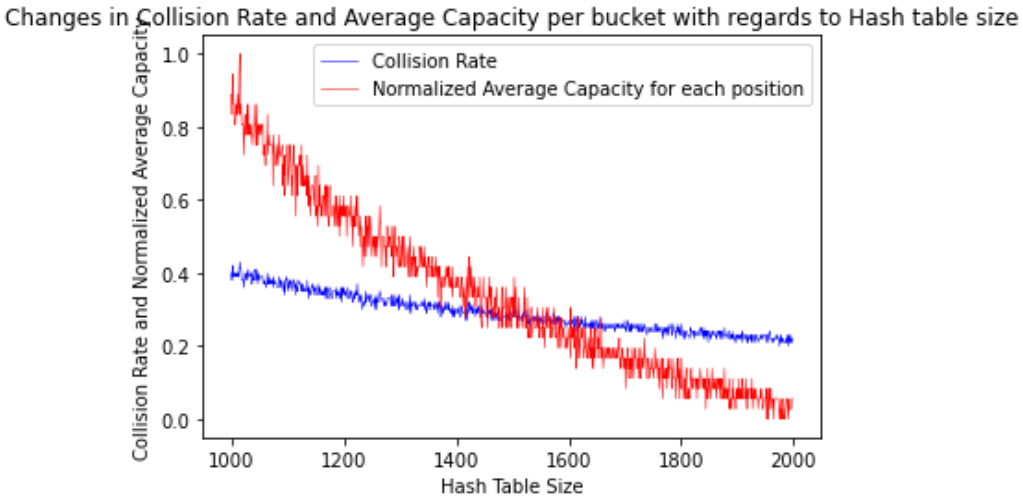



```
Length of cleaned inputted string 663
Potential sizes: [1319]
```

In [21]:

```
# k = 7
collision_regular = []
avg_capacity_regular = []
optimal_size = []
x = ''.join(random.choice(letters) for i in range(1000))
y = ''.join(random.choice(letters) for i in range(1000))

hashtable_sizes = [j for j in range(len(cleaning(x)), len(cleaning(x)) * 2)]

for sizes in hashtable_sizes:
    # store all collision rates and average capacity of buckets into two lists
    avg_capacity_regular.append(collisions(regular_get_match(x, y, 7, size = sizes)[0])[
0])
    collision_regular.append(collisions(regular_get_match(x, y, 7, size = sizes)[0])[1])

    # because we examine string matching within the substring's position, we care about l
ow average capacity in a slot
```

```
    if collisions(regular_get_match(x, y, 7, size = sizes)[0])[0] <= 1.25:
        optimal_size.append(sizes)

avg_capacity_regular = np.array(avg_capacity_regular)
normalized_avg_capacity_regular = (avg_capacity_regular - min(avg_capacity_regular)) / (
max(avg_capacity_regular) - min(avg_capacity_regular))

plt.plot(hashtable_sizes, collision_regular, label = 'Collision Rate', color = 'blue', l
inewidth = 0.5)
plt.plot(hashtable_sizes, normalized_avg_capacity_regular, label = 'Normalized Average Ca
pacity for each position', color = 'red', linewidth = 0.5)
plt.xlabel('Hash Table Size')
plt.ylabel('Collision Rate and Normalized Average Capacity')
plt.title('Changes in Collision Rate and Average Capacity per bucket with regards to Hash
table size')
plt.legend()
plt.show()
print("Length of cleaned inputted string", len(cleaning(x)))
print("Potential sizes:", optimal_size)
```




```
Length of cleaned inputted string 1000
Potential sizes: [1844, 1886, 1890, 1897, 1917, 1927, 1937, 1945, 1946, 1952, 1954, 1955,
1958, 1962, 1965, 1974, 1975, 1976, 1978, 1979, 1980, 1983, 1984, 1985, 1986, 1990, 1991,
1993, 1994, 1995, 1996]
```

From the two experiments above, the behavior is pretty the same. Both metrics reduce as the hash table size increases. For regular hashing, the hash table size is not necessarily a prime number.

As the hash table size gets larger, normalized average capacity for each bucket falls significantly while the collision rate is halved when the hash table size doubles the text length.

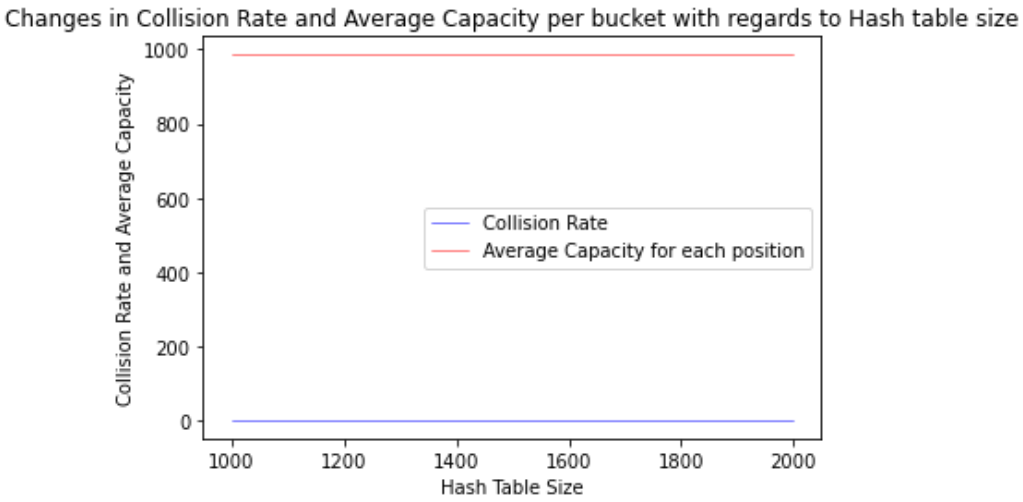
From all experiments on both algorithms, I can understand that the larger the hash table size, the fewer collisions happen. However, we need to balance with space consumption. This is when the normalization of average capacity plays out because we can see there will be a sweet spot where more space is no longer worth that reduction in collision rate. Also, because the ideal hash table size will mainly depend on the text length. I can also conclude that we should keep it only twice as large as the string length to both optimize the reduction in collision and save space consumption.

The strength of this analysis can be weakened because we only experiment on strings of 600 or 1000 length which corresponds to a short paragraph. Therefore, we might need more future experiments in the future to examine the ability of our plagiarism detector as well as the validity of choosing a hash table size twice as large as the text length.

In addition, I noticed that there is a constraint in k here to minimize collisions. To elaborate, in our hash value calculation, if k is a large number, then all substrings of the inputs actually will go to the same positions which is the first one. As a result, this points to the worst case scenario of the algorithm which we do not want to. Therefore, I have to conduct more experiments to figure out k = 7 is a better length of the pattern to work on. This constraint might become a limitation of this algorithm because 7 character length can only include around two words, which will be hard to conclude plagiarism.

In [22]:

```
# try with k = 15, we will see that all substrings go to one bucket, which is undesirable
collision_regular = []
avg_capacity_regular = []
optimal_size = []
x = ''.join(random.choice(letters) for i in range(1000))
y = ''.join(random.choice(letters) for i in range(1000))

hashtable_sizes = [j for j in range(len(cleaning(x)), len(cleaning(x)) * 2)]

for sizes in hashtable_sizes:
    # store all collision rates and average capacity of buckets into two lists
    avg_capacity_regular.append(collisions(regular_get_match(x, y, 15, size = sizes)[0])
[0])
    collision_regular.append(collisions(regular_get_match(x, y, 15, size = sizes)[0])[1]
)

    # because we examine string matching within the substring's position, we care about l
ow average capacity in a slot
    if collisions(regular_get_match(x, y, 15, size = sizes)[0])[0] <= 1.25:
        optimal_size.append(sizes)

# avg_capacity_regular = np.array(avg_capacity_regular)
# normalized_avg_capacity_regular = (avg_capacity_regular - min(avg_capacity_regular)) /
(max(avg_capacity_regular) - min(avg_capacity_regular))

plt.plot(hashtable_sizes, collision_regular, label = 'Collision Rate', color = 'blue', l
inewidth = 0.5)
plt.plot(hashtable_sizes, avg_capacity_regular, label = 'Average Capacity for each positi
on', color = 'red', linewidth = 0.5)
plt.xlabel('Hash Table Size')
plt.ylabel('Collision Rate and Average Capacity')
plt.title('Changes in Collision Rate and Average Capacity per bucket with regards to Hash
table size')
plt.legend()
plt.show()
print("Length of cleaned inputted string", len(cleaning(x)))
print("A list of potential sizes:", optimal_size)
```




```
Length of cleaned inputted string 1000
A list of potential sizes: []
```

We can see that the average capacity for each position is a line near the value of 1000 because all substrings go to one bucket only, which makes the average capacity become the capacity for this bucket only itself. Furthermore, the collision rate is close to 0, which is actually 1 because the number of overfilled slots equals that of filled slots.

In [23]:

```
# evidenced by the table

regular_get_match(x, y, 15, size = sizes)[0]
```

Out[23]:

```
{0: [('kcyjgebmkxydvoj', 0),
  ('cyjgebmkxydvojy', 1),
  ('yjgebmkxydvojyj', 2),
  ('jgebmkxydvojyjk', 3),
  ('gebmkxydvojyjkf', 4),
  ('ebmkxydvojyjkfu', 5),
  ('bmkxydvojyjkfut', 6),
  ('mkxydvojyjkfutc', 7),
  ('kxydvojyjkfutcc', 8),
  ('xydvojyjkfutccl', 9),
  ('ydvojyjkfutccly', 10),
  ('dvojyjkfutcclyc', 11),
  ('vojyjkfutcclycn', 12),
  ('ojyjkfutcclycng', 13),
  ('jyjkfutcclycngl', 14),
  ('yjkfutcclycnglf', 15),
  ('jkfutcclycnglfh', 16),
  ('kfutcclycnglfhk', 17),
  ('futcclycnglfhkb', 18),
  ('utcclycnglfhkbo', 19),
  ('tcclycnglfhkboi', 20),
  ('cclycnglfhkboin', 21),
  ('clycnglfhkboinm', 22),
  ('lycnglfhkboinmt', 23),
  ('ycnglfhkboinmto', 24),
  ('cnglfhkboinmtoi', 25),
  ('nglfhkboinmtoir', 26),
  ('glfhkboinmtoirr', 27),
  ('lfhkboinmtoirrm', 28),
  ('fhkboinmtoirrmn', 29),
  ('hkboinmtoirrmnr', 30),
  ('kboinmtoirrmnrj', 31),
  ('boinmtoirrmnrjq', 32),
  ('oinmtoirrmnrjqv', 33),
  ('inmtoirrmnrjqvu', 34),
  ('nmtoirrmnrjqvux', 35),
  ('mtoirrmnrjqvuxi', 36),
  ('toirrmnrjqvuxii', 37),
  ('oirrmnrjqvuxiiv', 38),
  ('irrmnrjqvuxiivv', 39),
  ('rrmnrjqvuxiivvt', 40),
  ('rmnrjqvuxiivvtx', 41),
  ('mnrjqvuxiivvtxk', 42),
  ('nrjqvuxiivvtxku', 43),
  ('rjqvuxiivvtxkuu', 44),
  ('jqvuxiivvtxkuuz', 45),
  ('qvuxiivvtxkuuzb', 46),
  ('vuxiivvtxkuuzba', 47),
  ('uxiivvtxkuuzbat', 48),
  ('xiivvtxkuuzbatl', 49),
  ('iivvtxkuuzbatle', 50),
  ('ivvtxkuuzbatlew', 51),
  ('vvtxkuuzbatlewx', 52),
  ('vtxkuuzbatlewxn', 53),
  ('txkuuzbatlewxnd', 54),
  ('xkuuzbatlewxndg', 55),
  ('kuuzbatlewxndgn', 56),
  ('uuzbatlewxndgnl', 57),
  ('uzbatlewxndgnll', 58),
  ('zbatlewxndgnllh', 59),
  ('batlewxndgnllhj', 60),
  ('atlewxndgnllhjf', 61),
  ('tlewxndgnllhjfs', 62),
  ('lewxndgnllhjfso', 63),
  ('ewxndgnllhjfsof', 64),
  ('wxndgnllhjfsofs', 65),
  ('xndgnllhjfsofsh', 66),
  ('ndgnllhjfsofshp', 67),
  ('dgnllhjfsofshpj', 68),
  ('gnllhjfsofshpjo', 69),
  ('nllhjfsofshpjos', 70),
```

```
('llhjfsofshpjosr', 71),
('lhjfsofshpjosrm', 72),
('hjfsofshpjosrmd', 73),
('jfsofshpjosrmdw', 74),
('fsofshpjosrmdwk', 75),
('sofshpjosrmdwkw', 76),
('ofshpjosrmdwkwy', 77),
('fshpjosrmdwkwyi', 78),
('shpjosrmdwkwyip', 79),
('hpjosrmdwkwyipx', 80),
('pjosrmdwkwyipxw', 81),
('josrmdwkwyipxwg', 82),
('osrmdwkwyipxwgl', 83),
('srmdwkwyipxwglj', 84),
('rmdwkwyipxwgljs', 85),
('mdwkwyipxwgljsg', 86),
('dwkwyipxwgljsgh', 87),
('wkwyipxwgljsghb', 88),
('kwyipxwgljsghby', 89),
('wyipxwgljsghbyr', 90),
('yipxwgljsghbyru', 91),
('ipxwgljsghbyruw', 92),
('pxwgljsghbyruwo', 93),
('xwgljsghbyruwof', 94),
('wgljsghbyruwofh', 95),
('gljsghbyruwofhi', 96),
('ljsghbyruwofhiu', 97),
('jsghbyruwofhium', 98),
('sghbyruwofhiumk', 99),
('ghbyruwofhiumkd', 100),
('hbyruwofhiumkdy', 101),
('byruwofhiumkdym', 102),
('yruwofhiumkdymu', 103),
('ruwofhiumkdymuy', 104),
('uwofhiumkdymuya', 105),
('wofhiumkdymuyat', 106),
('ofhiumkdymuyats', 107),
('fhiumkdymuyatsa', 108),
('hiumkdymuyatsaz', 109),
('iumkdymuyatsazv', 110),
('umkdymuyatsazvm', 111),
('mkdymuyatsazvmu', 112),
('kdymuyatsazvmup', 113),
('dymuyatsazvmupi', 114),
('ymuyatsazvmupis', 115),
('muyatsazvmupisd', 116),
('uyatsazvmupisdg', 117),
('yatsazvmupisdgm', 118),
('atsazvmupisdgms', 119),
('tsazvmupisdgmsn', 120),
('sazvmupisdgmsnl', 121),
('azvmupisdgmsnld', 122),
('zvmupisdgmsnlda', 123),
('vmupisdgmsnldao', 124),
('mupisdgmsnldaoi', 125),
('upisdgmsnldaoib', 126),
('pisdgmsnldaoibg', 127),
('isdgmsnldaoibga', 128),
('sdgmsnldaoibgad', 129),
('dgmsnldaoibgadi', 130),
('gmsnldaoibgadih', 131),
('msnldaoibgadihd', 132),
('snldaoibgadihde', 133),
('nldaoibgadihdea', 134),
('ldaoibgadihdeai', 135),
('daoibgadihdeaim', 136),
('aoibgadihdeaiml', 137),
('oibgadihdeaimle', 138),
('ibgadihdeaimleb', 139),
('bgadihdeaimlebv', 140),
('gadihdeaimlebvm', 141),
('adihdeaimlebvmv', 142),
```

```
('dihdeaimlebvmvb', 143),
('ihdeaimlebvmvbp', 144),
('hdeaimlebvmvbpp', 145),
('deaimlebvmvbppp', 146),
('eaimlebvmvbpppf', 147),
('aimlebvmvbpppfq', 148),
('imlebvmvbpppfqk', 149),
('mlebvmvbpppfqka', 150),
('lebvmvbpppfqkae', 151),
('ebvmvbpppfqkaeq', 152),
('bvmvbpppfqkaeqt', 153),
('vmvbpppfqkaeqty', 154),
('mvbpppfqkaeqtyn', 155),
('vbpppfqkaeqtync', 156),
('bpppfqkaeqtyncs', 157),
('pppfqkaeqtyncso', 158),
('ppfqkaeqtyncsoe', 159),
('pfqkaeqtyncsoeb', 160),
('fqkaeqtyncsoebt', 161),
('qkaeqtyncsoebtb', 162),
('kaeqtyncsoebtbi', 163),
('aeqtyncsoebtbiv', 164),
('eqtyncsoebtbivo', 165),
('qtyncsoebtbivor', 166),
('tyncsoebtbivork', 167),
('yncsoebtbivorks', 168),
('ncsoebtbivorksh', 169),
('csoebtbivorksha', 170),
('soebtbivorkshal', 171),
('oebtbivorkshalo', 172),
('ebtbivorkshaloe', 173),
('btbivorkshaloer', 174),
('tbivorkshaloerk', 175),
('bivorkshaloerkr', 176),
('ivorkshaloerkrq', 177),
('vorkshaloerkrqd', 178),
('orkshaloerkrqdf', 179),
('rkshaloerkrqdfl', 180),
('kshaloerkrqdflf', 181),
('shaloerkrqdflfg', 182),
('haloerkrqdflfgj', 183),
('aloerkrqdflfgjj', 184),
('loerkrqdflfgjjl', 185),
('oerkrqdflfgjjln', 186),
('erkrqdflfgjjlnt', 187),
('rkrqdflfgjjlntm', 188),
('krqdflfgjjlntmm', 189),
('rqdflfgjjlntmmc', 190),
('qdflfgjjlntmmca', 191),
('dflfgjjlntmmcac', 192),
('flfgjjlntmmcacv', 193),
('lfgjjlntmmcacvt', 194),
('fgjjlntmmcacvtg', 195),
('gjjlntmmcacvtge', 196),
('jjlntmmcacvtgen', 197),
('jlntmmcacvtgenc', 198),
('lntmmcacvtgenct', 199),
('ntmmcacvtgenctb', 200),
('tmmcacvtgenctbq', 201),
('mmcacvtgenctbqp', 202),
('mcacvtgenctbqpb', 203),
('cacvtgenctbqpbn', 204),
('acvtgenctbqpbnh', 205),
('cvtgenctbqpbnhk', 206),
('vtgenctbqpbnhkm', 207),
('tgenctbqpbnhkmv', 208),
('genctbqpbnhkmvr', 209),
('enctbqpbnhkmvrq', 210),
('nctbqpbnhkmvrqg', 211),
('ctbqpbnhkmvrqgx', 212),
('tbqpbnhkmvrqgxe', 213),
('bqpbnhkmvrqgxed', 214),
```

```
('qpbnhkmvrqgxeds', 215),
('pbnhkmvrqgxedso', 216),
('bnhkmvrqgxedsos', 217),
('nhkmvrqgxedsosk', 218),
('hkmvrqgxedsoskm', 219),
('kmvrqgxedsoskmw', 220),
('mvrqgxedsoskmwy', 221),
('vrqgxedsoskmwys', 222),
('rqgxedsoskmwysy', 223),
('qgxedsoskmwysyo', 224),
('gxedsoskmwysyof', 225),
('xedsoskmwysyofk', 226),
('edsoskmwysyofkk', 227),
('dsoskmwysyofkko', 228),
('soskmwysyofkkoo', 229),
('oskmwysyofkkoof', 230),
('skmwysyofkkoofc', 231),
('kmwysyofkkoofcf', 232),
('mwysyofkkoofcfq', 233),
('wysyofkkoofcfqq', 234),
('ysyofkkoofcfqqr', 235),
('syofkkoofcfqqrj', 236),
('yofkkoofcfqqrjz', 237),
('ofkkoofcfqqrjzd', 238),
('fkkoofcfqqrjzdr', 239),
('kkoofcfqqrjzdrb', 240),
('koofcfqqrjzdrbm', 241),
('oofcfqqrjzdrbmo', 242),
('ofcfqqrjzdrbmob', 243),
('fcfqqrjzdrbmobf', 244),
('cfqqrjzdrbmobfy', 245),
('fqqrjzdrbmobfyb', 246),
('qqrjzdrbmobfybm', 247),
('qrjzdrbmobfybmz', 248),
('rjzdrbmobfybmzs', 249),
('jzdrbmobfybmzsz', 250),
('zdrbmobfybmzszd', 251),
('drbmobfybmzszde', 252),
('rbmobfybmzszdet', 253),
('bmobfybmzszdetg', 254),
('mobfybmzszdetgn', 255),
('obfybmzszdetgnj', 256),
('bfybmzszdetgnjt', 257),
('fybmzszdetgnjtp', 258),
('ybmzszdetgnjtpq', 259),
('bmzszdetgnjtpqa', 260),
('mzszdetgnjtpqal', 261),
('zszdetgnjtpqald', 262),
('szdetgnjtpqaldx', 263),
('zdetgnjtpqaldxm', 264),
('detgnjtpqaldxmc', 265),
('etgnjtpqaldxmcs', 266),
('tgnjtpqaldxmcsq', 267),
('gnjtpqaldxmcsqm', 268),
('njtpqaldxmcsqmo', 269),
('jtpqaldxmcsqmoc', 270),
('tpqaldxmcsqmock', 271),
('pqaldxmcsqmockm', 272),
('qaldxmcsqmockmz', 273),
('aldxmcsqmockmzx', 274),
('ldxmcsqmockmzxk', 275),
('dxmcsqmockmzxkk', 276),
('xmcsqmockmzxkkc', 277),
('mcsqmockmzxkkcf', 278),
('csqmockmzxkkcfd', 279),
('sqmockmzxkkcfdg', 280),
('qmockmzxkkcfdga', 281),
('mockmzxkkcfdgau', 282),
('ockmzxkkcfdgauj', 283),
('ckmzxkkcfdgaujm', 284),
('kmzxkkcfdgaujmk', 285),
('mzxkkcfdgaujmkk', 286),
```

```
('zxkkcfdgaujmkks', 287),
('xkkcfdgaujmkksq', 288),
('kkcfdgaujmkksqx', 289),
('kcfdgaujmkksqxg', 290),
('cfdgaujmkksqxgz', 291),
('fdgaujmkksqxgzg', 292),
('dgaujmkksqxgzgr', 293),
('gaujmkksqxgzgri', 294),
('aujmkksqxgzgrix', 295),
('ujmkksqxgzgrixc', 296),
('jmkksqxgzgrixcf', 297),
('mkksqxgzgrixcfm', 298),
('kksqxgzgrixcfma', 299),
('ksqxgzgrixcfmaf', 300),
('sqxgzgrixcfmafv', 301),
('qxgzgrixcfmafve', 302),
('xgzgrixcfmafvek', 303),
('gzgrixcfmafvekr', 304),
('zgrixcfmafvekrd', 305),
('grixcfmafvekrdi', 306),
('rixcfmafvekrdim', 307),
('ixcfmafvekrdims', 308),
('xcfmafvekrdimsa', 309),
('cfmafvekrdimsaw', 310),
('fmafvekrdimsawg', 311),
('mafvekrdimsawgm', 312),
('afvekrdimsawgmu', 313),
('fvekrdimsawgmup', 314),
('vekrdimsawgmupf', 315),
('ekrdimsawgmupfv', 316),
('krdimsawgmupfvz', 317),
('rdimsawgmupfvzk', 318),
('dimsawgmupfvzky', 319),
('imsawgmupfvzkyr', 320),
('msawgmupfvzkyrj', 321),
('sawgmupfvzkyrjh', 322),
('awgmupfvzkyrjhc', 323),
('wgmupfvzkyrjhcc', 324),
('gmupfvzkyrjhccy', 325),
('mupfvzkyrjhccyz', 326),
('upfvzkyrjhccyzp', 327),
('pfvzkyrjhccyzph', 328),
('fvzkyrjhccyzphd', 329),
('vzkyrjhccyzphdm', 330),
('zkyrjhccyzphdmk', 331),
('kyrjhccyzphdmkf', 332),
('yrjhccyzphdmkfz', 333),
('rjhccyzphdmkfzz', 334),
('jhccyzphdmkfzzq', 335),
('hccyzphdmkfzzqy', 336),
('ccyzphdmkfzzqya', 337),
('cyzphdmkfzzqyas', 338),
('yzphdmkfzzqyasv', 339),
('zphdmkfzzqyasvr', 340),
('phdmkfzzqyasvrl', 341),
('hdmkfzzqyasvrll', 342),
('dmkfzzqyasvrllo', 343),
('mkfzzqyasvrllow', 344),
('kfzzqyasvrllowy', 345),
('fzzqyasvrllowyy', 346),
('zzqyasvrllowyyr', 347),
('zqyasvrllowyyrw', 348),
('qyasvrllowyyrws', 349),
('yasvrllowyyrwsr', 350),
('asvrllowyyrwsrh', 351),
('svrllowyyrwsrhy', 352),
('vrllowyyrwsrhyo', 353),
('rllowyyrwsrhyol', 354),
('llowyyrwsrhyolk', 355),
('lowyyrwsrhyolkd', 356),
('owyyrwsrhyolkda', 357),
('wyyrwsrhyolkday', 358),
```

```
('yyrwsrhyolkdayb', 359),
('yrwsrhyolkdaybo', 360),
('rwsrhyolkdaybou', 361),
('wsrhyolkdaybout', 362),
('srhyolkdayboutt', 363),
('rhyolkdaybouttf', 364),
('hyolkdaybouttfi', 365),
('yolkdaybouttfie', 366),
('olkdaybouttfieq', 367),
('lkdaybouttfieqw', 368),
('kdaybouttfieqwo', 369),
('daybouttfieqwot', 370),
('aybouttfieqwotw', 371),
('ybouttfieqwotwp', 372),
('bouttfieqwotwpu', 373),
('outtfieqwotwpum', 374),
('uttfieqwotwpuma', 375),
('ttfieqwotwpumaw', 376),
('tfieqwotwpumawu', 377),
('fieqwotwpumawur', 378),
('ieqwotwpumawuri', 379),
('eqwotwpumawuric', 380),
('qwotwpumawuricu', 381),
('wotwpumawuricug', 382),
('otwpumawuricuga', 383),
('twpumawuricugaq', 384),
('wpumawuricugaqa', 385),
('pumawuricugaqay', 386),
('umawuricugaqayp', 387),
('mawuricugaqaype', 388),
('awuricugaqayper', 389),
('wuricugaqayperm', 390),
('uricugaqaypermb', 391),
('ricugaqaypermbw', 392),
('icugaqaypermbwb', 393),
('cugaqaypermbwbb', 394),
('ugaqaypermbwbbe', 395),
('gaqaypermbwbbeu', 396),
('aqaypermbwbbeug', 397),
('qaypermbwbbeugh', 398),
('aypermbwbbeughv', 399),
('ypermbwbbeughvg', 400),
('permbwbbeughvgg', 401),
('ermbwbbeughvggt', 402),
('rmbwbbeughvggti', 403),
('mbwbbeughvggtii', 404),
('bwbbeughvggtiip', 405),
('wbbeughvggtiipm', 406),
('bbeughvggtiipmt', 407),
('beughvggtiipmtf', 408),
('eughvggtiipmtfk', 409),
('ughvggtiipmtfke', 410),
('ghvggtiipmtfkem', 411),
('hvggtiipmtfkemo', 412),
('vggtiipmtfkemoi', 413),
('ggtiipmtfkemoif', 414),
('gtiipmtfkemoifx', 415),
('tiipmtfkemoifxb', 416),
('iipmtfkemoifxbm', 417),
('ipmtfkemoifxbmu', 418),
('pmtfkemoifxbmum', 419),
('mtfkemoifxbmums', 420),
('tfkemoifxbmumsb', 421),
('fkemoifxbmumsbk', 422),
('kemoifxbmumsbkw', 423),
('emoifxbmumsbkwr', 424),
('moifxbmumsbkwrg', 425),
('oifxbmumsbkwrgq', 426),
('ifxbmumsbkwrgqq', 427),
('fxbmumsbkwrgqqg', 428),
('xbmumsbkwrgqqgb', 429),
('bmumsbkwrgqqgbt', 430),
```

```
('mumsbkwrgqqgbtq', 431),
('umsbkwrgqqgbtqm', 432),
('msbkwrgqqgbtqmc', 433),
('sbkwrgqqgbtqmcz', 434),
('bkwrgqqgbtqmczd', 435),
('kwrgqqgbtqmczdr', 436),
('wrgqqgbtqmczdrl', 437),
('rgqqgbtqmczdrld', 438),
('gqqgbtqmczdrldx', 439),
('qqgbtqmczdrldxt', 440),
('qgbtqmczdrldxtv', 441),
('gbtqmczdrldxtvq', 442),
('btqmczdrldxtvqt', 443),
('tqmczdrldxtvqty', 444),
('qmczdrldxtvqtyz', 445),
('mczdrldxtvqtyzz', 446),
('czdrldxtvqtyzzs', 447),
('zdrldxtvqtyzzsd', 448),
('drldxtvqtyzzsdb', 449),
('rldxtvqtyzzsdbr', 450),
('ldxtvqtyzzsdbrg', 451),
('dxtvqtyzzsdbrgf', 452),
('xtvqtyzzsdbrgfe', 453),
('tvqtyzzsdbrgfep', 454),
('vqtyzzsdbrgfepc', 455),
('qtyzzsdbrgfepcg', 456),
('tyzzsdbrgfepcgc', 457),
('yzzsdbrgfepcgcm', 458),
('zzsdbrgfepcgcmi', 459),
('zsdbrgfepcgcmif', 460),
('sdbrgfepcgcmifs', 461),
('dbrgfepcgcmifsq', 462),
('brgfepcgcmifsqh', 463),
('rgfepcgcmifsqhw', 464),
('gfepcgcmifsqhwm', 465),
('fepcgcmifsqhwmd', 466),
('epcgcmifsqhwmdj', 467),
('pcgcmifsqhwmdjz', 468),
('cgcmifsqhwmdjzj', 469),
('gcmifsqhwmdjzjd', 470),
('cmifsqhwmdjzjda', 471),
('mifsqhwmdjzjdad', 472),
('ifsqhwmdjzjdadu', 473),
('fsqhwmdjzjdaduf', 474),
('sqhwmdjzjdadufq', 475),
('qhwmdjzjdadufqn', 476),
('hwmdjzjdadufqnw', 477),
('wmdjzjdadufqnwz', 478),
('mdjzjdadufqnwzx', 479),
('djzjdadufqnwzxp', 480),
('jzjdadufqnwzxpw', 481),
('zjdadufqnwzxpwo', 482),
('jdadufqnwzxpwox', 483),
('dadufqnwzxpwoxn', 484),
('adufqnwzxpwoxnc', 485),
('dufqnwzxpwoxnch', 486),
('ufqnwzxpwoxnchl', 487),
('fqnwzxpwoxnchle', 488),
('qnwzxpwoxnchlew', 489),
('nwzxpwoxnchlewz', 490),
('wzxpwoxnchlewzk', 491),
('zxpwoxnchlewzkz', 492),
('xpwoxnchlewzkzf', 493),
('pwoxnchlewzkzfm', 494),
('woxnchlewzkzfmz', 495),
('oxnchlewzkzfmzg', 496),
('xnchlewzkzfmzgs', 497),
('nchlewzkzfmzgse', 498),
('chlewzkzfmzgsew', 499),
('hlewzkzfmzgsewn', 500),
('lewzkzfmzgsewnr', 501),
('ewzkzfmzgsewnrx', 502),
```

```
('wzkzfmzgsewnrxk', 503),
('zkzfmzgsewnrxkn', 504),
('kzfmzgsewnrxkng', 505),
('zfmzgsewnrxkngz', 506),
('fmzgsewnrxkngzi', 507),
('mzgsewnrxkngzif', 508),
('zgsewnrxkngzifd', 509),
('gsewnrxkngzifdp', 510),
('sewnrxkngzifdpa', 511),
('ewnrxkngzifdpaq', 512),
('wnrxkngzifdpaqw', 513),
('nrxkngzifdpaqwg', 514),
('rxkngzifdpaqwgt', 515),
('xkngzifdpaqwgtr', 516),
('kngzifdpaqwgtrm', 517),
('ngzifdpaqwgtrmx', 518),
('gzifdpaqwgtrmxd', 519),
('zifdpaqwgtrmxdb', 520),
('ifdpaqwgtrmxdbx', 521),
('fdpaqwgtrmxdbxz', 522),
('dpaqwgtrmxdbxzv', 523),
('paqwgtrmxdbxzvc', 524),
('aqwgtrmxdbxzvcz', 525),
('qwgtrmxdbxzvczj', 526),
('wgtrmxdbxzvczja', 527),
('gtrmxdbxzvczjaa', 528),
('trmxdbxzvczjaac', 529),
('rmxdbxzvczjaaca', 530),
('mxdbxzvczjaacaq', 531),
('xdbxzvczjaacaqx', 532),
('dbxzvczjaacaqxn', 533),
('bxzvczjaacaqxng', 534),
('xzvczjaacaqxngk', 535),
('zvczjaacaqxngkz', 536),
('vczjaacaqxngkze', 537),
('czjaacaqxngkzex', 538),
('zjaacaqxngkzexp', 539),
('jaacaqxngkzexpc', 540),
('aacaqxngkzexpcc', 541),
('acaqxngkzexpcck', 542),
('caqxngkzexpcckp', 543),
('aqxngkzexpcckpx', 544),
('qxngkzexpcckpxj', 545),
('xngkzexpcckpxjs', 546),
('ngkzexpcckpxjso', 547),
('gkzexpcckpxjsog', 548),
('kzexpcckpxjsogr', 549),
('zexpcckpxjsogre', 550),
('expcckpxjsogrem', 551),
('xpcckpxjsogremx', 552),
('pcckpxjsogremxc', 553),
('cckpxjsogremxcz', 554),
('ckpxjsogremxcza', 555),
('kpxjsogremxczaw', 556),
('pxjsogremxczaws', 557),
('xjsogremxczawsu', 558),
('jsogremxczawsun', 559),
('sogremxczawsune', 560),
('ogremxczawsunea', 561),
('gremxczawsuneac', 562),
('remxczawsuneacf', 563),
('emxczawsuneacfj', 564),
('mxczawsuneacfjo', 565),
('xczawsuneacfjop', 566),
('czawsuneacfjopp', 567),
('zawsuneacfjoppm', 568),
('awsuneacfjoppmn', 569),
('wsuneacfjoppmnd', 570),
('suneacfjoppmndb', 571),
('uneacfjoppmndbu', 572),
('neacfjoppmndbux', 573),
('eacfjoppmndbuxo', 574),
```

```
('acfjoppmndbuxow', 575),
('cfjoppmndbuxowu', 576),
('fjoppmndbuxowum', 577),
('joppmndbuxowumf', 578),
('oppmndbuxowumfc', 579),
('ppmndbuxowumfcz', 580),
('pmndbuxowumfczs', 581),
('mndbuxowumfczsx', 582),
('ndbuxowumfczsxb', 583),
('dbuxowumfczsxbq', 584),
('buxowumfczsxbqi', 585),
('uxowumfczsxbqia', 586),
('xowumfczsxbqiai', 587),
('owumfczsxbqiaie', 588),
('wumfczsxbqiaiev', 589),
('umfczsxbqiaievc', 590),
('mfczsxbqiaievcg', 591),
('fczsxbqiaievcgm', 592),
('czsxbqiaievcgms', 593),
('zsxbqiaievcgmsv', 594),
('sxbqiaievcgmsvv', 595),
('xbqiaievcgmsvvm', 596),
('bqiaievcgmsvvmq', 597),
('qiaievcgmsvvmqw', 598),
('iaievcgmsvvmqwa', 599),
('aievcgmsvvmqwah', 600),
('ievcgmsvvmqwahw', 601),
('evcgmsvvmqwahwr', 602),
('vcgmsvvmqwahwra', 603),
('cgmsvvmqwahwraz', 604),
('gmsvvmqwahwrazt', 605),
('msvvmqwahwraztt', 606),
('svvmqwahwraztto', 607),
('vvmqwahwrazttop', 608),
('vmqwahwrazttopq', 609),
('mqwahwrazttopqs', 610),
('qwahwrazttopqst', 611),
('wahwrazttopqstd', 612),
('ahwrazttopqstdu', 613),
('hwrazttopqstduf', 614),
('wrazttopqstdufy', 615),
('razttopqstdufyc', 616),
('azttopqstdufycs', 617),
('zttopqstdufycsh', 618),
('ttopqstdufycshi', 619),
('topqstdufycshig', 620),
('opqstdufycshigu', 621),
('pqstdufycshiguu', 622),
('qstdufycshiguuy', 623),
('stdufycshiguuyu', 624),
('tdufycshiguuyug', 625),
('dufycshiguuyugv', 626),
('ufycshiguuyugvs', 627),
('fycshiguuyugvsb', 628),
('ycshiguuyugvsbg', 629),
('cshiguuyugvsbgl', 630),
('shiguuyugvsbglh', 631),
('higuuyugvsbglhq', 632),
('iguuyugvsbglhqf', 633),
('guuyugvsbglhqfd', 634),
('uuyugvsbglhqfdo', 635),
('uyugvsbglhqfdok', 636),
('yugvsbglhqfdoke', 637),
('ugvsbglhqfdokeo', 638),
('gvsbglhqfdokeom', 639),
('vsbglhqfdokeomx', 640),
('sbglhqfdokeomxz', 641),
('bglhqfdokeomxzv', 642),
('glhqfdokeomxzvs', 643),
('lhqfdokeomxzvsq', 644),
('hqfdokeomxzvsqh', 645),
('qfdokeomxzvsqhb', 646),
```

```
('fdokeomxzvsqhbr', 647),
('dokeomxzvsqhbrn', 648),
('okeomxzvsqhbrnx', 649),
('keomxzvsqhbrnxl', 650),
('eomxzvsqhbrnxlf', 651),
('omxzvsqhbrnxlfv', 652),
('mxzvsqhbrnxlfvy', 653),
('xzvsqhbrnxlfvym', 654),
('zvsqhbrnxlfvymo', 655),
('vsqhbrnxlfvymov', 656),
('sqhbrnxlfvymovw', 657),
('qhbrnxlfvymovwg', 658),
('hbrnxlfvymovwgo', 659),
('brnxlfvymovwgog', 660),
('rnxlfvymovwgogm', 661),
('nxlfvymovwgogmt', 662),
('xlfvymovwgogmtj', 663),
('lfvymovwgogmtjv', 664),
('fvymovwgogmtjvi', 665),
('vymovwgogmtjviw', 666),
('ymovwgogmtjviwu', 667),
('movwgogmtjviwup', 668),
('ovwgogmtjviwupq', 669),
('vwgogmtjviwupqh', 670),
('wgogmtjviwupqhh', 671),
('gogmtjviwupqhhq', 672),
('ogmtjviwupqhhqw', 673),
('gmtjviwupqhhqwz', 674),
('mtjviwupqhhqwzz', 675),
('tjviwupqhhqwzzr', 676),
('jviwupqhhqwzzru', 677),
('viwupqhhqwzzrur', 678),
('iwupqhhqwzzruro', 679),
('wupqhhqwzzrurok', 680),
('upqhhqwzzrurokz', 681),
('pqhhqwzzrurokzt', 682),
('qhhqwzzrurokztk', 683),
('hhqwzzrurokztkm', 684),
('hqwzzrurokztkmr', 685),
('qwzzrurokztkmrs', 686),
('wzzrurokztkmrsz', 687),
('zzrurokztkmrsza', 688),
('zrurokztkmrszai', 689),
('rurokztkmrszair', 690),
('urokztkmrszairc', 691),
('rokztkmrszaircg', 692),
('okztkmrszaircgs', 693),
('kztkmrszaircgsw', 694),
('ztkmrszaircgswq', 695),
('tkmrszaircgswqs', 696),
('kmrszaircgswqsf', 697),
('mrszaircgswqsfk', 698),
('rszaircgswqsfkm', 699),
('szaircgswqsfkmr', 700),
('zaircgswqsfkmrk', 701),
('aircgswqsfkmrkm', 702),
('ircgswqsfkmrkmr', 703),
('rcgswqsfkmrkmrm', 704),
('cgswqsfkmrkmrmg', 705),
('gswqsfkmrkmrmgy', 706),
('swqsfkmrkmrmgyr', 707),
('wqsfkmrkmrmgyrg', 708),
('qsfkmrkmrmgyrgw', 709),
('sfkmrkmrmgyrgwo', 710),
('fkmrkmrmgyrgwoe', 711),
('kmrkmrmgyrgwoep', 712),
('mrkmrmgyrgwoepy', 713),
('rkmrmgyrgwoepyf', 714),
('kmrmgyrgwoepyfj', 715),
('mrmgyrgwoepyfji', 716),
('rmgyrgwoepyfjiz', 717),
('mgyrgwoepyfjizw', 718),
```

```
('gyrgwoepyfjizwq', 719),
('yrgwoepyfjizwqa', 720),
('rgwoepyfjizwqaw', 721),
('gwoepyfjizwqawc', 722),
('woepyfjizwqawcy', 723),
('oepyfjizwqawcyl', 724),
('epyfjizwqawcylp', 725),
('pyfjizwqawcylpm', 726),
('yfjizwqawcylpmd', 727),
('fjizwqawcylpmdn', 728),
('jizwqawcylpmdnn', 729),
('izwqawcylpmdnna', 730),
('zwqawcylpmdnnag', 731),
('wqawcylpmdnnagl', 732),
('qawcylpmdnnaglv', 733),
('awcylpmdnnaglvw', 734),
('wcylpmdnnaglvwl', 735),
('cylpmdnnaglvwlt', 736),
('ylpmdnnaglvwltc', 737),
('lpmdnnaglvwltco', 738),
('pmdnnaglvwltcoh', 739),
('mdnnaglvwltcohd', 740),
('dnnaglvwltcohdi', 741),
('nnaglvwltcohdiy', 742),
('naglvwltcohdiyv', 743),
('aglvwltcohdiyvr', 744),
('glvwltcohdiyvrf', 745),
('lvwltcohdiyvrfz', 746),
('vwltcohdiyvrfza', 747),
('wltcohdiyvrfzax', 748),
('ltcohdiyvrfzaxf', 749),
('tcohdiyvrfzaxfd', 750),
('cohdiyvrfzaxfdt', 751),
('ohdiyvrfzaxfdtb', 752),
('hdiyvrfzaxfdtbg', 753),
('diyvrfzaxfdtbgu', 754),
('iyvrfzaxfdtbgup', 755),
('yvrfzaxfdtbgupo', 756),
('vrfzaxfdtbgupoa', 757),
('rfzaxfdtbgupoag', 758),
('fzaxfdtbgupoagn', 759),
('zaxfdtbgupoagns', 760),
('axfdtbgupoagnsk', 761),
('xfdtbgupoagnskb', 762),
('fdtbgupoagnskbv', 763),
('dtbgupoagnskbvf', 764),
('tbgupoagnskbvfn', 765),
('bgupoagnskbvfnm', 766),
('gupoagnskbvfnmj', 767),
('upoagnskbvfnmjy', 768),
('poagnskbvfnmjyo', 769),
('oagnskbvfnmjyot', 770),
('agnskbvfnmjyotl', 771),
('gnskbvfnmjyotla', 772),
('nskbvfnmjyotlav', 773),
('skbvfnmjyotlavk', 774),
('kbvfnmjyotlavkl', 775),
('bvfnmjyotlavkln', 776),
('vfnmjyotlavklnm', 777),
('fnmjyotlavklnme', 778),
('nmjyotlavklnmef', 779),
('mjyotlavklnmefz', 780),
('jyotlavklnmefzx', 781),
('yotlavklnmefzxk', 782),
('otlavklnmefzxkb', 783),
('tlavklnmefzxkbk', 784),
('lavklnmefzxkbkn', 785),
('avklnmefzxkbknk', 786),
('vklnmefzxkbknks', 787),
('klnmefzxkbknkse', 788),
('lnmefzxkbknkseq', 789),
('nmefzxkbknkseqx', 790),
```

('mefzxkbknkseqxh', 791),
('efzxkbknkseqxhp', 792),
('fzxkbknkseqxhpk', 793),
('zxkbknkseqxhpkg', 794),
('xkbknkseqxhpkgb', 795),
('kbknkseqxhpkgbk', 796),
('bknkseqxhpkgbkv', 797),
('knkseqxhpkgbkva', 798),
('nkseqxhpkgbkvap', 799),
('kseqxhpkgbkvapm', 800),
('seqxhpkgbkvapmh', 801),
('eqxhpkgbkvapmhr', 802),
('qxhpkgbkvapmhrq', 803),
('xhpkgbkvapmhrql', 804),
('hpkgbkvapmhrqlz', 805),
('pkgbkvapmhrqlzj', 806),
('kgbkvapmhrqlzjw', 807),
('gbkvapmhrqlzjwo', 808),
('bkvapmhrqlzjwow', 809),
('kvapmhrqlzjwowb', 810),
('vapmhrqlzjwowbe', 811),
('apmhrqlzjwowbeh', 812),
('pmhrqlzjwowbehw', 813),
('mhrqlzjwowbehwk', 814),
('hrqlzjwowbehwke', 815),
('rqlzjwowbehwket', 816),
('qlzjwowbehwketq', 817),
('lzjwowbehwketqm', 818),
('zjwowbehwketqmm', 819),
('jwowbehwketqmmj', 820),
('wowbehwketqmmjd', 821),
('owbehwketqmmjdi', 822),
('wbehwketqmmjdiy', 823),
('behwketqmmjdiyo', 824),
('ehwketqmmjdiyoc', 825),
('hwketqmmjdiyocl', 826),
('wketqmmjdiyocll', 827),
('ketqmmjdiyoclle', 828),
('etqmmjdiyocllen', 829),
('tqmmjdiyocllenv', 830),
('qmmjdiyocllenvr', 831),
('mmjdiyocllenvro', 832),
('mjdiyocllenvrog', 833),
('jdiyocllenvrogh', 834),
('diyocllenvroghc', 835),
('iyocllenvroghcr', 836),
('yocllenvroghcrr', 837),
('ocllenvroghcrrg', 838),
('cllenvroghcrrgb', 839),
('llenvroghcrrgbs', 840),
('lenvroghcrrgbsg', 841),
('envroghcrrgbsgc', 842),
('nvroghcrrgbsgcw', 843),
('vroghcrrgbsgcww', 844),
('roghcrrgbsgcwwo', 845),
('oghcrrgbsgcwwof', 846),
('ghcrrgbsgcwwofs', 847),
('hcrrgbsgcwwofsj', 848),
('crrgbsgcwwofsjz', 849),
('rrgbsgcwwofsjzz', 850),
('rgbsgcwwofsjzzg', 851),
('gbsgcwwofsjzzgj', 852),
('bsgcwwofsjzzgju', 853),
('sgcwwofsjzzgjun', 854),
('gcwwofsjzzgjunf', 855),
('cwwofsjzzgjunfe', 856),
('wwofsjzzgjunfeq', 857),
('wofsjzzgjunfeqj', 858),
('ofsjzzgjunfeqjp', 859),
('fsjzzgjunfeqjpg', 860),
('sjzzgjunfeqjpgq', 861),
('jzzgjunfeqjpgqv', 862),

```
('zzgjunfeqjpgqvs', 863),
('zgjunfeqjpgqvsp', 864),
('gjunfeqjpgqvspo', 865),
('junfeqjpgqvspoz', 866),
('unfeqjpgqvspozd', 867),
('nfeqjpgqvspozdo', 868),
('feqjpgqvspozdok', 869),
('eqjpgqvspozdokk', 870),
('qjpgqvspozdokke', 871),
('jpgqvspozdokkez', 872),
('pgqvspozdokkezb', 873),
('gqvspozdokkezbb', 874),
('qvspozdokkezbbh', 875),
('vspozdokkezbbhg', 876),
('spozdokkezbbhgn', 877),
('pozdokkezbbhgne', 878),
('ozdokkezbbhgnew', 879),
('zdokkezbbhgnewe', 880),
('dokkezbbhgnewem', 881),
('okkezbbhgnewemy', 882),
('kkezbbhgnewemyc', 883),
('kezbbhgnewemycm', 884),
('ezbbhgnewemycmd', 885),
('zbbhgnewemycmdc', 886),
('bbhgnewemycmdcl', 887),
('bhgnewemycmdcla', 888),
('hgnewemycmdclad', 889),
('gnewemycmdcladw', 890),
('newemycmdcladww', 891),
('ewemycmdcladwws', 892),
('wemycmdcladwwsv', 893),
('emycmdcladwwsvd', 894),
('mycmdcladwwsvdl', 895),
('ycmdcladwwsvdlr', 896),
('cmdcladwwsvdlrx', 897),
('mdcladwwsvdlrxh', 898),
('dcladwwsvdlrxht', 899),
('cladwwsvdlrxhtp', 900),
('ladwwsvdlrxhtpf', 901),
('adwwsvdlrxhtpfr', 902),
('dwwsvdlrxhtpfrt', 903),
('wwsvdlrxhtpfrti', 904),
('wsvdlrxhtpfrtiy', 905),
('svdlrxhtpfrtiys', 906),
('vdlrxhtpfrtiysk', 907),
('dlrxhtpfrtiyskk', 908),
('lrxhtpfrtiyskki', 909),
('rxhtpfrtiyskkix', 910),
('xhtpfrtiyskkixb', 911),
('htpfrtiyskkixbp', 912),
('tpfrtiyskkixbpa', 913),
('pfrtiyskkixbpan', 914),
('frtiyskkixbpanv', 915),
('rtiyskkixbpanvi', 916),
('tiyskkixbpanvib', 917),
('iyskkixbpanvibs', 918),
('yskkixbpanvibsu', 919),
('skkixbpanvibsud', 920),
('kkixbpanvibsudr', 921),
('kixbpanvibsudrf', 922),
('ixbpanvibsudrft', 923),
('xbpanvibsudrftc', 924),
('bpanvibsudrftcx', 925),
('panvibsudrftcxz', 926),
('anvibsudrftcxzr', 927),
('nvibsudrftcxzrq', 928),
('vibsudrftcxzrqs', 929),
('ibsudrftcxzrqsa', 930),
('bsudrftcxzrqsaw', 931),
('sudrftcxzrqsawi', 932),
('udrftcxzrqsawin', 933),
('drftcxzrqsawiny', 934),
```

```
 ('rftcxzrqsawinyx', 935),
 ('ftcxzrqsawinyxl', 936),
 ('tcxzrqsawinyxlf', 937),
 ('cxzrqsawinyxlfs', 938),
 ('xzrqsawinyxlfsx', 939),
 ('zrqsawinyxlfsxa', 940),
 ('rqsawinyxlfsxav', 941),
 ('qsawinyxlfsxavq', 942),
 ('sawinyxlfsxavqp', 943),
 ('awinyxlfsxavqpo', 944),
 ('winyxlfsxavqpop', 945),
 ('inyxlfsxavqpopg', 946),
 ('nyxlfsxavqpopgd', 947),
 ('yxlfsxavqpopgdk', 948),
 ('xlfsxavqpopgdkc', 949),
 ('lfsxavqpopgdkcz', 950),
 ('fsxavqpopgdkczl', 951),
 ('sxavqpopgdkczlt', 952),
 ('xavqpopgdkczlts', 953),
 ('avqpopgdkczltss', 954),
 ('vqpopgdkczltssu', 955),
 ('qpopgdkczltssun', 956),
 ('popgdkczltssunx', 957),
 ('opgdkczltssunxd', 958),
 ('pgdkczltssunxdh', 959),
 ('gdkczltssunxdhi', 960),
 ('dkczltssunxdhiw', 961),
 ('kczltssunxdhiwy', 962),
 ('czltssunxdhiwyr', 963),
 ('zltssunxdhiwyrn', 964),
 ('ltssunxdhiwyrnh', 965),
 ('tssunxdhiwyrnhs', 966),
 ('ssunxdhiwyrnhss', 967),
 ('sunxdhiwyrnhssz', 968),
 ('unxdhiwyrnhsszp', 969),
 ('nxdhiwyrnhsszpu', 970),
 ('xdhiwyrnhsszpuj', 971),
 ('dhiwyrnhsszpuji', 972),
 ('hiwyrnhsszpujig', 973),
 ('iwyrnhsszpujigc', 974),
 ('wyrnhsszpujigcq', 975),
 ('yrnhsszpujigcqj', 976),
 ('rnhsszpujigcqjd', 977),
 ('nhsszpujigcqjdn', 978),
 ('hsszpujigcqjdnj', 979),
 ('sszpujigcqjdnjg', 980),
 ('szpujigcqjdnjgg', 981),
 ('zpujigcqjdnjggr', 982),
 ('pujigcqjdnjggri', 983),
 ('ujigcqjdnjggrij', 984),
 ('jigcqjdnjggrija', 985)],
1: [],
2: [],
3: [],
4: [],
5: [],
6: [],
7: [],
8: [],
9: [],
10: [],
11: [],
12: [],
13: [],
14: [],
15: [],
16: [],
17: [],
18: [],
19: [],
20: [],
21: [],
```

```
22: [],
23: [],
24: [],
25: [],
26: [],
27: [],
28: [],
29: [],
30: [],
31: [],
32: [],
33: [],
34: [],
35: [],
36: [],
37: [],
38: [],
39: [],
40: [],
41: [],
42: [],
43: [],
44: [],
45: [],
46: [],
47: [],
48: [],
49: [],
50: [],
51: [],
52: [],
53: [],
54: [],
55: [],
56: [],
57: [],
58: [],
59: [],
60: [],
61: [],
62: [],
63: [],
64: [],
65: [],
66: [],
67: [],
68: [],
69: [],
70: [],
71: [],
72: [],
73: [],
74: [],
75: [],
76: [],
77: [],
78: [],
79: [],
80: [],
81: [],
82: [],
83: [],
84: [],
85: [],
86: [],
87: [],
88: [],
89: [],
90: [],
91: [],
92: [],
93: [],
```

```
94: [],
95: [],
96: [],
97: [],
98: [],
99: [],
100: [],
101: [],
102: [],
103: [],
104: [],
105: [],
106: [],
107: [],
108: [],
109: [],
110: [],
111: [],
112: [],
113: [],
114: [],
115: [],
116: [],
117: [],
118: [],
119: [],
120: [],
121: [],
122: [],
123: [],
124: [],
125: [],
126: [],
127: [],
128: [],
129: [],
130: [],
131: [],
132: [],
133: [],
134: [],
135: [],
136: [],
137: [],
138: [],
139: [],
140: [],
141: [],
142: [],
143: [],
144: [],
145: [],
146: [],
147: [],
148: [],
149: [],
150: [],
151: [],
152: [],
153: [],
154: [],
155: [],
156: [],
157: [],
158: [],
159: [],
160: [],
161: [],
162: [],
163: [],
164: [],
165: [],
```

```
166: [],
167: [],
168: [],
169: [],
170: [],
171: [],
172: [],
173: [],
174: [],
175: [],
176: [],
177: [],
178: [],
179: [],
180: [],
181: [],
182: [],
183: [],
184: [],
185: [],
186: [],
187: [],
188: [],
189: [],
190: [],
191: [],
192: [],
193: [],
194: [],
195: [],
196: [],
197: [],
198: [],
199: [],
200: [],
201: [],
202: [],
203: [],
204: [],
205: [],
206: [],
207: [],
208: [],
209: [],
210: [],
211: [],
212: [],
213: [],
214: [],
215: [],
216: [],
217: [],
218: [],
219: [],
220: [],
221: [],
222: [],
223: [],
224: [],
225: [],
226: [],
227: [],
228: [],
229: [],
230: [],
231: [],
232: [],
233: [],
234: [],
235: [],
236: [],
237: [],
```

```
238: [],
239: [],
240: [],
241: [],
242: [],
243: [],
244: [],
245: [],
246: [],
247: [],
248: [],
249: [],
250: [],
251: [],
252: [],
253: [],
254: [],
255: [],
256: [],
257: [],
258: [],
259: [],
260: [],
261: [],
262: [],
263: [],
264: [],
265: [],
266: [],
267: [],
268: [],
269: [],
270: [],
271: [],
272: [],
273: [],
274: [],
275: [],
276: [],
277: [],
278: [],
279: [],
280: [],
281: [],
282: [],
283: [],
284: [],
285: [],
286: [],
287: [],
288: [],
289: [],
290: [],
291: [],
292: [],
293: [],
294: [],
295: [],
296: [],
297: [],
298: [],
299: [],
300: [],
301: [],
302: [],
303: [],
304: [],
305: [],
306: [],
307: [],
308: [],
309: [],
```

```
310: [],
311: [],
312: [],
313: [],
314: [],
315: [],
316: [],
317: [],
318: [],
319: [],
320: [],
321: [],
322: [],
323: [],
324: [],
325: [],
326: [],
327: [],
328: [],
329: [],
330: [],
331: [],
332: [],
333: [],
334: [],
335: [],
336: [],
337: [],
338: [],
339: [],
340: [],
341: [],
342: [],
343: [],
344: [],
345: [],
346: [],
347: [],
348: [],
349: [],
350: [],
351: [],
352: [],
353: [],
354: [],
355: [],
356: [],
357: [],
358: [],
359: [],
360: [],
361: [],
362: [],
363: [],
364: [],
365: [],
366: [],
367: [],
368: [],
369: [],
370: [],
371: [],
372: [],
373: [],
374: [],
375: [],
376: [],
377: [],
378: [],
379: [],
380: [],
381: [],
```

```
382: [],
383: [],
384: [],
385: [],
386: [],
387: [],
388: [],
389: [],
390: [],
391: [],
392: [],
393: [],
394: [],
395: [],
396: [],
397: [],
398: [],
399: [],
400: [],
401: [],
402: [],
403: [],
404: [],
405: [],
406: [],
407: [],
408: [],
409: [],
410: [],
411: [],
412: [],
413: [],
414: [],
415: [],
416: [],
417: [],
418: [],
419: [],
420: [],
421: [],
422: [],
423: [],
424: [],
425: [],
426: [],
427: [],
428: [],
429: [],
430: [],
431: [],
432: [],
433: [],
434: [],
435: [],
436: [],
437: [],
438: [],
439: [],
440: [],
441: [],
442: [],
443: [],
444: [],
445: [],
446: [],
447: [],
448: [],
449: [],
450: [],
451: [],
452: [],
453: [],
```

```
454: [],
455: [],
456: [],
457: [],
458: [],
459: [],
460: [],
461: [],
462: [],
463: [],
464: [],
465: [],
466: [],
467: [],
468: [],
469: [],
470: [],
471: [],
472: [],
473: [],
474: [],
475: [],
476: [],
477: [],
478: [],
479: [],
480: [],
481: [],
482: [],
483: [],
484: [],
485: [],
486: [],
487: [],
488: [],
489: [],
490: [],
491: [],
492: [],
493: [],
494: [],
495: [],
496: [],
497: [],
498: [],
499: [],
500: [],
501: [],
502: [],
503: [],
504: [],
505: [],
506: [],
507: [],
508: [],
509: [],
510: [],
511: [],
512: [],
513: [],
514: [],
515: [],
516: [],
517: [],
518: [],
519: [],
520: [],
521: [],
522: [],
523: [],
524: [],
525: [],
```

```
526: [],
527: [],
528: [],
529: [],
530: [],
531: [],
532: [],
533: [],
534: [],
535: [],
536: [],
537: [],
538: [],
539: [],
540: [],
541: [],
542: [],
543: [],
544: [],
545: [],
546: [],
547: [],
548: [],
549: [],
550: [],
551: [],
552: [],
553: [],
554: [],
555: [],
556: [],
557: [],
558: [],
559: [],
560: [],
561: [],
562: [],
563: [],
564: [],
565: [],
566: [],
567: [],
568: [],
569: [],
570: [],
571: [],
572: [],
573: [],
574: [],
575: [],
576: [],
577: [],
578: [],
579: [],
580: [],
581: [],
582: [],
583: [],
584: [],
585: [],
586: [],
587: [],
588: [],
589: [],
590: [],
591: [],
592: [],
593: [],
594: [],
595: [],
596: [],
597: [],
```

```
598: [],
599: [],
600: [],
601: [],
602: [],
603: [],
604: [],
605: [],
606: [],
607: [],
608: [],
609: [],
610: [],
611: [],
612: [],
613: [],
614: [],
615: [],
616: [],
617: [],
618: [],
619: [],
620: [],
621: [],
622: [],
623: [],
624: [],
625: [],
626: [],
627: [],
628: [],
629: [],
630: [],
631: [],
632: [],
633: [],
634: [],
635: [],
636: [],
637: [],
638: [],
639: [],
640: [],
641: [],
642: [],
643: [],
644: [],
645: [],
646: [],
647: [],
648: [],
649: [],
650: [],
651: [],
652: [],
653: [],
654: [],
655: [],
656: [],
657: [],
658: [],
659: [],
660: [],
661: [],
662: [],
663: [],
664: [],
665: [],
666: [],
667: [],
668: [],
669: [],
```

```
670: [],
671: [],
672: [],
673: [],
674: [],
675: [],
676: [],
677: [],
678: [],
679: [],
680: [],
681: [],
682: [],
683: [],
684: [],
685: [],
686: [],
687: [],
688: [],
689: [],
690: [],
691: [],
692: [],
693: [],
694: [],
695: [],
696: [],
697: [],
698: [],
699: [],
700: [],
701: [],
702: [],
703: [],
704: [],
705: [],
706: [],
707: [],
708: [],
709: [],
710: [],
711: [],
712: [],
713: [],
714: [],
715: [],
716: [],
717: [],
718: [],
719: [],
720: [],
721: [],
722: [],
723: [],
724: [],
725: [],
726: [],
727: [],
728: [],
729: [],
730: [],
731: [],
732: [],
733: [],
734: [],
735: [],
736: [],
737: [],
738: [],
739: [],
740: [],
741: [],
```

```
742: [],
743: [],
744: [],
745: [],
746: [],
747: [],
748: [],
749: [],
750: [],
751: [],
752: [],
753: [],
754: [],
755: [],
756: [],
757: [],
758: [],
759: [],
760: [],
761: [],
762: [],
763: [],
764: [],
765: [],
766: [],
767: [],
768: [],
769: [],
770: [],
771: [],
772: [],
773: [],
774: [],
775: [],
776: [],
777: [],
778: [],
779: [],
780: [],
781: [],
782: [],
783: [],
784: [],
785: [],
786: [],
787: [],
788: [],
789: [],
790: [],
791: [],
792: [],
793: [],
794: [],
795: [],
796: [],
797: [],
798: [],
799: [],
800: [],
801: [],
802: [],
803: [],
804: [],
805: [],
806: [],
807: [],
808: [],
809: [],
810: [],
811: [],
812: [],
813: [],
```

```
814: [],
815: [],
816: [],
817: [],
818: [],
819: [],
820: [],
821: [],
822: [],
823: [],
824: [],
825: [],
826: [],
827: [],
828: [],
829: [],
830: [],
831: [],
832: [],
833: [],
834: [],
835: [],
836: [],
837: [],
838: [],
839: [],
840: [],
841: [],
842: [],
843: [],
844: [],
845: [],
846: [],
847: [],
848: [],
849: [],
850: [],
851: [],
852: [],
853: [],
854: [],
855: [],
856: [],
857: [],
858: [],
859: [],
860: [],
861: [],
862: [],
863: [],
864: [],
865: [],
866: [],
867: [],
868: [],
869: [],
870: [],
871: [],
872: [],
873: [],
874: [],
875: [],
876: [],
877: [],
878: [],
879: [],
880: [],
881: [],
882: [],
883: [],
884: [],
885: [],
```

```
886: [],
887: [],
888: [],
889: [],
890: [],
891: [],
892: [],
893: [],
894: [],
895: [],
896: [],
897: [],
898: [],
899: [],
900: [],
901: [],
902: [],
903: [],
904: [],
905: [],
906: [],
907: [],
908: [],
909: [],
910: [],
911: [],
912: [],
913: [],
914: [],
915: [],
916: [],
917: [],
918: [],
919: [],
920: [],
921: [],
922: [],
923: [],
924: [],
925: [],
926: [],
927: [],
928: [],
929: [],
930: [],
931: [],
932: [],
933: [],
934: [],
935: [],
936: [],
937: [],
938: [],
939: [],
940: [],
941: [],
942: [],
943: [],
944: [],
945: [],
946: [],
947: [],
948: [],
949: [],
950: [],
951: [],
952: [],
953: [],
954: [],
955: [],
956: [],
957: [],
```

```
958: [],
959: [],
960: [],
961: [],
962: [],
963: [],
964: [],
965: [],
966: [],
967: [],
968: [],
969: [],
970: [],
971: [],
972: [],
973: [],
974: [],
975: [],
976: [],
977: [],
978: [],
979: [],
980: [],
981: [],
982: [],
983: [],
984: [],
985: [],
986: [],
987: [],
988: [],
989: [],
990: [],
991: [],
992: [],
993: [],
994: [],
995: [],
996: [],
997: [],
998: [],
999: [],
...}
```

Question 3: Algorithm Evaluation

1. **To investigate the extent to which plagiarism has been committed, we need to do more work on what we already have to understand the plagiarism percentage and where plagiarism occurs. What we currently have is a list of tuples detecting where there will be a match of k-length substrings in two inputs. To build upon this, we need to have a function calculating the percentage of plagiarism. We alsos need to construct a plagiarism distribution plot to understand more clearly which part in two strings witnesses more string similarities**
2. **Pitfalls and Assumptions: The algorithm heavily rests on a few assumptions, which a bit weakens its applicability in practice:**
    - **In my cleaning string function, I only accept normal letters. This assumes that plagiarism only happens to alphabetical letters, and excludes numbers, uppercase letters, punctuations or even symbols in different languages. As a result, the plagiarism detector cannot work well because it might fail to detect the real plagiarism in other aspects of writing. For example, if I included statistics or figures in my paper with changes in expressions and did not include the in-text citation as well, I should be counted as plagiarism. However, in this plagiarism detector, by eliminating numbers or double quotation marks, different expressions will save me from sufferring plagiarism. In addition, without punctuations, a lot of different meanings will count as one. As a result, plagiarism in this case is entirely wrong.**
    - **Also, the mechanism of this plagiarism detector is to find k-length string matching. In other words, as long as we have k contiguous symbols the same in two strings, this should be part of plagiarism. This does not make sense and may commit false positive. To exemplify, in some topics, there might be**

overarching key words we will mainly use. For example, regarding environmental sustainability, we will often use words like protection, ecofriendly, reusable or recycle. This will contribute to the plagiarism percentage when using this algorithm. This can be addressed by increasing k. However, as justified above, with large k, the multiplication method will not work effectively. Furthermore, with large k, we might somehow overlook other authentic plagiarism parts.

- Finally, this algorithm assumes that plagiarism only happens between two strings. This assumes only one source of information to compare the work with. Therefore, in the future, we need to construct multi-source plagiarism detector by activating multi-string matching algorithms. This also serves as a shortcoming of the two above algorithms

3. Comparison with brute-force approach:

- A brute-force approach will compare the first character of the pattern with the first character of the string we want to examine. If this is a match, we move to the second of the pattern and the string. Otherwise, we move to the second character of the string and compare it with the first character of the pattern. We iteratively do it until we go through the whole string and a plagiarism is caught only when there is exactly a pattern found in the string. This method is super computationally expensive and time consuming because we have to check all possible cases.
- Instead, our plagiarism detector computes hash values for each substring and only compares those with the same hash values. This is much more efficient. As we can see in the experimental analysis below, even with 10000 characters, the rolling hashing algorithm still takes less than one second to complete.

Question 4: Time complexity and Computational Critique

Time complexity for rolling hashing algorithm:

Let m, n be the length of two inputted strings: $len(x) = n$, $len(y) = m$. Suppose than $n > m$

- Cleaning inputs: Theta of $(m + n)$
- Compute hash values for k-length x-substrings: $O(n)$
  - The first substring: $O(k)$ because we have to hash every single character
  - The rest: $O(n - k)$ because we are using rolling, we just have to hash the next character
- Find position corresponding to the hash value in the hash table: $O(n - k + 1)$
  - Find position in the hash table for one substring: $O(1)$
  - There are $n - k + 1$ substrings
- Compute hash values for k-length y-substrings and find their position in the hash table: $O(m)$ (because $O(m) > O(m - k + 1)$)
- Suppose we have a matches in one position of the table, we have to compare k-length y-substring with k-length x-substring: $O(ak)$

In the best case scenario, there are no collisions and all substrings of x and y are uniformly distributed to different positions to the hash tables. Therefore, there are no matches and the time complexity will be $O(n + m)$. This can be better written as $O(n)$ given that one string is longer than the other and $(m + n)$ is bounded by $2n$. We can reduce $O(2n)$ to $O(n)$.

In the worst case scenario, this is similar to the naive matching algorithm when all substrings of x and y fall into the same bucket. An example for this is when one string is entirely part of the other string and all k-length substrings are the same. As a result, we have to compare all substrings of x and y together, resulting in a time complexity of $O(m * n)$

In the average case, the result will be $O(m + n)$ or better written as $O(n)$ given that $n > m$ because the hash function should reduce collisions, which makes the number of matches or the length of substring become constant factors and possible to be eliminated.

Therefore, we expect to see a linear runtime in the time complexity of rolling hashing algorithm

Time complexity for regular hashing algorithm:

The only difference between regular hashing and rolling hashing is in the step of computing hash values for all substrings.

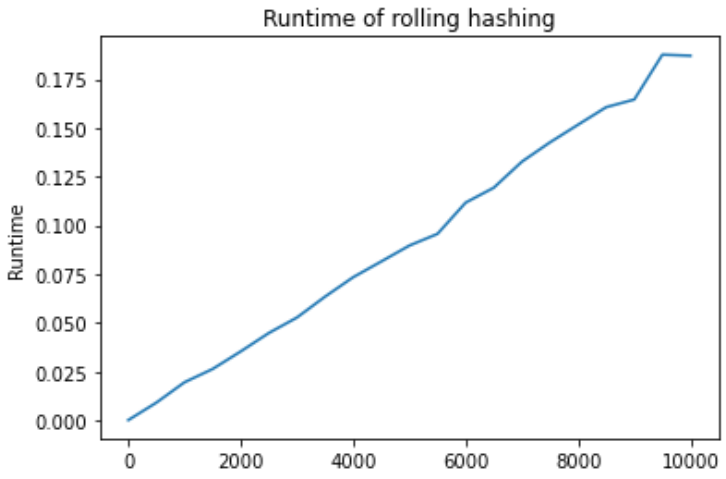
Instead of taking only $O(n)$ time to compute k-length x-substrings, it now takes $O(k*(n-k+1))$ because in each substring, we have to hash all characters in this k-length substring. The same applies to y-string.

However, because k is only a constant and in asymptotic analysis, we care more about the higher term. Therefore, we can also reduce the time complexity to only $O(n)$ in both best case and average case. In the worst case scenario, with the analysis above, the time complexity will also be $O(m*n)$.

We also expect a linear runtime of regular hashing for the above analysis. However, we will beware that we already remove more constant factors than we do for rolling hashing. Therefore, though both rolling and regular hashing take linear time, rolling hashing is supposed to take less time with smaller slope.

In [24]:

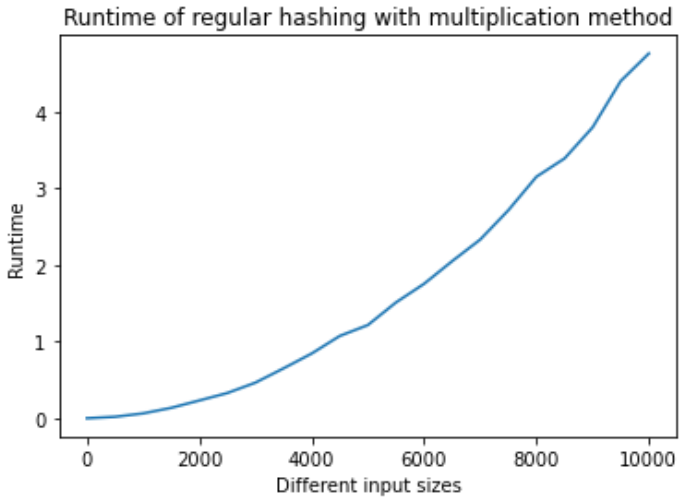
```
import string
import random
import time
import matplotlib.pyplot as plt

letters = string.ascii_lowercase

k = 15
size = 1319
rolling_runtime = []
input_sizes = [i for i in range(0, 10001, 500)]
for i in input_sizes:
    holder = 0
    for j in range(10):
        # get random text at different lengths
        x = ''.join(random.choice(letters) for i in range(i))
        y = ''.join(random.choice(letters) for i in range(i))

        start = time.time()
        rh_get_match(x, y, k, size)
        end = time.time()
        holder += (end - start)

    # get the average value
    rolling_runtime.append(holder / 10)

plt.plot(input_sizes, rolling_runtime)
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of rolling hashing')
plt.show()
```



Different input sizes

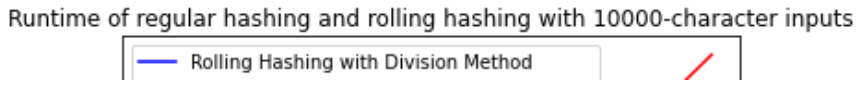
In [25]:

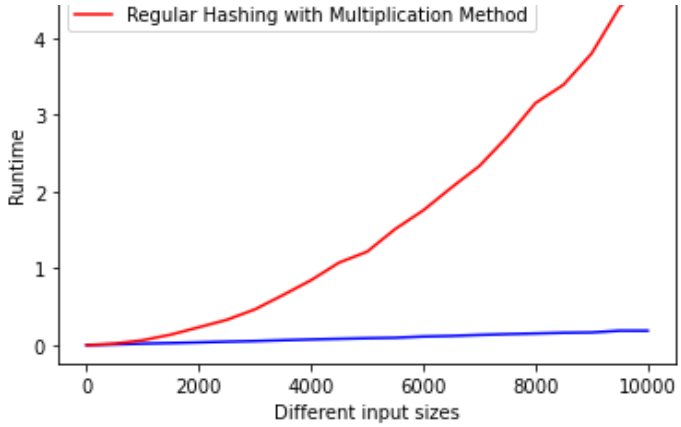
```
import string
import random
import time
import matplotlib.pyplot as plt
letters = string.ascii_lowercase

k = 15
size = 1319

multiplication = []

input_sizes = [i for i in range(0, 10001, 500)]

for i in input_sizes:

    holder_multiplication = 0

    for j in range(10):
        # get random text at different lengths
        x = ''.join(random.choice(letters) for iteration in range(i))
        y = ''.join(random.choice(letters) for iteration in range(i))

        start1 = time.time()
        regular_get_match(x, y, k)
        end1 = time.time()
        holder_multiplication += (end1 - start1)

    # get the average value
    multiplication.append(holder_multiplication / 10)

plt.plot(input_sizes, multiplication)
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of regular hashing with multiplication method')
plt.show()
```



In [26]:

```
plt.plot(input_sizes, rolling_runtime, color = 'blue', label = 'Rolling Hashing with Divi
sion Method')
plt.plot(input_sizes, multiplication, color = 'red', label = 'Regular Hashing with Multip
lication Method')
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of regular hashing and rolling hashing with 10000-character inputs')
plt.legend()
plt.show()
```

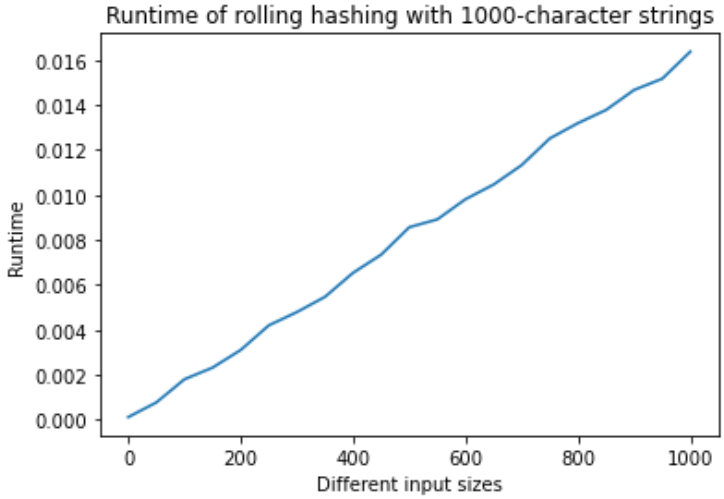




As expected, both algorithms take linear time to complete, yet with dramatic gaps in the slopes. Rolling hashing is apparently better, especially with larger input sizes. I noticed that at the beginning phase, the two algorithms seems to not differ that much. Therefore, I would love to conduct more experiments to zoom in this plot.

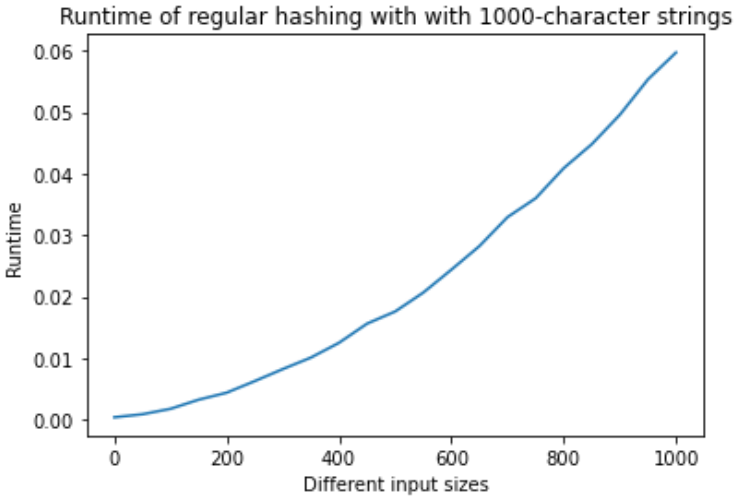
In [27]:

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

letters = string.ascii_lowercase

k = 15
size = 1319
rolling_runtime = []
input_sizes = [i for i in range(0, 1001, 50)]
for i in input_sizes:
    holder = 0
    for j in range(100):
        x = ''.join(random.choice(letters) for i in range(i))
        y = ''.join(random.choice(letters) for i in range(i))

        start = time.time()
        rh_get_match(x, y, k, size)
        end = time.time()
        holder += (end - start)

    rolling_runtime.append(holder / 100)

plt.plot(input_sizes, rolling_runtime)
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of rolling hashing with 1000-character strings')
plt.show()
```



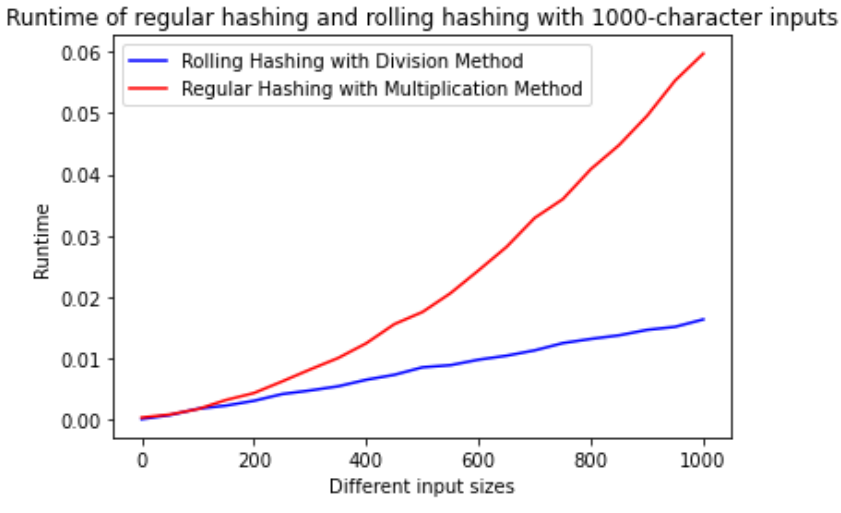
In [28]:

```
multiplication = []

input_sizes = [i for i in range(0, 1001, 50)]

for i in input_sizes:

    holder_multiplication = 0

    for j in range(100):

        x = ''.join(random.choice(letters) for iteration in range(i))
        y = ''.join(random.choice(letters) for iteration in range(i))

        start1 = time.time()
        regular_get_match(x, y, k)
        end1 = time.time()
        holder_multiplication += (end1 - start1)

    multiplication.append(holder_multiplication / 100)

plt.plot(input_sizes, multiplication)
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of regular hashing with with 1000-character strings')
plt.show()
```



In [29]:

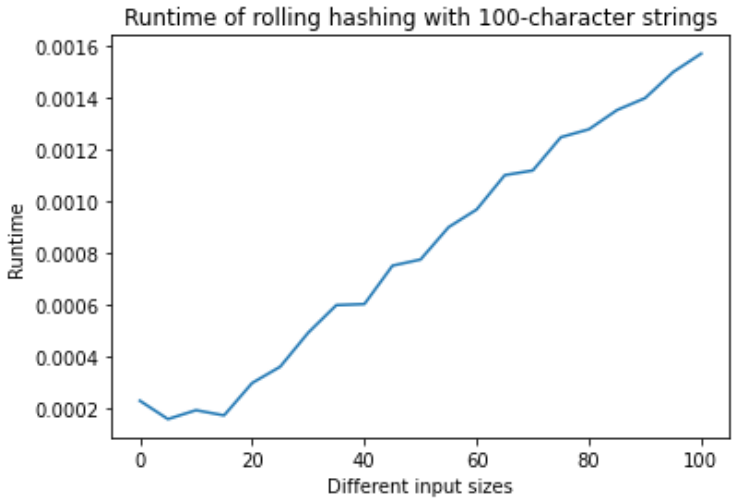
```
plt.plot(input_sizes, rolling_runtime, color = 'blue', label = 'Rolling Hashing with Division Method')
plt.plot(input_sizes, multiplication, color = 'red', label = 'Regular Hashing with Multiplication Method')
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of regular hashing and rolling hashing with 1000-character inputs')
plt.legend()
plt.show()
```



Linear time starts to become more clear in this graph, especially when I increase iterations to 100, which make the line smoother. With 1000-character inputs, the difference seems noticeable, yet not very considerable because it is only about 0.05-second difference. I would love to understand the algorithms' behavior with small size.

In [30]:

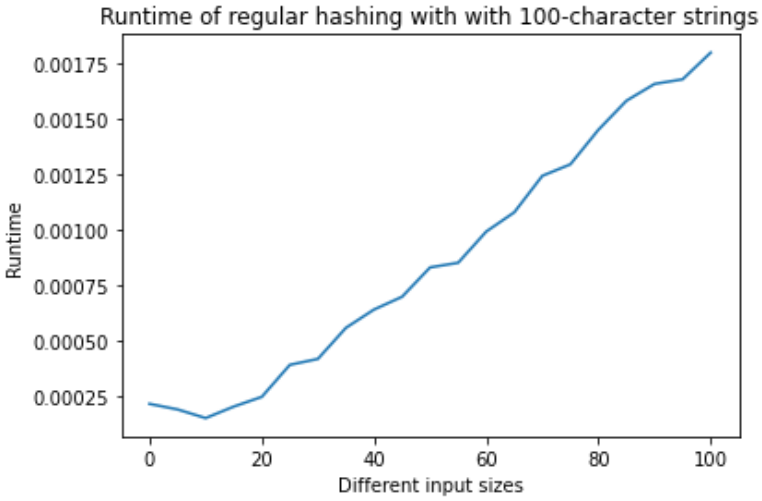
```
rolling_runtime = []
input_sizes = [i for i in range(0, 101, 5)]
for i in input_sizes:
    holder = 0
    for j in range(500):
        x = ''.join(random.choice(letters) for i in range(i))
        y = ''.join(random.choice(letters) for i in range(i))

        start = time.time()
        rh_get_match(x, y, k, size)
        end = time.time()
        holder += (end - start)

    rolling_runtime.append(holder / 500)

plt.plot(input_sizes, rolling_runtime)
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of rolling hashing with 100-character strings')
plt.show()
```



In [31]:

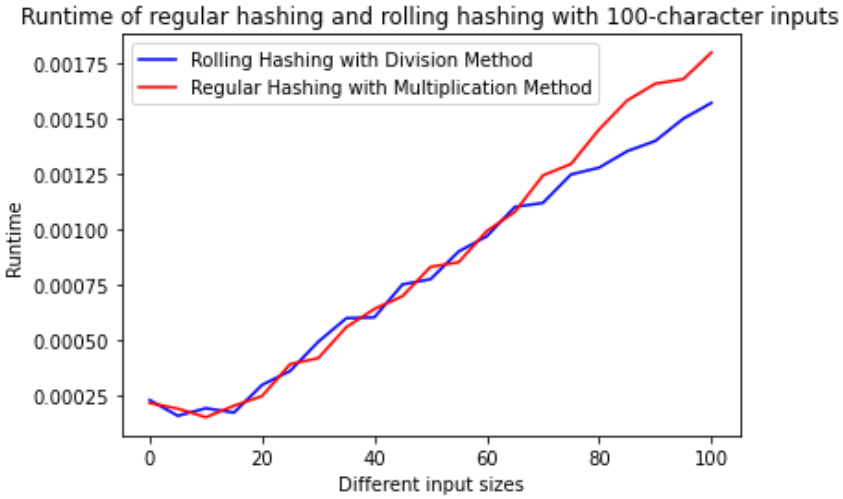
```
multiplication = []

input_sizes = [i for i in range(0, 101, 5)]

for i in input_sizes:

    holder_multiplication = 0

    for j in range(500):

        x = ''.join(random.choice(letters) for iteration in range(i))
        y = ''.join(random.choice(letters) for iteration in range(i))

        start1 = time.time()
        regular_get_match(x, y, k)
        end1 = time.time()
        holder_multiplication += (end1 - start1)

    multiplication.append(holder_multiplication / 500)

plt.plot(input_sizes, multiplication)
plt.xlabel('Different input sizes')
```

```
plt.ylabel('Runtime')
plt.title('Runtime of regular hashing with with 100-character strings')
plt.show()
```



In [32]:

```
plt.plot(input_sizes, rolling_runtime, color = 'blue', label = 'Rolling Hashing with Division Method')
plt.plot(input_sizes, multiplication, color = 'red', label = 'Regular Hashing with Multiplication Method')
plt.xlabel('Different input sizes')
plt.ylabel('Runtime')
plt.title('Runtime of regular hashing and rolling hashing with 100-character inputs')
plt.legend()
plt.show()
```



I even increased iterations in this experiment to 500 to make the line less bumpy. As predicted, the two algorithms now do not differ much at the beginning. Only from 60-character length and above, the regular hashing starts to develop greater slope. Therefore, we can consider it to be a threshold.

Question 5: LO and HC Appendix

LOs:

1. **#PythonProgramming: To get my code running with two algorithms, I tried to take full advantage of functionalities or operations I have learnt, from list manipulation for looping. This helps me so much in achieving my goals. Furthermore, in plotting the graph about two metrics of collision, I used techniques to normalize the whole list to the range from 0 to 1. This is helpful in conveying the behavior of the changes in two metrics.**
2. **#ComputationalCritique: To contrast the performance of two algorithms, rolling hashing and regular hashing, I tried my best to conduct different experiments on multiple natures of inputs. For both rolling and regular hashing, experiments on 10000 input size help me focus on the scaling behavior of two algorithms when the**

input sizes can massively big. In addition, to clearly understand the difference and closely engage in the two algorithms' performance, I gradually zoom in the plot with 1000-input size and 100-input size. This sheds light on the suitability of algorithms with different text lengths and helps me judge two algorithms more effectively. In additino, besides the experimental analysis, I also include the strong points of rolling hashing (simple calculation, good runtime, few collisions) as well as its limitations regarding space consumption. Furthermore, I also noticed the constraint of k-length substring in regular hashing because in this hash value computation, large k will not work effectively.

3. **#AlgorithmicStrategies: In the first question, I tried my best to describe my algorithmic strategy by dividing it into steps by steps and avoiding using jargon or coding terms. Instead, I used plain text and informal language to describe what the code does so that this will be more understood by the audience. Furthermore, I also focus on the nuances of the problem and justify why it is suitable to choose this algorithmic strategy. Along with that, I also compare the performance of this rolling hashing algorithm with the naive matching algorithm to strengthen the choice of rolling hashing. Comparing with competing alternatives by focusing on the nuances of the problem is such an effective combination of strategy in this scenario.**
4. **#CodeReadability: With the complementary explanation of HC #composition and #communicationdesign, I tried to include docstrings and code comments effectively and consistently to follow the conventions and help communicate with readers better.**

HCs:

1. **#dataviz: In this assignment, I conducted a lot of experiments to not only visualize the behavior of collision rate of each bucket's average capacity when the hash table changes but also contrast the runtime of two algorithms with multiple natures of input sizes. I tried my best to label the plots properly with correct title, axis labelling, color differentiation and effective normalization so that audience can easily understand what I am trying to communicate through these graphs. For example, in the graph about the collision rate and average capacity, I normalize the values of average capacity to the range from 0 to 1 so that viewers can easily identify the changes in two metrics with regards to hash table sizes. There might be a point where the reduction in collision is no longer worth the additional space we have to consume. That's the general idea I want to forewarn future users.**
2. **#communicationdesign: This assignment requires a lot of coding and experiments. Therefore, to better understand my work, I strategically divided codes into different blocks and different functions. Furthermore, I also included effective comments and markdown letters with different sizes to notify which part I am at. I believe that with clear divisions of different parts in the code and plotting part, viewers can understand better my thought process as well as my algorithm construction and evaluation in general.**
3. **#composition: I have practiced so long to write things succinctly so that conveying my message can be more effective. This is reflected in code comments and docstring. Sometimes, with complicated lines of code, I believe my comments are helpful with short key words and the docstring will also hep readers understand the general function of the code.**